OWASP
Open Web Application
Security Project

OWASP PROActive CONTROLS FOR DEVELOPERS

2018 v 3.0

10 Critical Security Areas That Software Developers Must Be Aware Of

PROJECT LEADERS

KATY ANTON
JIM MANICO
JIM BIRD

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.

FOREWORD

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.

AIM & OBJECTIF

The goal of the *OWASP Top 10 Proactive Controls project (OPC)* 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.

CALL TO ACTION

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.

COPYRIGHT AND LICENSE

PROJECT LEADERS

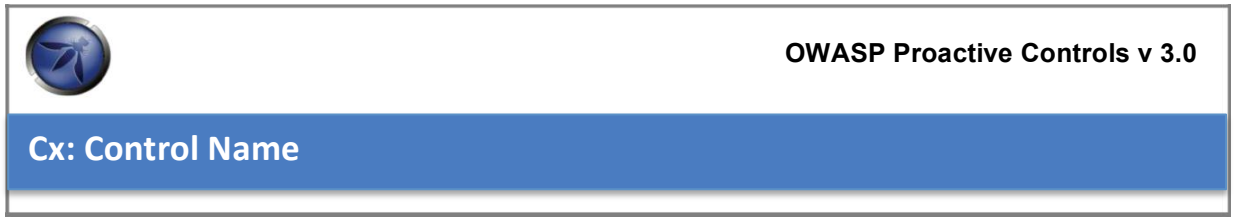
Katy Anton
Jim Bird
Jim Manico

CONTRIBUTORS

Chris Romeo
Dan Anderson
David Cybuck
Dave Ferguson
Josh Grossman
Osama Elnaggar
Colin Watson
Rick Mitchell
And many more…

DOCUMENT STRUCTURE

This document is structured as a list of security controls. Each control is described as follows:



Description

A detailed description of the control including some best practices to consider.

Implementation

Implementation best practices and examples to illustrate how to implement each control.

Vulnerabilities Prevented

List of prevented vulnerabilities or risks addressed (OWASP TOP 10 Risk, CWE, etc.)

References

List of references for further study (OWASP Cheat sheet, Security Hardening Guidelines, etc.)

Tools

Set of tools/projects to easily introduce/integrate security controls into your software.

INTRODUCTION

The OWASP Top Ten Proactive Controls 2018 is a list of security techniques that should be considered for every software development project. This document is written for developers to assist those new to secure development.

One of the main goals of this document is to provide concrete practical guidance that helps developers build secure software. These techniques should be applied proactively at the early stages of software development to ensure maximum effectiveness.

The Top 10 Proactive Controls

The list is ordered by importance with list item number 1 being the most important:

C1: Define Security Requirements

C2: Leverage Security Frameworks and Libraries

C3: Secure Database Access

C4: Encode and Escape Data

C5: Validate All Inputs

C6: Implement Digital Identity

C7: Enforce Access Controls

C8: Protect Data Everywhere

C9: Implement Security Logging and Monitoring

C10: Handle All Errors and Exceptions

How this List Was Created

This list was originally created by the current project leads with contributions from several volunteers. The document was then shared globally so even anonymous suggestions could be considered. Hundreds of changes were accepted from this open community process.

Target Audience

This document is primarily written for developers. However, development managers, product owners, Q/A professionals, program managers, and anyone involved in building software can also benefit from this document.

How to Use this Document

This document is intended to provide initial awareness around building secure software. This document will also provide a good foundation of topics to help drive introductory software security developer training. These controls should be used consistently and thoroughly throughout all applications. However, this document should be seen as a starting point rather than a comprehensive set of techniques and practices. A full secure development process should include comprehensive requirements from a standard such as the OWASP ASVS in addition to including a range of software development activities described in maturity models such as OWASP SAMM and BSIMM.

Link to the OWASP Top 10 Project

The OWASP Top 10 Proactive Controls is similar to the OWASP Top 10 but is focused on defensive techniques and controls as opposed to risks. Each technique or control in this document will map to one or more items in the *risk based* OWASP Top 10. This mapping information is included at the end of each control description.

OWASP Proactive Controls v 3.0

C1: Define Security Requirements

Description

A security requirement is a statement of needed security functionality that ensures one of many different security properties of software is being satisfied. Security requirements are derived from industry standards, applicable laws, and a history of past vulnerabilities. Security requirements define new features or additions to existing features to solve a specific security problem or eliminate a potential vulnerability.

Security requirements provide a foundation of vetted security functionality for an application. Instead of creating a custom approach to security for every application, standard security requirements allow developers to reuse the definition of security controls and best practices. Those same vetted security requirements provide solutions for security issues that have occurred in the past. Requirements exist to prevent the repeat of past security failures.

The OWASP ASVS

The OWASP Application Security Verification Standard (ASVS) is a catalog of available security requirements and verification criteria. OWASP ASVS can be a source of detailed security requirements for development teams.

Security requirements are categorized into different buckets based on a shared higher order security function. For example, the ASVS contains categories such as authentication, access control, error handling / logging, and web services. Each category contains a collection of requirements that represent the best practices for that category drafted as verifiable statements.

