Developer Guide

Open Worldwide Application Security Project (OWASP)

February 2023 onwards

OWASP Developer Guide

A Guide to Building Secure Web Applications and Web Services

Release version 4.1.5-RC1

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. The latest versions are 4.x because 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 security practices using the advice given in the OWASP Software Assurance Maturity Model (SAMM) and describes the OWASP projects referenced in the OWASP Application Security Wayfinder project.

This guide does not seek to replicate the many excellent sources on specific security topics; it will rarely tries to go into details on a subject and instead provides links for greater depth on these security topics. Instead the content of the Developer Guide aims to be accessible, introducing practical security concepts and providing enough detail to get developers started on various OWASP tools and documents.

All of the OWASP projects and tools described in this guide are free to download and use. All OWASP projects are open source; please do get involved if you are interested in improving application security.

Audience Developers should use this OWASP Developer Guide to help write applications that are more secure. The guide has been written by the security community to help software developers write solid, safe and secure applications. Most of the contributors to this guide are also software developers as well as security engineers, and this helps to keep the focus developer centric.

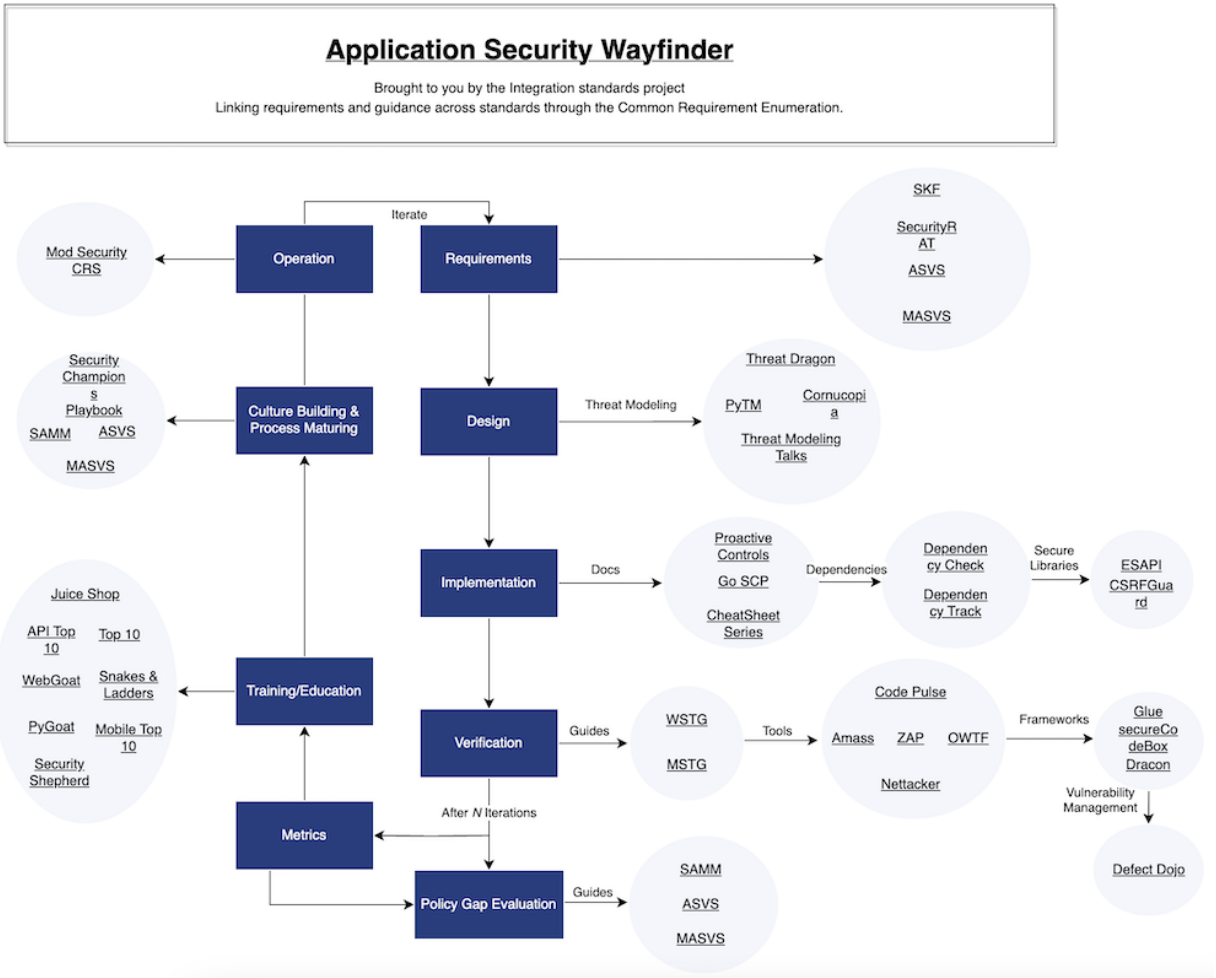
If you are in a hurry and want information on a specific subject then try the OpenCRE chat LLM for immediate answers.

What is the Developer Guide? You can think of this guide as a cross-reference source to the many tools and documents that OWASP provide for developers.

Or you can regard the purpose of this guide as answering the question: “I am a developer and I need a reference guide to navigate the numerous security tools and security activities that I know I should be doing.

Or think of it as a collection of articles that introduce developers to the wide domain of application security.

Or you can regard this guide as a companion document to the OWASP Integration Standards project: the Application Security Wayfinder maps out the many tools, projects and documents within OWASP and the Developer Guide provides some ‘wordy’ context.


Application Security Wayfinder
Brought to you by the Integration standards project
Linking requirements and guidance across standards through the Common Requirement Enumeration.
Iterate
Operation
Requirements
Mod Security CRS
SKF
SecurityRAT
ASVS
MASVS
Security Champions Playbook
SAMM
ASVS
MASVS
Culture Building & Process Maturing
Design
Threat Modeling
Threat Dragon
PyTM
Cornucopia
Threat Modeling Talks
Implementation
Docs
Proactive Controls
Go SCP
CheatSheet Series
Dependencies
Dependency Check
Dependency Track
Secure Libraries
ESAPI
CSRFGuard
Juice Shop
API Top 10
Top 10
WebGoat
Snakes & Ladders
PyGoat
Mobile Top 10
Security Shepherd
Training/Education
Verification
Guides
WSTG
MSTG
Tools
Code Pulse
Amass
ZAP
OWTF
Nettacker
Frameworks
Glue
secureCodeBox
Dracon
Vulnerability Management
Defect Dojo
After N Iterations
Metrics
Policy Gap Evaluation
Guides
SAMM
ASVS
MASVS

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. This Developer Guide can only give a brief overview of these concepts, for in-depth knowledge refer to the many texts on security such as the The Cyber Security Body Of Knowledge.

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 10

2.1 Security fundamentals

The fundamental principles of application security rely on the security concepts referenced in this developer guide. This section aims to provide an introduction to fundamental principles that any development team must be familiar with.

Software Assurance Maturity Model 

The Software Assurance Maturity Model (SAMM) provides context for the scope of software security and the foundations of good security practice:

- Governance
- Design
- Implementation
- Verification
- Operations

The SAMM model describes these foundations of software security as Business Functions, which are further divided into Business Practices. The OWASP Software Assurance Maturity Model (SAMM) is used throughout this Developer Guide; most of the sections in the Developer Guide reference at least one of the Business Functions or Practices from SAMM.

CIA triad 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.

CIA stands for Confidentiality, Integrity and Availability, and it is usually depicted as a triangle representing the strong bonds between its three tenets. This triad is considered the pillars of application security, often Confidentiality, Integrity or Availability are used as a properties of data or processes within a given system. The CIA triad can be extended with the AAA triad: 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 and applies to both data at rest and to data in transit. Confidentiality is also 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 triad The CIA triad is often extended with Authentication, Authorization and Auditing as these are closely linked to CIA concepts. CIA has a strong dependency on Authentication and Authorization; the confidentiality and integrity of sensitive data can not 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. For example the entity could be an automated client or a human actor; in either case authentication is required for a secure application.

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 that are being secured. Authorization is usually preceded by successful 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?"

Vulnerabilities NIST defines a vulnerability as 'Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.'

There are many weaknesses or bugs in every large application, but the term vulnerability is generally reserved for those weaknesses or bugs where there is a risk that a threat actor could exploit it using a threat vector.

Well known security vulnerabilities are :

- Clickjacking
- Credential Stuffing
- Cross-site leaks
- Denial of Service (DoS) attacks
- DOM based XSS attacks including DOM Clobbering
- IDOR (Insecure Direct Object Reference)
- Injection including OS Command injection and XXE
- LDAP specific injection attacks
- Prototype pollution
- SSRF attacks
- SQL injection and the use of Query Parameterization
- Unvalidated redirects and forwards
- XSS attacks and XSS Filter Evasion

HTTP and HTML Although not a security fundamental as such, web applications rely on HTTP communications and HTML. Both application developers and security engineers should have a good understanding of HTTP and the HTML language along with their various security controls.

Most application development teams will be familiar with HTTP communications and the HTML standard, but if necessary refer to the training from the W3 Consortium or the W3 Schools. The OWASP Cheat Sheet Series provide web application developers with the information needed to produce secure software :

- The HTML5 Security cheat sheet describes a wide range of controls, aligned with the current HTML Living Standard
- Refer to the Securing Cascading Style Sheets cheat sheet for CSS
- The HTTP headers need to be secure, see the HTTP Security Response Headers cheat sheet
- Strongly consider HTTP Strict Transport Security
- If the application has a file upload feature, follow the File Upload cheat sheet
- Ensure content security policy is in place with the Content Security Policy cheat sheet
- Using JWTs for a Java application? Refer to the JSON Web Token cheat sheet
- Storing or sending objects? Check out the Deserialization cheat sheet

References

- WHATWG HTML Living Standard
- OWASP Cheat Sheet Series
- OWASP Software Assurance Maturity Model (SAMM)

2.2 Secure development and integration

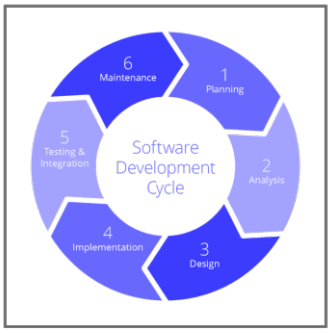
Secure development is described in the OWASP Software Assurance Maturity Model (SAMM) Design, Implementation and Verification business functions.

Prelude The best introduction to practical secure software development is the OWASP Application Security Fragmentation article :

Or how I worried less and stood on the shoulders of giants. - Spyros Gasteratos, Elie Saad

Much of the material in this section is drawn from this 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 notional Software Development LifeCycle (SDLC) is shown below, in practice there may be more or less phases according to the processes adopted by the organization.



A typical SDLC representation

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 from an existing software development lifecycle, it must always be the same development lifecycle as before but with security actions built into each phase, otherwise security actions may well 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 the SDLC 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 security built in to them. 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. Refer to the CI/CD Security Cheat Sheet for the latest DevSecOps security controls.

Secure development lifecycle Referring to the OWASP Application Security Wayfinder development cycle there are four iterative phases during application development: Requirements, Design, Implementation and Verification. The other phases are done less iteratively in the development cycle but 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 checked for completeness and validity, and it is worth considering various OWASP tools to help with this;
  - 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 application - it is never too late to do this but the earlier the better and easier to do. 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 at the end of this section.
- **Training**: development teams continually need security training. Although not part of the inner SDLC iterative loop training should still 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.
- **Operations**: the OWASP DevSecOps Guideline explains how to best implement a secure pipeline, using best practices and automation tools to help 'shift-left' security issues. Refer to the DevSecOps Guideline for more information on any of the topics within DevSecOps and in particular sections on Operations.
- **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 dependencies**: 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, and Dynamic Application Security Testing (DAST), which applies input to the application while running it in a sandbox or other isolated environments. Interactive Application Security Testing

(IAST) is designed to be run manually as well as being automated, and provides instant feedback on the tests as they are run.

Further reading from OWASP

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

OWASP verification projects

- Application Security Verification Standard (ASVS)
- Amass project
- Code Pulse
- Defect Dojo
- Mobile Application Security (MAS)
- Nettacker
- Offensive Web Testing Framework (OWTF)
- Web Security Testing Guide (WSTG)

OWASP training projects

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

OWASP resources

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

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.

Security by Design Security should not be an afterthought or add-on. When developing systems, you should begin with identifying relevant security requirements and treat them as an integral part of the overall process and system design. Begin with establishing and adopting relevant principles and policies as a foundation for your design, then build security into your development life cycle. Keep also in mind that the system you are building also will be needing maintenance and that system operators will need to securely manage and even shutdown and dispose of the system. Therefore, commit to secure operations by developing secure "operational management"[1] principles and practices.

Security by Default Secure by default means that the default configuration settings are the most secure settings possible. This is not necessarily the most user-friendly settings. Evaluate what the settings should be, based on both risk analysis and usability tests. As a result, the precise meaning is up to you to decide. Nevertheless, configure the system to only provide the least functionality and to specifically prohibit and/or restrict the use of all other functions, ports, protocols, and/or services. Also configure the defaults to be as restrictive as possible, according to best practices, without compromising the "Psychological acceptability" and "Usability and Manageability" of the system.

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 an entity 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, the term 'fail secure' is commonly used interchangeably with fail safe, which comes from physical security terminology. Failing safe also helps software resiliency in that the system / application can rapidly recover upon design or implementation flaws.

[1] Operational Management, (SAMM)

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 laterally or vertically. In order to apply this principle of least privilege proper granularity of privileges and permissions should be established.

Compartmentalize The principle of least privilege works better if access rights are not an "all or nothing" access model. Instead, compartmentalize the access to information on a "need-to-know" basis in order to perform certain tasks. The compartmentalization principle helps in minimizing the impact of a security breach in case of a successful breach attempt but must be used in moderation in order to prevent the system from becoming unmanageable. Therefore, follow also the principle of "Economy of Mechanism".

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, individually by themselves, for completing the task.

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 contrast 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 or processes are at different levels of privilege. This is important when defending against privilege escalation.

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

Usability and Manageability Is a principle related to psychological acceptability, but goes beyond just the perceived psychological acceptability to also include the design, implementation and operation of security controls. The configuration, administration and integration of security components should not be overly complex or obscure. Therefore, always use open standards for portability and interoperability, use common language in developing security requirements, design security to allow for regular adoption of new technology, ensure a secure and logical upgrade process exist, automate security management activities and strive for operational ease of use.

Secure the 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. Therefore, identifying the weakest component and addressing the most serious risk first, until an acceptable level of risk is attained, is considered good security practice.

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 components developed within the open source community have the further benefit of 'many eyes' and are therefore likely to be even more secure.

