Developer Guide

Open Worldwide Application Security Project (OWASP)

February 2023 onwards

OWASP Developer Guide (draft)

The draft version has the latest contributions to the Developer Guide and so the content is expected to frequently change.

Table of contents

1. Introduction

Welcome to the OWASP Development Guide.

The Open Worldwide Application Security Project (OWASP) is a nonprofit foundation that works to improve the security of software. It is an open community dedicated to enabling organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted.

Along with the OWASP Top Ten, the Developer Guide is one of the original resources published soon after the OWASP foundation was formed in 2001. Version 1.0 of the Developer Guide was released in 2002 and since then there have been various releases culminating in version 2.0 in 2005. Since then the guide has been revised extensively to bring it up to date and no longer has version numbers; notionally it is version 4.x as version 3.0 was never released.

The purpose of this guide is to provide an introduction to security concepts and a handy reference for application / system developers. Generally it describes the security practices within the Design and Implementation business functions of the OWASP Software Assurance Maturity Model (SAMM). Much of this guide is based on various OWASP sources:

- OWASP Top 10
- Top 10 Proactive Controls
- Application Security Verification Standard ((ASVS)
- Cheat Sheet Series
- Integration Standards project
- Mobile Application Security
- Software Assurance Maturity Model (SAMM)
- Web Security Testing Guide

These resources provide greater detail and wider context for the various sections in this guide, and are referenced throughout this developer guide.

The content of the main sections is aimed at an introductory level, with more detail provided in the following sections. This guide does not seek to replicate the many excellent sources on specific security topics; it will rarely deal with a subject at an advanced level and instead provides links for specialist security topics.

All of the OWASP projects and tools are free to download and use and the source code is open; if you are interested in improving application security then do get involved.

Audience The OWASP Developer Guide has been written by the security community to help software developers write solid, safe and secure applications. Application developers should try to be familiar with the entire guide; it is far harder to write solid applications than to destroy them.

You can regard the purpose of this guide as answering the question: "I am a developer and I need a reference source to navigate the numerous projects and describe the security activities I am *supposed* to be doing"

Or you can regard this guide as a companion document to the OWASP Wayfinder project; the Wayfinder maps out many OWASP tools and documents, the Developer Guide provides context for them.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

2. Foundations

There are various foundational concepts and terminology that are commonly used in software security. Although many of these concepts are complex to implement and are based on heavy-duty theory, the principles are often fairly straight forward and are accessible for every software engineer. A reasonable grasp of these foundational concepts allows development teams to understand and implement software security for the application or system under development.

Sections:

2.1 Security fundamentals
2.2 Secure development and integration)
2.3 Principles of security
2.4 Principles of cryptography
2.5 OWASP Top Ten
2.6 Security champions

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

2.1 Security fundamentals

The OWASP Software Assurance Maturity Model (SAMM) is used throughout this Developer Guide to provide context for each section. Refer to the section below for a brief introduction to SAMM.

Overview Security is simply about controlling who can interact with your information, what they can do with it, and when they can interact with it. These characteristics of security can be described using the CIA triad, and can be extended using the AAA triad.

CIA CIA stands for Confidentiality, Integrity and Availability, and it is usually depicted as a triangle representing the strong bonds between its three tenets. This trio is considered the pillars of application security, with CIA described as a property of some data or of a process. Often CIA is extended with AAA: Authorization, Authentication and Auditing.

Confidentiality Confidentiality is the protection of data against unauthorized disclosure; it is about ensuring that only those with the correct authorization can access the data. Confidentiality applies to data at rest, and should also be considered for data in motion. Confidentiality is related to the broader concept of data privacy.

Integrity Integrity is about protecting data against unauthorized modification, or assuring data trustworthiness. The concept contains the notion of data integrity (data has not been changed accidentally or deliberately) and the notion of source integrity (data came from or was changed by a legitimate source).

Availability Availability is about ensuring the presence of information or resources. This concept relies not just on the availability of the data itself, for example by using replication of data, but also on the protection of the services that provide access to the data, for example by using load balancing.

AAA CIA is often extended with Authentication, Authorization and Auditing as these are closely linked to CIA concepts. CIA has such a strong dependency on Authentication and Authorization that the confidentiality of the data in question can't be assured without them. Auditing is added as it can provide the mechanism to ensure proof of any interaction with the system.

Authentication Authentication is about confirming the identity of the entity that wants to interact with a secure system.

Authorization Authorization is about specifying access rights to secure resources (data, services, files, applications, etc). These rights describe the privileges or access levels related to the resources in question. It is normally preceded by *Authentication*.

Auditing Auditing is about keeping track of implementation-level events, as well as domain-level events taking place in a system. This helps to provide non-repudiation, which means that changes or actions on the protected system are undeniable. Auditing can provide not only technical information about the running system, but also proof that particular actions have been performed. The typical questions that are answered by auditing are "*Who* did *What When* and potentially *How*?"

Software Assurance Maturity Model The OWASP Software Assurance Maturity Model (SAMM) provides a good context for the scope of software security, and the foundations of SAMM rely on the security concepts in this section. The SAMM model describes the five fundamentals of software security, which it calls Business Functions:

- Governance
- Design
- Implementation
- Verification
- Operations

Each of these five fundamentals are further split into three Business Practices:

Business Function	Business Practices		
Governance	Strategy and Metrics	Policy and Compliance	Education and Guidance
Design	Threat Assessment	Security Requirements	Security Architecture
Implementation	Secure Build	Secure Deployment	Defect Management
Verification	Architecture Assessment	Requirements-driven Testing	Security Testing
Operations	Incident Management	Environment Management	Operational Management

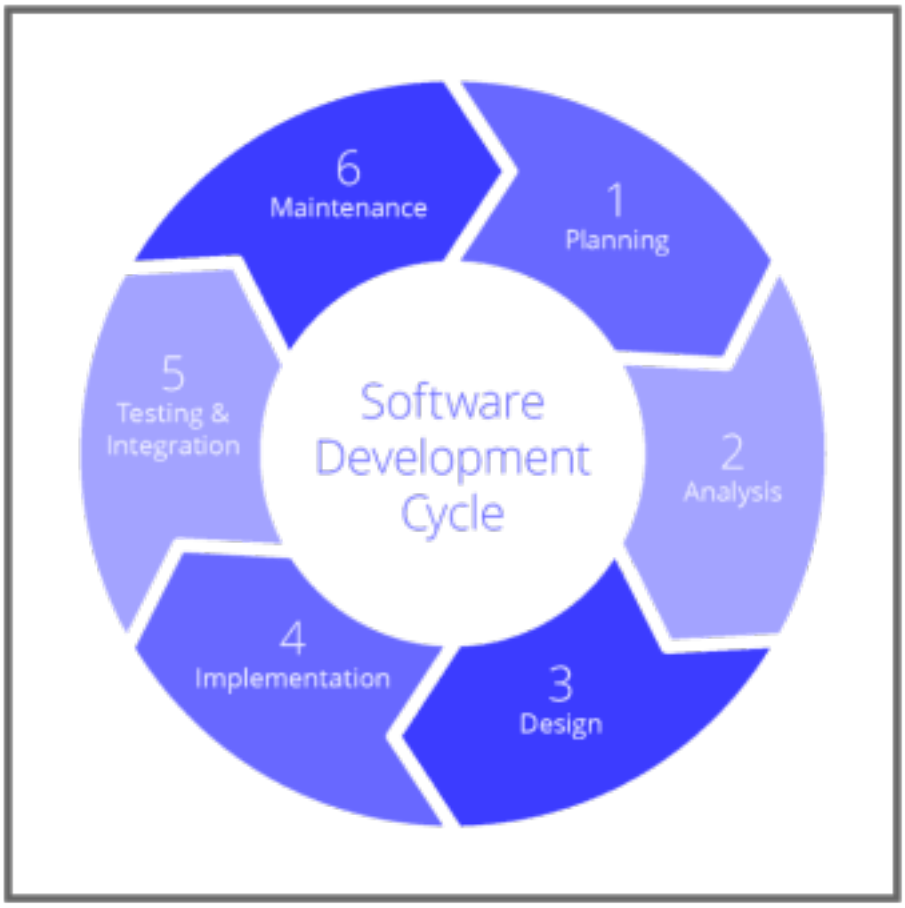
Each Business Practice is further subdivided into two streams, and the sections in the Developer Guide reference at least one of the Business Functions or Practices in SAMM.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

2.2 Secure development and integration

Secure development is described in the OWASP Software Assurance Maturity Model (SAMM) by the Design, Implementation and Verification business functions. Much of the material in this section is drawn from the OWASP Integration Standards project.

Overview Almost all modern software is developed in an iterative manner passing through phase to phase, such as identifying customer requirements, implementation and test. These phases are revisited in a cyclic manner throughout the lifetime of the application. A generic Software Development LifeCycle (SDLC) is shown below, and in practice there may be more or less phases according to the processes adopted by the business.



A typical SDLC representation

{: height="220px" }

With the increasing number and sophistication of exploits against almost every application or business system, most companies have adopted a secure Software Development LifeCycle (SDLC). The secure SDLC should never be a separate lifecycle, it must always be the same Software Development LifeCycle as before but with security actions built into each phase, otherwise the security actions will be set aside by busy development teams. Note that although the Secure SDLC could be written as 'SSDLC' it is almost always written as 'SDLC'.

DevOps integrates and automates many of the SDLC phases and implements Continuous Integration (CI) and Continuous Delivery/Deployment (CD) pipelines to provide much of this automation. DevOps and pipelines have been successfully exploited with serious large scale consequences and so, in a similar manner to the SDLC, much of the DevOps actions have also had to have security built in. Secure DevOps, or DevSecOps, builds security practices into the DevOps activities to guard against attack and to provide the SDLC with automated security testing. Examples of how DevSecOps is 'building security in' is the provision of Interactive, Static and Dynamic Application Security Testing (IAST, SAST & DAST) and

implementing supply chain security, and there are many other security activities that can be applied.

Secure development lifecycle Referring to the OWASP Application Wayfinder development cycle there are four phases during application development: Requirements, Design, Implementation and Verification. There are other phases that are done less often in the development cycle and these form an equally important part of the SDLC: Gap Analysis, Metrics, Operation and Training & Culture Building.

All of these phases of the SDLC should have security activities built into them, rather than done as separate activities. If security is built into these phases then the overhead becomes much less and the resistance from the development teams decreases. The goal is for the secure SDLC to become as familiar a process as before, with the development teams taking ownership of the security activities within each phase.

There are many OWASP tools and resources to help build security into the SDLC.

- Requirements: this phase determines the functional, non-functional and security requirements for the application. Requirements should be revisited periodically and check for completeness and validity, and it is worth considering various OWASP tools; the Application Security Verification Standard (ASVS) provides developers with a list of requirements for secure development, the Mobile Application Security project provides a security standard for mobile applications and SecurityRAT helps identify an initial set of security requirements.
- Design: it is important to design security into the development - it is never too late to do this but the earlier the better (and easier). OWASP provides two tools, Pythonic Threat Modeling and Threat Dragon, for threat modeling along with security gamification using Cornucopia.
- Implementation: the OWASP Top 10 Proactive Controls project states that they are "the most important control and control categories that every architect and developer should absolutely, 100% include in every project" and this is certainly good advice. Implementing these controls can provide a high degree of confidence that the application or system will be reasonably secure. OWASP provides two libraries that can be incorporated in web applications, the Enterprise Security API (ESAPI) security control library and CSRFGuard to mitigate the risk of Cross-Site Request Forgery (CSRF) attacks, that help implement these proactive controls. In addition the OWASP Cheat Sheet Series is a valuable source of information and advice on all aspects of applications security.
- Verification: OWASP provides a relatively large number of projects that help with testing and verification. This is the subject of a section in this Developer Guide, and the projects are listed below.
- Training: development teams continually need security training. Although not part of the inner SDLC iterative loop, training should be factored into the project lifecycle. OWASP provides many training environments and materials - see the list at the end of this section.
- Culture Building: a good security culture within a business organization will help greatly in keeping the applications and systems secure. There are many activities that all add up to create the security culture, the OWASP Security Culture project goes into more detail on these activities, and a good Security Champion program within the business is foundational to a good security posture. The OWASP Security Champions Guide provides guidance and material to create security champions within the development teams - ideally every team should have a security champion that has a special interest in security and has received further training, enabling the team to build security in.
- Operation: the OWASP DevSecOps Guideline explains how to best implement a secure pipeline, using best practices and introducing automation tools to help 'shift-left'. Refer to the DevSecOps Guideline for more information on any of the topics within DevSecOps and in particular sections on Operation.
- Supply chain: attacks that leverage the supply chain can be devastating and there have been several high profile of products being successfully exploited. A Software Bill of Materials (SBOM) is the first step in avoiding these attacks and it is well worth using the OWASP CycloneDX full-stack Bill of Materials (BOM) standard for risk reduction in the supply chain. In addition the OWASP

Dependency-Track project is a Continuous SBOM Analysis Platform which can help prevent these supply chain exploits by providing control of the SBOM.

- Third party libraries: keeping track of what third party libraries are included in the application, and what vulnerabilities they have, is easily automated. Many public repositories such as github and gitlab offer this service along with some commercial vendors. OWASP provides the Dependency-Check Software Composition Analysis (SCA) tool to track external libraries.
- Application security testing: there are various types of security testing that can be automated on pull-request, merge or nightlies - or indeed manually but they are most powerful when automated. Commonly there is Static Application Security Testing (SAST), which analyses the code without running it, Dynamic Application Security Testing (DAST), which applies input to the application while running it in a sandbox or other isolated environment. Interactive Application Security Testing (IAST) on the other hand is designed to be run manually as well, providing instant feedback on the tests as they are run.

Further reading from OWASP

- Cheat Sheet Series
- Cornucopia
- CycloneDX Bill of Materials (BOM) standard
- DevSecOps Guideline
- Security Champions Guide
- Security Culture project
- Top 10 Proactive Controls

OWASP verification projects

- Amass project
- Code Pulse
- Defect Dojo
- Mobile Application Security (MAS)
- Nettacker
- OWTF
- Web Security Testing Guide (WSTG)
- Zed Attack Proxy (ZAP)

OWASP training projects

- API Security Project
- Juice Shop
- Mobile Top 10
- Security Shepherd
- Snakes And Ladders
- Top Ten Web Application security risks
- WebGoat

OWASP resources

- Application Security Verification Standard (ASVS)
- CSRFGuard library
- Dependency-Check Software Composition Analysis (SCA)
- Dependency-Track Continuous SBOM Analysis Platform
- Enterprise Security API (ESAPI)
- Integration Standards project Application Wayfinder
- Mobile Application Security (MAS)
- Pythonic Threat Modeling
- Threat Dragon
- Security RAT (Security Requirement Automation Tool)

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

2.3 Principles of security

This section is a very brief introduction to some concepts used within the software security domain, as these may not be familiar to many application developers. The OWASP Cheat Sheet Series provides more in depth explanations for these security principles, see the further reading at the end of this section.

Overview There are various concepts and terms used in the security domain that are fundamental to the understanding and discussion of application security. Security architects and security engineers will be familiar with these terms and development teams will also need this understanding to implement secure applications.

No security guarantee One of the most important principles of software security is that **no** application or system is totally 100% guaranteed to be secure from all attacks. This may seem an unusually pessimistic starting point but it is merely acknowledging the real world; given enough time and enough resources any system can be compromised. The goal of software security is not '100% secure' but to make it hard enough and the rewards small enough that malicious actors look elsewhere for systems to exploit.

Defense in Depth Also known as layered defense, defense in depth is a security principle where defense against attack is provided by multiple security controls. The aim is that single points of complete compromise are eliminated or mitigated by the incorporation of a series or multiple layers of security safeguards and risk-mitigation countermeasures.

If one layer of defense turns out to be inadequate then, if diverse defensive strategies are in place, another layer of defense may prevent a full breach and if that one is circumvented then the next layer may block the exploit.

Fail Safe This is a security principle that aims to maintain confidentiality, integrity and availability when an error condition is detected. These error conditions may be a result of an attack, or may be due to design or implementation failures, in any case the system / applications should default to a secure state rather than an unsafe state.

For example unless a subject is given explicit access to an object, it should be denied access to that object by default. This is often described as 'Fail Safe Defaults' or 'Secure by Default'.

In the context of software security, fail secure is commonly used interchangeably with fail safe, which comes from physical security terminology. Failing safe also helps software resiliency, so that the system / application can rapidly recover upon design or implementation flaws.

Least Privilege A security principle in which a person or process is given only the minimum level of access rights (privileges) that is necessary for that person or process to complete an assigned operation. This right must be given only for a minimum amount of time that is necessary to complete the operation.

This helps to limits the damage when a system is compromised by minimising the ability of an attacker to escalate privileges both horizontally or vertically. In order to apply this principle of least privilege proper granularity of privileges and permissions should be established.

Separation of Duties Also known as separation of privilege, separation of duties is a security principle which requires that the successful completion of a single task is dependent upon two or more conditions that are insufficient for completing the task by itself.

There are many applications for this principle, for example limiting the damage an aggrieved or malicious insider can do, or by limiting privilege escalation.

Economy of Mechanism Also known as 'keep it simple', if there are multiple implementations then the simplest and most easily understood implementation should be chosen.

The likelihood of vulnerabilities increases with the complexity of the software architectural design and code, and increases further if it is hard to follow or review the code. The attack surface of the software is reduced by keeping the software design and implementation details simple and understandable.

Complete Mediation A security principle that ensures that authority is not circumvented in subsequent requests of an object by a subject, by checking for authorization (rights and privileges) upon every request for the object.

In other words, the access requests by a subject for an object are completely mediated every time, so that all accesses to objects must be checked to ensure that they are allowed.

Open Design The open design security principle states that the implementation details of the design should be independent of the design itself, allowing the design to remain open while the implementation can be kept secret. This is in contract to security by obscurity where the security of the software is dependent upon the obscuring of the design itself.

When software is architected using the open design concept, the review of the design itself will not result in the compromise of the safeguards in the software.

Least Common Mechanism The security principle of least common mechanisms disallows the sharing of mechanisms that are common to more than one user or process if the users and processes are at different levels of privilege.

Psychological acceptability A security principle that aims at maximizing the usage and adoption of the security functionality in the software by ensuring that the security functionality is easy to use and at the same time transparent to the user. Ease of use and transparency are essential requirements for this security principle to be effective.

Security mechanisms should not make the resource significantly more difficult to access than if the security mechanism were not present. If the security mechanism provides too much friction for the users then they may look for ways to defeat the mechanism and "prop the doors open".

Weakest Link This security principle states that the resiliency of your software against hacker attempts will depend heavily on the protection of its weakest components, be it the code, service or an interface.

Leveraging Existing Components This is a security principle that focuses on ensuring that the attack surface is not increased and no new vulnerabilities are introduced by promoting the reuse of existing software components, code and functionality.

Existing components are more likely to be tried and tested, and hence more secure, and also should have security patches available. In addition open source components have the benefit of 'many eyes' and are likely to be even more secure.

Further reading

- OWASP Cheat Sheet series
  - Authentication Cheat Sheet
  - Authorization_Cheat_Sheet
  - Secure Product Design Cheat Sheet

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

2.4 Principles of cryptography

Cryptography is fundamental to the Confidentiality and Integrity of applications and systems. The OWASP Cheat Sheet series describes the use of cryptography and some of these are listed in the further reading at the end of this section.

Overview This section provides a brief introduction to cryptography (often referred to as "crypto") and the terms used. Cryptography is a large subject and can get very mathematical, but fortunately for the majority of development teams a general understanding of the concepts is sufficient. This general understanding, with the guidance of security architects, should allow implementation of cryptography by the development team for the application or system.

Uses of cryptography Initially primarily restricted to the military and the realm of academia, cryptography has become ubiquitous thanks to the Internet. Common every day uses of cryptography include mobile phones, passwords, SSL VPNs, smart cards, and DVDs. Cryptography has permeated through everyday life, and is heavily used by many web applications.

Cryptography 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 solution architects.

The proper and accurate implementation of cryptography is extremely critical to its efficacy. A small mistake in configuration or coding will result in removing most of the protection and rending the crypto implementation useless.

A good understanding of crypto is required to be able to discern between solid products and snake oil. The inherent complexity of crypto makes it easy to fall for fantastic claims from vendors about their product. Typically, these are "a breakthrough in cryptography" or "unbreakable" or provide "military grade" security. If a vendor says "trust us, we have had experts look at this," chances are they weren't experts!

Confidentiality For the purposes of this section, we define "confidentiality" as "no unauthorized disclosure of information". Cryptography addresses this via encryption of either the data at rest or data en transit by protecting the information from all who do not hold the decryption key. We also use cryptographic hashes (i.e., secure, one way hashes) to prevent passwords from disclosure.