Augmenting Requirements with User Stories and Misuse Cases

The ASVS requirements are basic verifiable statements which can be expanded upon with user stories and misuse cases. The advantage of a user story or misuse case is that it ties the application to exactly what the user or attacker does to the system, versus describing what the system offers to the user.

Here is an example of expanding on an ASVS 3.0.1 requirement. From the "Authentication Verification Requirements" section of ASVS 3.0.1, requirement 2.19 focuses on default passwords.

> 2.19 Verify there are no default passwords in use for the application framework or any components used by the application (such as "admin/password").

This requirement contains both an action to verify that no default passwords exist, and also carries with it the guidance that no default passwords should be used within the application.

A user story focuses on the perspective of the user, administrator, or attacker of the system, and describes functionality based on what a user wants the system to do for them. A user story takes the form of "As a user, I can do x, y, and z".

```
As a user, I can enter my username and password to gain access to the
application.

As a user, I can enter a long password that has a maximum of 1023
characters.
```

When the story is focused on the attacker and their actions, it is referred to as a misuse case.

```
As an attacker, I can enter in a default username and password to gain
access.
```

This story contains the same message as the traditional requirement from ASVS, with additional user or attacker details to help make the requirement more testable.

Implementation

Successful use of security requirements involves four steps. The process includes discovering / selecting, documenting, implementing, and then confirming correct implementation of new security features and functionality within an application.

Discovery and Selection

The process begins with discovery and selection of security requirements. In this phase, the developer is understanding security requirements from a standard source such as ASVS and choosing which requirements to include for a given release of an application. The point of discovery and selection is to choose a manageable number of security requirements for this release or sprint, and then continue to iterate for each sprint, adding more security functionality over time.

Investigation and Documentation

During investigation and documentation, the developer reviews the existing application against the new set of security requirements to determine whether the application currently meets the requirement or if some development is required. This investigation culminates in the documentation of the results of the review.

Implementation and Test

After the need is determined for development, the developer must now modify the application in some way to add the new functionality or eliminate an insecure option. In this phase the developer first determines the design required to address the requirement, and then completes the code changes to meet the requirement. Test cases should be created to confirm the existence of the new functionality or disprove the existence of a previously insecure option.

Vulnerabilities Prevented

Security requirements define the security functionality of an application. Better security built in from the beginning of an applications life cycle results in the prevention of many types of vulnerabilities.

References

- OWASP Application Security Verification Standard (ASVS)
- OWASP Mobile Application Security Verification Standard (MASVS)
- OWASP Top Ten

OWASP Proactive Controls v 3.0

C2: Leverage Security Frameworks and Libraries

Description

Secure coding libraries and software frameworks with embedded security help software developers guard against security-related design and implementation flaws. A developer writing an application from scratch might not have sufficient knowledge, time, or budget to properly implement or maintain security features. Leveraging security frameworks helps accomplish security goals more efficiently and accurately.

Implementation Best Practices

When incorporating third party libraries or frameworks into your software, it is important to consider the following best practices:

1. Use libraries and frameworks from trusted sources that are actively maintained and widely used by many applications.
2. Create and maintain an inventory catalog of all the third party libraries.
3. Proactively keep libraries and components up to date. Use a tool like OWASP Dependency Check and Retire.JS to identify project dependencies and check if there are any known, publicly disclosed vulnerabilities for all third party code.
4. Reduce the attack surface by encapsulating the library and expose only the required behaviour into your software.

Vulnerabilities Prevented

Secure frameworks and libraries can help to prevent a wide range of web application vulnerabilities. It is critical to keep these frameworks and libraries up to date as described in the using components with known vulnerabilities Top Ten 2017 risk.

Tools

- OWASP Dependency Check - identifies project dependencies and checks for publicly disclosed vulnerabilities
- Retire.JS scanner for JavaScript libraries

OWASP Proactive Controls v 3.0

C3: Secure Database Access

Description

This section describes secure access to all data stores, including both relational databases and NoSQL databases. Some areas to consider:

1. Secure queries
2. Secure configuration
3. Secure authentication
4. Secure communication

Secure Queries

SQL Injection occurs when untrusted user input is dynamically added to a SQL query in an insecure manner, often via basic string concatenation. SQL Injection is one of the most dangerous application security risks. SQL Injection is easy to exploit and could lead to the entire database being stolen, wiped, or modified. The application can even be used to run dangerous commands against the operating system hosting your database, thereby giving an attacker a foothold on your network.

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'. This defense should be applied to SQL, OQL, as well as stored procedure construction.

A good list of query parameterization examples in ASP , ColdFusion , C# , Delphi, .NET , Go , Java , Perl , PHP , PL/SQL , PostgreSQL, Python , R , Ruby and Scheme can be found at http://bobby-tables.com and the OWASP Cheat Sheet on Query Parameterization.

Caution on Query Parameterization

Certain locations in a database query are not parameterizable. These locations are different for each database vendor. Be certain to do very careful exact-match validation or manual escaping when confronting database query parameters that cannot be bound to a parameterized query. Also, while the use of parameterized queries largely has a positive impact on performance, certain parameterized queries in specific database implementations will affect performance negatively. Be sure to test queries for performance; especially complex queries with extensive like clause or text searching capabilities.