References

- OWASP Cheat Sheet series
  - Authentication Cheat Sheet
  - Authorization Cheat Sheet
  - Secure Product Design Cheat Sheet
- OWASP Top 10 Proactive Controls
  - C5: Secure by Default Configurations
- Other
  - Compartmentalization (information security), (Wikipedia)
  - Least Functionality, (NIST)
  - Security by Design, (Open Group)
  - Usability and Manageability, (Open Group)

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 simply 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 Although cryptography was initially restricted primarily to the military and the realm of academia, cryptography has become ubiquitous in securing software applications. 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, confidentiality is defined as "no unauthorized disclosure of information". Cryptography addresses this via encryption of either the data at rest or data in transit by protecting the information from all who do not hold the decryption key. Cryptographic hashes (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. Cryptography is central to authentication:

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)

OpenID Connect is widely used as an identity layer on top of the OAuth 2.0 protocol, see the OAuth 2.0 Protocol Cheat Sheet.

Integrity Integrity ensures 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.

The term 'message authenticity' refers to ensuring the integrity of information, often using symmetric encryption and shared keys, but does not authenticate the sending party.

The term 'authenticated encryption' also ensures the integrity of information, and, if asymmetric encryption is used, can authenticate the sender.

Non-repudiation 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. It can be accomplished by having the sender or recipient 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 known as the 'hash value' or 'digest value'. These hash functions are many-to-one mappings that are compression functions.

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 provide two types of integrity: keyed hashes, often 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 operate 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 A cryptographic 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 Except when they are operating in a streaming mode, block ciphers generally operate on fixed size blocks. These block ciphers must also operate on messages of any size, not just those that are an integral multiple of the cipher's block size, and so the message can be padded to fit into the next fixed-size block.

Asymmetric ciphers 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. Generally the public key is widely shared and the private key is kept secure.

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 prevent 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.

References

- OWASP Cheat Sheet series
  - Authentication
  - Authorization
  - Cryptographic Storage
  - Key Management
  - OAuth 2.0 Protocol
  - SAML Security
  - Secure Product Design
  - User Privacy Protection

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 any 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
- Serverless 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

Other OWASP Top 10s are 'incubator' projects, which are work in progress, so this list will change over time.

3. Requirements

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.

The importance of understanding key security requirements is described in the Security Requirements practice that is part of the Design business function section within the OWASP SAMM model. Ideally structured software security requirements are available within with a security a requirements framework, and these are utilized by both developer teams and product teams. In addition suppliers to the organization must meet security requirements; build security into supplier agreements in order to ensure compliance with organizational security requirements.

In summary, security requirements exist to prevent the repeat of past security failures.

Sections:

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. Requirements will certainly help with the prevention of many types of vulnerabilities.

Requirements come from various sources, three common ones being:

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 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.

Consider using Abuse Cases to identify possible attacks and the controls required to mitigate them. This can then feed into the software security requirements.

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 regulations come into force, both of which may have a direct impact on the application.

References

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

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 common to implement a security control 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

References

- 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

---

{:
height="180px" }

3.3 OpenCRE

The Open Common Requirement Enumeration (OpenCRE) is a catalog of security requirements: enumerating security topics and providing links to various standards, cheat sheets and guides.

The OWASP Integration Standards project includes both the OpenCRE and Security and the Application Security Wayfinder, it is an OWASP documentation project with production status.

What is the Integration Standards project? The Integration Standards project is at the centre of the OWASP project community; it provides guidance on how to navigate and use the many projects within OWASP. It does this in two ways, first is the Application Security Wayfinder which provides a visual map of the most important OWASP projects - as of August 2024 there are 345 OWASP projects so this is a really useful visualization. The second is the Open Common Requirement Enumeration (OpenCRE) which provides a consolidated reference of standards, cheat sheets, tools and other enumerations (such as CWE).

The Integration Standards project has also produced OWASP Application Security Fragmentation write-up on OWASP and the secure Software Development LifeCycle (SDLC). This provides an overview of tools and techniques used for most SDLCs.

What is OpenCRE? OpenCRE is a catalog, or enumeration, of various standards and reference material, including:

- CAPEC
- CWE
- NIST Special Publications 800-53 and 800-63
- OWASP ASVS
- OWASP Top10
- OWASP Proactive Controls
- OWASP Cheat Sheets
- OWASP WSTG
- ZAP

The aim of this project is to 'Link all the things with OpenCRE' which will:

- make it easier for engineers, security officers, testers and procurement to find relevant information
- make it easier for standards makers to create and maintain references

Why use OpenCRE? OpenCRE: 'Everything organized'

OpenCRE is a powerful tool that can provide developers with links to many resources, and is easy to use. It provides a one-stop consolidated set of references on various security terms and domains, and crucially these are automatically kept up to date. The provides a handy security catalog that can be searched for various standards or security terms.

As well as being useful for day to day security questions, the OpenCRE can also be used as the reference section in documentation; linking across to the OpenCRE rather than providing a list of references means the links are kept up to date automatically.

How to use OpenCRE The OpenCRE catalog can be accessed in traditional ways such as using searches or linking across to it. For example OpenCRE references to the Common Weakness Enumeration can be accessed using the search facility or by linking across directly to a specific Open Common Requirement.

OpenCRE is also useful when providing references in documentation. OpenCRE can be used for these references instead of listing various references to a security concept or requirement. This will provide links to standards, cheat sheets, tools and other enumerations - along with other sources that have been added over time - and all kept up to date. So no more broken links or referring to out of date versions :)

This is now the age of large language models, and OpenCRE has embraced this technology. Immediate answers to security questions or searches can be provided by OpenCRE Chat.

For example, in answer to the question "*what use is the OWASP Developer Guide?*" OpenCRE Chat provides the agreeable answer:

"The OWASP Developer Guide provides a comprehensive overview of application security risks and how to mitigate them. It covers topics such as input validation, output encoding, secure coding practices, and secure design principles. The guide is a valuable resource for developers who want to create secure applications."

References

- OWASP OpenCRE
- Spotlight on OpenCRE
- OWASP Application Security Fragmentation
- OWASP Integration Standards project
- Understanding the Complete Chain of Application Security Using OpenCRE org

---

3.4 SecurityRAT

The OWASP SecurityRAT (Requirement Automation Tool) is used to generate and manage security requirements using information from the OWASP ASVS project. It also provides an automated approach to requirements management during development of frontend, server and mobile applications.

At present it is an OWASP Incubator project but it is likely to be upgraded soon to Laboratory status.

What is SecurityRAT? SecurityRAT is a companion tool for the ASVS set of requirements; it can be used to generate an initial set of requirements from the ASVS and then keep track of the status and updates for these requirements. It comes with documentation and instructions on how to install and run SecurityRAT.

To generate the initial list of requirements, SecurityRAT needs to be provided with three attributes defined by the ASVS:

- Application Security Verification Standard chapter ID - for example 'V2 - Authentication'
- Application Security Verification Level - the compliance level, for example 'L2'
- Authentication - whether Single sign-on (SSO) authentication is used or not

SecurityRAT then generates an initial list of recommended requirements. This list can be stored in a SecurityRAT database which allows tracking and update of the set of requirements. SecurityRAT also provides Atlassian JIRA integration for raising and tracking software issues.

The OWASP Spotlight series provides an overview of what Security Rat can do and how to use it: 'Project 5 - OWASP SecurityRAT'.

Why use it? At the time of writing the ASVS has more than 280 suggested requirements for secure software development. This number of requirements takes time to sort through and determine whether they are applicable to a given development project or not.

The use of SecurityRAT to create a more manageable subset of the ASVS requirements is a direct benefit to both security architects and the development team. In addition SecurityRAT provides for the tracking and update of this set of requirements throughout the development cycle, adding to the security of the application by helping to ensure security requirements are fulfilled.

How to use SecurityRAT Install both Production and Development SecurityRAT applications by downloading a release and installing on the Java Development Kit JDK11. Alternatively download and run the docker image from DockerHub. Configure SecurityRAT by referring to the deployment documentation; this is not that straightforward so to get started there is an online demonstration available.

Logging in to the demonstration site, using the credentials from the project page, you are presented with defining a set of requirements or importing an existing set. Assuming that we want a new set of requirements, give the requirements artifact a name and then either select specific ASVS sections/chapters from the list:

- V1 - Architecture, Design and Threat Modeling
- V2 - Authentication
- V3 - Session Management
- V4 - Access Control
- V5 - * Validation, Sanitization and Encoding
- V6 - Stored Cryptography
- V7 - Error Handling and Logging
- V8 - Data Protection
- V9 - Communication
- V10 - Malicious Code
- V11 - Business Logic
- V12 - Files and Resources
- V13 - API and Web Service
- V14 - Configuration

or leave blank to include all verification requirements.

Select the level using the ASVS defined security compliance levels:

- Level 1 is for low assurance levels and is completely penetration testable
- Level 2 is for applications that contain sensitive data and is the recommended level for most applications
- Level 3 is for the most critical applications

Finally select whether SSO authentication is being used, and generate a list of requirements. This requirements artifact is now stored in SecurityRAT ad can be retrieved in subsequent sessions.

SecurityRAT then presents an administration screen which allows tracking and editing of the ASVS verification requirements. Refer to the OWASP Spotlight on SecurityRAT for an explanation of how to integrate with Atlassian JIRA.

What is SecurityCAT? SecurityCAT (Compliance Automation Tool) is an extension for SecurityRAT meant for automatic testing of requirements. There is not an actual implementation of SecurityCAT, SecurityRAT provides an API that allows for a compliance tool to be created. and so this may be a future development for SecurityRAT.

References

- OWASP SecurityRAT
- OWASP SecurityRAT documentation
- OWASP SecurityCAT
- OWASP Application Security Verification Standard (ASVS)

3.5 Application Security Verification Standard

The Application Security Verification Standard (ASVS) is a long established OWASP flagship project, and is widely used to suggest security requirements as well as the core verification of web applications.

It can be downloaded from the OWASP project page in various languages and formats: PDF, Word, CSV, XML and JSON. Having said that, the recommended way to consume the ASVS is to access the github markdown pages directly - this will ensure that the latest version is used.

What is ASVS? The ASVS is an open standard that sets out the coverage and level of rigor expected when it comes to performing web application security verification. The standard also provides a basis for testing any technical security controls that are relied on to protect against vulnerabilities in the application.

The ASVS is split into various sections:

- V1 Architecture, Design and Threat Modeling
- V2 Authentication
- V3 Session Management
- V4 Access Control
- V5 Validation, Sanitization and Encoding
- V6 Stored Cryptography
- V7 Error Handling and Logging
- V8 Data Protection
- V9 Communication
- V10 Malicious Code
- V11 Business Logic
- V12 Files and Resources
- V13 API and Web Service
- V14 Configuration

The ASVS defines three levels of security verification:

1. applications that only need low assurance levels; these applications are completely penetration testable
2. applications which contain sensitive data that require protection; the recommended level for most applications
3. the most critical applications that require the highest level of trust

Most applications will aim for Level 2, with only those applications that perform high value transactions, or contain sensitive medical data, aiming for the highest level of trust at level 3.

Why use it? The ASVS is used by many organizations as a basis for the verification of their web applications. It is well established, the earlier versions were written in 2008, and it has been continually supported since then. The ASVS is comprehensive, for example version 4.0.3 has a list of 286 verification requirements, and these verification requirements have been created and agreed to by a wide security community.

For these reasons the ASVS is a good starting point for creating and updating security requirements for web applications. The widespread use of this open standard means that development teams and suppliers may already be familiar with the requirements, leading to easier adoption of the security requirements.

How to use it The OWASP Spotlight series provides an overview of the ASVS and its uses: 'Project 19 - OWASP Application Security Verification standard (ASVS)'.

The appropriate level of verification should be chosen from the ASVS levels:

- Level 1: First steps, automated, or whole of portfolio view
- Level 2: Most applications
- Level 3: High value, high assurance, or high safety

Tools such as SecurityRAT can help create a more manageable subset of the ASVS security requirements, allowing focus and decisions on whether each one is applicable to the web application or not.

The OWASP Cheat Sheets have been indexed specifically for each section of the ASVS, which can be used as documentation to help decide if a requirements category is to be included in the test scheme.

References

- OWASP Application Security Verification Standard (ASVS)
- OWASP Cheat Sheets for ASVS
- OWASP SecurityRAT

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 mobile applications. It provides a set of test cases that enables testers to deliver consistent and complete results. The OWASP MAS project provides both the Mobile Application Security Verification Standard (MASVS) for mobile applications and the Mobile Application Security Testing Guide (MASTG).

What is MASVS? The OWASP MASVS is used by mobile software architects and developers 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 / Cheat Sheets
- MASVS-CRYPTO / Cheat Sheets
- MASVS-AUTH / Cheat Sheets
- MASVS-NETWORK / Cheat Sheets
- MASVS-PLATFORM / Cheat Sheets
- MASVS-CODE / Cheat Sheets
- MASVS-RESILIENCE / Cheat Sheets
- MASVS-PRIVACY / Cheat Sheets

The last category, MASVS-PRIVACY, is being reworked so is subject to change.

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 Cheat Sheets have been indexed specifically for each category of the MASVS, which can be used as a guide to decide if the category should to be included in the test scheme.

References

- OWASP Mobile Application Security (MAS)
- MAS project
- MAS Checklist
- MAS Verification Standard (MASVS)
- OWASP Mobile Application Security cheat sheet

3.7 Security Knowledge Framework

The Security Knowledge Framework (SKF) is an expert system application that uses various open source projects to support development teams and security architects in building secure applications. The SKF builds on the OWASP Application Security Verification Standard (ASVS) to help developers in both pre-development and post-development phases and create applications that are secure by design.

Having been an OWASP flagship project for many years the SKF is now no longer within the OWASP organization; and it will continue to be referenced in the OWASP Wayfinder and other OWASP projects because it is a flagship project for any organization.

What is the Security Knowledge Framework? The SKF is a web application that is available from the github repo. There is a demo version of SKF that is useful for exploring the multiple benefits of the SKF. Note that SKF is in a process of migrating to a new repository so the download link may change.

The SKF provides training and guidance for application security:

- Requirements organizer
- Learning courses:
  - Developing Secure Software (LFD121)
  - Understanding the OWASP Top 10 Security Threats (SKF100)
  - Secure Software Development: Implementation (LFD105x)
- Practice labs
- Documentation on installing and using the SKF

Why use the SKF for requirements? The SKF organizes security requirements into various categories that provides a good starting point for application security.

- API and Web Service
- Access Control
- Architecture Design and Threat Modeling
- Authentication
- Business Logic
- Communication
- Configuration
- Data Protection
- Error Handling and Logging
- Files and Resources
- Malicious Code
- Session Management
- Stored Cryptography
- Validation Sanitization and Encoding

How to use the SKF for requirements Visit the requirements tool website and select the relevant requirements from the various categories. Export the selection to the format of your choice (Markdown, spreadsheet CSV or plain text) and use this as a starting point for the application security requirements.