Authentication Authentication is the process of verifying a claim that a subject is who it says it is via some provided corroborating evidence. We use cryptography for authentication in several different ways:

1. to protect the provided corroborating evidence (for example hashing of passwords for subsequent storage)
2. in authentication protocols often use cryptography to either directly authenticate entities or to exchange credentials in a secure manner.
3. to verify the identity one or both parties in exchanging messages, for example identity verification within Transport Layer Security (TLS)

Integrity We define "integrity" as ensuring that even authorized users have performed no accidental or malicious alternation of information. Cryptography can be used to prevent tampering by means of Message Authentication Codes (MACs) or Digital Signatures.

When you hear the term "message authenticity" being referred to, it is really referring to integrity. It sometimes is referred to as "authenticated encryption" as well, although in the case of symmetric encryption and shared keys, it does not authenticate the sending party. However, if asymmetric encryption is used, one can in fact use it to authenticate the sender.

Non-repudiation Non-repudiation comes in two flavors... non-repudiation of sender ensures that someone sending a message should not be able to deny later that they have sent it. Non-repudiation of receiver means that the receiver of a message should not be able to deny that they have received it.

Cryptography can be used to provide non-repudiation by providing unforgeable messages or replies to messages.

Non-repudiation is useful for financial, e-commerce, and contractual exchanges. Non-repudiation is accomplished by having the sender or recipient to digitally sign some unique transactional record.

Attestation Attestation is the act of "bearing witness" or certifying something for a particular use or purpose. Attestation is generally discussed in the context of a Trusted Platform Module (TPM), Digital Rights Management (DRM), and UEFI Secure Boot.

For example, Digital Rights Management is interested in attesting that your device or system hasn't been compromised with some back-door to allow someone to illegally copy DRM-protected content.

Cryptography can be used to provide a "chain of evidence" that everything is as it is expected to be, to prove to a challenger that everything is in accordance with the challenger's expectations. For example, remote attestation can be used to prove to a challenger that you are indeed running the software that you claim that you are running. Most often, attestation is done by providing a chain of digital signatures starting with a trusted (digitally signed) boot loader.

Cryptographic hashes Cryptographic hashes, also known as message digests, are functions that map arbitrary length bit strings to some fixed length bit string (referred to as the "hash value" or "digest value"). These hash functions are many-to-one mappings that are compression functions.

Such cryptographic hash functions are used to provide data integrity (i.e., to detect intentional data tampering), to store passwords or pass phrases, and to provide digital signatures in a more efficient manner than using asymmetric ciphers. Cryptographic hash functions are also used to extend a relatively small bit of entropy so that secure random number generators can be constructed.

When used to provide data integrity, cryptographic functions come in two flavors, keyed hashes (called "message authentication codes") and unkeyed hashes (called "message integrity codes").

Ciphers A cipher is an algorithm that performs encryption or decryption. Modern ciphers can be categorized in a couple of different ways. The most common distinctions between them are:

- Whether they work on fixed size number of bits (block ciphers) or on a continuous stream of bits (stream ciphers)
- Whether the same key is used for both encryption and decryption (symmetric ciphers) or separate keys for encryption and decryption (asymmetric ciphers

Symmetric Ciphers Symmetric ciphers encrypt and decrypt using the same key. This implies that if one party encrypts data that a second party must decrypt, those two parties must share a common key.

Symmetric ciphers come in two main types:

1. Block ciphers, which operate on a block of characters (typically 8 or 16 octets) at a time. An example of a block cipher is AES
2. Stream ciphers, which operate on a single bit (or occasionally a single byte) at a time. Examples of a stream ciphers are RC4 (aka, ARC4) and Salsa20

Note that all block ciphers can also operating in "streaming mode" by selecting the appropriate cipher mode.

Cipher Modes Block ciphers can function in different modes of operations known as "cipher modes". This cipher mode algorithmically describes how a cipher operates to repeatedly apply its encryption or decryption mechanism to a given cipher block. Cipher modes are important because they have an enormous impact on both the confidentiality and the message authenticity of the resulting ciphertext messages.

Almost all cryptographic libraries support the four original DES cipher modes of ECB, CBC (Cipher Block Chaining) OFB (Output Feedback), and CFB (Cipher Feedback). Many also support CTR (Counter) mode.

Initialization vector In cryptography, an initialization vector (IV) is a fixed size input to a block cipher's encryption / decryption primitive. The IV is recommended (and in many cases, required) to be random or at least pseudo-random.

Padding Block ciphers generally operate on fixed size blocks (the exception is when they are operating in a "streaming" mode). However, these ciphers must also operate on messages of any sizes, not just those that are an integral multiple of the cipher's block size.

Asymmetric cipher Asymmetric ciphers encrypt and decrypt with two different keys. One key generally is designated as the private key and the other is designated as the public key. The public key generally is widely shared.

Asymmetric ciphers are several orders of magnitude slower than symmetric ciphers. For this reason they are used frequently in hybrid cryptosystems, which combine asymmetric and symmetric ciphers. In such hybrid cryptosystems, a random symmetric "session" key is generated which is only used for the duration of the encrypted communication. This random session key is then encrypted using an asymmetric cipher and the recipient's private key. The plaintext data itself is encrypted with the session key. Then the entire bundle (encrypted session key and encrypted message) is all sent together. Both TLS and S/MIME are common cryptosystems using hybrid cryptography.

Digital signature Digital signatures are a cryptographically unique data string that is used to ensure data integrity and prove the authenticity of some digital message, and that associates some input message with an originating entity. A digital signature generation algorithm is a cryptographically strong algorithm that is used to generate a digital signature.

Key agreement protocol Key agreement protocols are protocols whereby N parties (usually two) can agree on a common key without actually exchanging the key. When designed and implemented properly, key agreement protocols are preventing adversaries from learning the key or forcing their own key choice on the participating parties.

Application level encryption Application level encryption refers to encryption that is considered part of the application itself; it implies nothing about where in the application code the encryption is actually done.

Key derivation A key derivation function (KDF) is a deterministic algorithm to derive a key of a given size from some secret value. If two parties use the same shared secret value and the same KDF, they should always derive exactly the same key.

Key wrapping Key wrapping is a construction used with symmetric ciphers to protect cryptographic key material by encrypting it in a special manner. Key wrap algorithms are intended to protect keys while held in untrusted storage or while transmitting keys over insecure communications networks.

Key exchange algorithms Key exchange algorithms (also referred to as key establishment algorithms) are protocols that are used to exchange secret cryptographic keys between a sender and receiver in a manner that meets certain security constraints. Key exchange algorithms attempt to address the problem of securely sharing a common secret key with two parties over an insecure communication channel in a manner that no other party can gain access to a copy of the secret key.

The most familiar key exchange algorithm is Diffie-Hellman Key Exchange. There are also password authenticated key exchange algorithms. RSA key exchange using PKI or webs-of-trust or trusted key servers are also commonly used.

Key transport protocols Key Transport protocols are where one party generates the key and sends it securely to the recipient(s).

Key agreement protocols Key Agreement protocols are protocols whereby N parties (usually two) can agree on a common key with all parties contributing to the key value. These protocols prevent adversaries from learning the key or forcing their own key choice on the participating parties.

Further reading

- OWASP Cheat Sheet series
  - Authentication Cheat Sheet
  - Authorization_Cheat_Sheet
  - Cryptographic Storage Cheat Sheet
  - Key Management Cheat Sheet
  - SAML Security Cheat Sheet
  - Secure Product Design Cheat Sheet
  - User Privacy Protection Cheat Sheet

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

2.5 OWASP Top Ten

The OWASP Top Ten is a very well known list of web application security risks, and is included by the OWASP Software Assurance Maturity Model (SAMM) in the Education & Guidance practice within the Governance business function.

Overview

The OWASP Top 10 Web Application Security Risks project is probably the most well known security concept within the security community, achieving wide spread acceptance and fame soon after its release in 2003. Often referred to as just the 'OWASP Top Ten', it is a list that identifies the most important threats to web applications and seeks to rank them in importance and severity.

The list has changed over time, with some threat types becoming more of a problem to web applications and other threats becoming less of a risk as technologies change. The latest version was issued in 2021 and each category is summarised below.

Note that there are various 'OWASP Top Ten' projects, for example the 'OWASP Top 10 for Large Language Model Applications', so to avoid confusion the context should be noted when referring to these lists.

A01:2021 Broken Access Control Access control involves the use of protection mechanisms that can be categorized as:

- Authentication (proving the identity of an actor)
- Authorization (ensuring that a given actor can access a resource)
- Accountability (tracking of activities that were performed)

Broken Access Control is where the product does not restrict, or incorrectly restricts, access to a resource from an unauthorized or malicious actor. When a security control fails or is not applied then attackers can compromise the security of the product by gaining privileges, reading sensitive information, executing commands, evading detection, etc.

Broken Access Control can take many forms, such as path traversal or elevation of privilege, so refer to both the Common Weakness Enumeration CWE-284 and A01 Broken Access Control and also follow the various OWASP Cheat Sheets related to access controls.

A02:2021 Cryptographic Failures Referring to OWASP Top 10 A02:2021, sensitive data should be protected when at rest and in transit. Cryptographic failures occur when the cryptographic security control is either broken or not applied, and the data is exposed to unauthorized actors - malicious or not.

It is important to protect data both at rest, when it is stored in an area of memory, and also when it is in transit such as being transmitted across a communication channel or being transformed. A good example of protecting data transformation is given by A02 Cryptographic Failures where sensitive data is properly encrypted in a database, but the export function automatically decrypts the data leading to sensitive data exposure.

Crypto failures can take many forms and may be subtle - a security control that looks secure may be easily broken. Follow the crypto OWASP Cheat Sheets to get the basic crypto controls in place and consider putting a crypto audit in place.

A03:2021 Injection A lack of input validation and sanitization can lead to injection exploits, and this risk has been a constant feature of the OWASP Top Ten since the first version was published in 2003. These vulnerabilities occur when hostile data is directly used within the application and can result in malicious data being used to subvert the application; see A03 Injection for further explanations.

The security control is straight forward: all input from untrusted sources should be sanitized and validated. See the Injection Cheat Sheets for the various types of input and their controls.

A04:2021 Insecure Design It is important that security is built into applications from the beginning and not applied as an afterthought. The A04 Insecure Design category recognizes this and advises that the use of threat modeling, secure design patterns, and reference architectures should be incorporated within the application design and architecture activities.

In practice this involves establishing a secure development lifecycle that encourages the identification of security requirements, the periodic use of threat modeling and consideration of existing secure libraries and frameworks. This category was introduced in the 2021 version and for now the supporting cheat sheets only cover threat modeling; as this category becomes more established it is expected that more supporting information will become available.

A05:2021 Security Misconfiguration Systems and large applications can be configurable, and this configuration is often used to secure the system/application. If this configuration is misapplied then the application may no longer be secure, and instead be vulnerable to well-known exploits. The A05 Security Misconfiguration page contains a common example of misconfiguration where default accounts and their passwords are still enabled and unchanged. These passwords and accounts are usually well-known and provide an easy way for malicious actors to compromise applications.

Both the OWASP Top 10 A05:2021 and the linked OWASP Cheat Sheets provide strategies to establish a concerted, repeatable application security configuration process to minimise misconfiguration.

A06:2021 Vulnerable and Outdated Components Perhaps one of the easiest and most effective security activities is keeping all the third party software dependencies up to date. If a vulnerable dependency is identified by a malicious actor during the reconnaissance phase of an attack then there are databases available, such as Exploit Database, that will provide a description of the exploit. These databases can also provide ready made scripts and techniques for attacking a given vulnerability, making it easy for vulnerable third party software dependencies to be exploited .

Risk A06 Vulnerable and Outdated Components underlines the importance of this activity, and recommends that fixes and upgrades to the underlying platform, frameworks, and dependencies are based on a risk assessment and done in a 'timely fashion'. Several tools can used to analyse dependencies and flag vulnerabilities, refer to the Cheat Sheets for these.

A07:2021 Identification and Authentication Failures Confirmation of the user's identity, authentication, and session management is critical to protect the system or application against authentication related attacks. Referring to risk A07 Identification and Authentication Failures, authorization can fail in several ways that often involve other OWASP Top Ten risks:

- broken access controls (A01)
- cryptographic failure (A02)
- default passwords (A05)
- out-dated libraries (A06)

Refer to the Cheat Sheets for the several good practices that are needed for secure authorization. There are also third party suppliers of Identity and Access Management (IAM) that will provide this as a service, consider the cost / benefit of using these (often commercial) suppliers.

A08:2021 Software and Data Integrity Failures Software and data integrity failures relate to code and infrastructure that does not protect against integrity violations. This is a wide ranging category that describes supply chain attacks, compromised auto-update and use of untrusted components for example. A07 Software and Data Integrity Failures was a new category introduced in 2021 so there is little information available from the Cheat Sheets, but this is sure to change for such an important threat.

A09:2021 Security Logging and Monitoring Failures Logging and monitoring helps detect, escalate, and respond to active breaches; without it breaches will not be detected. A09 Security Logging and Monitoring Failures lists various logging and monitoring techniques that should be familiar, but also others that may not be so common; for example monitoring the DevOps supply chain may be just as important as monitoring the application or system. The Cheat Sheets provide guidance on sufficient logging and also

provide for a common logging vocabulary. The aim of this common vocabulary is to provide logging that uses a common set of terms, formats and key words; and this allows for easier monitoring, analysis and alerting.

A10:2021 Server-Side Request Forgery Referring to A10 Server-Side Request Forgery (SSRF), these vulnerabilities can occur whenever a web application is fetching a remote resource without validating the user-supplied URL. These exploits allow an attacker to coerce the application to send a crafted request to an unexpected destination, even when protected by a firewall, VPN, or another type of network access control list. Fetching a URL has become a common scenario for modern web applications and as a result the incidence of SSRF is increasing, especially for cloud services and more complex application architectures.

This is a new category introduced in 2021 with a single (for now) Cheat Sheet that deals with SSRF.

OWASP top tens

There are various 'Top 10' projects created by OWASP that, depending on the context, may also be referred to as 'OWASP Top 10'. Here is a list of the stable 'OWASP Top 10' projects:

- API Security Top 10
- Data Security Top 10
- Low-Code/No-Code Top 10
- Mobile Top 10
- Top 10 CI/CD Security Risks
- Top 10 for Large Language Model Applications
- Top 10 Privacy Risks
- Top 10 Proactive Controls
- Top 10 Web Application Security Risks

There are other OWASP Top 10s that are still being worked on as 'incubator' projects so this list may change.

OWASP Top 10 versions

The OWASP Top 10 Web Application Security Risks document was originally published in 2003, making it one of the longest lived OWASP projects. Listed below are the versions up to the latest in 2021, which was released to mark 20 years of OWASP.

- Original 2003
- 2004
- 2007
- 2010
- 2013
- 2017
- Latest 2021

The OWASP Developer Guide is a community effort; if you see something that needs changing then submit an issue or edit on GitHub.

2.6 Security champions

A Security Champion program is a commonly used way of helping development teams successfully run a development lifecycle that is secure, and this is achieved by selecting members of teams to become Security Champions. The role of Security Champion is described by the OWASP Software Assurance Maturity Model (SAMM) Organization and Culture activities within the Governance business function of the Education & Guidance practice.

Overview Referring to the OWASP Security Culture project, it can be hard to introduce security across development teams using the application security team alone. Information security people do not scale across teams of developers. A good way to scale security and distribute security across development teams is by creating a security champion role and providing a Security Champions program to encourage a community spirit within the organization.

Security champions are usually individuals within each development team that show special interest in application security. The security champion provides a knowledgeable point of contact between the application security team and development, and in addition they can ensure that the development lifecycle has security built in. Often the security champion carries on with their original role within the development team, in addition to their new role, and so a Security Champions program is important for providing support and training and avoiding burn-out.

Security champion role Security champions are active members of a development team that act as the "voice" of security within their team. Security champions also provide visibility of their team's security activities to the application security team, and are seen as the first point of contact between developers and a central security team.

There is no universally defined role for a security champion, but the Security Culture project provides various suggestions:

- Evangelise security: promoting security best practice in their team, imparting security knowledge and helping to uplift security awareness in their team
- Contribute to security standards: provide input into organisational security standards and procedures
- Help run activities: promote activities such as Capture the Flag events or secure coding training
- Oversee threat modelling: threat modelling consists of a security review on a product in the design phase
- Oversee secure code reviews: raise issues of risk in the code base that arise from peer group code reviews
- Use security testing tools: provide support to their team for the use of security testing tools

The security champion role requires a passion and interest in application security, and so arbitrarily assigning this role is unlikely to work in practice. A better strategy is to provide a security champions program, so that developers who are interested can come forward; in effect they should self-select.

Security champions program It can be tough being a security champion: usually they are still expected to do their 'day job' but in addition they are expected to be knowledgeable on security matters and to take extra training. Relying on good will and cheerful interest will only go so far, and a Security Champions program should be put in place to identify, nurture, support and train the security champions.

The OWASP Security Champions Guide identifies ten key principles for a successful Security Champions program:

- Be passionate about security - identify the members of the teams that show interest in security
- Start with a clear vision for your program - be practical but ambitious, after if it works then it will work well
- Secure management support - as always, going it alone without management support is never going to work
- Nominate a dedicated captain - the program will take organisation and maintaining, so find someone willing to do that
- Trust your champions - they are usually highly motivated when it comes to the security of their own applications

- Create a community - it can be lonely, so provide a support network
- Promote knowledge sharing - if the community is in place, then encourage discussions and meet-ups
- Reward responsibility - they are putting extra work, so appreciate them
- Invest in your champions - the knowledge gained through extra training will pay for itself many times over
- Anticipate personnel changes - the security champion may move on, be alert to this and plan for it

A successful security champions program will increase the security of the applications / systems, allay developer's fears, increase the effectiveness of the application security team and improve the security posture of the organization as a whole.

Further reading

- Security Champions Playbook
- OWASP Security Champions Guide
- OWASP Security Culture project

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

3. Requirements

Referring to the OWASP Top Ten Proactive Controls, security requirements are statements of security functionality that ensure the different security properties of a software application are 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 potential vulnerabilities.

Security requirements also 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.

So you can look at it this way: requirements exist to prevent the repeat of past security failures.

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

3.1 Requirements in practice

This section deals with Security Requirements, which is a security practice in the Design business function section of the OWASP Software Assurance Maturity Model (SAMM). This security requirements practice has two activities, Software Requirements and Supplier Security, with regulatory and statutory requirements being an important subset of both these activities.

Overview Security requirements are part of every secure development process and form the foundation for the application's security posture - they will certainly help with the prevention of many types of vulnerabilities.

Requirements can come from many sources and in general there are three main sources:

1. Software-related requirements which specify objectives and expectations to protect the service and data at the core of the application
2. Requirements relative to supplier organizations that are part of the development context of the application
3. Regulatory and Statutory requirements

Ideally security requirements are built in at the beginning of development, but there is no wrong time to consider these security requirements and add new ones as necessary.

Software requirements The OWASP Top Ten Proactive Controls describes the most important categories of controls that architects and developers should include in every project. At the head of the list of controls is C1: Define Security Requirements and this reflects the importance of software security requirements: without them the development will not be secure.

Defining security requirements can be daunting at times, for example they may reference cryptographic techniques that can be misapplied, but it is perfectly acceptable to state these requirements in everyday language. For example a security requirement could be written as "Identify the user of the application at all times" and this is certainly sufficient to require that authentication is included in the design.

The SAMM Security Requirements practice lists maturity levels of software security requirements that specify objectives and expectations. Choose the level that is appropriate for the organization and the development team, with the understanding that any of these levels are perfectly acceptable.

The software security requirements maturity levels are:

1. High-level application security objectives are mapped to functional requirements
2. Structured security requirements are available and utilized by developer teams
3. Build a requirements framework for product teams to utilize

OWASP provides projects that can help in identifying security requirements that will protect the service and data at the core of the application. The Application Security Verification Standard provides a list of requirements for secure development, and this can be used as a starting point for the security requirements. The Mobile Application Security provides a similar set of standard security requirements for mobile applications.

Supplier security External suppliers involved in the development process need to be assessed for their security practices and compliance. Depending on their level of involvement these suppliers can have a significant impact on the security of the application so a set of security requirements will have to be negotiated with them.

SAMM lists maturity levels for the security requirements that will clarify the strengths and weaknesses of your suppliers. Note that supplier security is distinct from security of third-party software and libraries, and the use of third-party and open source software is discussed in its own section on dependency checking and tracking.