Secure Configuration

Unfortunately, database management systems do not always ship in a "secure by default" configuration. Care must be taken to ensure that the security controls available from the Database Management System (DBMS) and hosting platform are enabled and properly configured. There are standards, guides, and benchmarks available for most common DBMS.

Secure Authentication

All access to the database should be properly authenticated. Authentication to the DBMS should be accomplished in a secure manner. Authentication should take place only over a secure channel. Credentials must be properly secured and available for use.

Secure Communication

Most DBMS support a variety of communications methods (services, APIs, etc) - secure (authenticated, encrypted) and insecure (unauthenticated or unencrypted). It is a good practice to only use the secure communications options per the *Protect Data Everywhere* control.

Vulnerabilities Prevented

- OWASP Top 10 2017- A1: Injection
- OWASP Mobile Top 10 2014-M1 Weak Server Side Controls

References

- OWASP Cheat Sheet: Query Parameterization
- Bobby Tables: A guide to preventing SQL injection
- CIS Database Hardening Standards

OWASP Proactive Controls v 3.0

C4: Encode and Escape Data

Description

Encoding and escaping are defensive techniques meant to stop injection attacks. Encoding (commonly called "Output Encoding") involves translating special characters into some different but equivalent form that is no longer dangerous in the target interpreter, for example translating the "<" character into the < string when writing to an HTML page. Escaping involves adding a special character before the character/string to avoid it being misinterpreted, for example, adding a "\" character before a "**"**" (double quote) character so that it is interpreted as text and not as closing a string.

Output encoding is best applied just before the content is passed to the target interpreter. If this defense is performed too early in the processing of a request then the encoding or escaping may interfere with the use of the content in other parts of the program. For example if you HTML escape content before storing that data in the database and the UI automatically escapes that data a second time then the content will not display properly due to being double escaped.

Contextual Output Encoding

Contextual output encoding is a crucial security programming technique needed to stop XSS. This defense 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 will depend on the location (or context) in the document where data is being displayed or stored. The different types of encoding that would be used for building secure user interfaces includes HTML Entity Encoding, HTML Attribute Encoding, JavaScript Encoding, and URL Encoding.

Java Encoding Examples

For examples of the OWASP Java Encoder providing contextual output encoding see: OWASP Java Encoder Project Examples.

.NET Encoding Examples

Starting with .NET 4.5 , the Anti-Cross Site Scripting library is part of the framework, but not enabled by default. You can specify to use AntiXssEncoder from this library as the default encoder for your entire application using the web.conf settings. When applied is important to contextual encode your output - that means to use the right function from the AntiXSSEncoder library for the appropriate location of data in document.

PHP Encoding Examples

Zend Framework 2

In Zend Framework 2 (ZF2), Zend\Escaper can be used for encoding the output. For contextual encoding examples see Context-specific escaping with zend-escaper.

Other Types of Encoding and Injection Defense

Encoding/Escaping can be used to neutralize content against other forms of injection. For example, it's possible to neutralize certain special meta-characters when adding input to an operating system command. This is called "OS command escaping", "shell escaping", or similar. This defense can be used to stop "Command Injection" vulnerabilities.

There are other forms of escaping that can be used to stop injection such as XML attribute escaping stopping various forms of XML and XML path injection, as well as LDAP distinguished name escaping that can be used to stop various forms of LDAP injection.

Character Encoding and Canonicalization

Unicode Encoding is a method for storing characters with multiple bytes. Wherever input data is allowed, data can be entered using Unicode to disguise malicious code and permit a variety of attacks. RFC 2279 references many ways that text can be encoded.

Canonicalization is a method in which systems convert data into a simple or standard form. Web applications commonly use character canonicalization to ensure all content is of the same character type when stored or displayed.

To be secure against canonicalization related attacks means an application should be safe when malformed Unicode and other malformed character representations are entered.

Vulnerabilities Prevented

- OWASP Top 10 2017 - A1: Injection
- OWASP Top 10 2017 - A7: Cross Site Scripting (XSS)
- OWASP Mobile_Top_10_2014-M7 Client Side Injection

References

- XSS - General information
- OWASP Cheat Sheet: XSS Prevention - Stopping XSS in your web application
- OWASP Cheat Sheet: DOM based XSS Prevention
- OWASP Cheat Sheet: Injection Prevention

Tools

- OWASP Java Encoder Project
- AntiXSSEncoder
- Zend\Escaper - examples of contextual encoding

OWASP Proactive Controls v 3.0

C5: Validate All Inputs

Description

Input validation is a programming technique that ensures only properly formatted data may enter a software system component.

Syntax and Semantic Validity

An application should check that data is both *syntactically* and *semantically* valid (in that order) before using it in any way (including displaying it back to the user).

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 includes only accepting input that is within an acceptable range for the given application functionality and context. For example, a start date must be before an end date when choosing date ranges.

Whitelisting vs Blacklisting

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

- *Blacklisting* or *blacklist validation* attempts to check that given data does not contain "known bad" content. For example, a web application may block input that contains the exact text <SCRIPT> in order to help prevent XSS. However, this defense could be evaded with a lower case script tag or a script tag of mixed case.
- *Whitelisting* or *whitelist validation* attempts to check that a given data matches a set of "known good" rules. For example a whitelist validation rule for a US state would be a 2-letter code that is only one of the valid US states.