The OWASP Spotlight series provides an overview of the SKF: 'Project 7 - Security Knowledge Framework (SKF)'.

References

- Security Knowledge Framework (SKF)
- SKF courses and labs
- SKF requirements
- OWASP Application Security Verification Standard (ASVS)

4. Design

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.

Software application Design is one of the major business functions described in the Software Assurance Maturity Model (SAMM), and includes security practices:

- Threat Assessment
- Security Requirements
- Security Architecture

Sections:

4.1 Threat modeling

Referring to the Threat Modeling Cheat Sheet, threat modeling is a structured approach to 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 if during a later phase of the project refactoring is required to include risk mitigations. The outcomes 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:

4.1.1 Threat modeling in practice

This section discusses Threat Modeling, an activity described 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, and the philosophy of this section tries to follow the Threat Modeling Manifesto.

{: height="60px" }

Overview Threat modeling activities try to discover what can go wrong with a system and determine what to do about it. The deliverables from threat modeling take various forms including system models and diagrams, lists of threats, mitigations or assumptions, meeting notes, and more. This may be 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. 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
5. Trust: identify your trust boundaries, consider network segmentation
6. Agents: identify who the actors are (malicious or otherwise) and what they can do
7. Categorise: prioritise the threats taking into account probability, impact and any other factors
8. 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 Some final words 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 change control for their models.

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:

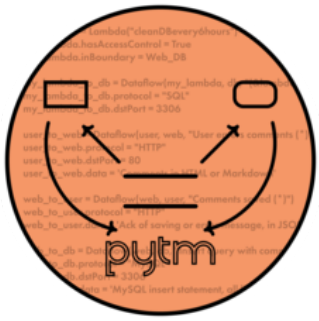
It is a good strategy to choose a threat categorisation methodology for the whole organisation and then try and keep to it. For example this could be STRIDE or LINDDUN, but if the CIA triad provides enough granularity then that is also a perfectly good choice.

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



4.1.2 pytm

The OWASP pytm (Pythonic Threat Modeling) 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 regular 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.

The OWASP Spotlight series provides an overview of pytm: 'Project 6 - OWASP pytm'.