The supplier security requirements maturity levels are:

1. Evaluate the supplier based on organizational security requirements
2. Build security into supplier agreements in order to ensure compliance with organizational requirements

3. Ensure proper security coverage for external suppliers by providing clear objectives

Regulatory and statutory requirements Regulatory requirements can include security requirements which then must be taken into account. Different industries are regulated are regulated to a lesser or greater extent, and the only general advice is to be aware and follow the regulations.

Various jurisdictions will have different statutory requirements that may result in security requirements. Any applicable statutory security requirement should be added to the application security requirements. Similarly to regulatory requirements, the only general advice is to be familiar with and follow the appropriate statutory requirements.

Periodic review The security requirements should be identified and recorded at the beginning of any new development and also when new features are added to an existing application. These security requirements should be periodically revisited and revised as necessary; for example security standards are updated and new standards come into force, both of which may have a direct impact on the application.

Further reading

- OWASP projects:
  - Software Assurance Maturity Model (SAMM)
  - Top Ten Proactive Controls
  - Application Security Verification Standard (ASVS)
  - Mobile Application Security

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

3.2 Risk profile

This section discusses the Application Risk Profile, an activity in the OWASP Software Assurance Maturity Model (SAMM). The risk profile activity is part of the Threat Assessment security practice in the Design business function.

Overview Risk management is the identification, assessment, and prioritization of risks to the application or system. The objective of risk management is to ensure that uncertainty does not deflect development activities away from the business goals.

Remediation is the strategy chosen in response to a risk to the business system, and these risks are identified using various techniques such as threat modeling and security requirements analysis.

Risk management can be split into two phases. First create a risk profile for the application and then provide solutions (remediate) to those risks in a way that is best for the business; risk management should always be business driven.

Application risk profile The application risk profile is created to understand the likelihood and also the impact of an attack. Over time various profiles could be created and these should be stored in a risk profile inventory, and ideally the risk profiles should be revisited as part of the organization's secure development strategy.

Quantifying risks is often difficult and there are many ways of approaching this; refer to the reading list below for various strategies in creating a risk rating model. The OWASP page on Risk Rating Methodology describes some steps in identifying risks and quantifying them:

1. Identifying a risk
2. Factors for estimating likelihood
3. Factors for estimating impact
4. Determining severity of the risk
5. Deciding what to fix
6. Customizing the risk rating model

The activities involved in creating a risk profile depend very much on the processes and maturity level of the organization, which is beyond the level of this Developer Guide, so refer to the further reading listed below for guidance and examples.

Remediation Risks identified during threat assessment, for example through the risk profile or through threat modeling, should have solutions or remedies applied.

There are four common ways to handle risk, often given the acronym TAME:

1. Transfer: the risk is considered serious but can be transferred to another party
2. Acceptance: the business is aware of the risk but has decided that no action needs to be taken; the level of risk is deemed acceptable
3. Mitigation: the risk is considered serious enough to require implementation of security controls to reduce the risk to an acceptable level
4. Eliminate / Avoid: the risk can be avoided or removed completely, often by removing the object with which the risk is associated

Examples:

1. Transfer: a common example of transferring risk is the use of third party insurance in response to the risk of RansomWare. Insurance premiums are paid but losses to the business are covered by the insurance
2. Acceptance: sometimes a risk is low enough in priority, or the outcome bearable, that it is not worth mitigating, an example might be where the version of software is revealed but this is acceptable (or even desirable)

3. Mitigation: it is usual to mitigate the impact of a risk, for example input sanitization or output encoding may be used for information supplied by an untrusted source, or the use of encrypted communication channels for transferring high risk information
4. Eliminate: an example may be that an application implements legacy functionality that is no longer used, if there is a risk of it being exploited then the risk can be eliminated by removing this legacy functionality

Further reading

- OWASP Risk Rating Methodology
- NIST 800-30 - Guide for Conducting Risk Assessments
- Government of Canada - The Harmonized Threat and Risk Assessment Methodology
- Mozilla's Risk Assessment Summary and Rapid Risk Assessment (RRA)
- Common Vulnerability Scoring System (CVSS) used for severity and risk ranking

---

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

3.3 Security Knowledge Framework

To Do: supply a couple of sentences on the Security Knowledge Framework (SKF), such as its status and where to find it. Although it is no longer an OWASP project, it was an important project for OWASP and is still referenced in the Wayfinder.

What is the Security Knowledge Framework? To Do: go into more detail about SKF so that a developer can gain an overview of what it can provide for them.

Why use it? To Do: provide more context for SKF that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how the SKF can provide value for development teams. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 1: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

3.4 Security RAT

To Do: supply a couple of sentences on the OWASP Security RAT (Requirement Automation Tool) project, such as its status as an OWASP project and where to find it.

What is Security RAT? To Do: go into more detail about Security RAT so that a developer can gain an overview of what the tool can provide for them.

Why use it? To Do: provide more context for Security RAT that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how Security RAT can provide value for the development teams. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 2: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

3.5 Application Security Verification Standard

To Do: supply a couple of sentences on the Application Security Verification Standard (ASVS) project, such as its status as an OWASP project and where to find it.

What is ASVS? To Do: go into more detail about ASVS so that a developer can gain an overview of what the tool can provide for them, specifically from a requirements point of view.

Why use it? To Do: provide more context for ASVS that allows developers to determine whether to use it in their requirements process.

How to use it To Do: give a brief outline of how ASVS can be used for the development teams and their requirements. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 3: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

3.6 Mobile Application Security

The OWASP Mobile Application Security (MAS) flagship project has the mission statement: "Define the industry standard for mobile application security".

The MAS project covers the processes, techniques, and tools used for security testing a mobile application, as well as an exhaustive set of test cases that enables testers to deliver consistent and complete results. The OWASP MAS project provides the Mobile Application Security Verification Standard (MASVS) for mobile applications and a comprehensive Mobile Application Security Testing Guide (MASTG).

What is MASVS? The OWASP MASVS is used by mobile software architects and developers seeking to develop secure mobile applications, as well as security testers to ensure completeness and consistency of test results. The MAS project has several uses; when it comes to defining requirements then the MASVS contains a list of security controls for mobile applications.

The security controls are split into several categories:

- MASVS-STORAGE
- MASVS-CRYPTO
- MASVS-AUTH
- MASVS-NETWORK
- MASVS-PLATFORM
- MASVS-CODE
- MASVS-RESILIENCE

Why use MASVS? The OWASP MASVS is the industry standard for mobile application security and it is expected that any given set of security requirements will satisfy the MASVS. When defining security requirements for mobile applications then each security control in the MASVS should be considered.

How to use MASVS MASVS can be accessed online and the links followed for each security control. In addition MASVS can be downloaded as a PDF which can, for example, be used for evidence or compliance purposes. Inspect each control within MASVS and regard it as a potential security requirement.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4. Design

Security Architecture is one of the practices that is described in the Software Assurance Maturity Model (SAMM). Referring to the Secure Product Design Cheat Sheet, the purpose of secure architecture and design is to ensure that all products meet or exceed the security requirements laid down by the organization, focusing on the security linked to components and technologies used during the development of the application.

Secure Architecture Design looks at the selection and composition of components that form the foundation of the solution. Technology Management looks at the security of supporting technologies used during development, deployment and operations, such as development stacks and tooling, deployment tooling, and operating systems and tooling.

A secure design will help establish secure defaults, minimise the attack surface area and fail securely to well-defined and understood defaults. It will also consider and follow various principles, such as:

- Least Privilege and Separation of Duties
- Defense-in-Depth
- Zero Trust
- Security in the Open

A Secure Development Lifecycle (SDLC) helps to ensure that all security decisions made about the product being developed are explicit choices and result in the correct level of security for the product design. Various secure development lifecycles can be used and they generally include threat modeling in the design process.

Checklists and Cheat Sheets are an important tool during the design process; they provide an easy reference of knowledge and help avoid repeating design errors and mistakes.

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

4.1 Threat modeling

Referring to the Threat Modeling Cheat Sheet, threat modeling is a structured approach of identifying and prioritizing potential threats to a system. The threat modeling process includes determining the value that potential mitigations would have in reducing or neutralizing these threats.

Assessing potential threats during the design phase of your project can save significant resources that might be needed to refactor the project to include risk mitigations during a later phase of the project. The outputs from the threat modeling activities generally include:

- Documenting how data flows through a system to identify where the system might be attacked
- Identifying as many potential threats to the system as possible
- Suggesting security controls that may be put in place to reduce the likelihood or impact of a potential threat

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

4.1.1 Threat modeling in practice

This section discusses Threat Modeling, an activity in the OWASP Software Assurance Maturity Model (SAMM). Threat modeling is part of the Threat Assessment security practice in the Design business function. Much of the material in this section is drawn from the OWASP Threat Model project.

Overview Threat modeling work to discover what can go wrong with a system and to determine what to do about it often results in threat model deliverables. Those take many forms including system models and diagrams, lists of threats, mitigations or assumptions, meeting notes, and more. Sometimes those are assembled into a single "threat model" document, a structured representation of all the information that affects the security of an application. In essence, it is a view of the application and its environment through security glasses.

Threat modeling is a process for capturing, organizing, and analyzing all of this information. and enables informed decision-making about application security risk. In addition to producing a model, typical threat modeling efforts also produce a prioritized list of *potential* security vulnerabilities in the concept, requirements, design, or implementation. Any potential vulnerabilities that have been identified from the model should then be remediated using one of the common strategies: mitigate, eliminate, transfer or accept the threat of being exploited.

There are many reasons for doing threat modeling but the most important one is that this activity is *useful* , it is probably the only stage in a development lifecycle where a team sits back and asks: 'What can go wrong?' There are other reasons for threat modeling, for example standards compliance or analysis for disaster recovery, but the main aim of threat modeling is to remedy (possible) vulnerabilities before the malicious actors can exploit them.

What is threat modeling Threat modeling works to identify, communicate, and understand threats and mitigations within the context of protecting something of value.

Threat modeling can be applied to a wide range of things, including software, applications, systems, networks, distributed systems, things in the Internet of things, business processes, etc. There are very few technical products which cannot be threat modeled; more or less rewarding, depending on how much it communicates, or interacts, with the world.

A threat model document is a record of the threat modeling process, and often includes:

- a description / design / model of what you're worried about
- a list of assumptions that can be checked or challenged in the future as the threat landscape changes
- potential threats to the system
- remediation / actions to be taken for each threat
- ways of validating the model and threats, and verification of success of actions taken

The threat model should be in a form that can be easily revised and modified during subsequent threat modeling discussions.

Why do it Like all engineering activities, effort spent on threat modeling has to be justifiable. Rarely any project or development has engineering effort that is going 'spare', and the benefits of threat modeling have to outweigh the engineering cost of this activity. Usually this is difficult to quantify; an easier way to approach it may be to ask what are the costs of *not* doing threat modeling? These costs may consist of a lack of compliance, an increased risk of being exploited, harm to reputation and so on.

The inclusion of threat modeling in the secure development activities can help:

- Build a secure design
- Efficient investment of resources; appropriately prioritize security, development, and other tasks
- Bring Security and Development together to collaborate on a shared understanding, informing development of the system
- Identify threats and compliance requirements, and evaluate their risk
- Define and build required controls.
- Balance risks, controls, and usability

- Identify where building a control is unnecessary, based on acceptable risk
- Document threats and mitigation
- Ensure business requirements (or goals) are adequately protected in the face of a malicious actor, accidents, or other causes of impact
- Identification of security test cases / security test scenarios to test the security requirements

Threat modeling also provides a clear "line of sight" across a project that can be used to justify other security efforts. The threat model allows security decisions to be made rationally, with all the information available, so that security decisions can be properly supported. The threat modeling process naturally produces an assurance argument that can be used to explain and defend the security of an application. An assurance argument starts with a few high level claims and then justifies them with either sub-claims or evidence.

When to threat model There is no wrong time to do threat modeling; the earlier it is done in the development lifecycle the more beneficial it is, but it threat modeling is also useful at any time during application development.

Threat modeling is best applied continuously throughout a software development project. The process is essentially the same at different levels of abstraction, although the information gets more and more granular throughout the development lifecycle. Ideally, a high-level threat model should be defined in the concept or planning phase, and then refined during the development phases. As more details are added to the system new attack vectors are identified, so the ongoing threat modeling process should examine, diagnose, and address these threats.

Note that it is a natural part of refining a system for new threats to be exposed. For example, when you select a particular technology, such as Java for example, you take on the responsibility to identify the new threats that are created by that choice. Even implementation choices such as using regular expressions for validation introduce potential new threats to deal with.

Threat modeling: *the sooner the better, but never too late*

Questions to ask Often threat modeling is a conceptual activity rather than a rigorous process, where development teams are brought together and asked to think up ways of subverting their system. To provide some structure it is useful to start with Shostack's Four Question Framework:

1 What are we working on?

As a starting point the scope of the Threat Model should be defined. This will require an understanding of the application that is being built, and some examples of inputs for the threat model could be:

- Architecture diagrams
- Dataflow transitions
- Data classifications

It is common to represent the answers to this question with one or more data flow diagrams and often supplemental diagrams like message sequence diagrams.

It is best to gather people from different roles with sufficient technical and risk awareness so that they can agree on the framework to be used during the threat modeling exercise.

2 What can go wrong?

This is a research activity to find the main threats that apply to your application. There are many ways to approach the question, including open discussion or using a structure to help think it through. Techniques and methodologies to consider include CIA, STRIDE, LINDDUN, cyber kill chains, PASTA, common attack patterns (CAPEC) and others.

There are resources available that will help with identifying threats and vulnerabilities. OWASP provide a set of cards, Cornucopia, that provide suggestions and explanations for general vulnerabilities. The game Elevation of Privileges threat modeling card game is an easy way to get started with threat modeling, and there is the OWASP version of Snakes and Ladders that truly gamifies these activities.

3 What are we going to do about that?

In this phase turn the threat model findings into specific actions. Consider the appropriate remediation for each threat identified: Transfer, Avoid, Mitigate or Eliminate.

4 Did we do a good enough job?

Finally, carry out a retrospective activity over the work identified to check quality, feasibility, progress, or planning.

The OWASP Threat Modeling Playbook goes into these practicalities in more detail and provides strategies for maintaining threat modeling within an organisation.

How to do it There is no one process for threat modeling. How it is done in practice will vary according to the organisation's culture, according to what type of system / application is being modeled and according to preferences of the development team itself. The various techniques and concepts are discussed in the Threat Modeling Cheat Sheet and can be summarised:

1. Terminology: try to use standard terms such as actors, trust boundaries, etc as this will help convey these concepts
2. Scope: be clear what is being modeled and keep within this scope
3. Document: decide which tools and what outputs are required to satisfy compliance, for example
4. Decompose: break the system being modeled into manageable pieces and identify your trust boundaries
5. Agents: identify who the actors are (malicious or otherwise) and what they can do
6. Categorise: prioritise the threats taking into account probability, impact and any other factors
7. Remediation: be sure to decide what to do about any threats identified, the whole reason for threat modeling

It is worth saying this again: there are many ways to do threat modeling, all perfectly valid, so choose the right process that works for a specific team.

Final advice Finally some advice on threat modeling.

Make it incremental:

Strongly consider using incremental threat modeling. It is almost certainly a bad idea trying to fully model an existing application or system; it can be very time consuming modeling a whole system, and by the time such a model was completed then it would probably be out of date. Instead incrementally model new features or enhancements as and when they are being developed.

Incremental threat modeling assumes that existing applications and features have already been attacked over time and these real world vulnerabilities have been remediated. It is the new features or new applications that pose a greater security risk; if they are vulnerable then they will reduce the security of the existing application or system. Concentrating on the new changes applies threat modeling effort at the place that it is needed most; at the very least the changes should not make the security worse - and ideally the security should be better.

Tools are secondary:

It is good to standardise threat modeling tools across an organisation, but also allow teams to choose how they record their threat models. If one team decides to use Threat Dragon, for example, and another wants to use a drawing board, then that is usually fine. The discussions had during the threat modeling process are more important than the tool used, although you might ask the team using the drawing board how they implement their change control.

Brevity is paramount:

It is very easy to create a threat model that looks a lot like a system diagram, with many components and data flows. This makes for a convincing diagram, but it is not a model specific to the threat of exploits. Instead concentrate on the attack / threat surfaces and be robust in consolidating multiple system components into one threat model component. This will keep the number of components and dataflows manageable, and focuses the discussion on what matters most: malicious actors (external or internal) trying to subvert your system.

Choose your methodology:

It is a good strategy to choose a threat categorisation methodology for the whole organisation and then try and keep to it. This could be STRIDE or LINDDUN, but if the CIA triad gives enough granularity then that is a perfectly good choice.

Further reading

- Threat Modeling Manifesto
- OWASP Threat Model project
- OWASP Threat Modeling Cheat Sheet
- OWASP Threat Modeling Playbook (OTMP)
- OWASP Attack Surface Analysis Cheat Sheet
- OWASP community pages on Threat Modeling and the Threat Modeling Process
- The Four Question Framework For Threat Modeling 60 second video
- Lockheed's Cyber Kill Chain
- VerSprite's Process for Attack Simulation and Threat Analysis (PASTA)
- Threat Modeling: Designing for Security
- Threat Modeling: A Practical Guide for Development Teams

Resources

- Shostack's Four Question Framework
- OWASP pytm Pythonic Threat Modeling tool
- OWASP Threat Dragon threat modeling tool using dataflow diagrams
- Threagile, an open source project that provides for Agile threat modeling
- Microsoft Threat Modeling Tool, a widely used tool throughout the security community and free to download
- threatspec, an open source tool based on comments inline with code
- Mitre's Common Attack Pattern Enumeration and Classification (CAPEC)
- NIST Common Vulnerability Scoring System Calculator

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.1.2 Pythonic Threat Modeling

The OWASP Pythonic Threat Modeling (pytm) project is a framework for threat modeling and its automation. The goal of pytm is to shift threat modeling to the left, making threat modeling more automated and developer-centric.

Pytm is an OWASP Lab Project with a community of contributors creating several releases.

What is pytm? Pytm is a Java library that provides programmatic way of threat modeling; the application model itself is defined as a python3 source file and follows Python program syntax. Findings are included in the application model python program with threats defined as rows in an associated text file. The threat file can be reused between projects and provides for accumulation of a knowledge base.

Using pytm the model and threats can be programmatically output as a dot data flow diagram which is displayed using Graphviz, an open source graph visualization software utility. Alternatively the model and threats can be output as a PlantUML file which can then be displayed, using Java and the PlantUML `.jar`, as a sequence diagram.

If a report document is required then a pytm script can output the model, threats and findings as markdown. Programs such as pandoc can then take this markdown file and provide the document in various formats such as PDF, ePub or html.

Why use pytm? The pytm development team state that traditional threat modeling often comes too late in the development process, and sometimes may not happen at all. In addition, creating manual / diagrammatic data flows and reports can be extremely time-consuming. These are very good points, and pytm attempts to get threat modeling to 'shift-left' in the development lifecycle.

Many traditional threat modeling tools such as OWASP Threat Dragon provide a graphical way of creating diagrams and entering threats. These applications store the models as text, for example JSON and YAML, but the main entry method is via the application.

Pytm is different - the primary method of creating and updating the threat models is through code. This source code completely defines the model along with its findings, threats and remediations. Diagrams and reports are regarded as outputs of the model; not the inputs to the model. This makes pytm a powerful tool for describing a system or application, and allows the model to be built up over time.

This focus on the model as code and programmatic outputs makes Pytm particularly useful in automated environments, helping the threat model to be built in to the design process from the start, as well as in the more traditional threat modeling sessions.

How to use pytm The best description of how to use pytm is given in chapter 4 of the book Threat Modeling: a practical guide for development teams which is written by two of the main contributors to the pytm project.

Pytm is code based within a program environment, rather than run as a single application, so there are various components that have to be installed on the target machine to allow pytm to run. At present it does not work on Windows, only Linux or MacOS, so if you need to run Windows then use a Linux VM or follow the instructions to create a Docker container.

The following tools and libraries need to be installed:

- Python 3.x
- Graphviz package
- Java, such as OpenJDK 10 or 11
- the PlantUML executable JAR file
- and of course pytm itself: clone the pytm project repo

Once the environment is installed then navigate to the top directory of your local copy of the project.

Follow the example given by the pytm project repo and run the suggested scripts to output the data flow diagram, sequence diagram and report:

```
mkdir -p tm
./tm.py --report docs/basic_template.md | pandoc -f markdown -t html > tm/report.html
./tm.py --dfd | dot -Tpng -o tm/dfd.png
./tm.py --seq | java -Djava.awt.headless=true -jar $PLANTUML_PATH -tpng -pipe > tm/seq.png
```

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.1.3 Threat Dragon

The OWASP Threat Dragon project provides a diagrammatic tool for threat modeling applications, APIs and software systems. It is an OWASP Lab Project with several releases and is in active development.