When building secure software, whitelisting is the recommended minimal approach. Blacklisting is prone to error and can be bypassed with various evasion techniques and can be dangerous when depended on by itself. Even though blacklisting can often be evaded it can

often useful to help detect obvious attacks. So while *whitelisting* helps limit the attack surface by ensuring data is of the right syntactic and semantic validity, *blacklisting* helps detect and potentially stop obvious attacks.

Client side and Server side Validation

Input validation must always be done on the server-side for security. While client side validation can be useful for both functional and some security purposes it can often be easily bypassed. This makes server-side validation even more fundamental to security. For example, JavaScript validation may alert the user that a particular field must consist of numbers but the server side application must validate that the submitted data only consists of numbers in the appropriate numerical range for that feature.

Regular Expressions

Regular expressions offer a way to check whether data matches a specific pattern. Let's start with a basic example.

The following regular expression is used to define a whitelist rule to validate usernames.

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

This regular expression allows only lowercase letters, numbers and the underscore character. The username is also restricted to a length of 3 and 16 characters.

Caution: Potential for Denial of Service

Care should be exercised when creating regular expressions. Poorly designed expressions may result in potential denial of service conditions (aka ReDoS). Various tools can test to verify that regular expressions are not vulnerable to ReDoS.

Caution: Complexity

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 programmatically which can be easier to maintain for some developers.

Limits of Input Validation

Input validation does not always make data “safe” since certain forms of complex input may be "valid" but still dangerous. For example a valid email address may contain a SQL injection attack or a valid URL may contain a Cross Site Scripting attack. Additional defenses besides input validation should always be applied to data such as query parameterization or escaping.

Challenges of Validating Serialized Data

Some forms of input are so complex that validation can only minimally protect the application. For example, it's dangerous to deserialize untrusted data or data that can be manipulated by an attacker. The only safe architectural pattern is to not accept serialized objects from untrusted sources or to only deserialize in limited capacity for only simple data types. You should avoid processing serialized data formats and use easier to defend formats such as JSON when possible.

If that is not possible then consider a series of validation defenses when processing serialized data.

- Implement integrity checks or encryption of the serialized objects to prevent hostile object creation or data tampering.
- Enforce strict type constraints during deserialization before object creation; typically code is expecting a definable set of classes. Bypasses to this technique have been demonstrated.
- Isolate code that deserializes, such that it runs in very low privilege environments, such as temporary containers.
- Log security deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.
- Restrict or monitor incoming and outgoing network connectivity from containers or servers that deserialize.
- Monitor deserialization, alerting if a user deserializes constantly.

Unexpected User Input (Mass Assignment)

Some frameworks support automatic binding of HTTP requests parameters to server-side objects used by the application. This auto-binding feature can allow an attacker to update server-side objects that were not meant to be modified. The attacker can possibly modify their access control level or circumvent the intended business logic of the application with this feature.

This attack has a number of names including: mass assignment, autobinding and object injection.

As a simple example, if the user object has a field privilege which specifies the user's privilege level in the application, a malicious user can look for pages where user data is modified and add privilege=admin to the HTTP parameters sent. If auto-binding is enabled in an insecure fashion, the server-side object representing the user will be modified accordingly.

Two approaches can be used to handle this:

- Avoid binding input directly and use Data Transfer Objects (DTOs) instead.
- Enable auto-binding but set up whitelist rules for each page or feature to define which fields are allowed to be auto-bound.

More examples are available in the OWASP Mass Assignment Cheat Sheet.

Validating and Sanitizing HTML

Consider an application that needs to accept HTML from users (via a WYSIWYG editor that represents content as HTML or features that directly accept HTML in input). In this situation validation or escaping will not help.

- Regular expressions are not expressive enough to understand the complexity of HTML5.
- Encoding or escaping HTML will not help since it will cause the HTML to not render properly.

Therefore, you need a library that can parse and clean HTML formatted text. Please see the XSS Prevention Cheat Sheet on HTML Sanitization for more information on HTML Sanitization.

Validation Functionality in Libraries and Frameworks

All languages and most frameworks provide validation libraries or functions which should be leveraged to validate data. Validation libraries typically cover common data types, length

requirements, integer ranges, "is null" checks and more. Many validation libraries and frameworks allow you to define your own regular expression or logic for custom validation in a way that allows the programmer to leverage that functionality throughout your application. Examples of validation functionality include PHP's filter functions or the Hibernate Validator for Java. Examples of HTML Sanitizers include Ruby on Rails sanitize method, OWASP Java HTML Sanitizer or DOMPurify.

Vulnerabilities Prevented

- Input validation reduces the attack surface of applications and can sometimes make attacks more difficult against an application.
- Input validation is a technique that provides security to certain forms of data, specific to certain attacks and cannot be reliably applied as a general security rule.
- Input validation should not be used as the primary method of preventing XSS, SQL Injection and other attacks.