Why use pytm? The pytm development team make the good point 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 certainly valid observations, and so 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 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
```

References

- OWASP Pythonic Threat Modeling (pytm)
- Graphviz
- pandoc
- PlantUML
- pytm repository
- Spotlight on pytm
- Threat Modeling: a practical guide for development teams

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 - various different types of teams can all benefit from Threat Dragon ease of use

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

- STRIDE
- LINDDUN
- PLOT4ai
- CIA
- DIE

and it can be used by all sorts of development teams.

How to use it The OWASP Spotlight series provides an overview of Threat Dragon and how to use it: 'Project 22 - OWASP Threat Dragon'.

It is straightforward to start using Threat Dragon; the latest version is available to use online:

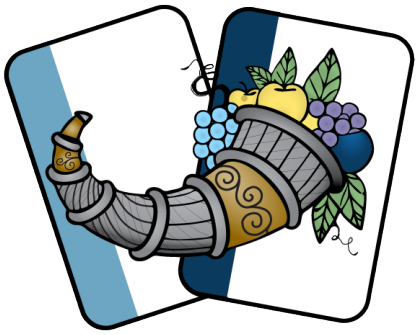
1. select 'Login to Local Session'
2. select 'Explore a Sample Threat Model'
3. select 'Version 2 Demo Model'
4. you are then presented with the threat model meta-data which can be edited
5. click on the diagram 'Main Request Data Flow' to display the data flow diagram
6. the diagram components can be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

Threat Dragon is distributed as a cross platform desktop application as well a web application. 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.

References

- OWASP Threat Dragon



4.1.4 Cornucopia

OWASP Cornucopia is a card game used to help derive application security requirements during the software development life cycle. Cornucopia is an OWASP Lab project, and can be downloaded from its project page.

What is Cornucopia? Cornucopia provides a set of cards designed to gamify threat modeling activities, helping agile development teams to identify weaknesses in applications and then record remediations or requirements.

There are three versions of the Cornucopia deck of threat modeling cards:

- Website App Edition
- Mobile App Edition
- Enterprise App Edition

The decks come with several suits according to the application, and always contain an overall 'Cornucopia' suit.

Cornucopia can be played in many different ways, there is no one way, and there is a suggested set of rules to start the game off. Cornucopia provides a score sheet to can help keep track of the game session and to record outcomes.

Website App Edition Each card in the Website App deck describes a common error or anti-pattern that allows systems to be vulnerable to attack. Vulnerabilities are arranged in domains as five suits with the additional Cornucopia suit ranging across these domains:

- Data Validation and Encoding
- Authentication
- Session Management
- Authorization
- Cryptography
- Cornucopia

To provide context the Cornucopia Website App cards reference other projects:

- OWASP Application Security Verification Standard (ASVS)
- OWASP Secure Coding Practices (SCP]) quick reference guide
- OWASP AppSensor
- Mitre's Common Attack Pattern Enumeration and Classification (CAPEC)
- SAFEcode

The SCP quick reference guide has now been incorporated as part of this Developer Guide.

Mobile App Edition Similarly to the website application deck, the mobile application deck has five domains/suits, with Cornucopia cross domain:

- Platform and Code

- Authentication and Authorization
- Network and Storage
- Resilience
- Cryptography
- Cornucopia

For context the Cornucopia Mobile App cards reference these other projects:

- OWASP Mobile Application Security Verification Standard (MASVS)
- OWASP Mobile Application Security Testing Guide (MASTG)
- Mitre's Common Attack Pattern Enumeration and Classification (CAPEC)
- SAFEcode

Ecommerce Website Edition This is the original Cornucopia deck and has the same domains/suits, including the Cornucopia cross domain suit, as the Website App Edition. Some of the vulnerabilities are specific to Ecommerce, but it references the same projects as the website edition.

Why use it? Cornucopia is useful for both requirements analysis and threat modeling, providing gamification of these activities within the development lifecycle. It is targeted towards agile development teams and provides a different perspective to these tasks.

The outcome of the game is to identify possible threats and propose remediations.

How to use Cornucopia The OWASP Spotlight series provides an excellent overview of Cornucopia and how it can be used for gamification: 'Project 16 - Cornucopia'.

Ideally Cornucopia is played in person using physical cards, with the development team and security architects in the same room. The application should already have been described by an architecture diagram or data flow diagram so that the players have something to refer to during the game.

The suggested order of play is:

1. Pre-sort: the deck, some cards may not be relevant for the web application
2. Deal: the cards equally to the players
3. Play: the players take turns to select a card
4. Describe: the player describes the possible attack using the card played
5. Convince: the other players have to be convinced that the attack is valid
6. Score: award points for a successful attack
7. Follow suit: the next player has to select a card from the same suit
8. Winner: the player with the most points
9. Follow up: each valid threat should be recorded and acted upon

Remember that the outcome of the game is to identify possible threats and propose remediations, as well as having a good time.

References

- AppSensor
- Application Security Verification Standard, ASVS
- Common Attack Pattern Enumeration and Classification, CAPEC
- Cornucopia
- Mobile Application Security Verification Standard, MASVS)
- Mobile Application Security Testing Guide, MASTG)
- Secure Coding Practices quick reference guide
- SAFEcode
- Spotlight on Cornucopia

4.1.5 LINDDUN GO

LINNDUN GO is a card game used to help derive privacy requirements during the software development life cycle. The LINNDUN GO card set can be downloaded as a PDF and then printed out.

What is LINDDUN GO? LINDDUN GO helps identify potential privacy threats based on the key LINDDUN threats to privacy:

- Linking
- Identifying
- Non-repudiation
- Detecting
- Data Disclosure
- Unawareness
- Non-compliance

LINNDUN GO is similar to OWASP Cornucopia in that it takes the form of a set of cards that can be used to gamify the process of identifying application privacy / security requirements. The deck of 33 cards are arranged in suits that match each category of threats to privacy, and there is a set of rules to structure the game sessions. Each LINDDUN GO card illustrates a single common privacy threat and suggested remediations.

Why use it? LINDDUN is an approach to threat modeling from a privacy perspective. It is a methodology that is useful to structure and guide the identification of threats to privacy, and also helps with suggestions for the mitigation of any threats.

LINDDUN GO gamifies this approach to privacy with a set of cards and rules to guide the identification process for threats to the privacy provided by the application. This is a change to other established processes and provides a different and useful perspective to the system.

How to use LINDDUN GO The idea for a LINDDUN GO is that it is played in person by a diverse team with as varied a set of viewpoints as possible. The advice from the LINDDUN GO 'getting started' instructions is that this team contains some or all of:

- domain experts
- system architects
- developers
- the Data Protection Officer (DPO)
- legal experts
- the Chief Information Security Officer (CISO)
- privacy champions

The application should have already been described by an architecture diagram or data flow diagram so that the players have something to refer to during the game. Download and printout the deck of cards.

Follow the set of rules to structure the game session, record the outcome and act on it. The outcome of the game is to identify possible privacy threats and propose remediations; as well as having a good time of course.

4.1.6 Threat Modeling toolkit

There is no one technique or tool that fits every threat modeling process. The process can be tactical or architectural, subjective or automated, attack tree or data flow diagram, all are perfectly valid for different organizations, teams and situations.

The OWASP Threat Modeling toolkit presentation at OWASP AppSec California 2018 gives a good overview of the range of concepts and techniques that can be regarded as threat modeling.

Advice on Threat Modeling In addition to the Threat Modeling toolkit there are OWASP community pages on Threat Modeling and the OWASP Threat Modeling Project, both of which provide context and overviews of threat modeling - in particular Shostack's Four Question Framework.

Threat Modeling step by step The Threat Modeling Process suggests steps that should be taken when threat modeling:

1. Decompose the Application
2. Determine and Rank Threats
3. Determine Countermeasures and Mitigation

and goes into detail on each concept :

- External Dependencies
- Entry Points
- Exit Points
- Assets
- Trust Levels
- Threat Categorization
- Threat Analysis
- Ranking of Threats
- Remediation for threats / vulnerabilities

The OWASP Threat Modeling Playbook (OTMP) is an OWASP Incubator project that describes how to create and nurture a good threat modeling culture within the organisation itself.

Cheat Sheets for Threat Modeling The OWASP series of Cheat Sheets is a primary source of advice and techniques on all things security, with the OWASP Threat Modeling Cheat Sheet and OWASP Attack Surface Analysis Cheat Sheet providing practical suggestions along with explanations of both the terminology and the concepts involved.

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 starting point for a checklist 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:

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 C4: Address Security form the Start and its cheatsheets for more context from the OWASP Top 10 Proactive Controls project, and use the lists below as suggestions for a checklist that has been tailored for the individual project.

1. System configuration

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

2. Cryptographic practices

1. Use peer reviewed and open solution cryptographic modules
2. All cryptographic functions used to protect secrets from the application user must be implemented on a trusted system
3. Cryptographic modules must fail securely
4. Ensure all random elements such as numbers, file names, UUID and strings are generated using the cryptographic module approved random number generator
5. Cryptographic modules used by the application are compliant to FIPS 140-2 or an equivalent standard
6. Establish and utilize a policy and process for how cryptographic keys will be managed
7. Ensure that any secret key is protected from unauthorized access
8. Store keys in a proper secrets vault as described below
9. Use independent keys when multiple keys are required
10. Build support for changing algorithms and keys when needed
11. Build application features to handle a key rotation

3. File management

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

References

- OWASP Application Security Verification Standard (ASVS)

- OWASP Mobile Application Security
- OWASP Top 10 Proactive Controls

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 C4: Address Security from the Start and its cheatsheets for more context from the OWASP Top 10 Proactive Controls project.

For technology specific checklists refer to the appropriate OWASP Cheat Sheets:

- AJAX_Security
- C-Based toolchain hardening
- Django security
- Django REST framework
- Docker security
- DotNet security
- GraphQL security
- Infrastructure as Code
- Java security
- Javascript management
- Kubernetes
- Laravel security
- Microservices security
- NPM security best practices
- Node.js security
- Node.js security for Docker
- PHP configuration
- REST APIs and how to assess them
- Ruby on Rails security
- Symfony framework
- Web Services
- XML security

and use them as the starting point for a checklist that is tailored for the technology used by the project.

In addition consider the following extra checks for frameworks and libraries.

1. Security Frameworks and Libraries

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

References

- OWASP Dependency Check

- OWASP Top 10 Proactive Controls

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: Validate all Input & Handle Exceptions and its cheatsheets for more context from the OWASP Top 10 Proactive Controls project, and use the list below as suggestions for a checklist that has been tailored for the individual project.

1. Secure queries

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

2. Secure configuration

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

3. Secure authentication

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

References

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

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 C3: Validate all Input & Handle Exceptions and its cheatsheets for more context from the OWASP Top 10 Proactive Controls project, and use the list below as suggestions for a checklist that has been tailored for the individual project.

1. Character encoding and canonicalization

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

2. 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.

1. Contextually encode all data returned to the client from untrusted sources
2. 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

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 C3: Validate All Input & Handle Exceptions and its cheatsheets for more context from the OWASP Top 10 Proactive Controls project, and use the list below as suggestions for a checklist that has been tailored for the individual project.

1. Syntax and semantic validity

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

2. Libraries and frameworks

1. Conduct all input validation on a trusted system [2]
2. Use a centralized input validation library or framework for the whole application
3. If the standard validation routine cannot address some inputs then use extra discrete checks
4. If any potentially hazardous input *must* be allowed then implement additional controls
5. Validate for expected data types using an allow-list rather than a deny-list

3. Validate serialized data

1. Implement integrity checks or encryption of the serialized objects to prevent hostile object creation or data tampering
2. Enforce strict type constraints during deserialization before object creation; typically a definable set of classes is expected
3. Isolate features that deserialize so that they run in very low privilege environments such as temporary containers
4. Log security deserialization exceptions and failures
5. Restrict or monitor incoming and outgoing network connectivity from containers or servers that deserialize
6. 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

[2] Secure Coding Practices checklist

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 C7: Implement Digital Identity and its cheatsheets for more context from the OWASP Top 10 Proactive Controls project, and use the list below as suggestions for a checklist that has been tailored for the individual project.

1. Authentication

1. Design access control authentication thoroughly up-front
2. Force all requests to go through access control checks unless public
3. Do not hard code access controls that are role based
4. Log all access control events
5. Use Multi-Factor Authentication (MFA) for sensitive or high value transactional accounts

2. Passwords

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

3. Cryptographic based authentication

1. Use the server or framework's session management controls
2. Session identifier creation must always be done on a trusted system
3. Session management controls should use well vetted algorithms that ensure sufficiently random session identifiers
4. Set the domain and path for cookies containing authenticated session identifiers to an appropriately restricted value for the site
5. Logout functionality should fully terminate the associated session or connection
6. Logout functionality should be available from all pages protected by authorization
7. Establish a session inactivity timeout that is as short as possible, based on balancing risk and business functional requirements
8. Disallow persistent logins and enforce periodic session terminations, even when the session is active
9. If a session was established before login, close that session and establish a new session after a successful login
10. Generate a new session identifier on any re-authentication
11. Do not allow concurrent logins with the same user ID
12. Do not expose session identifiers in URLs, error messages or logs
13. Implement appropriate access controls to protect server side session data from unauthorized access from other users of the server
14. Generate a new session identifier and deactivate the old one periodically
15. Generate a new session identifier if the connection security changes from HTTP to HTTPS, as can occur during authentication
16. Set the `secure` attribute for cookies transmitted over an TLS connection
17. 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: Choosing and Using Security Questions
- OWASP Cheat Sheet: Forgot Password
- OWASP Cheat Sheet: Multifactor Authentication
- OWASP Cheat Sheet: Password Storage
- OWASP Cheat Sheet: Session Management
- OWASP Top 10 Proactive Controls

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 C1: Implement Access Controls and its cheatsheets for more context from the OWASP Top 10 Proactive Controls project, and use the list below as suggestions for a checklist that has been tailored for the individual project.

1. Authorization

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

2. Access control

1. Enforce authorization controls on every request
2. Use only trusted system objects for making access authorization decisions
3. Use a single site-wide component to check access authorization
4. Access controls should fail securely
5. Deny all access if the application cannot access its security configuration information
6. Segregate privileged logic from other application code
7. 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
8. If long authenticated sessions are allowed, periodically re-validate a user's authorization
9. Implement account auditing and enforce the disabling of unused accounts
10. The application must support termination of sessions when authorization ceases

References

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

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 C2: Use Cryptography the proper way and its cheatsheets for more context from the OWASP Top 10 Proactive Controls project, and use the list below as suggestions for a checklist that has been tailored for the individual project.

1. Data protection

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

2. Memory management

1. Explicitly initialize all variables and data stores
2. Check that any buffers are as large as specified
3. Check buffer boundaries if calling the function in a loop and protect against overflow
4. Specifically close resources, don't rely on garbage collection
5. Use non-executable stacks when available
6. Properly free allocated memory upon the completion of functions and at all exit points
7. Overwrite any sensitive information stored in allocated memory at all exit points from the function
8. 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

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 and its cheatsheets for more context from the OWASP Top 10 Proactive Controls project, and use the list below as suggestions for a checklist that has been tailored for the individual project.

1. Security logging

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

2. Security logging design

1. Protect log integrity
2. Ensure log entries that include untrusted data will not execute as code in the intended log viewing interface or software
3. Restrict access to logs to only authorized individuals
4. Utilize a central routine for all logging operations
5. Forward logs from distributed systems to a central, secure logging service
6. Follow a common logging format and approach within the system and across systems of an organization
7. Synchronize across nodes to ensure that timestamps are consistent
8. All logging controls should be implemented on a trusted system
9. 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

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 C3: Validate all Input & Handle Exceptions and its cheatsheets for more context from the OWASP Top 10 Proactive Controls project, and use the list below as suggestions for a checklist that has been tailored for the individual project.

1. Errors and exceptions

1. Manage exceptions in a centralized manner to avoid duplicated try/catch blocks in the code
2. Ensure that all unexpected behavior is correctly handled inside the application
3. Ensure that error messages displayed to users do not leak critical data, but are still verbose enough to enable the proper user response
4. Ensure that exceptions logs give enough information for support, QA, forensics or incident response teams
5. Carefully test and verify error handling code
6. Do not disclose sensitive information in error responses, for example system details, session identifiers or account information
7. Use error handlers that do not display debugging or stack trace information
8. Implement generic error messages and use custom error pages
9. The application should handle application errors and not rely on the server configuration
10. Properly free allocated memory when error conditions occur
11. 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

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 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 a given MASVS control. This MAS 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? The OWASP MASVS is the industry standard for mobile application security. 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.

References

- Mobile Application Security (MAS) project
- MAS Checklist