What is Threat Dragon? Threat Dragon is a tool that can help development teams with their threat modeling process. It provides for creating and modifying data flow diagrams which provide the context and direction for the threat modeling activities. It also stores the details of threats identified during the threat modeling sessions, and these are stored alongside the threat model diagram in a text-based file. Threat Dragon can also output the threat model diagram and the associated threats as a PDF report.

Why use it? Threat Dragon is a useful tool for small teams to create threat models quickly and easily. Threat Dragon aims for:

- Simplicity - you can install and start using Threat Dragon very quickly
- Flexibility - the diagramming and threat generation allows all types of threat to be described
- Accessibility - different types of teams can benefit from Threat Dragon's simplicity and flexibility

It supports various methodologies and threat categorizations used during the threat modeling activities:

- STRIDE
- LINDDUN
- CIA
- DIE

and it can be used by a wide range of developers and teams:

- Simplicity - installation is easy and teams can start using Threat Dragon quickly
- Flexibility - the diagramming and threat generation allows all types of threat to be described
- Accessibility - various different types of teams can all benefit from Threat Dragon ease of use

How to use it Threat Dragon is distributed as a cross platform desktop application as well a web application. It is straightforward to start using Threat Dragon; the latest version is available to use online:

1. select Login to Local Session
2. select Explore a Sample Threat Model
3. select Version 2 Demo Model
4. you are presented with the threat model meta-data which can be edited
5. click on the diagram Main Request Data Flow to display the data flow diagram
6. the diagram components can be inspected, and their associated threats are displayed
7. components can be added and deleted, along with editing their properties

The desktop application can be downloaded for Windows, Linux and MacOS. The web application can be run using a Docker container or from the source code.

An important feature of Threat Dragon is the PDF report output which can be used for documentation and GRC compliance purposes; from the threat model meta-data window click on the Report button.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.1.4 Threat Modeling and Cornucopia

To Do: supply a couple of sentences on the Cornucopia card game, including its status as an OWASP project and where to download or buy it.

What is pytm? To Do: go into more detail about Cornucopia so that a developer can gain an overview of what it is, and its use during threat modeling.

Why use it? To Do: provide more context on Cornucopia that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how to use Cornucopia, but do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 4: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

4.1.5 Threat Modeling toolkit

To Do: we need to go through the various sources shown in the OWASP Threat Modeling toolkit presentation.

Probably also should describe:

- OWASP Threat Modeling Project
- OWASP Threat Modeling Cheat Sheet
- OWASP Threat Modeling Playbook (OTMP)
- OWASP Attack Surface Analysis Cheat Sheet
- OWASP community pages on Threat Modeling and the Threat Modeling Process

but this is open to discussion

Figure 5: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

4.2. Web application checklist

Checklists are a valuable resource for development teams. They provide structure for establishing good practices and processes and are also useful during code reviews and design activities.

The checklists that follow are general lists that are categorised to follow the controls listed in the OWASP Top 10 Proactive Controls project. These checklists provide suggestions that certainly should be tailored to an individual project's requirements and environment; they are not meant to be followed in their entirety.

Probably the best advice on checklists is given by the Application Security Verification Standard (ASVS). The ASVS can be used to provide a framework for an initial checklist, according to the security verification level, and this initial ASVS checklist can then be expanded using the following checklist sections.

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

4.2.1 Checklist: Define Security Requirements

A security requirement is a statement of security functionality that ensures software security is being satisfied. Security requirements are derived from industry standards, applicable laws, and a history of past vulnerabilities.

Refer to proactive control C1: Define Security Requirements for more context from the OWASP Top 10 Proactive Controls project, and use the checklists below as suggestions for the checklist that has been tailored for the individual project.

System configuration

- Restrict applications, processes and service accounts to the least privileges possible
- If the application must run with elevated privileges, raise privileges as late as possible, and drop as soon as possible
- Remove all unnecessary functionality and files
- Remove test code or any functionality not intended for production, prior to deployment
- The security configuration store for the application should be available in human readable form to support auditing
- Isolate development environments from production and provide access only to authorized development and test groups
- Implement a software change control system to manage and record changes to the code both in development and production

Cryptographic practices

- Use peer reviewed and open solution cryptographic modules
- All cryptographic functions used to protect secrets from the application user must be implemented on a trusted system
- Cryptographic modules must fail securely
- Ensure all random elements such as numbers, file names, UUID and strings are generated using the cryptographic module approved random number generator
- Cryptographic modules used by the application are compliant to FIPS 140-2 or an equivalent standard
- Establish and utilize a policy and process for how cryptographic keys will be managed
- 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

File management

- Do not pass user supplied data directly to any dynamic include function
- Require authentication before allowing a file to be uploaded
- Limit the type of files that can be uploaded to only those types that are needed for business purposes
- Validate uploaded files are the expected type by checking file headers rather than by file extension
- Do not save files in the same web context as the application
- Prevent or restrict the uploading of any file that may be interpreted by the web server.
- Turn off execution privileges on file upload directories
- When referencing existing files, use an allow-list of allowed file names and types
- Do not pass user supplied data into a dynamic redirect
- Do not pass directory or file paths, use index values mapped to pre-defined list of paths
- Never send the absolute file path to the client
- Ensure application files and resources are read-only
- Scan user uploaded files for viruses and malware

References

- OWASP Application Security Verification Standard (ASVS)

- OWASP Mobile Application Security
- OWASP Top 10 Proactive Controls

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.2.2 Checklist: Leverage Security Frameworks and Libraries

Secure coding libraries and software frameworks with embedded security help software developers guard against security-related design and implementation flaws.

Refer to proactive control C2: Leverage Security Frameworks and Libraries for more context from the OWASP Top 10 Proactive Controls project, and use the checklist below as suggestions for the checklist that has been tailored for the individual project.

Security Frameworks and Libraries

- Ensure servers, frameworks and system components are running the latest approved versions and patches
- Use libraries and frameworks from trusted sources that are actively maintained and widely used
- Review all secondary applications and third party libraries to determine business necessity
- Validate safe functionality for all secondary applications and third party libraries
- Create and maintain an inventory catalog of all third party libraries using Software Composition Analysis (SCA)
- Proactively keep all third party libraries and components up to date
- Reduce the attack surface by encapsulating the library and expose only the required behaviour into your software
- Use tested and approved managed code rather than creating new unmanaged code for common tasks
- Utilize task specific built-in APIs to conduct operating system tasks
- Do not allow the application to issue commands directly to the Operating System
- Use checksums or hashes to verify the integrity of interpreted code, libraries, executables, and configuration files
- Restrict users from generating new code or altering existing code
- Implement safe updates using encrypted channels

References

- OWASP Dependency Check
- OWASP Top 10 Proactive Controls

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.2.3 Checklist: Secure Database Access

Ensure that access to all data stores is secure, including both relational databases and NoSQL databases.

Refer to proactive control C3: Secure Database Access for more context from the OWASP Top 10 Proactive Controls project, and use the checklists below as suggestions for the checklist that has been tailored for the individual project.

Secure queries

- Use Query Parameterization to prevent untrusted input being interpreted as part of a SQL command
- Use strongly typed parameterized queries
- Utilize input validation and output encoding and be sure to address meta characters
- Do not run the database command if input validation fails
- Ensure that variables are strongly typed
- Connection strings should not be hard coded within the application
- Connection strings should be stored in a separate configuration file on a trusted system and they should be encrypted

Secure configuration

- The application should use the lowest possible level of privilege when accessing the database
- Use stored procedures to abstract data access and allow for the removal of permissions to the base tables in the database
- Close the database connection as soon as possible
- Turn off all unnecessary database functionality
- Remove unnecessary default vendor content, for example sample schemas
- Disable any default accounts that are not required to support business requirements

Secure authentication

- Remove or change all default database administrative passwords
- The application should connect to the database with different credentials for every trust distinction (for example user, read-only user, guest, administrators)
- Use secure credentials for database access

References

- OWASP Cheat Sheet: Query Parameterization
- OWASP Cheat Sheet: Database Security
- OWASP Top 10 Proactive Controls

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.2.4 Checklist: Encode and Escape Data

Encoding and escaping of output data are defensive techniques meant to stop injection attacks on a target system or application which is receiving the output data.

The target system may be another software component or it may be reflected back to the initial system, such as operating system commands, so encoding and escaping output data helps to provide defense in depth for the system as a whole.

Refer to proactive control C4: Encode and Escape Data for more context from the OWASP Top 10 Proactive Controls project, and use the checklists below as suggestions for the checklist that has been tailored for the individual project.

Character encoding and canonicalization

- Apply output encoding just before the content is passed to the target system
- Conduct all output encoding on a trusted system
- Utilize a standard, tested routine for each type of outbound encoding
- Specify character sets, such as UTF-8, for all outputs
- Apply canonicalization to convert unicode data into a standard form
- Ensure the output encoding is safe for all target systems
- In particular sanitize all output used for operating system commands

Contextual output encoding Contextual output encoding of data is based on how it will be utilized by the target. The specific methods vary depending on the way the output data is used, such as HTML entity encoding.

- Contextually encode all data returned to the client from untrusted sources
- Contextually encode all output of untrusted data to queries for SQL, XML, and LDAP

References

- OWASP Cheat Sheet: Injection Prevention
- OWASP Java Encoder Project
- OWASP Top 10 Proactive Controls

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.2.5 Checklist: Validate All Inputs

Input validation is a collection of techniques that ensure only properly formatted data may enter a software application or system component.

It is vital that input validation is performed to provide the starting point for a secure application or system. Without input validation the software application/system will continue to be vulnerable to new and varied attacks.

Refer to proactive control C5: Validate All Inputs for more context from the OWASP Top 10 Proactive Controls project, and use the checklists below as suggestions for the checklist that has been tailored for the individual project.

Syntax and semantic validity

- Identify all data sources and classify them into trusted and untrusted
- Validate all input data from untrusted sources such as client provided data
- Encode input to a common character set before validating
- Specify character sets, such as UTF-8, for all input sources
- If the system supports UTF-8 extended character sets then validate after UTF-8 decoding is completed
- Verify that protocol header values in both requests and responses contain only ASCII characters
- Validate data from redirects
- Validate data range and also data length
- Utilize canonicalization to address obfuscation attacks
- All validation failures should result in input rejection

Libraries and frameworks

- Conduct all input validation on a trusted system
- Use a centralized input validation library or framework for the whole application
- If the standard validation routine cannot address some inputs then use extra discrete checks
- If any potentially hazardous input *must* be allowed then implement additional controls
- Validate for expected data types using an allow-list rather than a deny-list

Validate 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 a definable set of classes is expected
- Isolate features that deserialize so that they run in very low privilege environments such as temporary containers
- Log security deserialization exceptions and failures
- Restrict or monitor incoming and outgoing network connectivity from containers or servers that deserialize
- Monitor deserialization, for example alerting if a user agent constantly deserializes

References

- OWASP Cheat Sheet: Input Validation
- OWASP Java HTML Sanitizer Project
- OWASP Top 10 Proactive Controls

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.2.6 Checklist: Implement Digital Identity

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.

Refer to proactive control C6: Implement Digital Identity for more context from the OWASP Top 10 Proactive Controls project, and use the checklists below as suggestions for the checklist that has been tailored for the individual project.

Authentication

- Design access control authentication thoroughly up-front
- Force all requests to go through access control checks unless public
- Do not hard code access controls that are role based
- Log all access control events
- Use Multi-Factor Authentication for highly sensitive or high value transactional accounts

Passwords

- Require authentication for all pages and resources, except those specifically intended to be public
- All authentication controls must be enforced on a trusted system
- Establish and utilize standard, tested, authentication services whenever possible
- Use a centralized implementation for all authentication controls
- Segregate authentication logic from the resource being requested and use redirection to and from the centralized authentication control
- All authentication controls should fail securely
- Administrative and account management must be at least as secure as the primary authentication mechanism
- If your application manages a credential store, use cryptographically strong one-way salted hashes
- Password hashing must be implemented on a trusted system
- Validate the authentication data only on completion of all data input
- Authentication failure responses should not indicate which part of the authentication data was incorrect
- Utilize authentication for connections to external systems that involve sensitive information or functions
- Authentication credentials for accessing services external to the application should be stored in a secure store
- Use only HTTP POST requests to transmit authentication credentials
- Only send non-temporary passwords over an encrypted connection or as encrypted data
- Enforce password complexity and length requirements established by policy or regulation
- Enforce account disabling after an established number of invalid login attempts
- Password reset and changing operations require the same level of controls as account creation and authentication
- Password reset questions should support sufficiently random answers
- If using email based resets, only send email to a pre-registered address with a temporary link/password
- Temporary passwords and links should have a short expiration time
- Enforce the changing of temporary passwords on the next use
- Notify users when a password reset occurs
- Prevent password re-use
- The last use (successful or unsuccessful) of a user account should be reported to the user at their next successful login
- Change all vendor-supplied default passwords and user IDs or disable the associated accounts
- Re-authenticate users prior to performing critical operations
- If using third party code for authentication inspect the code carefully to ensure it is not affected by any malicious code

Cryptographic based authentication

- Use the server or framework's session management controls
- Session identifier creation must always be done on a trusted system
- Session management controls should use well vetted algorithms that ensure sufficiently random session identifiers
- Set the domain and path for cookies containing authenticated session identifiers to an appropriately restricted value for the site
- Logout functionality should fully terminate the associated session or connection
- Logout functionality should be available from all pages protected by authorization
- Establish a session inactivity timeout that is as short as possible, based on balancing risk and business functional requirements
- Disallow persistent logins and enforce periodic session terminations, even when the session is active
- If a session was established before login, close that session and establish a new session after a successful login
- Generate a new session identifier on any re-authentication
- Do not allow concurrent logins with the same user ID
- Do not expose session identifiers in URLs, error messages or logs
- Implement appropriate access controls to protect server side session data from unauthorized access from other users of the server
- Generate a new session identifier and deactivate the old one periodically
- Generate a new session identifier if the connection security changes from HTTP to HTTPS, as can occur during authentication
- Set the `secure` attribute for cookies transmitted over an TLS connection
- Set cookies with the `HttpOnly` attribute, unless you specifically require client-side scripts within your application to read or set a cookie value

References

- OWASP Cheat Sheet: Authentication
- OWASP Cheat Sheet: Password Storage
- OWASP Cheat Sheet: Forgot Password
- OWASP Cheat Sheet: Session Management
- OWASP Top 10 Proactive Controls

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.2.7 Checklist: Enforce Access Controls

Access Control (or Authorization) is the process of granting or denying specific requests from a user, program, or process.

Refer to proactive control C7: Enforce Access Controls for more context from the OWASP Top 10 Proactive Controls project, and use the checklists below as suggestions for the checklist that has been tailored for the individual project.

Authorization

- Design access control / authorization thoroughly up-front
- Force all requests to go through access control checks unless public
- Deny by default; if a request is not specifically allowed then it is denied
- Apply least privilege, providing the least access as is necessary
- Log all authorization events

Access control

- Enforce authorization controls on every request
- Use only trusted system objects for making access authorization decisions
- Use a single site-wide component to check access authorization
- Access controls should fail securely
- Deny all access if the application cannot access its security configuration information
- Segregate privileged logic from other application code
- Limit the number of transactions a single user or device can perform in a given period of time, low enough to deter automated attacks but above the actual business requirement
- If long authenticated sessions are allowed, periodically re-validate a user's authorization
- Implement account auditing and enforce the disabling of unused accounts
- The application must support termination of sessions when authorization ceases

References

- OWASP Cheat Sheet: Authorization
- OWASP Top 10 Proactive Controls

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.2.8 Checklist: Protect Data Everywhere

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 General Data Protection Regulation GDPR), financial data protection rules such as PCI Data Security Standard (PCI DSS) or other regulations.

Refer to proactive control C8: Protect Data Everywhere for more context from the OWASP Top 10 Proactive Controls project, and use the checklists below as suggestions for the checklist that has been tailored for the individual project.

Data protection

- Classify data according to the level of sensitivity
- Implement appropriate access controls for sensitive data
- Encrypt data in transit
- Ensure secure communication channels are properly configured
- Avoid storing sensitive data when at all possible
- Ensure sensitive data at rest is cryptographically protected to avoid unauthorized disclosure and modification
- Purge sensitive data when that data is no longer required
- Store application-level secrets in a secrets vault
- Check that secrets are not stored in code, config files or environment variables
- Implement least privilege, restricting access to functionality, data and system information
- Protect all cached or temporary copies of sensitive data from unauthorized access
- Purge those temporary copies of sensitive data as soon as they are no longer required

Memory management

- Explicitly initialize all variables and data stores
- Check that any buffers are as large as specified
- Check buffer boundaries if calling the function in a loop and protect against overflow
- Specifically close resources, don't rely on garbage collection
- Use non-executable stacks when available
- Properly free allocated memory upon the completion of functions and at all exit points
- Overwrite any sensitive information stored in allocated memory at all exit points from the function
- Protect shared variables and resources from inappropriate concurrent access

References

- OWASP Cheat Sheet: Cryptographic Storage
- OWASP Cheat Sheet: Secrets Management
- OWASP Top 10 Proactive Controls

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.2.9 Checklist: Implement Security Logging and Monitoring

Logging is recording security information during the runtime operation of an application. Monitoring is the live review of application and security logs using various forms of automation.

Refer to proactive control C9: Implement Security Logging and Monitoring for more context from the OWASP Top 10 Proactive Controls project, and use the checklists below as suggestions for the checklist that has been tailored for the individual project.

Security logging

- Log submitted data that is outside of an expected numeric range.
- Log submitted data that involves changes to data that should not be modifiable
- Log requests that violate server-side access control rules
- Encode and validate any dangerous characters before logging to prevent log injection attacks
- Do not log sensitive information
- Logging controls should support both success and failure of specified security events
- Do not store sensitive information in logs, including unnecessary system details, session identifiers or passwords
- Use a cryptographic hash function to validate log entry integrity

Security logging design

- Protect log integrity
- Ensure log entries that include untrusted data will not execute as code in the intended log viewing interface or software
- Restrict access to logs to only authorized individuals
- Utilize a central routine for all logging operations
- Forward logs from distributed systems to a central, secure logging service
- Follow a common logging format and approach within the system and across systems of an organization
- Synchronize across nodes to ensure that timestamps are consistent
- All logging controls should be implemented on a trusted system
- Ensure that a mechanism exists to conduct log analysis

References

- OWASP Cheat Sheet: Logging
- OWASP Cheat Sheet: Application Logging Vocabulary
- OWASP Top 10 Proactive Controls

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.2.10 Checklist: Handle all Errors and Exceptions

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.

Refer to proactive control C10: Handle all Errors and Exceptions for more context from the OWASP Top 10 Proactive Controls project, and use the checklist below as suggestions for the checklist that has been tailored for the individual project.

Errors and exceptions

- 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 logs give enough information for support, QA, forensics or incident response teams
- Carefully test and verify error handling code
- Do not disclose sensitive information in error responses, for example system details, session identifiers or account information
- Use error handlers that do not display debugging or stack trace information
- Implement generic error messages and use custom error pages
- The application should handle application errors and not rely on the server configuration
- Properly free allocated memory when error conditions occur
- Error handling logic associated with security controls should deny access by default

References

- OWASP Code Review Guide: Error Handling
- OWASP Improper Error Handling
- OWASP Top 10 Proactive Controls

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

4.3 Mobile application checklist

The OWASP Mobile Application Security (MAS) flagship project has the mission statement: "Define the industry standard for mobile application security".

The MAS project covers the processes, techniques, and tools used for security testing a mobile application, as well as an exhaustive set of test cases that enables testers to deliver consistent and complete results. The OWASP MAS project provides the Mobile Application Security Verification Standard (MASVS) for mobile applications and a comprehensive Mobile Application Security Testing Guide (MASTG).

The Mobile Application Security Checklist contains links to the MASTG test cases for each MASVS control.

What is MAS Checklist? The MAS Checklist provides a checklist that keeps track of the MASTG test cases for each MASVS control, and the checklist is split out into categories that match the MASVS categories:

- MASVS-STORAGE sensitive data storage
- MASVS-CRYPTO cryptography best practices
- MASVS-AUTH authentication and authorization mechanisms
- MASVS-NETWORK network communications
- MASVS-PLATFORM interactions with the mobile platform
- MASVS-CODE platform and data entry points along with third-party software
- MASVS-RESILIENCE integrity and running on a trusted platform

In addition to the web links there is a downloadable spreadsheet.

Why use it? If the MASTG is being applied to a mobile application then the MAS Checklist is a handy reference that can also be used for compliance purposes.