References

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

Tools

- OWASP Java HTML Sanitizer Project
- Java JSR-303/JSR-349 Bean Validation
- Java Hibernate Validator
- JEP-290 Filter Incoming Serialization Data
- Apache Commons Validator
- PHP's filter functions

OWASP Proactive Controls v 3.0

C6: Implement Digital Identity

Description

Digital Identity is the unique representation of a user (or other subject) as they engage in an online transaction. Authentication is the process of verifying that an individual or entity is who they claim to be. Session management is a process by which a server maintains the state of the users authentication so that the user may continue to use the system without re-authenticating. The NIST Special Publication 800-63B: Digital Identity Guidelines (Authentication and Lifecycle Management provides solid guidance on implementing digital identity, authentication and session management controls.

Below are some recommendations for secure implementation.

Authentication Levels

NIST 800-63b describes three levels of a authentication assurance called a authentication assurance level (AAL). AAL level 1 is reserved for lower-risk applications that do not contain PII or other private data. At AAL level 1 only single-factor authentication is required, typically through the use of a password.

Level 1 : Passwords

Passwords are really really important. We need policy, we need to store them securely, we need to sometimes allow users to reset them.

Password Requirements

Passwords should comply with the following requirements at the very least:

- be at least 8 characters in length if multi-factor authentication (MFA) and other controls are also used. If MFA is not possible, this should be increased to at least 10 characters
- all printing ASCII characters as well as the space character should be acceptable in memorized secrets
- encourage the use of long passwords and passphrases

- remove complexity requirements as these have been found to be of limited effectiveness. Instead, the adoption of MFA or longer password lengths is recommended
- ensure that passwords used are not commonly used passwords that have been already been leaked in a previous compromise. You may choose to block the top 1000 or 10000 most common passwords which meet the above length requirements and are found in compromised password lists. The following link contains the most commonly found passwords: https://github.com/danielmiessler/SecLists/tree/master/Passwords

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, it may ask a security question - something they know, and then send a generated token to a device - something they own.

Please see the Forgot_Password_Cheat_Sheet and Choosing_and_Using_Security_Questions_Cheat_Sheet for further details.

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.