- MAS Verification Standard (MASVS)
- OWASP Mobile Application Security cheat sheet

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

5.1 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

5.1.1 Top 10 Proactive Controls

The OWASP Top 10 Proactive Controls describes the most important controls and control categories that security architects and development teams should consider in web application projects.

What are the Top 10 Proactive Controls? The OWASP Top 10 Proactive Controls is a list of security techniques that should be considered for web applications. They are ordered by order of importance, with control number 1 being the most important:

- C1: Implement Access Control, ref Cheat Sheets
- C2: Use Cryptography the proper way, ref Cheat Sheets
- C3: Validate all Input & Handle Exceptions, ref Cheat Sheets
- C4: Address Security from the Start, ref Cheat Sheets
- C5: Secure By Default Configurations, ref Cheat Sheets
- C6: Keep your Components Secure, ref Cheat Sheets
- C7: Implement Digital Identity, ref Cheat Sheets
- C8: Leverage Browser Security Features, ref Cheat Sheets
- C9: Implement Security Logging and Monitoring, ref Cheat Sheets
- C10: Stop Server Side Request Forgery, ref Cheat Sheets

Why use them? The Proactive Controls are a well established list of security controls, first published in 2014 and revised in 2018, so considering these controls can be seen as best practice. Following best practice is always encouraged: at the very least an organization should avoid the avoidable exploits.

Putting these proactive controls in place can help remediate common security vulnerabilities, for example:

- Clickjacking
- Credential Stuffing
- Cross-site leaks
- Denial of Service (DoS) attacks
- DOM based XSS attacks including DOM Clobbering
- IDOR (Insecure Direct Object Reference)
- Injection including OS Command injection and XXE
- LDAP specific injection attacks
- Prototype pollution
- SSRF attacks
- SQL injection and the use of Query Parameterization
- Unvalidated redirects and forwards
- XSS attacks and XSS Filter Evasion

How to apply them The OWASP Spotlight series provides an overview of how to use this documentation project: 'Project 8 - Proactive Controls'.

During development of a web application, consider using each security control described in the sections of the Proactive Controls that are relevant to the application.

The OWASP Cheat Sheets have been indexed specifically for each Proactive Control, which can be used as additional information on implementing the control.

References

- OWASP Proactive Controls project
- OWASP Cheat Sheet Proactive Controls index

5.1.2 Go Secure Coding Practices

The OWASP Go Secure Coding Practices (Go-SCP) is a set of software secure coding practices for the Go programming language.

The Go-SCP documentation project is an OWASP Incubator Project that has enough long term support to achieve Lab status soon. The published document can be downloaded in various formats from the github repo.

What is Go-SCP? Go-SCP provides examples and recommendations to help developers avoid common mistakes and pitfalls, including code examples in Go that provide practical guidance on implementing the recommendations. Go-SCP covers the OWASP Secure Coding Practices Quick Reference Guide topic-by-topic:

- Input Validation
- Sanitization Output Encoding
- Authentication and Password Management
- Session Management
- Access Control
- Cryptographic Practices
- Error Handling and Logging
- Data Protection
- Communication Security
- System Configuration
- Database Security
- File Management
- Memory Management
- General Coding Practices

The Go Secure Coding Practices book is available in various formats:

- PDF
- ePub
- DocX
- MOBI

Why use Go-SCP? Development teams often need help and support in getting the security right for web applications, and part of this help comes from secure coding guidelines and best practices. Go-SCP provides this guidance for a wide range of secure coding topics as well as providing practical code examples for each coding practice.

How to use Go-SCP? The primary audience of the Go Secure Coding Practices Guide is developers, particularly those with previous experience in other programming languages.

Download the Go-SCP document in one of the formats: PDF, ePub, DocX and MOBI. Refer to the specific topic chapter and then use the example Go code snippets for practical guidance on secure coding using Go.

5.1.3 Cheat Sheet Series

The OWASP Cheat Sheet Series provide a concise collection of high value information on a wide range of specific application security topics. The cheat sheets have been created by a community of application security professionals who have expertise in each specific topic.

The Cheat Sheet Series documentation project is an OWASP Flagship Project which is constantly being kept up to date.

What are the Cheat Sheets? The OWASP Cheat Sheets are a common body of knowledge created by the software security community for a wide audience that is not confined to the security community.

The Cheat Sheets are a series of self contained articles written by the security community on a specific subject within the security domain. The range of topics covered by the cheat sheets is wide, almost from A to Z: from AJAX Security to XS (Cross Site) vulnerabilities. Each cheat sheet provides an introduction to the subject and provides enough information to understand the basic concept. It will then go on to describe its subject in more detail, often supplying recommendations or best practices.

Why use them? The OWASP Cheat Sheet Series provide developers and security engineers with most, and perhaps all, of the information on security topics that they will need to do their job. In addition the Cheat Sheets are regarded as authoritative: it is recommended to follow the advice in these Cheat Sheets. If a web application does not follow the recommendations in a cheat sheet, for example, then the implementation could be challenged during testing or review processes.

How to use them The OWASP Spotlight series provides a good overview of using this documentation: 'Project 4 - Cheat Sheet Series'.

There are many cheat sheets in the OWASP Cheat Sheet Series; 91 of them as of March 2024 and this number is set to increase. The OWASP community recognises that this may become overwhelming at first, and so has arranged them in various ways:

- Alphabetically
- Indexed to follow the ASVS project or the MASVS project
- Arranged in sections of the OWASP Top 10 or the OWASP Proactive Controls

The cheat sheets are continually being updated and are always open to contributions from the security community.

References

- OWASP Cheat Sheet Series
- OWASP Cheat Sheet ASVS index
- OWASP Cheat Sheet MASVS index
- OWASP Cheat Sheet Proactive Controls index
- OWASP Cheat Sheet Top 10 index
- OWASP Cheat Sheet project

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 application in a production environment. This can be achieved by Software Composition Analysis (SCA) to identify the third party dependencies.

A Software Bill of Materials (SBOM) provides a record of the dependencies within the system / application, and 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 specific Common Vulnerability and Exposure (CVE), or what CVEs are present in a particular application.

Sections:

{: height="150px" }

5.2.1 Dependency-Check

OWASP Dependency-Check is a tool that provides Software Composition Analysis (SCA) from the command line. It identifies the third party libraries in a web application project and checks if these libraries are vulnerable using the NVD database.

Dependency-Check is an OWASP Flagship project and can be downloaded from the github releases area. Dependency-Check was started in September 2012 and since then has been continuously supported with regular releases.

What is Dependency-Check? Dependency-Check is a Software Composition Analysis (SCA) tool that attempts to detect publicly disclosed vulnerabilities contained within a project's dependencies. It does this by determining if there is a Common Platform Enumeration (CPE) identifier for a given dependency.

The core engine contains a series of analyzers that inspect the project dependencies and identify the CPE for the given dependency. If a CPE is identified then it is cross referenced to the NIST CVE database and any associated Common Vulnerability and Exposure (CVE) entries are listed in the report.

Dependency-Check's core analysis engine can be used as:

- an Ant Task
- a Command Line Tool
- Gradle Plugin
- Jenkins Plugin
- Maven Plugin
- SBT Plugin

Why use it? Checking for vulnerable components, 'A06 Vulnerable and Outdated Components', is in the OWASP Top Ten and is one of the most straight-forward and effective security activities to implement. The Dependency-Check tool provides checks for vulnerable components that can be run from the command line.

This is useful for development teams; the ability to check for vulnerable application components from the command line gives immediate feedback to the developer without having to wait for a pipeline to run.

Dependency-Check also provides plugins to check for vulnerable components for CI/CD pipelines.

How to use it The OWASP Spotlight series provides an example of the risks involved in using out of date and vulnerable libraries, and how to use Dependency-Check: 'Project 2 - OWASP Dependency Check'.

Refer to the Dependency-Check documentation to get started running from the command line:

- ensure Java is installed, for example from Eclipse Adoptium
- download and unzip the latest Dependency-Check release

- navigate to the Dependency-Check 'bin' directory and run, using threat Dragon as an example: `./dependency-check.sh --project "Threat Dragon" --scan ~/github/threat-dragon`
- open the html output file and act on the findings

The command line is useful for immediate debugging development. Depending on what automation environment is in place a plugin can also be installed into a pipeline which can then generate the SCA reports.

References

- OWASP Dependency-Check project
- OWASP Dependency-Check documentation
- OWASP CI/CD Security Cheat Sheet
- OWASP Top 10

5.2.2 Dependency-Track

OWASP Dependency-Track is an intelligent platform for Component Analysis including Third Party Software. It allows organizations to identify and reduce risk in the software supply chain using its ability to analyse a Software Bill of Materials (SBOM).

The Dependency-Track is an OWASP Flagship project and can be installed using a docker-compose file from the Dependency-Track website.

What is Dependency-Track? The Dependency-Track tool provides an organization with a dashboard to analyze, oversee and control the components for all of its projects. It tracks component usage across every application in an organizations portfolio by analyzing exports from multiple projects within the organization, via CycloneDX SBOMs and Vulnerability Exploitability Exchange.

It provides full-stack support for all types of component, including hardware and services. Dependency-Track identifies multiple forms of risk, including components with known vulnerabilities, by integrating with multiple sources of vulnerability intelligence such as the National Vulnerability Database (NVD), GitHub advisories and others.

It has built-in repository support for various repository types, and will provide risk and compliance for security, risk and operations. See the Documentation for more information on the features provided by Dependency-Track.

Why use it? By leveraging the capabilities of Software Bill of Materials (SBOM), Dependency-Track provides capabilities that traditional Software Composition Analysis (SCA) solutions are unlikely to achieve.

The Dependency-Track dashboard has the ability to analyze all software projects within an organization. It integrates with numerous notification platforms, for example Slack and Microsoft Teams, and can feed results to various vulnerability aggregator tools such as DefectDojo or Fortify.

Dependency-Track is feature rich, it provides integrations and features that most organizations will need; see the Documentation Introduction for a full list of these features.

How to use it The OWASP Spotlight series provides an overview of tracking dependencies and inspecting SBOMs using Dependency-Track: 'Project 15 - OWASP Dependency-Track'.

Follow the getting started guide to install the Dependency-Track tool, using the recommended deployment of a Docker container.

Although Dependency-Track will run with its default configuration it should should be configured for the organization's specific needs. The Dependency-Track configuration file is important for optimally running the tool but this is outside the scope of the Developer Guide - see the Dependency-Track documentation for a step by step guide to this configuration process.

5.2.3 CycloneDX

OWASP CycloneDX is a full-stack Bill of Materials (BOM) standard that provides advanced supply chain capabilities for cyber risk reduction. This project is one of the OWASP flagship projects.

What is CycloneDX? CycloneDX is a widely used standard for various types of Bills of Materials. It provides an organization's supply chain with software security risk reduction. The specification supports:

- Software Bill of Materials (SBOM)
- Software-as-a-Service Bill of Materials (SaaSBOM)
- Hardware Bill of Materials (HBOM)
- Machine-learning Bill of Materials (ML-BOM)
- Manufacturing Bill of Materials (MBOM)
- Operations Bill of Materials (OBOM)
- Bill of Vulnerabilities (BOV)
- Vulnerability Disclosure Reports (VDR)
- Vulnerability Exploitability eXchange (VEX)
- Common Release Notes format
- Syntax for Bill of Materials linkage (BOM-Link)

The CycloneDX project provides standards in XML, JSON, and Protocol Buffers. There is a large collection of official and community supported tools that consume and create CycloneDX BOMs or interoperate with the CycloneDX standard.

Why use it? CycloneDX is a very well established standard for SBOMs and various other types of BOM. There is a huge ecosystem built around CycloneDX and it is used globally by many companies. In addition SBOMs are mandatory for many industries and various governments - at some point every organization will have to provide SBOMs for their customers and CycloneDX is an accepted standard for this.

CycloneDX also provides standards for other types of BOMs that may be required in the supply chain along with standards for release notes and responsible disclosure. It is useful to use CycloneDX throughout the supply chain as it promotes interoperability between the various tools.

How to use it The OWASP Spotlight series provides an overview of CycloneDX along with the a demonstration of using SBOMs: 'Project 21 - OWASP CycloneDX'.

CycloneDX is an easy to understand standard that can be augmented to suit all parts of a supply chain, and there are many tools (more than 220 as of February 2024) that interoperate with CycloneDX.

The easiest way to use CycloneDX is to select tools from this list for any of the supported BOM types, with both proprietary/commercial and open source tools included in the list. A common example is for a customer to request that an SBOM is provided for a web application, and various tools can be chosen that are able to export the SBOM in various formats.

5.3 Secure libraries

The use of secure libraries is part of the technology management that helps to fulfil security requirements. Standard libraries enable the adoption of common design patterns and security solutions, and provide standardized technologies and frameworks that can be used throughout 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:

5.3.1 Enterprise Security API library

The OWASP Enterprise Security API (ESAPI) library is a security control library for web applications written in Java.

The ESAPI library is an OWASP Lab project that is under active development for Java security controls with regular releases.

What is the ESAPI library? The OWASP Enterprise Security API (ESAPI) library provides a set of security control interfaces which define types of parameters that are passed to the security controls.

The ESAPI is an 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 help programmers retrofit security into existing Java applications, and the library also serves as a solid foundation for new development.

Why use it? The use of the ESAPI Java library is not easy to justify, although its use should certainly be considered. The engineering decisions a development team will need to make when using ESAPI are discussed in the 'Should I use ESAPI?' documentation.

For new projects or for modifying an existing project then alternatives should be strongly considered:

- Output encoding: OWASP Java Encoder project
- General HTML sanitization: OWASP Java HTML Sanitizer
- Validation: JSR-303/JSR-349 Bean Validation
- Strong cryptography: Google Tink or Keyczar
- Authentication & authorization: Apache Shiro, authentication using Spring Security
- CSRF protection: OWASP CSRFGuard project

Consideration could be given for using ESAPI if multiple security controls provided by this library are used in a project, it then may be useful to use the monolithic ESAPI library rather than multiple disparate class libraries.

How to use it If the engineering decision is to use the ESAPI library then it can be downloaded as a Java Archive (.jar) package file. There is a reference implementation for each security control.

References

- ESAPI for Java
- ESAPI documentation
- ESAPI project
- OWASP Java Encoder project
- OWASP Java HTML Sanitizer
- Spring Security

5.3.2 CSRFGuard library

OWASP CSRFGuard is a security control that helps protect Java applications against Cross-Site Request Forgery (CSRF) attacks.

The CSRFGuard Builder/Breaker Tool project is an OWASP Production Project and is being actively maintained by a pool of international volunteers.

What is CSRFGuard? 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 are submitted with the corresponding HTTP request. CSRFGuard ensures the token is present and is valid for the current HTTP request.

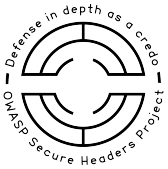
Why use it? The OWASP CSRFGuard library is widely used for Java applications, and will help mitigate against CSRF.

How to use it Pre-compiled versions of the CSRFGuard library can be downloaded from the Maven Central repository or the OSS Sonatype Nexus repository.

Follow the instructions to build CSRFGuard into the Java application using Maven.

References

- OWASP CSRFGuard
- OWASP Cross-Site Request Forgery Prevention Cheat Sheet



5.3.3 OWASP Secure Headers Project

The OWASP Secure Headers Project (OSHP) provides information on HTTP response headers to increase the security of a web application.

The OSHP documentation project is an OWASP Lab Project and raises awareness of secure headers and their use.

What is OSHP? The OSHP project) provides explanations for the HTTP response headers that an application can use to increase the security of the application. Once set the HTTP response headers can restrict modern browsers from running into easily preventable vulnerabilities.

OSHP contains guidance and downloads on:

- Response headers explanations and usage
- Links to individual browser support
- Guidance and best practices
- Technical resources in the form of tools and documents
- Code snippets to help working with HTTP security headers

Why use it? The OSHP is a documentation project that explains the reasoning and usage of HTTP response headers. It is the go-to document for guidance and best practices; the information on HTTP response headers is the best advice, in one location, and is kept up to date.

How to use it The OWASP Spotlight series provides an overview of this project and its uses: 'Project 24 - OWASP Security Headers Project'.

OSHP provides links to development libraries that provide for secure HTTP response headers in a range of languages and frameworks: DotNet, Go, HAPI, Java, NodeJS, PHP, Python, Ruby, Rust. The OSHP also lists various tools useful for inspection, analysis and scanning of HTTP response headers.

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, and also includes other review and evaluation activities.

Verification activities should include:

- Architecture assessment, validation and mitigation
- Requirements-driven testing
- 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 MAS Testing Guide
6.1.3 Application Security Verification Standard
6.2 Tools
6.2.1 DAST tools
6.2.2 Amass
6.2.3 Offensive Web Testing Framework
6.2.4 Nettacker
6.2.5 OWASP Secure Headers Project
6.3 Frameworks
6.3.1 secureCodeBox
6.4 Vulnerability management
6.4.1 DefectDojo

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:

6.1.1 Web Security Testing Guide
6.1.2 MAS Testing Guide
6.1.3 Application Security Verification Standard

6.1.1 Web Security Testing Guide

The OWASP Web Security Testing Guide (WSTG) is a comprehensive guide to testing the security of web applications and web services.

The WSTG documentation project is an OWASP Flagship Project and can be accessed as a web based document.

What is WSTG? The Web Security Testing Guide (WSTG) document is a comprehensive guide to testing the security of web applications and web services. The WSTG provides a framework of best practices commonly used by external penetration testers and organizations conducting in-house testing.

The WSTG document describes a suggested web application test framework and also provides general information on how to test web applications with good testing practice.

The tests are split out into domains:

1. Configuration and Deployment Management
2. Identity Management
3. Authentication
4. Authorization
5. Session Management
6. Input Validation
7. Error Handling
8. Weak Cryptography
9. Business Logic
10. Client-side
11. API

Each test in each domain has enough information to understand and run the test including:

- Summary
- Test objectives
- How to test
- Suggested remediation
- Recommended tools and references

The tests are identified with a unique reference number, for example 'WSTG-APIT-01' refers to the first test in the 'API Testing' domain provided in the WSTG document. These references are widely used and understood by the test and security communities.