How to use it The online version is useful to list the MASVS controls and which MASTG tests apply. Follow the links to access the individual controls and tests.

The spreadsheet download allows the status of each test to be recorded, with a separate sheet for each MASVS category. This record of test results can be used as evidence for compliance purposes.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

5. Implementation

The Implementation business function is described by the OWASP Software Assurance Maturity Model (SAMM). Implementation is focused on the processes and activities related to how an organization builds and deploys software components and its related defects. Implementation activities have the most impact on the daily life of developers, and an important goal of Implementation is to ship reliably working software with minimum defects.

Implementation should include security practices such as :

- Secure Build
- Secure Deployment
- Defect Management

Implementation is where the application / system begins to take shape; source code is written and tests are created. The implementation of the application follows a secure development lifecycle, with security built in from the start.

The implementation will use a secure method of source code control and storage to fulfil the design security requirements. The development team will be referring to documentation advising them of best practices, they will be using secure libraries wherever possible in addition to checking and tracking external dependencies.

Much of the skill of implementation comes from experience, and taking into account the Do's and Don'ts of secure development is an important knowledge activity in itself.

Sections:

5.1 Documentation
5.1.1 Top 10 Proactive Controls
5.1.2 Go Secure Coding Practices
5.1.3 Cheatsheet Series
5.2 Dependencies 5.2.1 Dependency-Check
5.2.2 Dependency-Track
5.2.3 CycloneDX
5.3 Secure Libraries
5.3.1 Enterprise Security API library
5.3.2 CSRFGuard library
5.3.3 OWASP Secure Headers Project
5.4 Implementation Do's and Don'ts
5.4.1 Container security
5.4.2 Secure coding
5.4.3 Cryptographic practices
5.4.4 Application spoofing
5.4.5 Content Security Policy (CSP)
5.4.6 Exception and error handling
5.4.7 File management
5.4.8 Memory management

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

5.1 Documentation

Documentation is used here as part of the SAMM Training and Awareness activity, which in turn is part of the SAMM Education & Guidance security practice within the Governance business function.

It is important that development teams have good documentation on security techniques, frameworks, tools and threats. Documentation helps to promote security awareness for all teams involved in software development, and provides guidance on building security into applications and systems.

Sections:

5.1.1 Top 10 Proactive Controls
5.1.2 Go Secure Coding Practices
5.1.3 Cheatsheet Series

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

5.1.1 Top 10 Proactive Controls

To Do: supply a couple of sentences on the OWASP Top 10 Proactive Controls documentation project, including its status as an OWASP project and where to find it.

What are the Top 10 Proactive Controls? To Do: go into more detail about the Proactive Controls so that a developer can gain an overview of what this documentation project can provide for them.

Why use them? To Do: provide more context for the Proactive Controls that allows developers to determine whether to use them in their project.

How to apply them To Do: give a brief outline of how applying the Top 10 Proactive Controls documentation provides value for a development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 6: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

5.1.2 Go Secure Coding Practices

To Do: supply a couple of sentences on the OWASP Go Secure Coding Practices (SCP) documentation project, including its status as an OWASP project and where to find it.

What is Go SCP? To Do: go into more detail about the Go SCP project so that a developer can gain an overview of what this documentation project can provide for them.

Why use Go SCP? To Do: provide more context for Go SCP that allows developers to determine whether to use it in their project.

How to apply Go SCP To Do: give a brief outline of how applying the Go SCP project provides value for a development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 7: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

5.1.3 Cheat Sheet Series

The OWASP Cheat Sheet Series was created to provide a concise collection of high value information on a wide range of specific application security topics. The cheat sheets are created by various application security professionals who have expertise in specific topics.

The Cheat Sheet Series documentation project is an OWASP Flagship Project in active development.

What is the Cheat Sheet Series? To Do: go into more detail about the Cheat Sheet Series so that a developer can gain an overview of what this documentation project can provide for them.

Why use them? To Do: provide more context for the Cheat Sheet Series that allows developers to determine whether to use them in their project.

How to apply them To Do: give a brief outline of how applying the Cheat Sheet Series documentation provides value for a web development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 8: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

5.2 Dependencies

Management of software dependencies is described by the SAMM Software Dependencies activity, which in turn is part of the SAMM Secure Build security practice within the Implementation business function.

It is important to record all dependencies used throughout the target production environment. This record is often referred to as a Software Bill of Materials (SBOM). An ideal SBOM provides information on each dependency so that it can be tracked:

- Where it is used or referenced
- Version used
- License
- Source information and repository
- Support and maintenance status of the dependency

Having an SBOM provides the ability to quickly find out which applications are affected by a particular CVE, for example.

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

5.2.1 Dependency-Check

To Do: supply a couple of sentences on the OWASP Dependency-Check Software Composition Analysis (SCA) tool, including its status as an OWASP project and where to find it.

What is Dependency-Check? To Do: go into more detail about the Dependency-Check tool so that a developer can gain an overview of what this tool can provide for them.

Why use it? To Do: provide more context for the Dependency-Check tool that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how applying the Dependency-Check tool provides value for a web development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 9: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

5.2.2 Dependency-Track

To Do: supply a couple of sentences on the OWASP Dependency-Track component analysis platform tool and its use of the Software Bill of Materials (SBOM), including its status as an OWASP project and where to find it.

What is Dependency-Track? To Do: go into more detail about the Dependency-Track tool so that a developer can gain an overview of what this tool can provide for them.

Why use it? To Do: provide more context for the Dependency-Track tool and SBOMs that allows developers to determine whether to use them in their project.

How to use it To Do: give a brief outline of how applying the Dependency-Track tool provides value for a web development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 10: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

5.2.3 CycloneDX

To Do: supply a couple of sentences on the OWASP CycloneDX full-stack Bill of Materials (BOM) standard and its use for Software Bill of Materials (SBOMs), including its status as an OWASP project and where to find it.

What is CycloneDX? To Do: go into more detail about the CycloneDX standard so that a developer can gain an overview of what this standard can provide for them.

Why use it? To Do: provide more context for the CycloneDX standard and SBOMs that allows developers to determine whether to use them in their project.

How to use it To Do: give a brief outline of how applying the CycloneDX standard provides value for a web development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 11: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

5.3 Secure libraries

The use of secure libraries is part of the technology management that helps to fulfil security requirements. Standard libraries help with the adoption of common design patterns and security solutions, and help standardize technologies and frameworks used throughout the different applications.

Technology Management for the software applications is described by SAMM as an activity within the SAMM Security Architecture security practice which in turn is part of the Design business function.

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

5.3.1 Enterprise Security API library

The OWASP Enterprise Security API (ESAPI) library is a free, open source, web application security control library that makes it easier for Java programmers to write lower-risk applications. The ESAPI Java library is designed to make it easier for programmers to retrofit security into existing Java applications, and also serve as a solid foundation for new development.

The ESAPI library is an OWASP Lab project that is under active development for Java only.

What is the ESAPI library? To Do: go into more detail about the ESAPI library so that a developer can gain an overview of what this Java library can provide for them.

Why use it? To Do: provide more context for the ESAPI Java library that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how applying the ESAPI library provides valuable application security for a development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 12: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

5.3.2 CSRFGuard library

OWASP CSRFGuard is a library that implements a variant of the synchronizer token pattern to mitigate the risk of Cross-Site Request Forgery (CSRF) attacks for Java applications.

The OWASP CSRFGuard library is integrated through the use of a JavaEE Filter and exposes various automated and manual ways to integrate per-session or pseudo-per-request tokens into HTML. When a user interacts with this HTML, CSRF prevention tokens (i.e. cryptographically random synchronizer tokens) are submitted with the corresponding HTTP request.

CSRFGuard is an OWASP Flagship Project and is being actively maintained by a pool of international volunteers.

What is CSRFGuard? To Do: go into more detail about the CSRFGuard library so that a developer can gain an overview of what this library can provide for them.

Why use it? To Do: provide more context for the CSRFGuard library that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how applying the CSRFGuard library provides increased application security. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 13: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

5.3.3 OWASP Secure Headers Project

The OWASP Secure Headers Project (also called OSHP) describes HTTP response headers that your application can use to increase the security of your application. Once set, these HTTP response headers can restrict modern browsers from running into easily preventable vulnerabilities. The OWASP Secure Headers Project intends to raise awareness and use of these headers.

OSHP is an OWASP Lab Project site contains guidance and downloads on:

- Response Headers
- Browser Support
- Best Practices
- Technical Resources
- Code Snippets

What is OSHP? To Do: go into more detail about OSHP so that a developer / penetration tester can gain an overview of how this can provide an HTTP security header configuration.

Why use it? To Do: provide more context for OSHP that allows developers to determine whether to use it.

How to use it To Do: give a brief outline of how OSHP provides value for a development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 14: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

5.4 Implementation Do's and Don'ts

Implementation demands technical knowledge, skill and experience. There is no substitute for experience, but learning from past mistakes and the experience of others can go a long way. This section of the Developer Guide is a collection of Do's and Don'ts, some of which may be directly relevant to any given project and some of which will be less so. It is worth considering all of these Do's and Don'ts and picking out the ones that will be of most use.

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

5.4.1 Container security

Here is a collection of Do's and Don'ts when it comes to container security, gathered from practical experiences. Some of these are language specific and others have more general applicability.

Container image security, host security, client security, daemon security, runtime security:

- Choose the right base image
- Include only the required packages in the image
- If using Docker images, use multi-stage builds
- Use layer caching and multi stage builds to:
  - Separate build-time dependencies from runtime dependencies
  - Remove special permissions from images
  - `find / -perm /6000 -type f -exec ls -ld {} \;`
  - RUN `find / -xdev -perm /6000 -type f -exec chmod a-s {} \; || true`
- Reduce overall image size by shipping only what your app needs to run
- Please see the Docker documentation for more information
- Remove unused images with prune
- docker image prune [OPTIONS]
- Do not embed any secrets, passwords, keys, credentials, etc in images
- Use a read-only file system
- Sign images with cryptographic keys and not with username/password combination
- Secure your code and its dependencies
- Test your images for vulnerabilities
- Monitor container runtimes
- Docker Content Trust (DCT) is enabled on Docker clients
- Check freshness security of images with the provided timestamp key that is associated with the registry.
- Create the timestamp key by Docker and store on the server
- Use tagging keys associated with a registry. Such that a poisoned image from a different registry cannot be pushed into a registry.
- Use offline keys to sign the tagging keys.
- Offline keys are owned by the organisation and secured in an out-of-band location.
- Scan images frequently for any vulnerabilities. Rebuilt all images to include patches and instantiate new containers from them
- Remove `setuid` and `setgid` permissions from the images.
- Where applicable, use 'copy' instruction in place of 'add' instruction.
- Verify authenticity of packages before installing them into images
- Use namespaces and control groups for containers
- Use bridge interfaces for the host
- Authenticity of packages is verified before installing them into images
- Mount files on a separate partition to address any situation where the mount becomes full, but the host still remains usable
- Mark registries as private and only use signed images.
- Pass commands through the authorization plugin to ensure that only authorised client connects to the daemon
- TLS authentication is configured to restrict access to the Docker daemon
- Namespaces are enabled to ensure that
- Leave control groups (cgroups) at default setting to ensure that tampering does not take place with excessive resource consumption.
- Do not enable experimental features for Docker.
- set docker.service file ownership to root:root.
- Set docker.service file permissions to either 644 or to a more restrictive value.
- Set docker.socket file ownership and group ownership to root.
- Set file permissions on the docker.socket file to 644 or more restrictively.
- Set /etc/docker directory ownership and group ownership to root.
- Set /etc/docker directory permissions to 755 or more restrictively.
- Set ownership of registry certificate files (usually found under `/etc/docker/certs.d/<registry-name>`

directory) to individual ownership and is group owned by root.
- Set registry certificate files (usually found under `/etc/docker/certs.d/<registry-name>` directory) permissions to 444 or more restrictively.
- Acquire and ship daemon logs to SIEM for monitoring
- Inter-container network connections are restricted and enabled on a requirement basis. By default containers cannot capture packets that have other containers as destination.
- Where hairpin NAT is enabled, userland proxy is disabled.
- Docker daemon is run as a non-root user to mitigate lateral privilege escalation due to any possible compromise of vulnerabilities.
- `No_new_priv` is set (but not to false) to ensure that containers cannot gain additional privileges via `suid` or `sgid`
- Default SECCOMP profile is applied for access control.
- TLS CA certificate file on the image host (the file that is passed along with the `--tlscacert` parameter) is individually owned and group owned by root
- TLS CA certificate file on the image host (the file that is passed along with the `--tlscacert` parameter) has permissions of 444 or is set more restrictively
- Containers should run as a non-root user.
- Containers should have as small a footprint as possible, and should not contain unnecessary software packages which could increase their attack surface
- Docker default bridge 'docker0' is not used to avoid ARP spoofing and MAC flooding attacks.
- Either Dockers AppArmor policy is enabled or the Docker hosts AppArmor is enabled.
- SELinux policy is enabled on the Docker host.
- Linux kernel capabilities are restricted within containers
- privileged containers are not used
- sensitive host system directories are not mounted on containers
- `sshd` is not run within containers
- privileged ports are not mapped within containers (TCP/IP port numbers below 1024 are considered privileged ports)
- only needed ports are open on the container.
- the hosts network namespace is not shared.
- containers root filesystem is mounted as read only
- Do not use docker exec with the `--privileged` option.
- docker exec commands are not used with the user=root option
- cgroup usage is confirmed
- The `no_new_priv` option prevents LSMs like SELinux from allowing processes to acquire new privileges
- Docker socket is not mounted inside any containers to prevent processes running within the container to execute Docker commands which would effectively allow for full control of the host.
- incoming container traffic is bound to a specific host interface
- hosts process namespace is not shared to ensure that processes are separated.
- hosts IPC namespace is not shared to ensure that inter-process communications does not take place.
- host devices are not directly exposed to containers
- hosts user namespaces are not shared to ensure isolation of containers.
- CPU priority is set appropriately on containers
- memory usage for containers is limited.
- 'on-failure' container restart policy is set to '5'
- default `ulimit` is overwritten at runtime if needed
- container health is checked at runtime
- PIDs cgroup limit is used (limit is set as applicable)
- The Docker host is hardened to ensure that only Docker services are run on the host
- Secure configurations are applied to ensure that the containers do not gain access to the host via the Docker daemon.
- Docker is updated with the latest patches such that vulnerabilities are not compromised.
- The underlying host is managed to ensure that vulnerabilities are identified and mitigated with patches.
- Docker server certificate file (the file that is passed along with the `--tlscert` parameter) is individual

owned and group owned by root.
- Docker server certificate file (the file that is passed along with the `--tlscert` parameter) has permissions of 444 or more restrictive permissions.
- Docker server certificate key file (the file that is passed along with the `--tlskey` parameter) is individually owned and group owned by root.
- Docker server certificate key file (the file that is passed along with the `--tlskey` parameter) has permissions of 400.
- Docker socket file is owned by root and group owned by docker.
- Docker socket file has permissions of 660 or are configured more restrictively.
- `daemon.json` file individual ownership and group ownership is correctly set to root, if it is in use.
- daemon.json file is present its file permissions are correctly set to 644 or more restrictively.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

5.4.2 Secure coding

Here is a collection of Do's and Don'ts when it comes to secure coding, gathered from practical experiences. Some of these are language specific and others have more general applicability.

- Authentication
  - User
    - Require authentication for all pages and resources, except those specifically intended to be public
    - Perform all authentication on server side. Send credentials only on encrypted channel (HTTPS)
    - Use a centralised implementation for all authentication controls, including libraries that call external authentication services. Use security vetted libraries for federation (Okta / PING / etc). If using third party code for authentication, inspect the code carefully to ensure it is not affected by any malicious code
    - Segregate authentication logic from the resource being requested and use redirection to and from the centralised authentication control
    - Validate the authentication data only on completion of all data input, especially for sequential authentication implementations
    - Authentication failure responses should not indicate which part of the authentication data was incorrect. For example, instead of "Invalid username" or "Invalid password", just use "Invalid username and/or password" for both. Error responses must be truly identical in both display and source code
    - Utilise authentication for connections to external systems that involve sensitive information or functions
    - Authentication credentials for accessing services external to the application should be encrypted and stored in a protected location on a trusted system (e.g., Secrets Manager). The source code is NOT a secure location.
    - Do not store passwords in code or in configuration files. Use Secrets Manager to store passwords
    - Use only HTTP POST requests to transmit authentication credentials
    - Implement monitoring to identify attacks against multiple user accounts, utilising the same password. This attack pattern is used to bypass standard lockouts, when user IDs can be harvested or guessed
    - Re-authenticate users prior to performing critical operations
    - Use Multi-Factor Authentication for highly sensitive or high value transactional accounts
    - If using third party code for authentication, inspect the code carefully to ensure it is not affected by any malicious code
    - Restrict the user if a pre-defined number of failed logon attempts exceed. Restrict access to a limited number of attempts to prevent brute force attacks
    - Partition the portal into restricted and public access areas
    - Restrict authentication cookies to HTTPS connections
    - If the application has any design to persist passwords in the database, hash and salt the password before storing in database. Compare hashes to validate password
    - Authenticate the user before authorising access to hidden directories
    - Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
  - Server
    - When using SSL/TLS, ensure that the server identity is established by following a trust chain to a known root certificate
    - When using SSL/TLS, validate the host information of the server certificate.
    - If weak client authentication is unavoidable, perform it only over a secure channel
    - Do not rely upon IP numbers or DNS names in establishing identity.
    - Ensure all internal and external connections (user and entity) go through an appropriate and adequate form of authentication. Be assured that this control cannot be bypassed.
    - For the account that runs the web server:
      - Grant permissions to only those folders that the application needs to access