PHP Example for Password Storage

Below is an example for secure 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] );
?>
```

Please see the OWASP Password Storage Cheat Sheet for further details.

Level 2 : Multi-Factor Authentication

NIST 800-63b AAL level 2 is reserved for higher-risk applications that contain "self-asserted PII or other personal information made available online." At AAL level 2 multi-factor authentication is required including OTP or other forms of multi-factor implementation.

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

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

Using passwords as a sole factor provides weak security. Multi-factor solutions provide a more robust solution by requiring an attacker to acquire more than one element to authenticate with the service..

It is worth noting that biometrics, when employed as a single factor of authentication, are not considered acceptable secrets for digital authentication. They can be obtained online or by taking a picture of someone with a camera phone (e.g., facial images) with or without their knowledge, lifted from objects someone touches (e.g., latent fingerprints), or captured with high resolution images (e.g., iris patterns). Biometrics must be used only as part of multi-factor authentication with a physical authenticator (something you own). For example, accessing a multi-factor one-time password (OTP) device that will generate a one-time password that the user manually enters for the verifier.

Level 3 : Cryptographic Based Authentication

NIST 800-63b Authentication Assurance Level 3 (AAL3) is required when the impact of compromised systems could lead to personal harm, significant financial loss, harm the public interest or involve civil or criminal violations. AAL3 requires authentication that is "based on proof of possession of a key through a cryptographic protocol." This type of authentication is used to achieve the strongest level of authentication assurance. This is typically done though hardware cryptographic modules.

Session Management

Once the initial successful user authentication has taken place, an application may choose to track and maintain this authentication state for a limited amount of time. This will allow the user to continue using the application without having to keep re-authentication with each request. Tracking of this user state is called Session Management.

Session Generation and Expiration

User state is tracked in a session. This session is typically stored on the server for traditional web based session management. A session identifier is then given to the user so the user can identify which server-side session contains the correct user data. The client only needs to maintain this session identifier, which also keeps sensitive server-side session data off of the client.

Here are a few controls to consider when building or implementing session management solutions:

- Ensure that the session id is long, unique and random.
- The application should generate a new session or at least rotate the session id during authentication and re-authentication.
- The application should implement an idle timeout after a period of inactivity and an absolute maximum lifetime for each session, after which users must re-authenticate. The length of the timeouts should be inversely proportional with the value of the data protected.

Please see the Session Management Cheat Sheet further details. ASVS Section 3 covers additional session management requirements.

Browser Cookies

Browser cookies are a common method for web application to store session identifiers for web applications implementing standard session management techniques. Here are some defenses to consider when using browser cookies.

- When browser cookies are used as the mechanism for tracking the session of an authenticated user, these should be accessible to a minimum set of domains and paths and should be tagged to expire at, or soon after, the session's validity period.
- The 'secure' flag should be set to ensure the transfer is done via secure channel only (TLS).

- HttpOnly flag should be set to prevent the cookie from being accessed via JavaScript.
- Adding "samesite" attributes to cookies prevents some modern browsers from sending cookies with cross-site requests and provides protection against cross-site request forgery and information leakage attacks.

Tokens

Server-side sessions can be limiting for some forms of authentication. "Stateless services" allow for client side management of session data for performance purposes so they server has less of a burden to store and verify user session. These "stateless" applications generate a short-lived access token which can be used to authenticate a client request without sending the user's credentials after the initial authentication.

JWT (JSON Web Tokens)

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. A JWT token is created during authentication and is verified by the server (or servers) before any processing.

However, JWT's are often not saved by the server after initial creation. JWT's are typically created and then handed to a client without being saved by the server in any way. The integrity of the token is maintained through the use of digital signatures so a server can later verify that the JWT is still valid and was not tampered with since its creation.

This approach is both stateless and portable in the way that client and server technologies can be different yet still interact.

Caution

Digital identity, authentication and session management are very big topics. We're scratching the surface of the topic of Digital Identity here. Ensure that your most capable engineering talent is responsible for maintaining the complexity involved with most Identity solutions.

Vulnerabilities Prevented

- OWASP Top 10 2017 A2- Broken Authentication and Session Management
- OWASP Mobile Top 10 2014-M5- Poor Authorization and Authentication

References

- OWASP Cheat Sheet: Authentication
- OWASP Cheat Sheet: Password Storage
- OWASP Cheat Sheet: Forgot Password
- OWASP Cheat Sheet: Choosing and Using Security Questions
- OWASP Cheat Sheet: Session Management
- OWASP Cheat Sheet: IOS Developer
- OWASP Testing Guide: Testing for Authentication
- NIST Special Publication 800-63 Revision 3 - Digital Identity Guidelines

Tools

- Daniel Miessler: Most commonly found passwords

OWASP Proactive Controls v 3.0

C7: Enforce Access Controls

Description

Access Control (or Authorization) is the process of granting or denying *specific requests* from a user, program, or process. Access control also involves the act of *granting and revoking those privileges*.

It should be noted that authorization (verifying access to specific features or resources) is not equivalent to authentication (verifying identity).

Access Control functionality often spans many areas of software depending on the complexity of the access control system. For example, managing access control metadata or building caching for scalability purposes are often additional components in an access control system that need to be built or managed.

There are several different types of access control design that should be considered.

- Discretionary Access Control (DAC) is a means of restricting access to objects (e.g., files, data entities) based on the identity and need-to-know of subjects (e.g., users, processes) and/or groups to which the object belongs.
- Mandatory Access Control (MAC**)** is a means of restricting access to system resources based on the sensitivity (as represented by a label) of the information contained in the system resource and the formal authorization (i.e., clearance) of users to access information of such sensitivity.
- Role Based Access Control (RBAC) is a model for controlling access to resources where permitted actions on resources are identified with roles rather than with individual subject identities.
- Attribute Based Access Control (ABAC) will grant or deny user requests based on arbitrary attributes of the user and arbitrary attributes of the object, and environment conditions that may be globally recognized and more relevant to the policies at hand.

Access Control Design Principles

The following "positive" access control design requirements should be considered at the initial stages of application development.