The WSTG also provides a suggested Web Security Testing Framework which can be tailored for a particular organization's processes or can provide a generally accepted reference framework.

Why use it? The WSTG document is widely used and has become the defacto standard on what is required for comprehensive web application testing. An organization's security testing process should consider the contents of the WSTG, or have equivalents, which help the organization conform to general expectation of the security community. The WSTG reference document can be adopted completely, partially or not at all; according to an organization's needs and requirements.

How to use it The OWASP Spotlight series provides an overview of how to use the WSTG: 'Project 1 - Applying OWASP Testing Guide'.

The WSTG is accessed via the online web document. The section on principles and techniques of testing provides foundational knowledge, along with advice on testing within typical Secure Development Lifecycle (SDLC) and penetration testing methodologies.

The individual tests described in the various testing domains should be selected or discarded as necessary; not every test will be relevant to every web application or organizational requirement, and the tests should be tailored to provide at least the minimum test coverage while not expending too much test effort.

References

- OWASP Web Security Testing Guide (WSTG) project
- WSTG downloads

6.1.2 MAS testing guide

The MAS Verification Standard (MASVS) explains the processes, techniques and tools used for security testing a mobile application.

The OWASP MAS project provides the Mobile Application Security Testing Guide (MASTG) which describes technical processes that can be used for verification of the mobile application controls .

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 application 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 OWASP Spotlight series provides an overview of using the MASTG: 'Project 13 - OWASP Mobile Security Testing Guide (MSTG)'.

The MASTG project contains a large number of resources that can be used during verification and testing of mobile applications; pick and choose the resources that are applicable to specific 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

References

- OWASP Mobile Application Security (MAS) project
- OWASP MAS Testing Guide (MASTG)
- OWASP MAS Checklists
- OWASP MAS Verification Standard (MASVS)
- OWASP Mobile Application Security cheat sheet

6.1.3 Application Security Verification Standard

The Application Security Verification Standard (ASVS) is a long established OWASP flagship project, and is widely used as a guide during the verification of web applications.

It can be downloaded from the OWASP project page in various languages and formats: PDF, Word, CSV, XML and JSON. Having said that, the recommended way to consume the ASVS is to access the github markdown pages directly - this will ensure that the latest version is used.

What is ASVS? The ASVS is an open standard that sets out the coverage and 'level of rigor' expected when it comes to performing web application security verification. The standard also provides a basis for testing any technical security controls that are relied on to protect against vulnerabilities in the application.

The ASVS is split into various sections:

- V1 Architecture, Design and Threat Modeling
- V2 Authentication
- V3 Session Management
- V4 Access Control
- V5 Validation, Sanitization and Encoding
- V6 Stored Cryptography
- V7 Error Handling and Logging
- V8 Data Protection
- V9 Communication
- V10 Malicious Code
- V11 Business Logic
- V12 Files and Resources
- V13 API and Web Service
- V14 Configuration

The ASVS defines three levels of security verification:

1. applications that only need low assurance levels; these applications are completely penetration testable
2. applications which contain sensitive data that require protection; the recommended level for most applications
3. the most critical applications that require the highest level of trust

Most applications will aim for Level 2, with only those applications that perform high value transactions, or contain sensitive medical data, aiming for the highest level of trust at level 3.

Why use it? The ASVS is used by many organizations as a basis for the verification of their web applications. It is well established, the earlier versions were written in 2008, and it has been continually supported since then.

The ASVS is comprehensive, for example version 4.0.3 has a list of 286 verification requirements, and these verification requirements have been created and agreed to by a wide security community. Using the ASVS as a guide provides a firm basis for the verification process.

How to use it The OWASP Spotlight series provides an overview of the ASVS and its uses: 'Project 19 - OWASP Application Security Verification standard (ASVS)'.

The ASVS should be used as a guide for the verification process, with the appropriate level of verification chosen from:

- Level 1: First steps, automated, or whole of portfolio view
- Level 2: Most applications
- Level 3: High value, high assurance, or high safety

Use the ASVS as guidance rather than trying to implement every possible control. Tools such as SecurityRAT can help create a more manageable subset of the ASVS requirements.

The ASVS guidance will help developers build security controls that will satisfy the application security requirements.

The OWASP Cheat Sheets have been indexed specifically for each section of the ASVS, which can be used as documentation to help decide if a requirements category is to be included in verification.

References

- OWASP Application Security Verification Standard (ASVS)
- OWASP ASVS Index
- OWASP SecurityRAT project

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:

Dynamic application security testing (DAST) represents a non-functional testing process to identify security weaknesses and vulnerabilities in applications. The testing process can be carried out manually or be automated. Manual assessment of an application involves human intervention to identify security flaws which might slip from an automated tool. Usually business logic errors, race condition checks, and certain zero-day vulnerabilities can only be identified using manual assessments.

6.2.1 DAST tools

DAST tools are programs which communicates with a web application through the web front-end in order to identify potential security vulnerabilities in the web application and architectural weaknesses. It performs a black-box test. Unlike static application security testing tools, DAST tools do not have access to the source code and therefore detect vulnerabilities by actually performing attacks.

Different DAST tools The OWASP Community projects contains a list of DAST tools that can be used to conduct DAST. All of these tools have their own strengths and weaknesses. If you are interested in the effectiveness of DAST tools, check out the OWASP Benchmark project, which attempts to scientifically measure the effectiveness of all types of vulnerability detection tools, including DAST.

Why use it? The big advantage of these types of tools are that they can scan year-round to be constantly searching for vulnerabilities. With new vulnerabilities being discovered regularly this allows companies to find and patch vulnerabilities before they can become exploited.

Cons Because these tools does dynamic testing, it cannot cover 100% of the source code of the application and then, the application itself. The penetration tester should look at the coverage of the web application or of its attack surface to know if the tool was configured correctly or was able to understand the web application.

References

- Dynamic application security testing
- Vulnerability Scanning Tools

6.2.2 Amass

The OWASP Amass is a tool that provides attack surface management for an organization's web sites and applications. It used during penetration testing for network mapping of attack surfaces and external asset discovery by integrating various existing security tools.

The Amass breaker/tool project is an OWASP Flagship Project and installers can be downloaded from the project's github repository release area.

What is Amass? Amass is a command line tool that provides information on an organization's web sites, using various open source information gathering tools and active reconnaissance techniques.

It is run from the command line with subcommands :

1. 'amass intel' collects intelligence on the target organization
2. 'amass enum' performs DNS enumeration and network mapping to populate the results database
3. 'amass db'

Each command comes with a wide set of options that controls the tools used and the format of the findings.

Why use it? Amass is an important tool for security test teams. Amass is included in the Kali Linux distribution, which is widely used by penetration testing teams, with Amass providing a straightforward way of running a wide set of reconnaissance and enumeration tools.

In addition Amass is an easily used tool that is available to both legitimate test teams and malicious actors. It is very likely that any given organization has been scanned and enumerated by Amass at some point, either maliciously or legitimately, so it is important that the tool is run to determine what information a malicious actor can obtain.

How to use it If Kali Linux is being used then Amass comes ready installed, otherwise a wide set of installers is provided for other platforms.

The extensive Amass tutorial provides the best way of learning to use Amass and its features.

6.2.3 Offensive Web Testing Framework

OWASP Offensive Web Testing Framework (OWTF) is a penetration test tool that provides pen-testers with a framework for organising and running security test suites. It also helps align the pen-testing to various standards and security guides, allowing the testing to be more creative and comprehensive.

The OWTF defender/tool project is an OWASP Flagship Project and can be downloaded from the project's github repository release area.

What is OWTF? The OWTFtool is a penetration test framework used to organise and run suites of security and pen-testing tools. It is designed to be run on Kali Linux; it can also be run on MacOS but with some modification of scripts and paths.

OWTF is very much a penetration tester's tool; there is an expectation that the user has a reasonable expertise and grasp of penetration testing environments and tools. The documentation on installing and running OWTF requires is not extensive, and some in-depth knowledge on the target system is required to configure the tool.

Why use it? OWTF is easily configurable and plugins can be created or new tests added using the configuration files. It can be quickly installed on Kali Linux, a distribution of Ubuntu that is widely used by pen-testers, and allows for a whole suite of tests to be directed against the target.

How to use it The OWTF documentation is relatively old, last updated in 2016, and the install instructions may need adapting to run on MacOS or Kali.

6.2.4 Nettacker

OWASP Nettacker is a command line utility for automated network and vulnerability scanning. It can be used during penetration testing for both internal and external security assessments of networks.

The Nettacker breaker/tool project is an OWASP Incubator Project; the latest version can be downloaded from the project's github repository.

What is Nettacker? Nettacker is an automated penetration testing tool. It is used to scan a network to discover nodes and servers on the network including subdomains. Nettacker can then identify servers, services and port numbers in use.

Nettacker is a modular python application that that can be extended with other scanning functions. The many modules available are grouped into domains:

- Scan modules for reconnaissance
- Vulnerability modules that attempt specific exploits
- Brute force modules

Nettacker runs on Windows, Linux and MacOS.

Why use it? Nettacker is easy to use from the command line, making it easy to use in scripts, and also comes with a web browser interface for easy navigation of the results. This makes it a quick and reliable way to gain information from a network.

Nettacker can be used both for auditing purposes and also for penetration testing.

How to use it The OWASP Spotlight series provides an overview of attack surface management using Nettacker: 'Project 11 - Nettacker'.

The documentation for Nettacker is provided in the repository wiki pages; follow these instructions to install it.

Nettacker is a flexible and modular scanning tool that can be used in many ways and with many options. The best way to start using it is by following the introduction video and then taking it from there.

6.2.5 Secure Headers Project

The OWASP Secure Headers Project (OSHP) provides information on HTTP response headers to increase the security of a web application.

The OSHP documentation project is an OWASP Lab Project and raises awareness of secure headers and their use.

What is OSHP? The OSHP project) provides explanations for the HTTP response headers that an application can use to increase the security of the application. Once set the HTTP response headers can restrict modern browsers from running into easily preventable vulnerabilities.

OSHP contains guidance and downloads on:

- Response headers explanations and usage
- Links to individual browser support
- Guidance and best practices
- Technical resources in the form of tools and documents
- Code snippets to help working with HTTP security headers

Why use it? The OSHP is a documentation project that explains the reasoning and usage of HTTP response headers. It is the go-to document for guidance and best practices; the information on HTTP response headers is the best advice, in one location, and is kept up to date.

How to use it The OWASP Spotlight series provides an overview of this project and its uses: 'Project 24 - OWASP Security Headers Project'.

OSHP documents various tools useful for inspection, analysis and scanning of HTTP response headers:

- hsecscan
- humble
- SecurityHeaders.com
- Mozilla Observatory
- Recx Security Analyser
- testssl.sh
- DrHEADer
- csp-evaluator

OSHP also provides links to development libraries that provide for secure HTTP response headers in a range of languages and frameworks.

6.3 Verification frameworks

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

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 secureCodeBox

{: height="180px" }

6.3.1 secureCodeBox OWASP secureCodeBox is a kubernetes based modularized toolchain that provides continuous security scans of an organizations' projects and web applications.

The secureCodeBox builder/tool project is an OWASP Lab Project and is installed using the Helm ChartMuseum.

What is secureCodeBox? OWASP secureCodeBox combines existing security tools from the static analysis, dynamic analysis and network analysis domains. It uses these tools to provide a comprehensive overview of threats and vulnerabilities affecting an organization's network and applications.

OWASP secureCodeBox orchestrates a range of security-testing tools in various domains:

- Container analysis:
  - Trivy container vulnerability scanner
  - Trivy SBOM container dependency scanner
- Content Management System analysis:
  - CMSeeK detecting the Joomla CMS and its core vulnerabilities
  - Typo3Scan detecting the Typo3 CMS and its installed extensions
  - WPScan Wordpress vulnerability scanner
- Kubernetes analysis:
  - Kube Hunter vulnerability scanner
  - Kubeaudit configuration Scanner
- Network analysis:
  - Amass subdomain enumeration scanner
  - doggo DNS client
  - Ncrack network authentication bruteforcing
  - Nmap network discovery and security auditing
  - Whatweb website identification
- Repository analysis:
  - Git Repo Scanner discover Git repositories
  - Gitleaks find potential secrets in repositories
  - Semgrep static code analysis
- SSH/TLS configuration and policy scanning with SSH-audit and SSLyze
- Web Application analysis:
  - ffuf web server and web application elements and content discovery
  - Nikto web server vulnerability scanner
  - Nuclei template based vulnerability scanner.
  - Screenshooter takes screenshots of websites

- ZAP Advanced web application & OpenAPI vulnerability scanner

Other tools may be added over time.

Why use it? OWASP secureCodeBox provides the power of leading open source security testing tools with a multi-scanner platform. This provides the ability to run routine scans continuously and automatically on an organization's network infrastructure and applications.

OWASP secureCodeBox is fully scalable and can be separately configured for multiple teams, systems or clusters.

How to use it OWASP secureCodeBox runs on Kubernetes and uses Helm to install using the Helm ChartMuseum. There is an excellent 'Starting your First Scans' guide to getting started with secureCodeBox, with the rest of the documentation providing clear information on configuring and running secureCodeBox.

References

- OWASP secureCodeBox
- Kubernetes container orchestration
- Helm package manager for Kubernetes

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 using security unit tests to provide regression testing gives some degree of confidence that applications are not vulnerable to known exploits.

Sections:

6.4.1 DefectDojo

{:
height="160px" }

6.4.1 DefectDojo

OWASP DefectDojo is a DevSecOps tool for vulnerability management. It provides one platform to orchestrate end-to-end security testing, vulnerability tracking, deduplication, remediation, and reporting.

DefectDojo is an OWASP Flagship project and is well established; the project was started in 2013 and has been in continuous development / release since then.

What is DefectDojo? DefectDojo is an open source vulnerability management tool that streamlines the testing process by integration of templating, report generation, metrics, and baseline self-service tools.

DefectDojo streamlines the testing process through several 'models' that an admin can manipulate with Python code. The core models include:

- engagements
- tests
- findings

DefectDojo has supplemental models that facilitate :

- metrics
- authentication
- report generation
- tools

A good introduction to DefectDojo is the We Hack Purple discussion between Matt Tesauro and Tanya Janca.

Why use it? DefectDojo integrates with many open-source and proprietary/commercial tools from various domains:

- Dynamic Application Security Testing (DAST)
- Static Application Security Testing (SAST)
- Software Composition Analysis (SCA)
- Software Bills of Materials (SBOMs)
- Scanning of infrastructure and APIs

It also integrates with the Threagile Threat Modeling tool, and with time more integrations with threat modeling tools will become available.

How to use it Testing or installing DefectDojo is straight forward using the installation instructions. An instance of DefectDojo can be setup using docker compose along with the associated scripts that handle the dependencies, configure the database, create users and so on. Refer to the DefectDojo documentation for all the information on alternative deployments, setting up, usage and integrations.

References

- OWASP DefectDojo
- We Hack Purple discussion
- Threagile Threat Modeling

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:

7.1 Vulnerable Applications
7.1.1 Juice Shop
7.1.2 WebGoat
7.1.3 PyGoat
7.1.4 Security Shepherd
7.2 Secure Coding Dojo
7.3 Security Knowledge Framework
7.4 SamuraiWTF
7.5 OWASP Top 10 project
7.6 Mobile Top 10
7.7 API Top 10
7.8 WrongSecrets
7.9 OWASP Snakes and Ladders

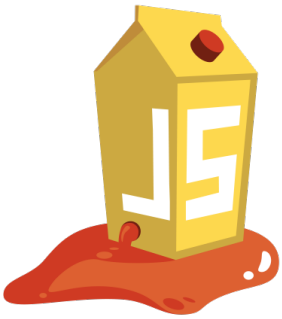
7.1 Vulnerable Applications

Vulnerable applications are useful for the Training and Education activities described 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 intentionally-vulnerable applications provide a safe environment where various vulnerable targets can be attacked. This provides practice in using various penetration tools available to a tester, without the risk of attack traffic triggering intrusion detection systems. The OWASP Vulnerable Web Applications Directory Project (VWAD) provides a comprehensive list of available intentionally-vulnerable web applications:

- Vulnerable mobile applications
- Offline vulnerable web applications
- Containerized vulnerable web applications
- vulnerable web applications available Online

Sections:



7.1.1 Juice Shop

The OWASP flagship project Juice Shop is a deliberately insecure web application. Juice Shop encompasses vulnerabilities from the entire OWASP Top Ten along with many other security flaws found in real-world applications.

What is Juice Shop? Juice Shop is an Open Source web application that is free to download and use, and is intentionally insecure. It is easy to get started with Juice Shop; it includes Hacking Instructor scripts with an optional tutorial mode to guide newcomers through several challenges while explaining the underlying vulnerabilities.

Juice Shop is easily installed using a Docker image and runs on Windows/Mac/Linux as well as all major cloud providers. There are various ways to run Juice Shop:

- From source
- Packaged Distributions
- Docker Container
- Vagrant
- Amazon EC2 Instance
- Azure Container Instance
- Google Compute Engine Instance

Juice Shop is written in JavaScript using Node.js, Express and Angular.

Why use it? Juice Shop has several uses:

- As the basis for security training programs, with integration for other training systems via a WebHook
- As practice for pentesters and hackers, including many built in coding challenges
- To provide awareness demos, with customizable rebranding for specific corporations or customers
- Support for Capture the Flag (CTF) events using flag codes
- As a guinea pig for security tools

For example pentesting proxies or security scanners can use Juice Shop as a 'guinea pig' application to check how well their tools cope with JavaScript-heavy application frontends and REST APIs.

How to use it There is no 'one way' to use Juice Shop, and so a good starting point is the overview of Juice Shop provided by the OWASP Spotlight series: 'Project 25 - OWASP Juice Shop'.

Get started by downloading and installing the Docker image. The Docker daemon will have to be running to do this; get the Docker Engine from the download site.