            - Grant only those privileges that the account needs
        - Disable HTTP TRACE. It can help in bypassing WAF because of it inherent nature of TRACE response includes all headers on its route. Please see - Three Minutes with the HTTP TRACE Method - for further details
        - Disable WEBDav feature unless it is required for business reasons. If it is, perform a risk assessment for enabling the feature on your environment.
        - Ensure that authentication credentials are sent on an encrypted channel
        - Ensure development/debug backdoors are not present in production code.
    - Password policy
        - Provide a mechanism for self-reset and do not allow for third-party reset.
        - If the application has any design to persist passwords in the database, hash and salt the password before storing in database. Compare hashes to validate password.
        - Rate limit bad password guesses to a fixed number(5) in a given time period (5-minute period)
        - Provide a mechanism for users to check the quality of passwords when they set or change it.
        - Only send non-temporary passwords over an encrypted connection or as encrypted data, such as in an encrypted email. Temporary passwords associated with email resets may be an exception
        - Enforce password complexity requirements established by policy or regulation. Authentication credentials should be sufficient to withstand attacks that are typical of the threats in the deployed environment. (e.g., requiring the use of alphabetic as well as numeric and/or special characters)
        - Enforce password length requirements established by policy or regulation. Eight characters is commonly used, but 16 is better or consider the use of multi-word pass phrases
        - Password entry should be obscured on the user's screen. (e.g., on web forms use the input type "password")
        - Enforce account disabling after an established number of invalid login attempts (e.g., five attempts is common). The account must be disabled for a period of time sufficient to discourage brute force guessing of credentials, but not so long as to allow for a denial-of-service attack to be performed
        - Password reset and changing operations require the same level of controls as account creation and authentication.
        - Password reset questions should support sufficiently random answers. (e.g., "favorite book" is a bad question because "The Bible" is a very common answer)
        - If using email based resets, only send email to a pre-registered address with a temporary link/password
        - Temporary passwords and links should have a short expiration time
        - Enforce the changing of temporary passwords on the next use
        - Notify users when a password reset occurs
        - Prevent password re-use
        - For high risk application (for example banking applications or any application the compromise of credentials of which may lead to identity theft), passwords should be at least one day old before they can be changed, to prevent attacks on password re-use
        - Enforce password changes based on requirements established in policy or regulation. Critical systems may require more frequent changes. The time between resets must be administratively controlled
        - Disable "remember me" functionality for password fields
        - Avoid sending authentication information through E-mail, particularly for existing users.
- Authorisation
    - Access control
        - Build authorisation on rules based access control. Persist the rules as a matrix (for example as a list of strings which is passed as a parameter to a method that is run when the user first access the page, based on which access is granted). Most frameworks today, support this kind of matrix.
        - Check if the user is authenticated before checking the access matrix. If the user is not authenticated, direct the user to the login page. Alternatively, use a single site-

wide component to check access authorisation. This includes libraries that call external authorisation services
    * Ensure that the application has clearly defined the user types and the privileges for the users.
    * Ensure there is a least privilege stance in operation. Add users to groups and assign privileges to groups
    * Scan the code for development/debug backdoors before deploying the code to production.
    * Re-Authenticate the user before authorising the user to perform business critical activities
    * Re-Authenticate the user before authorising the user to admin section of the application
    * Do not include authorisation in the query string. Direct the user to the page via a hyperlink on a page. Authenticate the user before granting access. For example if `admin.php` is the admin page for `www.example.com` do not create a query string like `www.example.com/admin.php`. Instead include a hyperlink to `admin.php` on a page and control authorisation to the page
    * Prevent forced browsing with role based access control matrix
    * Ensure Lookup IDs are not accessible even when guessed and lookup IDs cannot be tampered with
    * Enforce authorisation controls on every request, including those made by server side scripts, "includes" and requests from rich client-side technologies like AJAX and Flash
    * Server side implementation and presentation layer representations of access control rules must match
    * Implement access controls for POST, PUT and DELETE especially when building an API
    * Use the "referer" header as a supplemental check only, it should never be the sole authorisation check, as it is can be spoofed
    * Ensure it is not possible to access sensitive URLs without proper authorisation. Resources like images, videos should not be accessed directly by simply specifying the correct path
    * Test all URLs on administrator pages to ensure that authorisation requirements are met. If verbs are sent cross domain, pin the OPTIONS request for non-GET verbs to the IP address of subsequent requests. This will be a first step toward mitigating DNS Rebinding and TOCTOU attacks.
- Session management
    * Creation of session: Use the server or framework's session management controls. The application should only recognise these session identifiers as valid
    * Creation of session: Session identifier creation must always be done on a trusted system (e.g., The server)
    * Creation of session: If a session was established before login, close that session and establish a new session after a successful login
    * Creation of session: Generate a new session identifier on any re-authentication
    * Random number generation: Session management controls should use well vetted algorithms that ensure sufficiently random session identifiers. Rely on CSPRNG rather than PRNG for random number generation
    * Domain and path: Set the domain and path for cookies containing authenticated session identifiers to an appropriately restricted value for the site
    * Logout: Logout functionality should fully terminate the associated session or connection
    * Session timeout: Establish a session inactivity timeout that is as short as possible, based on balancing risk and business functional requirements. In most cases it should be no more than several hours
    * Session ID: Do not expose session identifiers in URLs, error messages or logs. Session identifiers should only be located in the HTTP cookie header. For example, do not pass session identifiers as GET parameters
    * Session ID: Supplement standard session management for sensitive server-side operations, like account management, by utilising per-session strong random tokens or parameters. This method can be used to prevent Cross Site Request Forgery attacks
- JWT
    * Reject tokens set with 'none' algorithm when a private key was used to issue them (`alg: ""none""`). This is because an attacker may modify the token and hashing algorithm to

indicate, through the 'none' keyword, that the integrity of the token has already been verified, fooling the server into accepting it as a valid token
    * Use appropriate key length (e.g. 256 bit) to protect against brute force attacks. This is because attackers may change the algorithm from 'RS256' to 'HS256' and use the public key to generate a HMAC signature for the token, as server trusts the data inside the header of a JWT and doesn't validate the algorithm it used to issue a token. The server will now treat this token as one generated with 'HS256' algorithm and use its public key to decode and verify it
    * Adjust the JWT token validation time depending on required security level (e.g. from few minutes up to an hour). For extra security, consider using reference tokens if there's a need to be able to revoke/invalidate them
    * Use HTTPS/SSL to ensure JWTs are encrypted during client-server communication, reducing the risk of the man-in-the-middle attack. This is because sensitive information may be revealed, as all the information inside the JWT payload is stored in plain text
    * Only use up-to-date and secure libraries and choose the right algorithm for requirements
    * Verify all tokens before processing the payload data. Do not use unsigned tokens. For the tokens generated and consumed by the portal, sign and verify tokens
    * Always check that the `aud` field of the JWT matches the expected value, usually the domain or the URL of your APIs. If possible, check the "sub" (client ID) - make sure that this is a known client. This may not be feasible however in a public API situation (e.g., we trust all clients authorised by Google).
    * Validate the issuer's URL (`iss`) of the token. It must match your authorisation server.
    * If an authorisation server provides X509 certificates as part of its JWT, validate the public key using a regular PKIX mechanism
    * Make sure that the keys are frequently refreshed/rotated by the authorisation server.
    * Make sure that the algorithms you use are sanctioned by JWA (RFC7518)
    * There is no built in mechanism to revoke a token manually, before it expires. One way to ensure that the token is force expired build a service that can be called on log out. In the mentioned service, block the token.
    * Restrict accepted algorithms to the ONE you want to use
    * Restrict URLs of any JWKS/X509 certificates
    * Use the strongest signing process you can afford the CPU time for
    * Use asymmetric keys if the tokens are used across more than one server
  - SAML
- Input data validation
  - Identify input fields that form a SQL query. Check that these fields are suitably validated for type, format, length, and range.
  - To prevent SQL injection use bind variables in stored procedures and SQL statements. Also referred as prepared statements / parameterization of SQL statements. DO NOT concatenate strings that are an input to the database. The key is to ensure that raw input from end users is not accepted without sanitization. When converting data into a data structure (deserializing), perform explicit validation for all fields, ensuring that the entire object is semantically valid. Many technologies now come with data access layers that support input data validation. These layers are usually in the form of a library or a package. Ensure to add these libraries / dependencies / packages to the project file such that they are not missed out.
  - Use a security vetted library for input data validation. Try not to use hard coded allow-list of characters. Validate all data from a centralised function / routine. In order to add a variable to a HTML context safely, use HTML entity encoding for that variable as you add it to a web template.
    * Validate HTTP headers. Dependencies that perform HTTP headers validation are available in technologies.
    * Validate post backs from javascript.
    * Validate data from http headers, input fields, hidden fields, drop down lists & other web components
    * Validate data retrieved from database. This will help mitigate persistent XSS.
    * Validate all redirects. Unvalidated redirects may lead to data / credential exfiltration.

Evaluate any URL encodings before trying to use the URL.
    - Validate data received from redirects. The received data may be from untrusted source.
    - If any potentially hazardous characters must be allowed as input, be sure that you implement additional controls like output encoding, secure task specific APIs and accounting for the utilization of that data throughout the application. Examples of common hazardous characters include `< > " ' % ( ) & + \ \' \"`
    - If your standard validation routine cannot address the following inputs, then they should be checked discretely
        - Check for null bytes `%00`
        - Check for new line characters `%0d, %0a, \r, \n`
        - Check for "dot-dot-slash" `../` or `..\` path alterations characters. In cases where UTF-8 extended character set encoding is supported, address alternate representation like: `%c0%ae%c0%ae/` (Utilise canonicalization to address double encoding or other forms of obfuscation attacks)
    - Client-side storage (`localStorage`, `SessionStorage`, `IndexedDB`, WebSQL): If you use client-side storage for persistence of any variables, validate the date before consuming it in the application
    - Reject all input data that has failed validation.
    - If used, don't involve user parameters in calculating the destination. This can usually be done. If destination parameters can't be avoided, ensure that the supplied value is valid, and authorised for the user. It is recommended that any such destination parameters be a mapping value, rather than the actual URL or portion of the URL, and that server side code translate this mapping to the target URL. Applications can use ESAPI to override the `sendRedirect()` method to make sure all redirect destinations are safe.
- Output data encoding
    - If your code echos user input or URL parameters back to a Web page, validate input data as well as output data. It will help you prevent persistent as well as reflective cross-site scripting. Pay particular attention to areas of the application that permit users to modify configuration or personalization settings. Also pay attention to persistent free-form user input, such as message boards, forums, discussions, and Web postings. Encode javascript to prevent injection by escaping non-alphanumeric characters. Use quotation marks like " or ' to surround your variables. Quoting makes it difficult to change the context a variable operates in, which helps prevent XSS
    - Conduct all encoding on a trusted system (e.g., The server) Utilise a standard, tested routine for each type of outbound encoding Contextually output encode all data returned to the client that originated outside the application's trust boundary. HTML entity encoding is one example, but does not work in all cases Encode all characters unless they are known to be safe for the intended interpreter Contextually sanitise all output of untrusted data to queries for SQL, XML, and LDAP Sanitise all output of untrusted data to operating system commands
    - Output encoding is not always perfect. It will not always prevent XSS. Some contexts are not secure. These include: Callback functions Where URLs are handled in code such as this CSS `{ background-url : "javascript:alert(test)"; }` All JavaScript event handlers (`onclick()`, `onerror()`, `onmouseover()`). Unsafe JavaScript functions like `eval()`, `setInterval()`, `setTimeout()` Don't place variables into these contexts as even with output encoding, it will not prevent an XSS attack fully
    - Do not rely on client-side validation. Perform validation on server side to prevent second order attacks.
- Canonicalisation
    - Convert all input data to an accepted/decided format like UTF-8. This will help prevent spoofing of character
- Test all URLs with different parameter values. Spider and check the site/product/application/portal for redirects.
- Connection with backend Assign required permissions and privileges for accounts / roles used by the application to connect to the database. In the event of any compromise of the account / role, the malicious actor would be able to do whatever the account /role has permissions for.
- Insecure direct object references

- Unvalidated redirects Test all URLs with different parameter values to validate any redirects If used, do not allow the URL as user input for the destination. Where possible, have the user provide short name, ID or token which is mapped server-side to a full target URL. This provides the protection against the URL tampering attack. Be careful that this doesn't introduce an enumeration vulnerability where a user could cycle through IDs to find all possible redirect targets If user input can't be avoided, ensure that the supplied value is valid, appropriate for the application, and is authorised for the user. Sanitise input by creating a list of trusted URLs (lists of hosts or a regex). This should be based on an allow-list approach, rather than a block list. Force all redirects to first go through a page notifying users that they are going off of your site, with the destination clearly displayed, and have them click a link to confirm.
- JSON For JSON, verify that the Content-Type header is application/json and not text/html to prevent XSS Do not use duplicate keys. Usage of duplicate keys may be processed differently by parsers. For example last-key precedence versus first-key precedence.
- Generate fatal parse errors on duplicate keys. Do not perform character truncation. Instead, replace invalid Unicode with placeholder characters (e.g., unpaired surrogates should be displayed as the Unicode replacement character `U+FFFD`). Truncating may break sanitization routines for multi-parser applications."
- Produce errors when handling integers or floating-point numbers that cannot be represented faithfully
- Do not use `eval()` with JSON. This opens up for JSON injection attacks. Use JSON.parse() instead Data from an untrusted source is not sanitised by the server and written directly to a JSON stream. This is referred to as server-side JSON injection. Data from an untrusted source is not sanitised and parsed directly using the JavaScript `eval` function. This is referred to as client-side JSON injection. To prevent server-side JSON injections, sanitise all data before serialising it to JSON Escape characters like ":", "",","@","'","%","?","–",">","<","&"

JSON Vulnerability Protection

A JSON vulnerability allows third party website to turn your JSON resource URL into JSONP request under some conditions. To counter this your server can prefix all JSON requests with following string `")]}',\n"`. AngularJS will automatically strip the prefix before processing it as JSON.

For example if your server needs to return: `['one','two']` which is vulnerable to attack, your server can return: `)]}', ['one','two']`

Refer to JSON vulnerability protection Always have the outside primitive be an object for JSON strings

Exploitable: `[{""object"": ""inside an array""}]`

Not exploitable: `{""object"": ""not inside an array""}`

Also not exploitable: `{""result"": [{""object"": ""inside an array""}]}"`

- Avoid manual build of JSON, use an existing framework
- Ensure calling function does not convert JSON into a javascript and JSON returns its response as a non-array json object
- Wrap JSON in () to force the interpreter to think of it as JSON and not a code block
- When using node.js, on the server use a proper JSON serializer to encode user-supplied data properly to prevent the execution of user-supplied input on the browser.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

5.4.3 Cryptographic practices

Here is a collection of Do's and Don'ts when it comes to cryptographic practices, gathered from practical experiences.

- The basis for usage of PKI is to address (using encryption and hashing)
- Confidentiality
- Integrity
- Authentication
- Non-repudiation
- Cryptography is used for the following:
  - Data-at-rest protection using data encrypting keys and key encrypting keys. For which,
- Do not use custom cryptographic algorithms / deprecated algorithms
- Do not use passwords as cryptographic keys
- Do not hard-code cryptographic keys in the application
- Persist secret keys in a secure vault like HSM, KMS, Secrets Manager
- Manage encryption keys through the lifecycle, including key retirement/replacement when someone who has access leaves the organisation
- Rotate keys on a regular basis. However this depends on the key strength and the algorithm used. If the key strength is low, the rotation period will be smaller
- Maintain a contingency plan to recover data in the event of an encrypted key being lost
- Ensure the code eliminates secrets from memory.
- Maintain a contingency plan that can recover data in the event of an encrypted key being lost
- Store keys away from the data
- Do not use IV twice for a fixed key
- Communication security
- Ensure no sensitive data is transmitted in the clear, internally or externally.
- Validate certificates properly against the hostnames/users for whom they are meant
- Failed TLS connections should not fall back to an insecure connection
- Do not use IV twice for a fixed key
- Cryptography in general
- All protocols and algorithms for authentication and secure communication should be well vetted by the cryptographic community.
- Perform Message integrity checking by using a combined mode of operation, or a MAC based on a block cipher.
- Do not use key sizes less than 128 bits or cryptographic hash functions with output sizes less than 160 bits.
- Do not use custom cryptographic algorithms that have not been vetted by cryptography community
- Do not hardcode cryptographic keys in applications?
- Issue keys using a secure means.
- Maintain a key lifecycle for the organisation (Creation, Storage, Distribution and installation, Use, Rotation, Backup, Recovery, Revocation, Suspension, Destruction)
- Lock and unlock symmetric secret keys securely
- Maintain CRL (Certificate Revocation Lists) maintained on a real-time basis
- Validate certificates properly against the hostnames/users for whom they are meant
- Ensure the code eliminates secrets from memory
- Specific encryption, in addition to SSL
- Mask or remove keys from logs
- Use salted hashes when using MD5 or any such less secure algorithms
- Use known encryption algorithms, do not create your own as they will almost certainly be less secure
- Persist secret keys in a secure vault like HSM, KMS, Secrets Manager
- Do not use IV twice for a fixed key
- Ensure that cryptographic randomness is used where appropriate, and that it has not been seeded in a predictable way or with low entropy. Most modern APIs do not require the developer to seed the CSPRNG to get security.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

5.4.4 Application Spoofing

Here is a collection of Do's and Don'ts when it comes to application spoofing, gathered from practical experiences. Some of these are language specific and others have more general applicability.

What is application spoofing:

- A threat actor including an application in a malicious iFrame
- A threat actor creating dependencies with similar names as legitimate ones (typo squatting)

How can it be addressed:

Application spoofing / clickjacking Set `X-FRAME-OPTIONS` header to `SAMEORIGIN` or `DENY`, depending on what the business requirement is for rendering the web page. This will help prevent a malicious actor including your application in an iFrame to capture credentials/exfiltrate data. As a caveat, this will not work with Meta Tags. X-FRAME-OPTIONS must be applied as HTTP Response Header

Use Content Security Policy:

Common uses of CSP frame-ancestors:

Content-Security-Policy: frame-ancestors 'none';

This prevents any domain from framing the content. This setting is recommended unless a specific need has been identified for framing.

Content-Security-Policy: frame-ancestors 'self';

This only allows the current site to frame the content.

Content-Security-Policy: frame-ancestors 'self' `*.somesite.com https://myfriend.site.com;`

This allows the current site, as well as any page on `somesite.com` (using any protocol), and only the page `myfriend.site.com`, using HTTPS only on the default port (443).

Use `SameSite` Cookies

Use `httpOnly` cookies

Domain squatting / typo squatting What is domain squatting (also known as cybersquatting):

- A threat actor creating a malicious domain with the same spelling as a legitimate domain but use different UTF characters (domain squatting)
- A threat actor registering, trafficking in, or using an Internet domain name, with an intent to profit from the goodwill of a trademark belonging to someone else.
- Though domain squatting impacts brand value directly, it has an impact from a security perspective.
- It can result in the following kind of scenario: (also known as typosquatting)

Wherein the domain with U+00ED may be a malicious application trying to harvest credentials.

- Typo squatting is achieved with supply chain manipulation.

How can it be addressed:

- Use threat intelligence to monitor lookalikes for your domain
- In the event a dispute needs to be raised, it can be done with URDP
- Verify packages in registries before using them

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

5.4.5 Content Security Policy

Here is a collection of Do's and Don'ts when it comes to Content Security Policy, gathered from practical experiences. Some of these are language specific and others have more general applicability.

Content Security Policy (CSP) helps in allow-listing the sources that are allowed to be executed by clients.

To this effect CSP helps in addressing vulnerabilities that are the target of scripts getting executed from different domains (namely XSS, ClickJacking)

1. The policy elements listed below is restrictive. Third party libraries can be allow-listed as a part of `script-src`, `default-src`, `frame-src` or `frame-ancestors`.
2. I assume fonts / images / media / plugins are not loaded from any external sources.
3. Do not use `\*\` as an attribute for any of the components of the policy.

CSP considers two types of content:

Passive content - resources which cannot directly interact with or modify other resources on a page: images, fonts, audio, and video for example

Active content - content which can in some way directly manipulate the resource with which a user is interacting.

SCOPE

The scope of this policy / procedure / whatever includes (but not limited to):

- Applications that are displayed in browsers
  - On desktops
  - On laptops
  - On mobile devices
- Mobile Applications
  - iOS
  - Android

Policy for content security should be set in «add SSDLC Policy / Secure Coding Policy / any others that is applicable. Unless otherwise specified by the customer, third party sources should not be allowed to connect from the deployed solutions

Web Applications For web applications, the source of all content is set to self.

- `default-src` 'self'
- `script-src` 'self';
- `script-src` `unsafe-inline` `unsafe-eval` https:; (I am fairly sure this is used to block unsafe inline scripts and `eval` but to be checked) - Have checked now and `unsafe-inline` should not be used
- `connect-src` 'self';
- `img-src` 'self';
- `style-src` 'self'
- `style-src` 'unsafe-inline' should not be used
- `font-src` 'self';
- `frame-src` https:;
- `frame-ancestors` 'none' (This is to prevent ClickJacking equivalent to X-FRAME-OPTIONS = SAME-ORIGIN)
- `frame-ancestors` 'self' (This is to prevent ClickJacking equivalent to X-FRAME-OPTIONS = SAME-ORIGIN)
- `frame-ancestors` `example.com` (This component allows content to be loaded only from `example.com`)
- `media-src` 'self':;
- `object-src` 'self:;
- `report-uri` «» (insert the URL where the report for policy violations should be sent)

- sandbox (this is something to be tried out specifies an HTML sandbox policy that the user agent applies to the protected resource)
- `plugin-types` «» (insert the list of plugins that the protected resource can invoke)
- `base-uri` (restricts the URLs that can be used to specify the document base URL, but I do not know how this is used)
- `child-src` 'self'

An Example:

```
<add name="Content-Security-Policy" value="script-src *.google-analytics.com maps.googleapis.com apis
" script-src 'self' font-src 'self' frame-ancestors 'toyota.co.uk' object-src 'self' />
```

For display on desktops and laptops: add `name="Content-Security-Policy"` value

For display on other mobile devices that use HTML5: `meta http-equiv="Content-Security-Policy"`

Mobile Application

iOS iOS framework has capability to restrict connecting to sites that are not a part of the allow-list on the application, which is the `NSExceptionDomains`. Use this setting to restrict the content that gets executed by the application

```
NSAppTransportSecurity : Dictionary {
    NSAllowsArbitraryLoads : Boolean
    NSAllowsArbitraryLoadsForMedia : Boolean
    NSAllowsArbitraryLoadsInWebContent : Boolean
    NSAllowsLocalNetworking : Boolean
    NSExceptionDomains : Dictionary {
        <domain-name-string> : Dictionary {
            NSIncludesSubdomains : Boolean
            NSExceptionAllowsInsecureHTTPLoads : Boolean
            NSExceptionMinimumTLSVersion : String
            NSExceptionRequiresForwardSecrecy : Boolean
            NSRequiresCertificateTransparency : Boolean
        }
    }
}
```

Android Setting rules for Android application:

- If your application doesn't directly use JavaScript within a WebView, do not call `setJavaScriptEnabled()`
- By default, WebView does not execute JavaScript, so cross-site-scripting is not possible
- Use `addJavaScriptInterface()` with particular care because it allows JavaScript to invoke operations that are normally reserved for Android applications. If you use it, expose `addJavaScriptInterface()` only to web pages from which all input is trustworthy
- Expose `addJavaScriptInterface()` only to JavaScript that is contained within your application APK
- When sharing data between two apps that you control or own, use signature-based permissions

```
<manifest xmlns:android=<link to android schemas ...>
    package="com.example.myapp">
    <permission android:name="my_custom_permission_name"
                android:protectionLevel="signature" />
```

- Disallow other apps from accessing Content Provider objects

```
<manifest xmlns:android=<link to android schemas ...>
    package="com.example.myapp">
    <application ... >
        <provider
            android:name="android.support.v4.content.FileProvider"
```

```
            android:authorities="com.example.myapp.fileprovider"
            ...
            android:exported="false">
            <!-- Place child elements of <provider> here. -->
        </provider>
        ...
    </application>
</manifest>
```

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

5.4.6 Exception and Error Handling

Here is a collection of Do's and Don'ts when it comes to exception and error handling, gathered from practical experiences. Some of these are language specific and others have more general applicability.