1) Design Access Control Thoroughly Up Front

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 thoroughly designed up front, especially when addressing requirements like multi-tenancy and horizontal (data dependent) access control.

Access Control design may start simple but can often grow into a complex and feature-heavy security control. When evaluating access control capability of software frameworks, ensure that your access control functionality will allow for customization for your specific access control feature need.

2) Force All Requests to Go Through Access Control Checks

Ensure that all request go through some kind of access control verification layer. Technologies like Java filters or other automatic request processing mechanisms are ideal programming artifacts that will help ensure that all requests go through some kind of access control check.

3) Deny by Default

Deny by default is the principle that if a request is not specifically allowed, it is denied. There are many ways that this rule will manifest in application code. Some examples of these are:

1. Application code may throw an error or exception while processing access control requests. In these cases access control should always be denied.
2. When an administrator creates a new user or a user registers for a new account, that account should have minimal or no access by default until that access is configured.
3. When a new feature is added to an application all users should be denied to use that feature until it's properly configured.

4) Principle of Least Privilege

Ensure that all users, programs, or processes are only given as least or as little necessary access as possible. Be wary of systems that do not provide granular access control configuration capabilities.

5) Don't Hardcode Roles

Many application frameworks default to access control that is role based. It is common to find application code that is filled with checks of this nature.

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

Be careful about this type of role-based programming in code. It has the following limitations or dangers.

- Role based programming of this nature is fragile. It is easy to create incorrect or missing role checks in code.
- Role based programming does not allow for multi-tenancy. Extreme measures like forking the code or added checks for each customer will be required to allow role based systems to have different rules for different customers.
- Role based programming does not allow for data-specific or horizontal access control rules.
- Large codebases with many access control checks can be difficult to audit or verify the overall application access control policy.

Instead, please consider the following access control programming methodology:

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

Attribute or feature-based access control checks of this nature are the starting point to building well-designed and feature-rich access control systems. This type of programming also allows for greater access control customization capability over time.

6) Log All Access Control Events

All access control failures should be logged as these may be indicative of a malicious user probing the application for vulnerabilities.

Vulnerabilities Prevented

- OWASP Top 10 2017-A5-Broken Access Control
- OWASP Mobile Top 10 2014-M5 Poor Authorization and Authentication

References

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

Tools

- OWASP ZAP with the optional Access Control Testing add-on

OWASP Proactive Controls v 3.0

C8: Protect Data Everywhere

Description

Sensitive data such as passwords, credit card numbers, health records, personal information and business secrets require extra protection, particularly if that data falls under privacy laws (EU's General Data Protection Regulation GDPR), financial data protection rules such as PCI Data Security Standard (PCI DSS) or other regulations.

Attackers can steal data from web and webservice applications in a number of ways. For example, if sensitive information in sent over the internet without communications security, then an attacker on a shared wireless connection could see and steal another user's data. Also, an attacker could use SQL Injection to steal passwords and other credentials from an applications database and expose that information to the public.

Data Classification

It's critical to classify data in your system and determine which level of sensitivity each piece of data belongs to. Each data category can then be mapped to protection rules necessary for each level of sensitivity. For example, public marketing information that is not sensitive may be categorized as public data which is ok to place on the public website. Credit card numbers may be classified as private user data which may need to be encrypted while stored or in transit.

Encrypting Data in Transit

When transmitting sensitive data over any network, end-to-end communications security (or encryption-in-transit) of some kind should be considered. TLS is by far the most common and widely supported cryptographic protocol for communications security. It is used by many types of applications (web, webservice, mobile) to communicate over a network in a secure fashion. TLS must be properly configured in a variety of ways in order to properly defend secure communications.

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.

Encrypting Data at Rest

The first rule of sensitive data management is to avoid storing sensitive data when at all possible. If you must store sensitive data then make sure it's cryptographically protected in some way to avoid unauthorized disclosure and modification.

Cryptography (or crypto) is one of the more advanced topics of information security, and one whose understanding requires the most schooling and experience. It is difficult to get right because there are many approaches to encryption, each with advantages and disadvantages that need to be thoroughly understood by web solution architects and developers. In addition, serious cryptography research is typically based in advanced mathematics and number theory, providing a serious barrier to entry.

Instead of building cryptographic capability from scratch, it is strongly recommended that peer reviewed and open solutions be used, such as the Google Tink project, Libsodium, and secure storage capability built into many software frameworks and cloud services.

Mobile Application: Secure Local Storage

Mobile applications are at particular risk of data leakage because mobile devices are regularly lost or stolen yet contain sensitive data.

As a general rule, only the minimum data required should be stored on the mobile device. But if you must store sensitive data on a mobile device, then sensitive data should be stored within each mobile operating systems specific data storage directory. On Android this will be the Android keystore and on iOS this will be the iOS keychain.

Key Lifecycle

Secret keys are used in applications number of sensitive functions. For example, secret keys can be used to to sign JWTs, protect credit cards, provide various forms of authentication as well as facilitation other sensitive security features. In managing keys, a number of rules should be followed including:

- Ensure that any secret key is protected from unauthorized access
- Store keys in a proper secrets vault as described below
- Use independent keys when multiple keys are required
- Build support for changing algorithms and keys when needed
- Build application features to handle a key rotation

Application Secrets Management

Applications contain numerous "secrets" that are needed for security operations. These include certificates, SQL connection passwords, third party service account credentials, passwords, SSH keys, encryption keys and more. The unauthorized disclosure or modification of these secrets could lead to complete system compromise. In managing application secrets, consider the following.

- Don't store secrets in code, config files or pass them through environment variables. Use tools like GitRob or TruffleHog to scan code repos for secrets.
- Keep keys and your other application-level secrets in a secrets vault like KeyWhiz or Hashicorp's Vault project or Amazon KMS to provide secure storage and access to application-level secrets at run-time.

Vulnerabilities Prevented