- Ensure that all method/function calls that return a value have proper error handling and return value checking.
- Ensure that exceptions and error conditions are properly handled.
- Ensure that no system errors can be returned to the user.
- Ensure that the application fails in a secure manner.
- Ensure resources are released if an error occurs.
- Ensure that stack trace is not thrown to the user.
- Swallowing exceptions into an empty catch() block is not advised as an audit trail of the cause of the exception would be incomplete.
- Code that might throw exceptions should be in a try block and code that handles exceptions in a catch block.
- If the language in question has a finally method, use it. The finally method is guaranteed to always be called.
- The finally method can be used to release resources referenced by the method that threw the exception.
- This is very important. An example would be if a method gained a database connection from a pool of connections, and an exception occurred without finally, the connection object shall not be returned to the pool for some time (until the timeout).
- This can lead to pool exhaustion. finally() is called even if no exception is thrown.
- Handle errors and exception conditions in the code
- Do not expose sensitive information in user sessions
- When working with a multi-threaded or otherwise asynchronous environment, ensure that proper locking APIs are used to lock before the if statement; and unlock when it has finished.
- Types of errors:
  - The result of business logic conditions not being met.
  - The result of the environment wherein the business logic resides fails.
  - The result of upstream or downstream systems upon which the application depends fail.
  - Technical hardware / physical failure.
- Failures are never expected, but they do occur.

In the event of a failure, it is important not to leave the "doors" of the application open and the keys to other "rooms" within the application sitting on the table. In the course of a logical workflow, which is designed based upon requirements, errors may occur which can be programmatically handled, such as a connection pool not being available, or a downstream server not being contactable.

- This is a very tricky guideline.

To fail securely, areas of failure should be examined during the course of the code review. It should be examined if all resources should be released in the case of a failure and during the thread of execution if there is any potential for resource leakage, resources being memory, connection pools, file handles etc. Include a statement that defaults to safe failure

- The review of code should also include pinpointing areas where the user session should be terminated or invalidated. Sometimes errors may occur which do not make any logical sense from a business logic perspective or a technical standpoint;

e.g: ""A logged in user looking to access an account which is not registered to that user and such data could not be inputted in the normal fashion.""

- Examine the application for 'main()' executable functions and debug harnesses/backdoors. In their basic form, backdoors are user id / password combination with the required privileges, embedded in the code, which can be used later on by the developer to get into the system without having to request for login credentials.
- Search for commented out code, commented out test code, which may contain sensitive information.
- Search for any calls to the underlying operating system or file open calls and examine the error possibilities.

Logging

- Ensure that no sensitive information is logged in the event of an error.
- Ensure the payload being logged is of a defined maximum length and that the logging mechanism enforces that length.
- Ensure no sensitive data can be logged; E.g. cookies, HTTP GET method, authentication credentials.
- Examine if the application will audit the actions being taken by the application on behalf of the client (particularly data manipulation/Create, Read, Update, Delete (CRUD) operations).
- Ensure successful and unsuccessful authentication is logged.
- Ensure application errors are logged.
- Examine the application for debug logging with the view to logging of sensitive data.
- Ensure change in sensitive configuration information is logged along with user who modified it. Ensure access to secure storage areas including crypto keys are logged.
- Credentials and sensitive user data should not be logged
- Does the code include poor logging practice of not declaring Logger object as static and final?
- Does the code allow entering invalidated user input to the log file?
- Capture following details for the events:
  - User identification.
  - Type of event.
  - Date and time.
  - Success and failure indication.
  - Origination of event.
  - Identity or name of affected data, system component, resource, or service (for example, name and protocol).
- Log file access, privilege elevation, and failures of financial transactions.
- Log all administrators actions. Log all actions taken after privileges are elevated - `runas` / `sudo`
- Log all input validation failures
- Log all authentication attempts, especially failures
- Log all access control failures
- Log all apparent tampering events, including unexpected changes to state data
- Log attempts to connect with invalid or expired session tokens
- Log all system exceptions
- Log all administrative functions, including changes to the security configuration settings
- Log all backend TLS connection failures

- Log cryptographic module failures
- Use a cryptographic hash function to validate log entry integrity

---

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

5.4.7 File Management

Here is a collection of Do's and Don'ts when it comes to file management, gathered from practical experiences.

- Validate all filenames and directories before use, ensuring that there are no special characters that might lead to accessing an unintended file.
- Use safe directories for all file access except those initiated by the end user e.g. document saving and restoring to a user-chosen location.
- Use a sub-domain with one way trust for the downloaded files. Such that any compromise of the sub-domain does not impact the main domain. Do not save files in the same web context as the application. Files should either go to the content server or in the database.
- Have at least 64 bits of randomness in all temporary file names.
- where applicable, require authentication before allowing a file to be uploaded
- Limit the type of files that can be uploaded to only those types that are needed for business purposes
- Validate uploaded files are the expected type by checking file headers.
- Prevent or restrict the uploading of any file that may be interpreted by the web server.
- Turn off execution privileges on file upload directories
- Implement safe uploading in UNIX by mounting the targeted file directory as a logical drive using the associated path or the chrooted environment
- When referencing existing files, use an allow list of allowed file names and types. Validate the value of the parameter being passed and if it does not match one of the expected values, either reject it or use a hard coded default file value for the content instead
- Do not pass user supplied data into a dynamic redirect. If this must be allowed, then the redirect should accept only validated, relative path URLs
- Do not pass directory or file paths, use index values mapped to pre-defined list of paths
- Never send the absolute file path to the client
- Ensure application files and resources are read-only
- Scan user uploaded files for viruses and malware

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

5.4.8 Memory Management

Here is a collection of Do's and Don'ts when it comes to memory management, gathered from practical experiences.

- Check that the buffer is as large as specified
- When using functions that accept a number of bytes to copy, such as `strncpy()`, be aware that if the destination buffer size is equal to the source buffer size, it may not NULL-terminate the string
- Check buffer boundaries if calling the function in a loop and make sure there is no danger of writing past the allocated space
- Truncate all input strings to a reasonable length before passing them to the copy and concatenation functions
- Specifically close resources, do not rely on garbage collection. (for example connection objects, file handles, etc.)
- Properly free allocated memory upon the completion of functions and at all exit points.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

6. Verification

Verification is one of the business functions described by the OWASP SAMM.

Verification focuses on the processes and activities related to how an organization checks and tests artifacts produced throughout software development. This typically includes quality assurance work such as testing, but it can also include other review and evaluation activities.

Verification activities should include:

- Architecture assessment, validation and mitigation
- Requirements-driven testing including security control verification and misuse/abuse testing
- Automated security testing and baselining
- Manual security testing and penetration testing

These activities are supported by:

- Security guides
- Test tools
- Test frameworks
- Vulnerability management
- Checklists

Sections:

6.1 Guides
6.1.1 Web Security Testing Guide
6.1.2 Mobile Application Security
6.2 Tools
6.2.1 Zed Attack Proxy
6.2.2 Code Pulse
6.2.3 Amass
6.2.4 Offensive Web Testing Framework
6.2.5 Nettacker
6.2.6 OWASP Secure Headers Project
6.3 Frameworks
6.3.1 Glue
6.3.2 secureCodeBox
6.3.3 Dracon
6.4 Vulnerability management
6.4.1 DefectDojo
6.5 Do's and Don'ts 6.5.1 Secure environment
6.5.2 System hardening
6.5.3 Open Source software

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

6.1 Verification guides

Verification is one of the business functions described by the OWASP SAMM. The verification activities are wide ranging, and will include:

- Testing of security controls
- Review of controls and security mechanisms
- Evaluation and assessment of the security architecture
- and others

Given the breadth of techniques and knowledge required, guides are an important resource for verification activities.

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

6.1.1 Web Security Testing Guide

To Do: supply a couple of sentences on the Web Security Testing Guide (WSTG) testing project, including its status as an OWASP project and where to find it.

What is WSTG? To Do: go into more detail about Web Security Testing Guide (WSTG) so that a developer can gain an overview of what this testing resource can provide for them.

Why use it? To Do: provide more context for WSTG that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how the WSTG testing resource can provide value for web application development teams. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 15: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

6.1.2 Mobile Application Security

The OWASP Mobile Application Security (MAS) flagship project has the mission statement: "Define the industry standard for mobile application security".

The MAS project covers the processes, techniques, and tools used for security testing a mobile application, as well as an exhaustive set of test cases that enables testers to deliver consistent and complete results. The OWASP MAS project provides the Mobile Application Security Verification Standard (MASVS) for mobile applications and a comprehensive Mobile Application Security Testing Guide (MASTG).

What is MASTG? The OWASP Mobile Application Security Testing Guide is a comprehensive manual for mobile application security testing and reverse engineering. It describes the technical processes used for verifying the controls listed in the OWASP MASVS.

The MASTG provides several resources for testing the controls:

- Sections detailing the concepts and theory behind testing of both Android and iOS platforms
- Lists of tests for each section of the MASVS
- Descriptions of techniques for Android or iOS used during testing
- Lists of generic tools and also ones specific for Android or iOS
- Reference applications that can be used as training material

Why use MASTG? The OWASP MASVS is the industry standard for mobile app security, and provides a list of security controls that are expected in a mobile application. If the application does not implement these controls correctly then it could be vulnerable; the MASTG tests that the application has the controls listed in the MASVS.

How to use MASTG The MASTG project contains a large number of resources that can be used during verification and testing of mobile applications, so pick and choose the resources that are applicable to your application.

- Refer to the MASTG section on the concepts and theory to ensure good understanding of the testing process
- Select the MASTG tests that are applicable to the application and its platform OS
- Use the section on MASTG techniques to run the selected tests correctly
- Become familiar with the range of MASTG tools available and select the ones that you need
- Use the MAS Checklists to provide evidence of compliance

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

6.2 Verification tools

Verification is one of the business functions described by the OWASP SAMM.

The SAMM Security Testing activity describes the use of both automated security testing and manual expert security testing to discover security defects. This security testing should be automated as part of the development, build and deployment processes; and can be complemented with regular manual security penetration tests.

Automated security testing tools are fast and scale well to numerous applications, whereas manual security testing of high-risk components requires good knowledge of the application and its business logic.

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

6.2.1 Zed Attack Proxy

To Do: supply a couple of sentences on the Zed Attack Proxy (ZAP) verification and testing project, including its status as a previous OWASP project, now a project with the Linux Foundation, and where to find it.

What is ZAP? To Do: go into more detail about ZAP so that a developer can gain an overview of what this testing resource can provide for them.

Why use it? To Do: provide more context for ZAP that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how the ZAP testing resource can provide value for web application development teams. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 16: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

6.2.2 Code Pulse

To Do: supply a couple of sentences on the OWASP Code Pulse verification and reporting project, including its status as an OWASP project and where to find it.

What is Code Pulse? To Do: go into more detail about Code Pulse so that a developer can gain an overview of what this reporting resource can provide for them.

Why use it? To Do: provide more context for Code Pulse that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how Code Pulse provides value for web application development teams. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 17: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

6.2.3 Amass

To Do: supply a couple of sentences on the OWASP Amass attack surface management project, including its status as an OWASP project and where to find it.

What is Amass? To Do: go into more detail about Amass so that a developer can gain an overview of what this attack surface management project can provide for them.

Why use it? To Do: provide more context for Amass that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how Amass provides value for a web application development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 18: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

6.2.4 Offensive Web Testing Framework

To Do: supply a couple of sentences on the OWASP Offensive Web Testing Framework (OWTF), including its status as an OWASP project and where to find it.

What is OWTF? To Do: go into more detail about OWTF so that a developer / penetration tester can gain an overview of what this attack surface management project can provide for them.

Why use it? To Do: provide more context for OWTF that allows developers or penetration testers to determine whether to use it.

How to use it To Do: give a brief outline of how OWTF provides value for a penetration tester. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 19: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

6.2.5 Nettacker

To Do: supply a couple of sentences on the OWASP Nettacker automated information & vulnerability scanner, including its status as an OWASP project and where to find it.

What is Nettacker? To Do: go into more detail about Nettacker so that a developer / penetration tester can gain an overview of what this attack surface management project can provide for them.

Why use it? To Do: provide more context for Nettacker that allows developers or penetration testers to determine whether to use it.

How to use it To Do: give a brief outline of how Nettacker provides value for a penetration tester. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 20: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

6.2.6 Secure Headers Project

To Do: supply a couple of sentences on the OWASP Secure Headers Project (OSHP) including its status as an OWASP project and where to find it.

What is OSHP? To Do: go into more detail about OSHP so that a developer / penetration tester can gain an overview of how this tool can validate an HTTP security header configuration.

Why use it? To Do: provide more context for OSHP that allows developers to determine whether to use it.

How to use it To Do: give a brief outline of how OSHP provides value for a development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 21: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

6.3 Verification frameworks

Verification is one of the business functions described by the OWASP SAMM and testing is an important part of verification, both Security Testing and Requirements-driven Testing.

Verification testing can benefit from using frameworks to support continuous and automated security testing. Use of a framework can provide:

- automation of a security analysis pipeline
- flexibility to run a series of tools in a pipeline
- scalability for multiple security scanners
- control interfaces

Sections:

6.3.1 Glue
6.3.2 secureCodeBox
6.3.3 Dracon

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

6.3.1 Glue

To Do: supply a couple of sentences on the OWASP Glue security analysis tool, including its status as an OWASP project and where to find it.

What is Glue? To Do: go into more detail about Glue so that a developer can gain an overview of what this tool can provide for them.

Why use it? To Do: provide more context for Glue that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how Glue provides value for a web application development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 22: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

6.3.2 secureCodeBox

To Do: supply a couple of sentences on the OWASP secureCodeBox automated security testing tool, including its status as an OWASP project and where to find it.

What is secureCodeBox? To Do: go into more detail about secureCodeBox so that a developer can gain an overview of what this security testing tool can provide for them.

Why use it? To Do: provide more context for secureCodeBox that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how secureCodeBox provides value for a web application development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 23: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

6.3.3 Dracon

To Do: supply a couple of sentences on the OWASP Dracon security pipelines tool, including its status as an OWASP project and where to find it.

What is Dracon? To Do: go into more detail about Dracon so that a developer can gain an overview of what this tool can provide for them.

Why use it? To Do: provide more context for Dracon that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how Dracon provides value for a web application development team. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 24: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

6.4 Verification vulnerability management

Verification is one of the business functions described by the OWASP SAMM. Vulnerability management helps maintain the application security level after bug fixes, changes or during maintenance.

The SAMM Requirements-driven Testing practice describes the outcomes for effective vulnerability management, and why it is necessary to have these processes in place. For example having regression testing in place, using security unit tests, provides some degree of confidence that applications are not vulnerable to know exploits.

Sections:

6.4.1 DefectDojo

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

6.4.1 DefectDojo

To Do: supply a couple of sentences on the OWASP DefectDojo DevSecOps and vulnerability management project, including its status as an OWASP project and where to find it.

What is DefectDojo? To Do: go into more detail about DefectDojo so that a developer can gain an overview of what this vulnerability management can provide for them.

Why use it? To Do: provide more context for DefectDojo that allows developers to determine whether to use it in their project.

How to use it To Do: give a brief outline of how DefectDojo provides value for web application development teams. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 25: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

6.5 Verification Do's and Don'ts

Verification is one of the business functions described by the OWASP SAMM.

Verification takes skill and knowledge, so it is important to build on the existing experience contained in these Do's and Dont's.

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

6.5.1 Secure environment

Here is a collection of Do's and Don'ts when it comes to creating a secure environment, gathered from practical experiences. Some of these are language specific and others have more general applicability.

- The WEB-INF directory tree contains web application classes, pre-compiled JSP files, server side libraries, session information, and files such as `web.xml` and `webapp.properties`. So be sure the code base is identical to production. Ensuring that we have a "secure code environment" is also an important part of an application secure code inspection.
- Use a "deny all" rule to deny access and then grant access on need basis.
- In Apache HTTP server, ensure directories like WEB-INF and META-INF are protected. If permissions for a directory and subdirectories are specified in `.htaccess` file, ensure that it is protected using the "deny all" rule.
- While using Struts framework, ensure that JSP files are not accessible directly by denying access to `*.jsp` files in `web.xml`.
- Maintain a clean environment. remove files that contain source code but are not used by the application.
- Ensure production environment does not contain any source code / development tools and that the production environment contains only compiled code / executables.
- Remove test code / debug code (that might contain backdoors). Commented code can also be removed as at times, it might contain sensitive data. Remove file metadata e.g., .git
- Set "Deny All" in security constraints (for the roles being set up) while setting up the application on the web server.
- The listing of HTTP methods in security constraints works in a similar way to deny-listing. Any verb not explicitly listed is allowed for execution. Hence use "Deny All" and then allow the methods for the required roles. This setting carries weightage while using "Anonymous User" role. For example, in Java, remove all `<http-method>` elements from `web.xml` files.
- Configure web and application server to disallow HEAD requests entirely.
- Comments on code and Meta tags pertaining to the IDE used or technology used to develop the application should be removed. Some comments can divulge important information regarding bugs in code or pointers to functionality. This is particularly important with server side code such as JSP and ASP files.
- Search for any calls to the underlying operating system or file open calls and examine the error possibilities.
- Remove unused dependencies, unnecessary features, components, files, and documentation.
- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component
- Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

6.5.2 System hardening

Here is a collection of Do's and Don'ts when it comes to system hardening, gathered from practical experiences. Some of these are language specific and others have more general applicability.

- The WEB-INF directory tree contains web application classes, pre-compiled files, server side libraries, session information, and files such as `web.xml` and `webapp.properties`. Secure these files
- In Apache HTTP server, ensure directories like WEB-INF and META-INF are protected. If permissions for a directory and subdirectories are specified in `.htaccess` file, ensure that it is protected using the "deny all" rule.
- While using Struts framework, ensure that JSP files are not accessible directly by denying access to `.jsp` files in `web.xml`.
- Maintain a clean environment. Remove files that contain source code but are not used by the application. Remove unused dependencies, unnecessary features, components, files, and documentation.
- Ensure production environment does not contain any source code / development tools and that the production environment contains only compiled code / executables.
- Remove test code / debug code (that might contain backdoors). Commented code can also be removed as at times it might contain sensitive data. Remove file metadata (e.g. `.git`)
- Set "Deny All" in security constraints (for the roles being set up) while setting up the application on the web server.
- The listing of HTTP methods in security constraints works in a similar way to deny-listing. Any verb not explicitly listed is allowed for execution. Hence use "Deny All" and then allow the methods for the required roles. This setting is particularly important using "Anonymous User" role. For example, in Java, remove all `<http-method>` elements from `web.xml` files.
- Prevent disclosure of your directory structure in the robots.txt file by placing directories not intended for public indexing into an isolated parent directory. Then ""Disallow"" that entire parent directory in the robots.txt file rather than disallowing each individual directory
- Configure web and application server to disallow HEAD requests entirely.
- Comments on code and Meta tags pertaining to the IDE used or technology used to develop the application should be removed.
- Some comments can divulge important information regarding bugs in code or pointers to functionality. This is particularly important with server side code such as JSP and ASP files.
- Search for any calls to the underlying operating system or file open calls and examine the error possibilities.
- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component
- Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.
- Remove backup or old files that are not in use
- Change/disable all default account passwords

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

6.5.3 Open Source software

Here is a collection of Do's and Don'ts when it comes to Open Source software, gathered from practical experiences. Some of these are language specific and others have more general applicability.