- OWASP Top 10 2017 - A3: Sensitive Data Exposure
- OWASP Mobile Top 10 2014-M2 Insecure Data Storage

References

- OWASP Cheat Sheet: Transport Layer Protection
- Ivan Ristic: SSL/TLS Deployment Best Practices
- OWASP Cheat Sheet: HSTS
- OWASP Cheat Sheet: Cryptographic Storage
- OWASP Cheat Sheet: Password Storage
- OWASP Cheat Sheet: IOS Developer - Insecure Data Storage
- OWASP Testing Guide: Testing for TLS

Tools

- SSLyze - SSL configuration scanning library and CLI tool
- SSLLabs - Free service for scanning and checking TLS/SSL configuration
- OWASP O-Saft TLS Tool - TLS connection testing tool
- GitRob - Command line tool to find sensitive information in publicly available files on GitHub
- TruffleHog - Searches for secrets accidentally committed
- KeyWhiz - Secrets manager
- Hashicorp Vault - Secrets manager
- Amazon KMS - Manage keys on Amazon AWS

OWASP Proactive Controls v 3.0

C9: Implement Security Logging and Monitoring

Description

Logging is a concept that most developers already use for debugging and diagnostic purposes. Security logging is an equally basic concept: to log security information during the runtime operation of an application. Monitoring is the live review of application and security logs using various forms of automation. The same tools and patterns can be used for operations, debugging and security purposes.

Benefits of Security Logging

Security logging can be used for:

1) Feeding intrusion detection systems
2) Forensic analysis and investigations
3) Satisfying regulatory compliance requirements

Security Logging Implementation

The following is a list of security logging implementation best practices.

- Follow a common logging format and approach within the system and across systems of an organization. An example of a common logging framework is the Apache Logging Services which helps provide logging consistency between Java, PHP, .NET, and C++ applications.
- Do not log too much or too little. For example, make sure to always log the timestamp and identifying information including the source IP and user-id, but be careful not to log private or confidential data.
- Pay close attention to time syncing across nodes to ensure that timestamps are consistent.

Logging for Intrusion Detection and Response

Use logging to identify activity that indicates that a user is behaving maliciously. Potentially malicious activity to log includes:

- Submitted data that is outside of an expected numeric range.
- Submitted data that involves changes to data that should not be modifiable (select list, checkbox or other limited entry component).
- Requests that violate server-side access control rules.
- A more comprehensive list of possible detection points is available here.

When your application encounters such activity, your application should at the very least log the activity and mark it as a high severity issue. Ideally, your application should also respond to a possible identified attack, by for example invalidating the user's session and locking the user's account. The response mechanisms allows the software to react in realtime to possible identified attacks.

Secure Logging Design

Logging solutions must be built and managed in a secure way. Secure Logging design may include the following:

- Encode and validate any dangerous characters before logging to prevent log injection or log forging attacks.
- Do not log sensitive information. For example, do not log password, session ID, credit cards, or social security numbers.
- Protect log integrity. An attacker may attempt to tamper with the logs. Therefore, the permission of log files and log changes audit should be considered.
- Forward logs from distributed systems to a central, secure logging service. This will sure log data cannot be lost if one node is compromised. This also allows for centralized monitoring.

References

- OWASP AppSensor Detection Points - Detection points used to identify a malicious user probing for vulnerabilities or weaknesses in application.
- OWASP Log injection
- OWASP Log forging
- OWASP Cheat Sheet: Logging How to properly implement logging in an application
- OWASP Development Guide: Logging
- OWASP Code Review Guide: Reviewing Code for Logging Issues

Tools

- OWASP Security Logging Project
- Apache Logging Services

OWASP Proactive Controls v 3.0

C10: Handle all Errors and Exceptions

Description

Exception handling is a programming concept that allows an application to respond to different error states (like network down, or database connection failed, etc) in various ways. Handling exceptions and errors correctly is critical to making your code reliable and secure.

Error and exception handling occurs in all areas of an application including critical business logic as well as security features and framework code.

Error handling is also important from an intrusion detection perspective. Certain attacks against your application may trigger errors which can help detect attacks in progress.

Error Handling Mistakes

Researchers at the University of Toronto have found that even small mistakes in error handling or forgetting to handle errors can lead to catastrophic failures in distributed systems.

Mistakes in error handling can lead to different kinds of security vulnerabilities.

- Information leakage: Leaking sensitive information in error messages can unintentionally help attackers. For example, an error that returns a stack trace or other internal error details can provide an attacker with information about your environment. Even small differences in handling different error conditions (e.g., returning "invalid user" or "invalid password" on authentication errors) can provide valuable clues to attackers. As described above, be sure to log error details for forensics and debugging purposes, but don't expose this information, especially to an external client.
- TLS bypass: The Apple goto "fail bug" was a control-flow error in error handling code that lead to a complete compromise of TLS connections on apple systems.
- DOS: A lack of basic error handling can lead to system shutdown. This is usually a fairly easy vulnerability for attackers to exploit. Other error handling problems could lead to increased usage of CPU or disk in ways that could degrade the system.

Positive Advice

- Manage exceptions in a centralized manner to avoid duplicated try/catch blocks in the code. Ensure that all unexpected behavior is correctly handled inside the application.
- Ensure that error messages displayed to users do not leak critical data, but are still verbose enough to enable the proper user response.
- Ensure that exceptions are logged in a way that gives enough information for support, QA, forensics or incident response teams to understand the problem.
- Carefully test and verify error handling code.

References

- OWASP Code Review Guide: Error Handling
- OWASP Testing Guide: Testing for Error Handling
- OWASP Improper Error Handling
- CWE 209: Information Exposure Through an Error Message
- CWE 391: Unchecked Error Condition

Tools

- Error Prone - A static analysis tool from Google to catch common mistakes in error handling for Java developers
- One of the most famous automated tools for finding errors at runtime is Netflix's Chaos Monkey, which intentionally disables system instances to ensure that the overall service will recover correctly.

Final word

This document should be seen as a starting point rather than a comprehensive set of techniques and practices. We want to again emphasize that this document is intended to provide initial awareness around building secure software.

Good next steps to help build an application security program include:

1) To understand some of the risks in web application security please review the OWASP Top Ten and the OWASP Mobile Top Ten.
2) Per Proactive Control #1, a secure development program should include a *comprehensive list of security requirements* based on a standard such as the OWASP (Web) ASVS and the OWASP (Mobile) MASVS.
3) To understand the core building blocks of a secure software program from a more macro point of view please review the OWASP OpenSAMM project.

If you have any questions for the project leadership team please sign up for our mailing list at https://lists.owasp.org/mailman/listinfo/owasp_proactive_controls.

OWASP
Open Web Application
Security Project

FOR DEVELOPERS