- Static Code Analysis (for licensing and dependencies)
  - Consuming open source software has a heavy dependency on the license under which the open source software is available.
  - Following are some URLs to licensing details:
    - `https://choosealicense.com/licenses/`
    - `https://tldrlegal.com/`
    - `https://creativecommons.org/licenses/by/4.0/`

It is important for the organisation to have a policy statement for consumption of open source software. From a licensing perspective and the implication of using a open source software incorrectly, maintain a procedure for approval of usage of selected open source software. This could be in the form of a workflow or obtaining security approvals for the chosen open source software We realise it could be challenging, but if feasible, maintain a list of approved open source software

- Address vulnerabilities with: Binaries / pre-compiled code / packages where source code sharing is not a part of the license (Examples executables / NuGets)
  - Where possible use version pinning
  - Where possible use integrity verification
  - Check for vulnerabilities for the selected binaries in vulnerability disclosure databases like
    - CVE database (`https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=bouncy+castle`)
    - VulnDB (`https://vuldb.com/?id.173918`)
  - If within the budget of your organisation, use an SCA tool to scan for vulnerabilities
  - Always vet and perform due-diligence on third-party modules that you install in order to confirm their health and credibility.
  - Hold-off on upgrading immediately to new versions; allow new package versions some time to circulate before trying them out.
  - Before upgrading, make sure to review change log and release notes for the upgraded version.
  - When installing packages make sure to add the –ignore-scripts suffix to disable the execution of any scripts by third-party packages.
  - Consider adding ignore-scripts to your `.npmrc` project file, or to your global npm configuration.
  - If you use npm, run `npm outdated`, to see which packages are out of date
  - Typosquatting is an attack that relies on mistakes made by users, such as typos. With typosquatting, bad actors could publish malicious modules to the npm registry with names that look much like existing popular modules.To address this vulnerability verify your packages before consuming them
- Address vulnerabilities with: where source code sharing is a part of the license
  - GitHub CodeQL / third party tool
  - If within the budget of your organisation, use an SCA tool to scan for vulnerabilities
- Security Testing: Binaries / pre-compiled code / packages where source code sharing is not a part of the license (Examples executables / NuGets)
  - Perform Dynamic application analysis
  - Perform Pen testing
  - Verify which tokens are created for your user or revoke tokens in cases of emergency; use npm token list or npm token revoke respectively.
- Security Testing: where source code sharing is a part of the license
  - Perform Static code analysis
  - Perform Dynamic application analysis
  - Perform Pen testing.
- Third Party Software and Libraries (hive off to OWASP Dependency Tracker)
  - Address supply chain risk with: Binaries / pre-compiled code / packages where source code sharing is not a part of the license (Examples executables / NuGets)
  - Use signed binaries / packages
  - Reference private feed in your code

  - Use controlled scopes
  - Lock files
  - Avoid publishing secrets to the npm registry (secrets may end up leaking into source control or even a published package on the public npm registry)
- Address supply chain risk with: where source code sharing is a part of the license
  - GitHubCheck for dependency graph
  - GitHubDependabot alerts
  - GitHubCommit and tag signatures
- Monitor Dependencies: Binaries / pre-compiled code / packages where source code sharing is not a part of the license (Examples executables / NuGets)
  - Use dependency graphs
  - Enable repeatable package restores using lock files
- Monitor Dependencies: where source code sharing is a part of the license
  - GitHubCheck for dependency graph
  - GitHub Secret scanning
- Maintaining open source software/components: Binaries / pre-compiled code / packages where source code sharing is not a part of the license
  - Monitor for deprecated packages
  - Use dependency graphs
  - Lock files
  - Monitor vulnerabilities with:
    - Check for vulnerabilities for the selected binaries in vulnerability disclosure databases like
      - CVE database (`https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=bouncy+castle`)
      - VulnDB (`https://vuldb.com/?id.173918`)
  - If within the budget of your organisation, use an SCA tool to scan for vulnerabilities
- Copying source code off public domain (internet) For example source code that is on a blog or in discussion forums like stacktrace or snippets of example on writeups *******Don't do it!!!*******

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

7. Training and Education

Training and Education activities are described by in the SAMM Training and Awareness section, which in turn is part of the SAMM Education & Guidance security practice within the Governance business function.

The goal of security training and education is to increase the awareness of application security threats and risks along with security best practices and secure software design principles. The security awareness training should be customised for all roles currently involved in the management, development, testing, or auditing of the applications and systems. In addition a Learning Management System or equivalent should be in place to track the employee training and certification processes.

OWASP provides various resources and environments that can help with this security training and education.

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

7.1 Juice Shop

To Do: supply a couple of sentences on the OWASP Juice Shop project, including its status as an OWASP project and where to find it.

What is Juice Shop? To Do: go into more detail about Juice Shop so that a developer can gain an overview of what this training tool can provide for them.

Why use it? To Do: provide more context for Juice Shop that allows developers to determine whether to use it.

How to use it To Do: give a brief outline of how to run Juice Shop. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 26: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

7.2 WebGoat

The OWASP WebGoat project is a deliberately insecure web application that can be used to attack common application vulnerabilities in a safe environment. It can also be used to exercise application security tools, such as OWASP ZAP, to practice scanning and identifying the various vulnerabilities built into WebGoat.

WebGoat is a well established OWASP project and achieved Lab Project status many years ago.

What is WebGoat? WebGoat is primarily a training aid to help development teams put into practice common attack patterns. It provides an environment where a web application can be safely attacked without traversing a network or upsetting a website owner.

The environment is self contained using a container and this ensures attack traffic does not leak to other systems; this traffic should look like a malicious attack to a corporate intrusion detection system and will certainly light it up. The WebGoat container contains WebWolf, an attacker client, which further ensures that attack traffic stays within the container.

In addition there is another WebGoat container available that includes a Linux desktop with some attack tools pre-installed.

Why use WebGoat? WebGoat is one of those tools that has many uses; certainly during training but also when presenting demonstrations, testing out security tools and so on. When ever you need a deliberately vulnerable web application running in a self contained and safe environment then WebGoat should be one of the first to consider.

Reasons to use WebGoat:

- Practical learning how to exploit web applications
- Ready made target during talks and demonstration on penetration testing
- Evaluating dynamic application security testing (DAST) tools; they should identify the known vulnerabilities
- Practising penetration testing skills
- and there will be more

How to use WebGoat The easiest way to run WebGoat is to use the provided Docker images to run containers. WebGoat can also be run as a standalone Java application using the downloaded Java Archive file or from the source itself; this requires various dependencies, whereas all dependencies are provided within the Docker images.

Access to WebGoat is via the port 8080 on the running Docker container and this will need to be mapped to a port on the local machine. Note that mapping to port 80 can be blocked on corporate laptops so it is suggested to map the port to localhost 8080.

1. The Docker daemon will have to be running to do this, get the Docker Engine from the download site.
2. Download the WebGoat docker image using command `docker pull webgoat/webgoat`
3. Run the container with `docker run --name webgoat -it -p 127.0.0.1:8080:8080 -p 127.0.0.1:9090:9090 webgoat/webgoat`
4. Use a browser to navigate to `localhost:8080/WebGoat` - note that there is no page served on `localhost:8080/`
5. You are then prompted to login, so first thing to do is create a test account from this login page
6. The accounts are retained when the container is stopped, but removed if the container is deleted
7. Creating insecure username/password combinations, such as `kalikali` with `kalikali`, is allowed

The browser should now be displaying the WebGoat lessons, such as 'Hijack a session' under 'Broken Access Control'.

How to use WebWolf WebWolf is provided alongside both the WebGoat docker images and the WebGoat JAR file. WebWolf is accessed using port 9090 on the Docker container, and this can usually be mapped to localhost port 9090 as in the example given above.

Use a browser to navigate to `http://localhost:9090/WebWolf`, there is no page served on URL `localhost:9090`. Login to WebWolf using one of the accounts created when accessing the WebGoat account management pages, such as username `kalikali` and password `kalikali`. All going well you will now have the WebWolf home page displayed.

WebWolf provides:

- File upload area
- Email test mailbox
- JWT tools
- Display of http requests

Where to go from here? Try all the WebGoat lessons, they will certainly inform and educate. Use WebGoat in demonstrations of your favourite attack chains. Exercise Zap and Burp Suite against WebGoat, or other attack tools you have with you.

Try out the WebGoat desktop environment by running `docker run -p 127.0.0.1:3000:3000 webgoat/webgoat-desktop` and navigating to `http://localhost:3000/`.

There are various ways of configuring WebGoat, see the github repo for more details.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

7.3 PyGoat

To Do: supply a couple of sentences on the OWASP PyGoat project, including its status as an OWASP project and where to find it.

What is PyGoat? To Do: go into more detail about PyGoat so that a developer can gain an overview of what this training tool can provide for them.

Why use it? To Do: provide more context for PyGoat that allows developers to determine whether to use it.

How to use it To Do: give a brief outline of how to run PyGoat. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 27: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

7.4 OWASP Top 10

To Do: supply a couple of sentences on the OWASP Top 10 project, including its status as an OWASP project and where to find it.

What is the OWASP Top 10? To Do: go into more detail about the OWASP Top 10 so that a developer can gain an overview of how it provides training and education.

Why use it? To Do: provide more context for the OWASP Top 10 that allows developers to determine whether to use it in their training and education program.

Figure 28: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

7.5 Mobile Top 10

To Do: supply a couple of sentences on the OWASP Mobile Top 10 project, including its status as an OWASP project and where to find it.

What is the Mobile Top 10? To Do: go into more detail about the Mobile Top 10 so that a developer can gain an overview of how it provides training and education.

Why use it? To Do: provide more context for the Mobile Top 10 that allows developers to determine whether to use it in their training and education program.

Figure 29: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

7.6 API Top 10

To Do: supply a couple of sentences on the API Top 10 project, including its status as an OWASP project and where to find it.

What is the API Top 10? To Do: go into more detail about the API Top 10 so that a developer can gain an overview of how it provides training and education.

Why use it? To Do: provide more context for the API Top 10 that allows developers to determine whether to use it in their training and education program.

Figure 30: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

7.7 Security Shepherd

To Do: supply a couple of sentences on the Security Shepherd project, including its status as an OWASP project and where to find it.

What is Security Shepherd? To Do: go into more detail about Security Shepherd so that a developer can gain an overview of what the training tool can provide for them.

Why use it? To Do: provide more context for Security Shepherd that allows developers to determine whether to use it for their training.

How to use it To Do: give a brief outline of how to run Security Shepherd and how it provides value for the development teams. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 31: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

7.8 OWASP Snakes and Ladders

To Do: supply an introduction on the OWASP Snakes & Ladders project, including its status as an OWASP project and where to find it, where to purchase it.

Why use it? To Do: provide more context for OWASP Snakes & Ladders that allows developers to determine whether to use it in their training and education programs, and its use for threat modeling.

Figure 32: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

7.9 Wrong Secrets

To Do: supply a couple of sentences on the OWASP Wrong Secrets project, including its status as an OWASP project and where to find it.

What is Wrong Secrets? To Do: go into more detail about Wrong Secrets so that a developer can gain an overview of what the training tool/game can provide for them.

Why use it? To Do: provide more context for Wrong Secrets that allows developers to determine whether to use it in their training.

How to use it To Do: give a brief outline of how to run Wrong Secrets and how it provides value for the development teams. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 33: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

8. Culture building and Process maturing

Culture building and Process maturing is described by the SAMM Organization and Culture activity, which in turn is part of the SAMM Education & Guidance security practice within the Governance business function.

The maturity of security processes and culture is wide ranging, with indicators of a mature process and culture including:

- Security champions have been identified for each development team
- A program is in place to support the security champions
- Secure coding practices are in place to define standards and improve software development
- Developers and application security professionals across the organization are able to communicate and share best practice

Sections:

8.1 Security Champions Playbook
8.2 Software Assurance Maturity Model
8.3 Application Security Verification Standard
8.4 Mobile Application Security

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

8.1 Security Champions Playbook

To Do: supply a couple of sentences on the Security Champions Playbook project, including its status as an OWASP project and where to find it.

What is the Security Champions Playbook? To Do: go into more detail about the Security Champions Playbook project, and Security Champions in general,so that a developer can gain an overview of what it can provide for their culture building and process maturing activities.

Why use it? To Do: provide more context for the Security Champions Playbook project that allows developers to determine whether it is suitable for their culture building and process maturing activities.

How to use it To Do: give a brief outline of how the Security Champions Playbook project can be used by the development teams for their culture building and process maturing. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 34: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

8.2 Software Assurance Maturity Model

To Do: supply a couple of sentences on the Software Assurance Maturity Model (SAMM) project, including its status as an OWASP project and where to find it.

What is SAMM? To Do: go into more detail about the SAMM project so that a developer can gain an overview of what it can provide for their culture building and process maturing activities.

Why use it? To Do: provide more context for the SAMM model that allows developers to determine whether to use it in their culture building and process maturing activities.

How to use it To Do: give a brief outline of how SAMM can be used by the development teams for their culture building and process maturing. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 35: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

8.3 Application Security Verification Standard

To Do: supply a couple of sentences on the Application Security Verification Standard (ASVS) project, including its status as an OWASP project and where to find it.

What is ASVS? To Do: go into more detail about ASVS so that a developer can gain an overview of what the tool can provide for them, specifically for the culture building and process maturing activities.

Why use it? To Do: provide more context for ASVS that allows developers to determine whether to use it in their culture building and process maturing activities.

How to use it To Do: give a brief outline of how ASVS can be used by the development teams for their culture building and process maturing. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 36: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

8.4 Mobile Application Security

The OWASP Mobile Application Security (MAS) flagship project has the mission statement: "Define the industry standard for mobile application security".

OWASP MAS Crackmes, also known as UnCrackable Apps, is a collection of reverse engineering challenges for the MAS project.

The MAS project covers the processes, techniques, and tools used for security testing a mobile application, as well as an exhaustive set of test cases that enables testers to deliver consistent and complete results. The OWASP MAS project provides the Mobile Application Security Verification Standard (MASVS) for mobile applications and a comprehensive Mobile Application Security Testing Guide (MASTG).

What is MAS Crackmes? OWASP MAS Crackmes is a set of reverse engineering challenges for mobile applications. These challenges are used as examples throughout the OWASP Mobile Application Security Testing Guide (MASTG) and, of course, you can also solve them for fun.

There are challenges for Android and a couple for Apple iOS.

Why use MAS Crackmes? Working through the challenges will improve understanding of mobile application security, and will also give an insight into the examples provided in the MASTG.

How to try the challenges

1. Select and download a challenge into your mobile application environment
2. Satisfy the individual challenge exercise
3. Have fun

Each challenge has various solutions provided by the community; these can be used to compare with your solution.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

9. Operation

Operations are those activities necessary to ensure confidentiality, integrity, and availability are maintained throughout the operational lifetime of an application and its associated data. The aim of Operations is to provide greater assurance that the organization is resilient in the face of operational disruptions, and responsive to changes in the operational landscape. This is described by the OWASP SAMM model for the Operations business function.

Operations generally cover the security practices:

- Incident Management of security breaches and incidents
- Environment Management such as configuration hardening, patching and updating
- Operational Management which includes data protection and system / legacy management

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

9.1 ModSecurity Core Rule Set

To Do: supply a couple of sentences on the OWASP ModSecurity Core Rule Set (CRS) project, including its status as an OWASP project and where to find it.

What is Coraza and ModSecurity? To Do: go into more detail about the Coraza / ModSecurity module and the Apache web server, so that a development team can gain an overview of how this works as a WAF.

What is the Core Rule Set? To Do: go into more detail about the Coraza / ModSecurity Core Rule Set so that a development team can determine understand its use with ModSecurity.

Why use it? To Do: provide more context for ModSecurity and the CRS to allows developer to determine whether to use it in the WAF.

How to use it To Do: give a brief outline of how CRS can provide value for the development teams. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 37: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

9.2 Coraza Web Application Firewall

OWASP Coraza is a golang enterprise-grade Web Application Firewall framework that supports the seclang language by ModSecurity and is 100% compatible with the OWASP Core Ruleset. Coraza is an OWASP Production project and is in active development.

What is Coraza? To Do: go into more detail about Coraza Web Application Firewall framework, so that a development team can gain an overview of how to use this as a WAF.

Why use it? To Do: provide more context for Coraza to allows developer to determine whether to use it.

How to use it To Do: give a brief outline of how Coraza can provide value for the development teams. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

For example, Coraza can be imported as a library or used with one of the Coraza connectors like coraza-server (GRPC and SPOA), coraza-caddy (web server, reverse proxy) and docker (using connector).

Figure 38: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

10. Metrics

At present the OWASP Integration Standards project Application Wayfinder project does not identify any OWASP projects that gather or process metrics, but this may change in the future.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

11. Security gap analysis

A security gap analysis is an activity where the information security posture of an organization is assessed and any shortfalls or operation gaps are identified. This activity can also be combined with a security gap evaluation where the existing controls and processes are assessed for effectiveness and relevance. Security gap analysis is required to gain or maintain certification to a management system standard such as ISO 27001 'Information security, cybersecurity and privacy protection'.

The security gap analysis is often associated with Governance, Risk & Compliance activities, where the compliance with a management system standard is periodically reviewed and updated. Guides and tools are useful for these compliance activities and the OWASP projects SAMM, MASVS and ASVS provide information and advice in meeting management system standards.

Sections:

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

11.1 Security gap analysis guides

Security gap analysis and security gap evaluation are central to Governance, Risk & Compliance activities and are used to gain and maintain certification to a management system standard such as ISO 27001 'Information security, cybersecurity and privacy protection'.

Guidance is important for these analysis and evaluation activities, with the OWASP projects SAMM, MASVS and ASVS providing this information and advice.

Sections:

11.1.1 Software Assurance Maturity Model
11.1.2 Application Security Verification Standard
11.1.3 Mobile Application Security

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue.

11.1.1 Software Assurance Maturity Model

To Do: supply a couple of sentences on the Software Assurance Maturity Model (SAMM) project, including its status as an OWASP project and where to find it.

What is SAMM? To Do: go into more detail about the SAMM project so that a developer can gain an overview of what the model can provide for them, specifically from a security gap analysis point of view.

Why use it? To Do: provide more context for the SAMM model that allows developers to determine whether to use it for their security gap analysis.

How to use it To Do: give a brief outline of how SAMM can be used by the development teams for their security gap analysis. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 39: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

11.1.2 Application Security Verification Standard

To Do: supply a couple of sentences on the Application Security Verification Standard (ASVS) project, including its status as an OWASP project and where to find it.

What is ASVS? To Do: go into more detail about ASVS so that a developer can gain an overview of what the tool can provide for them, specifically from a security gap analysis point of view.

Why use it? To Do: provide more context for ASVS that allows developers to determine whether to use it in their security gap analysis process.

How to use it To Do: give a brief outline of how ASVS can be used by the development teams and their security gap analysis. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 40: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.

11.1.3 Mobile Application Security

The OWASP Mobile Application Security (MAS) flagship project has the mission statement: "Define the industry standard for mobile application security".

The MAS project covers the processes, techniques, and tools used for security testing a mobile application, as well as an exhaustive set of test cases that enables testers to deliver consistent and complete results. The OWASP MAS project provides the Mobile Application Security Verification Standard (MASVS) for mobile applications and a comprehensive Mobile Application Security Testing Guide (MASTG).

What is MASVS? The OWASP MASVS is the industry standard for mobile app security. It can be used by mobile software architects and developers seeking to develop secure mobile applications, as well as security testers to ensure completeness and consistency of test results.

The MAS project has several uses; when it comes to security gap analysis then the MASVS contains a list of security controls for mobile applications that are expected to be present / implemented.

The security controls are split into several categories:

- MASVS-STORAGE
- MASVS-CRYPTO
- MASVS-AUTH
- MASVS-NETWORK
- MASVS-PLATFORM
- MASVS-CODE
- MASVS-RESILIENCE

Why use MASVS? The OWASP MASVS provides a list of security controls that are expected in a mobile application. If the application does not implement any of the controls then this could become a compliance issue, given that MASVS is the industry standard for mobile applications, so any omissions need to be justified.

How to use MASVS MASVS can be accessed online and the links followed for the security controls; the mobile application can then be inspected for compliance with each control. In addition MASVS can be downloaded as a PDF which can, for example, be used for evidence or compliance purposes.

The OWASP Developer Guide is a community effort; if there is something that needs changing then submit an issue or edit on GitHub.

11.2 Bug Logging Tool

To Do: supply a couple of sentences on the OWASP Bug Logging Tool (BLT) project, including its status as an OWASP project and where to find it.

What is BLT? To Do: go into more detail about Bug Logging Tool, so that an organization / development team can gain an overview on why to deploy it.

Why use it? To Do: provide more context for BLT to allows an organization to determine whether to use it.

How to use it To Do: give a brief outline of how BLT can provide value for an organization. Do not repeat the project documentation itself; ideally provide a primer and a pointer to the documentation.

Figure 41: Developer Guide

The OWASP Developer Guide is a community effort and this page needs some content to be added. If you have suggestions then submit an issue and the project team can assign it to you, or provide new content direct on GitHub.