```
docker pull bkimminich/juice-shop
docker run --rm -p 3000:3000 bkimminich/juice-shop
```

Using a browser access `http://localhost:3000/#/` and note that you are now interacting with a deliberately insecure 'online' shopping web application, so be suspicious of everything you see :)

Once Juice Shop is running the next step is to follow the Official Companion Guide that can be downloaded from the Juice Shop shop. This guide provides overviews of each Juice Shop application vulnerability and includes hints on how to spot and exploit them. In the appendix there is a complete step-by-step solution to every challenge for when you are stuck or just curious.

7.1.2 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 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 Java-based 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. Whenever you need a deliberately vulnerable web application running in a self contained and safe environment then WebGoat is 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 'Kali1234', 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 'Kali1234'. 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 available attack tools against WebGoat.

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.

References

- OWASP WebGoat and WebWolf
- Docker

7.1.3 PyGoat

The OWASP PyGoat project is an intentionally insecure web application, and is written in python using the Django framework. PyGoat is used to practice attacking a python-based web application in an isolated and secure environment

PyGoat is a relatively new OWASP project, its first commit was in May 2020, and although it is presently an Incubator project it should soon gain Lab project status.

What is PyGoat? The purpose of PyGoat is to give both developers and testers an isolated platform for learning how to test applications and how to code securely. It provides examples of traditional web application exploits that follow the OWASP Top Ten vulnerabilities, as well as providing a vulnerable web application for further exploitation / testing.

PyGoat also provides a view of the python source code to determine where the mistake was made that caused the vulnerability. This allows you to make changes to the web application and test whether these changes have secured it.

PyGoat can be installed from source repository via scripts or pip, from a Docker hub image, or using a Docker image built locally.

Why use it? PyGoat is an easy to use demonstration and learning platform that provides a secure way to try and attack web applications. It is less fully featured than either Juice Shop or WebGoat, and this makes it a simple and direct learning experience. PyGoat provides example labs for the complete OWASP Top Ten (both 2021 and 2017 versions) along with explanatory notes.

PyGoat also provides a python / Django web application which complements Juice Shop's Node.js and WebGoat's Java applications; it is important to learn how to attack all of these frameworks.

So if you are looking for a direct and easy way to start attacking a vulnerable web application then PyGoat is one of the first to consider.

How to use it The easiest way to run PyGoat is by downloading the Docker image and running it as a Docker container. The Docker daemon will have to be running to do this, so get the Docker Engine from the download site.

Follow the instructions from the Docker hub project page:

```
docker pull pygoat/pygoat
docker run --rm -p 8000:8000 pygoat/pygoat
```

The internal container port 8000 is mapped to external port 8000, so browse to `http://127.0.0.1:8000/login/`.

A user account needs to be set up before the labs can be accessed. This account is local to the container; it will be deleted if the Docker container is stopped or deleted. It can be any test account such as username 'user' and password 'Kali1234'. To set up a user account access the login page and click on the 'Here' text within 'Click Here to register' (it is not entirely obvious at first) and then enter the new username and password.

Once a user account has been set up then login to access the labs. A handy feature of PyGoat is the inclusion of the 2021 version of the OWASP Top Ten as well as the 2017 version, these are provided side by side and aid cross referencing to the latest OWASP Top Ten.

7.1.4 Security Shepherd

OWASP Security Shepherd is a web and mobile application security training platform that helps to foster and improve security awareness for development teams.

The Security Shepherd tool project is an OWASP Flagship Project and can be downloaded from the project's github repository.

What is Security Shepherd? Security Shepherd is a teaching tool that provides lessons and an environment to learn how to attack both web and mobile applications. This enables users to learn or to improve upon existing their manual penetration testing skills.

Security Shepherd is run on a web server such as Apache Tomcat and this can be installed manually. There is also a pre-built virtual machine available or a docker image can be composed to run as a container.

Why use it? Security Shepherd can train inexperienced pen-testers to security expert level by sharpening their testing skill-set. Pen-testing is often included as a required stage in a organization's secure software development lifecycle (SDLC).

How to use it Security Shepherd can be run as a Docker container, as a Virtual Machine or manually on top of a web server.

The Security Shepherd wiki has step by step installation instructions:

- either compose the Docker image and run the container
- or download the virtual machine and run on a hypervisor such as Virtual Box
- or install on a Tomcat web server
- or install on windows using a Tomcat web server

Once installed and logged in, the lessons and vulnerable applications are available to use. Security Shepherd has modes which it can be used for different training goals:

- CTF (Capture the Flag) Mode
- Open Floor Mode
- Tournament Mode

7.2 Secure Coding Dojo

The OWASP Secure Coding Dojo is a platform for delivering secure coding training to software development teams. Secure Coding Dojo is an OWASP Lab project and has been continuously supported and developed since 2017.

What is the Secure Coding Dojo? The aim of Secure Coding Dojo is to teach developers how to recognize security flaws during code reviews.

The training platform has a set of training lessons and also blocks of code where the developer has to identify which block of code is written in an insecure way. A leader board is provided for the development teams to track their progress.

Each lesson is built as an attack/defense pair. The developers can observe the software weaknesses by conducting the attack and after solving the challenge they learn about the associated software defenses. The predefined lessons are based on the MITRE most dangerous software errors (also known as SANS 25) so the focus is on software errors rather than attack techniques.

The training platform can be customized to integrate with custom vulnerable websites and other CTF challenges.

Why use it? Development teams are often required to have Secure Coding training, and this may be an annual compliance requirement. The Secure Coding Dojo provides this compliant training in reviewing software for security bugs in representative source code.

How to use it The OWASP Spotlight series provides an overview of the developer training provided by the Secure Coding Dojo: 'Project 14 - OWASP Secure Coding Dojo'.

There is a demonstration site for Secure Coding Dojo which provides access to the training modules, code blocks and a public leader board. Note that the demonstration site does not provide the deliberately insecure web sites, such as the 'Insecure.Inc' Java site, because this would encourage attack traffic across a public network.

Ideally Secure Coding Dojo is deployed by the organization providing the training, rather than by using the demo site, because development teams can then log in securely to the Dojo. Deployment is straight forward, consisting of cloning the repository and running `docker-compose` with environment variables. This also allows deployment of the associated deliberately insecure web site to practice penetration testing.

7.3 Security Knowledge Framework training

The Security Knowledge Framework (SKF) is an expert system application that uses various open source projects to support development teams and security architects in building secure applications. The Security Knowledge Framework uses the OWASP Application Security Verification Standard (ASVS) with code examples to help developers in pre-development and post-development phases and create applications that are secure by design.

Having been an OWASP flagship project for many years the SKF is now no longer within the OWASP organization; it will continue to be referenced in the OWASP Wayfinder and other OWASP projects because it is certainly a flagship project for any organization.

What is the Security Knowledge Framework? The SKF is a web application that is available from the github repo. There is a demo version of SKF that is useful for exploring the multiple benefits of the SKF. Note that SKF is in a process of migrating to a new repository so the download link may change.

The SKF provides training and guidance for application security:

- Requirements organizer
- Learning courses
- Practice labs
- Documentation on installing and using the SKF

Why use the SKF? The SKF provides both learning courses and practice labs that are useful for development teams to practice secure coding skills.

The following learning courses are available (as of December 2023):

- Developing Secure Software (LFD121)
- Understanding the OWASP Top 10 Security Threats (SKF100)
- Secure Software Development: Implementation (LFD105x)

and there are plans for more training courses. All of these courses (LFD121, SKF100 and LFD105x) are provided by the Linux Foundation.

In addition to the training courses there are a wide range of practice labs (64 as of December 2023).

How to use the SKF The easiest way to get started with the SKF training is to try the online demo. This will provide access to the practice labs, the training courses and also to the requirements tool.

{:
height="160px" }

7.4 SamuraiWTF

The OWASP SamuraiWTF (Web Training and Testing Framework) is a linux desktop distribution that is intended for application security training.

The SamuraiWTF breaker/tool project is an OWASP Laboratory Project and the desktop can be downloaded as a pre-built virtual machine from the website.

What is SamuraiWTF? Samurai Web Training Framework is similar in spirit to the widely used Kali Linux distribution; it is a distribution of an Ubuntu desktop that integrates many open-source tools used for penetration testing.

SamuraiWTF is different to Kali in that it is meant as a training environment for attacking web applications rather than as a more general and comprehensive pen-testers toolkit. It was originally a web testing framework tool, but has been migrated to a training tool for penetration testing. For this reason it integrates a different set of tools from Kali; it focuses only on the tools used during a web penetration test.

Samurai-Dojo is a set of vulnerable web applications that can be used to exercise the SamuraiWTF testing framework. In addition there is the Katana which provides configuration to install specific tools and targets. This allows instructors to set up a classroom lab, for example, that can be distributed to their students.

Why use it? SamuraiWTF is easy to use and comes as a virtual machine, which makes it ideal in a teaching environment or as an attack tool targeted specifically against web applications. The teaching environment can be tailored for a particular set of lessons using the command line tool 'katana'.

The applications provided by Samurai-Dojo provides a set of real world applications to attack; these applications are contained within the Samurai Web Training Framework virtual machine. This provides a teaching environment where none of the attack traffic will leak from the environment, and so avoids triggering network intrusion detection systems.

How to use it The OWASP Spotlight series provides an overview of training provided by SamuraiWTF: 'Project 26 - OWASP SamuraiWTF'.

Getting started with SamuraiWTF is described in the github README :

- either download the virtual machine for Oracle VirtualBox
- or download the Hyper-V for Windows
- or build an Amazon Workspace

Run the Samurai Web Training Framework and login as the super-user ‘samurai’. From a command prompt run ‘katana’ to start configuring SamuraiWTF for your training purposes, for example ‘katana list’.

References

- OWASP SamuraiWTF main site
- SamuraiWTF Dojo
- SamuraiWTF Katana
- SamuraiWTF downloads
- SamuraiWTF OWASP project

7.5 OWASP Top Ten project

The OWASP Top 10 is a standard awareness document for developers and web application security. It represents a broad consensus about the most critical security risks to web applications.

The OWASP Top Ten is a flagship documentation project and is one of the very first OWASP projects.

What is the OWASP Top 10? 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 OWASP Top 10 is periodically revised to keep it up to date with the latest threat landscape. The latest version was released in 2021 to mark twenty years of OWASP:

- A01:2021-Broken Access Control
- A02:2021-Cryptographic Failures
- A03:2021-Injection
- A04:2021-Insecure Design
- A05:2021-Security Misconfiguration
- A06:2021-Vulnerable and Outdated Components
- A07:2021-Identification and Authentication Failures
- A08:2021-Software and Data Integrity Failures
- A09:2021-Security Logging and Monitoring Failures
- A10:2021-Server-Side Request Forgery

The project itself is actively maintained by a project team. The list is based on data collected from identified application vulnerabilities and from a variety of sources; security vendors and consultancies, bug bounties, along with company/organizational contributions. The data is normalized to allow for level comparison between 'Human assisted Tooling and Tooling assisted Humans'.

How to use it The OWASP Top 10 has various uses that are foundational to application security:

- as a training aid on the most common web application vulnerabilities
- as a starting point when testing web applications
- to raise awareness of vulnerabilities in applications in general
- as a set of demonstration topics

There is not one way to use this documentation project; use it in any way that promotes application security. The OWASP Spotlight series provides an overview of the Top Ten: 'Project 10 - Top10'.

OWASP Top 10 versions The OWASP Top 10 Web Application Security Risks document was originally published in 2003, making it one of (or even the most) longest lived OWASP project, and since then has been in active and continuous development. Listed below are the versions up to the latest in 2021, which was released to mark 20 years of OWASP.

- Original 2003
- Update 2004
- Update 2007
- Release 2010
- Release 2013
- Release 2017
- Latest version 2021

7.6 Mobile Top 10

The OWASP Mobile Top 10 is a list of the most prevalent vulnerabilities found in mobile applications. In addition to the list of risks it also includes a list of security controls used to counter these vulnerabilities.

This documentation project is an OWASP Lab project, aimed at security builders and defenders.

What is the Mobile Top 10? The Mobile Top 10 identifies and lists the top ten vulnerabilities found in mobile applications. These risks of application vulnerabilities have been determined by the project team from various sources including incident reports, vulnerability databases, and security assessments. The list has been built using a data-based approach from unbiased sources, an approach detailed in the repository read-me.

- M1: Improper Credential Usage
- M2: Inadequate Supply Chain Security
- M3: Insecure Authentication/Authorization
- M4: Insufficient Input/Output Validation
- M5: Insecure Communication
- M6: Inadequate Privacy Controls
- M7: Insufficient Binary Protections
- M8: Security Misconfiguration
- M9: Insecure Data Storage
- M10: Insufficient Cryptography

The project also provides a comprehensive list of security controls and techniques that should be applied to mobile applications to provide a minimum level of security:

1. Identify and protect sensitive data on the mobile device
2. Handle password credentials securely on the device
3. Ensure sensitive data is protected in transit
4. Implement user authentication,authorization and session management correctly
5. Keep the backend APIs (services) and the platform (server) secure
6. Secure data integration with third party services and applications
7. Pay specific attention to the collection and storage of consent for the collection and use of the user's data
8. Implement controls to prevent unauthorized access to paid-for resources (wallet, SMS, phone calls etc)
9. Ensure secure distribution/provisioning of mobile applications
10. Carefully check any runtime interpretation of code for errors

The list of mobile controls has been created and maintained by a collaboration of OWASP and the European Network and Information Security Agency (ENISA) to build a joint set of controls.

Why use it? It is important to have awareness of the types of attack mobile applications are exposed to, and the types of vulnerabilities that may be present in any given mobile application.

The Mobile Top 10 provides a starting point for this training and education, and it should be noted that the risks to mobile applications do not stop at the Top 10; this list is only the more important ones and in practice there are many more risks.

In addition the Mobile Top 10 provides a list of controls that should be considered for mobile applications; ideally at the requirements stage of the development cycle (the sooner the better) but they can be applied at any time during development.

Mobile Top 10 versions The Mobile Top 10 was first released in 2014, updated in 2016 with the latest version released in 2024.

The list of mobile application controls were originally published in 2011 as the 'Smartphone Secure Development Guideline'. This was then revised during 2016 and released in February 2017 to inform the latest set of mobile application controls.

7.7 API Top 10

The OWASP API Security Project (API Top 10) explains strategies and solutions to help the understanding and mitigation of the unique vulnerabilities and security risks of Application Programming Interfaces (APIs).

The API Top 10 is an OWASP Laboratory Project which is accessed as a web based document.

What is the API Top 10? The use of Application Programming Interfaces (APIs) comes with security risks. Given that APIs are widely used in various types of applications, the OWASP API Security Project created and maintains the Top 10 API Security Risks document as well as a documentation portal for best practices when creating or assessing APIs.

- API1:2023 - Broken Object Level Authorization
- API2:2023 - Broken Authentication
- API3:2023 - Broken Object Property Level Authorization
- API4:2023 - Unrestricted Resource Consumption
- API5:2023 - Broken Function Level Authorization
- API6:2023 - Unrestricted Access to Sensitive Business Flows
- API7:2023 - Server Side Request Forgery
- API8:2023 - Security Misconfiguration
- API9:2023 - Improper Inventory Management
- API10:2023 - Unsafe Consumption of APIs

Why use it? Most software projects use APIs in some form or another. Developers and security engineers should be encouraged to refer to the API Security Top 10 to assist them when acting as security builders, breakers, and defenders for an organization.

.image-right { height: 180px; display: block; margin-left: auto; margin-right: auto; float: right; }

7.8 WrongSecrets

OWASP WrongSecrets is a production status project and provides challenges focused on secrets management using an intentionally vulnerable application and environment. The project offers standalone and Capture-the-flag modes, with a demo on Heroku.

What is WrongSecrets? WrongSecrets goals are to:

- Educate on secret management and its pitfalls
- Help people reflect on their secrets management strategy
- Promote secrets management as an important facet of security

The project provides challenges around secrets management across several layers:

- A Spring Boot Java application
- Application configuration
- Docker
- Kubernetes
- Vault
- AWS, GCP, or Azure
- Binaries / Reverse engineering

Scenarios vary in difficulty, and you can solve some of them just by using the browser on your mobile phone. For others, you would need knowledge of cloud security or reverse engineering tools and cryptography.

Why use it? If you, your team or your organization want to learn about secrets management and potential pitfalls, you can do so with WrongSecrets' challenges.

Alternatively, you can use WrongSecrets as a secret detector testbed/benchmark.

How to use it You can set WrongSecrets up in standalone or in capture the flag (CTF) mode on Docker, Kubernetes, AWS, GCP or Azure.

Set-up guides for the standalone version are available in the project README.

For the CTF, the project also provides set-up guides and a Helm chart.

References

- OWASP WrongSecrets
- Secure_Cloud_Architecture cheat sheet
- WrongSecrets demo

{% include breadcrumb.html %}

7.9 OWASP Snakes and Ladders

OWASP Snakes & Ladders is an educational project based on the popular board game. It uses gamification to promote awareness of application security controls and risks, and in particular knowledge of other OWASP documents and tools.

This documentation project is an OWASP Lab project, aimed at security builders and defenders.

What is it? Yes, it really is the snakes & ladders game, but for web and mobile application security. It is played by two competing teams, possibly accompanied by beer and pretzels.

In the board game for web applications, the virtuous behaviours (ladders) are secure coding practices (using the OWASP Proactive Controls) and the vices (snakes) are application security risks from the OWASP Top Ten 2017 version.

The web application version can be downloaded for various languages:

- German (DE)
- English (EN)
- Spanish (ES)
- French (FR)
- Japanese (JA)
- Turkish (TR)
- Chinese (ZH)

The board game for mobile applications uses the mobile controls detailed in the OWASP Mobile Top 10 as the virtuous behaviours. The vices are the Mobile Top 10 risks from the 2014 version of the project.

The mobile application version is available as a download in English and Japanese

Why use it? This board game was created so that it could be used as an ice-breaker in application security training. It also has wider appeal as learning materials for developers or simply as a promotional hand-out.

To cover all of that, the Snakes & Ladders project team summarise it as:

“OWASP Snakes and Ladders is meant to be used by software programmers, big and small”

The game is quite lightweight; so it is meant to be just some fun with some learning attached, and is not intended to have the same rigour or depth as the card game Cornucopia.

When the project was first created there was a print run of the game on heavy duty paper. These were available at conferences and meetings - they were also available to be purchased online but this last option no longer seems to be available.

References

- OWASP [Snakes & Ladders][snakes
- OWASP Proactive Controls
- OWASP Top Ten 2017 version
- OWASP Mobile Top 10
- OWASP Cornucopia.

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 Culture
8.2 Security Champions
8.2.1 Security champions program
8.2.2 Security Champions Guide
8.2.3 Security Champions Playbook
8.3 Software Assurance Maturity Model
8.4 Application Security Verification Standard
8.5 Mobile Application Security

8.1 Security Culture

Most organizations have an application development lifecycle in place with security activities built into it, this goes a long way to reducing the security issues present in applications and systems. The OWASP Security Culture project is a guide that considers security at each stage of the application security development lifecycle, with the aim of creating and nurturing secure development practices throughout the lifecycle.

The Security Culture guide is an OWASP incubator project and version 1.0 is available as a web document.

What is the OWASP Security Culture project The OWASP Security Culture project is a collection of explanations and practical advice arranged under various topic headings.

- Why add security in development teams
- Setting maturity goals
- Security team collaboration
- Security champions
- Activities
- Threat modelling
- Security testing
- Security related metrics

The OWASP Security Culture project is focused on establishing/persisting a positive security posture within the application development lifecycle and references other OWASP projects in a similar way to the OWASP Developer Guide.

Encouraging a Security Culture The philosophy of a security culture is as important as the technical aspects; the application development teams need to be onboard to adopt a good security posture. The Security Culture project recognizes this and devotes a section to the importance of building security into the development lifecycle.

As well as onboard development teams there has to be buy-in from the higher management: without this any security champion program is certain to fail and the security culture of the organization will suffer. The Security Culture project provides information on goals, metrics and maturity models that are a necessary prerequisite for management approval of security activities. In addition the Security Culture project highlights the importance of security teams, management and development teams all working together - all are necessary for a good security culture.

Security Champions are an important way of encouraging security within an organization - an organization can have a healthy security culture without security champions but it is a lot easier with a security champion program in place. Selecting and nurturing security champions has to be tailored to the individual organisation, no security champion program will be the same as another one and close reference should be made to the Security Champions Playbook.

Threat modelling is an activity that in itself is important within an organization, and it also has the benefit of helping communication between the security teams and development teams. Security testing (such as SAST, DAST and IAST) is another area where close collaboration is required within the organization: management, security, development and pipeline teams will all be involved. This has the added benefit, as with threat modeling, of promoting a good security culture / awareness within the organization - and can be a good indicator of where the security culture is succeeding.

Metrics are important for a healthy security culture to persist and grow with an organization. Without metrics to show the benefits of the security culture then interest and buy-in from the various teams involved will wane, leading to a decline in the culture and leading in turn to a poor security posture. Metrics will provide the justification for investment and nurturing of a good security culture.

8.2 Security Champions

A 'Security Champion' is a member of a software development team who is the liaison between Information Security and developers. This helps to embed security into the development organization.

Security Champions and the necessary supporting program are described in the SAMM Organization and Culture section, which in turn is part of the SAMM Education & Guidance security practice within the Governance business function.

Depending on the development team the Security Champion may be a software developer, tester, product manager or any role within the team; what matters most is an enthusiasm for software security and a willingness to learn. Security Champions can assist with researching, verifying, and prioritizing security and compliance related software defects within the application/product.

Security Champions will usually be involved in risk/threat assessments and architectural reviews and can often help identify opportunities to remediate security defects; making the architecture of the application more resilient and reducing the attack threat surface. Security Champions also participate in periodic briefings to increase awareness and expertise in different security disciplines.

The two goals of the Security Champion program are to increase effectiveness of application security and compliance and to strengthen the relationship between development teams and Information Security teams. The program should supply Security Champions with additional training to help develop their role as a software security subject matter expert. If possible the Security Champion should be provided with time for Information Security related activities, and this may well have to be negotiated with the development management hierarchy.

Importantly it should be recognized that Security Champions are often taking on an extra role in addition to their existing one, and it is important that support is provided by the program for their well-being.

Sections:

8.2.1 Security champions program

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.

References

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

OWASP SECURITY CHAMPIONS GUIDE

8.2.2 Security Champions Guide

The OWASP Security Champions Guide is a guidebook that helps organizations build a security champions program that can succeed over the long term.

It is a relatively new OWASP Incubator project and is available as a web document.

Overview Security Champions are an important part of an organization's security posture; they provide development teams with subject matter experts in application security and can be the first point of contact for information security teams. It is widely recognized that a program needs to be in place to actively support the security champions, otherwise there is a risk of disillusionment or even burn-out; to counter this risk a Security Champions Program will help identify and nurture security champions.

The Security Champions Guide provides two resources that explain what a security champion program is and how it can be put into practice:

- The Security Champions Manifesto sets out a philosophy for a good security champions program
- The Security Champions Guide explains each point in the manifesto and illustrates it with practical advice

The Security Champions Guide is not proscriptive, an individual organization should select freely from the suggestions to create its own program - and perhaps revisit the guide as its security champions program matures over time.

The Security Champions Manifesto The manifesto defines ten key principles for a successful security champions program:

- Be passionate about security
- Start with a clear vision for your program
- Secure management support
- Nominate a dedicated captain
- Trust your champions
- Create a community
- Promote knowledge sharing
- Reward responsibility
- Invest in your champions
- Anticipate personnel changes

This manifesto is a set of guiding principles that will certainly help with the creating the program and can also improve an existing security champions program.

The Security Champions Guide If the security champions program is in the process of being put in place then consider each principle/section of the guide in turn and decide if it can be part of the program. Each principle is generally applicable - as every program will be different in practice - so pick and choose the elements the organization can adopt or leverage to create a customized program.

Each principle is split into topics: the What, Why and How. Some sections also contain checklists or templates that can be used to create or improve the program. For example the section on investing in

security champions explains what this entails: 'Invest in the personal growth and development of your Security Champions'. It then goes on to describe why this is important (ensuring the health of the Security Champions community) and then gives suggestions on what this means in practice: webinars, conferences, recognition etc. The other sections are similarly helpful and provide a range of practical advice.

The guide is also useful for an existing security champions program, providing advice on what can be further achieved. It is worth noting that some security champions programs are initially successful but can then fail over time for various reasons, perhaps through change of personnel or budgetary pressure. The suggestions in the Security Champions Guide can be used as a justification for investing in the program further and will help to sustain the existing program.

8.2.3 Security Champions Playbook

The Security Champions Playbook is a project that describes the process of establishing a Security Champions program within an organization.

What are Security Champions? Security Champions are active members of a team that act as a core element of the security assurance process within a product or service. They are often are the initial point of contact within the team when it comes to security concerns and incidents.

Some advantages of encouraging Security Champions within a team are :

- Scaling security through multiple teams
- Engaging non-security engineers in security
- Establishing the security culture throughout an organization

The Security Champion should be given extra training to carry out this role, which is often in addition to their existing responsibilities.

How to use the playbook Security Champions Playbook lists six steps which include general recommendations:

1. Identify teams
2. Define the role
3. Nominate Champions
4. Set up communication channels
5. Build solid knowledge base
6. Maintain interest

Use these recommendations to build up a Security Champions program that is tailored to the needs of the organization.

8.3 Software Assurance Maturity Model

The Software Assurance Maturity Model (SAMM) project provides an effective and measurable way for an organization to analyze and improve their secure development lifecycle processes. SAMM is one of the OWASP's flagship projects, and can be downloaded from the SAMM project site.

What is SAMM? SAMM can be regarded as the prime maturity model for software assurance. SAMM provides an effective and measurable way for all types of organizations to analyze and improve their software security posture. SAMM supports the entire secure software development lifecycle and is technology and process agnostic.

The SAMM model is hierarchical. At the highest level SAMM defines five business functions; activities that software development must fulfill to some degree:

- Governance
- Design
- Implementation
- Verification
- Operations

Each business function in turn has three security practices, which are areas of security-related activities that build assurance for the related business function.

Security practices have activities, grouped in logical flows and divided into two streams (A and B). Streams cover different aspects of a practice and have their own objectives, aligning and linking the activities in the practice over the different maturity levels.

For each security practice, SAMM defines three maturity levels which generalize to foundational, mature and advanced. Each level has a successively more sophisticated objective with specific activities, and more strict success metrics.

Why use it? The structure and setup of the SAMM model support:

- assessment of the organization's current software security posture
- definition of the organization's targets
- definition of an implementation roadmap to get there
- prescriptive advice on how to implement particular activities

These provide suggestions for improving processes and building security practices into the culture of the organization.

How to use it The OWASP Spotlight series provides an overview of using the SAMM to guide development: 'Project 9 - Software Assurance Maturity Model (SAMM)'.

The SAMM Fundamentals Course provides training on the high level SAMM Business Functions and provides guidance on each Security Practice. The SAMM assessment tools measure the quality of an organization's software assurance maturity process, which can be used as feedback into the culture of the organization.

References

- OWASP Software Assurance Maturity Model (SAMM)
- SAMMY management tool
- OWASP SAMM project

8.4 Application Security Verification Standard

The Application Security Verification Standard (ASVS) is a long established OWASP flagship project, and is widely used to build a culture of security as well as verification of web applications.

It can be downloaded from the OWASP project page in various languages and formats: PDF, Word, CSV, XML and JSON. Having said that, the recommended way to consume the ASVS is to access the github markdown pages directly - this will ensure that the latest version is used.

What is ASVS? The ASVS is an open standard that sets out the coverage and level of rigor expected when it comes to performing web application security verification. The standard also provides a basis for testing any technical security controls that are relied on to protect against vulnerabilities in the application.

The ASVS is split into various sections:

- V1 Architecture, Design and Threat Modeling
- V2 Authentication
- V3 Session Management
- V4 Access Control
- V5 Validation, Sanitization and Encoding
- V6 Stored Cryptography
- V7 Error Handling and Logging
- V8 Data Protection
- V9 Communication
- V10 Malicious Code
- V11 Business Logic
- V12 Files and Resources
- V13 API and Web Service
- V14 Configuration

The ASVS defines three levels of security verification:

1. applications that only need low assurance levels; these applications are completely penetration testable
2. applications which contain sensitive data that require protection; the recommended level for most applications
3. the most critical applications that require the highest level of trust

Most applications will aim for Level 2, with only those applications that perform high value transactions, or contain sensitive medical data, aiming for the highest level of trust at level 3.

Why use it? The ASVS is well established, the earlier versions were written in 2008, and it has been continually supported since then. The ASVS is used to generate security requirements, guide the verification process and also to identify gaps in the application security.

The ASVS can also be used as a metric on how mature application security processes are; it is a yardstick with which to assess the degree of trust that can be placed in the web application. This helps provide a good security culture: the application developers and application owners can see how they are doing and be confident in the maturity of their processes in comparison with other teams and organizations.

How to use it The OWASP Spotlight series provides an overview of the ASVS and its uses: 'Project 19 - OWASP Application Security Verification standard (ASVS)'.

The appropriate ASVS level should be chosen from:

- Level 1: First steps, automated, or whole of portfolio view
- Level 2: Most applications
- Level 3: High value, high assurance, or high safety

This is not a judgemental ranking, for example if an application needs only Level 1 protection then that is a valid choice. Tools such as SecurityRAT can then help create a subset of the ASVS security requirements for consideration.

Application developer teams and application owners can then gain familiarity with the appropriate security requirements and incorporate them into the process and culture of the organization. To help navigate the ASVS, the OWASP Cheat Sheets have been indexed specifically for each section of the ASVS which can be used to explain and expand on each requirements category.

8.5 Mobile Application Security

The MAS Verification Standard (MASVS) explains the processes, techniques and tools used for security testing a mobile application.

The OWASP MAS Crackmes, also known as UnCrackable Apps, is a collection of reverse engineering challenges for the OWASP Mobile Application Security (MAS).

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 also 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.

References

- OWASP Mobile Application Security (MAS)
- MAS project
- MAS Crackmes UnCrackable Apps
- MAS Testing Guide (MASTG)
- MAS Verification Standard (MASVS)
- OWASP Mobile Application Security cheat sheet

9. Operations

Operations are those activities necessary to ensure that 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 Operations business function in the OWASP SAMM model.

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

OWASP projects provide the CRS that is used for both Coraza and ModSecurity web application firewalls, which are widely used for data and system management.

Sections:

9.1 DevSecOps Guideline

The OWASP DevSecOps Guideline project explains how to best implement a secure pipeline, using best practices and introducing automation tools to help 'shift-left' security issues.

The DevSecOps Guideline is in active development as an OWASP Production documentation project and can be accessed from the web document or downloaded as a PDF.

What is the DevSecOps Guideline? The DevOps (combining software Development and release Operations) pipelines use automation to integrate various established activities within the development and release processes into pipeline steps. This enables the use of Continuous integration / Continuous Delivery/Deployment (CI/CD) within an organization. DevSecOps (combining security with DevOps) seeks to add steps into the existing CI/CD pipelines to build security into the development and release process.

The DevSecOps Guideline is a collection of advice and theory that explains how to embed security into DevOps. It covers various foundational topics such as Threat Modeling pipelines, Secrets Management and Linting Code. It then explains and illustrates various vulnerability scanning steps commonly used in CI/CD pipelines :

- Static Application Security Testing (SAST)
- Dynamic Application Security Testing (DAST)
- Interactive Application Security Testing (IAST)
- Software Composition Analysis (SCA)
- Infrastructure Vulnerability Scanning
- Container Vulnerability Scanning

The DevSecOps Guideline is a concise guide that provides the foundational knowledge to implement DevSecOps.

How to use the DevSecOps Guideline The DevSecOps Guideline is document can be accessed from the web document or downloaded as a PDF. It is concise enough that all the sections can be read within a short time, and it provides enough knowledge to understand the concept behind DevSecOps and what activities are involved.

It provides an excellent overview of DevSecOps which shows how the steps of a typical CI/CD pipeline fit together and what sort of tools can be applied in each step to secure the pipeline. Many of the pages in the DevSecOps Guideline contain lists of tools that can be applied to the pipeline step.

The DevSecOps Guideline document is in the process of being expanded and updated which will build on the existing 2023 version.

References

- OWASP DevSecOps Guideline project
- OWASP CI/CD Security Cheat Sheet

{: height="160px" }

9.2 Coraza Web Application Firewall

The OWASP Coraza project provides a golang enterprise-grade Web Application Firewall framework that supports the ModSecurity seclang language and is completely compatible with OWASP CRS. Coraza is in active development as an OWASP Production code project, with the first stable version released in September 2021 and several releases since then.

What is Coraza? The Coraza Web Application Firewall framework is used to enforce policies, providing a first line of defense to stop attack on web applications and servers. Coraza can be configured using the OWASP CRS and also custom policies can be created.

Coraza can be deployed:

- as a library in an existing web server
- within an application server acting as a WAF
- as a reverse proxy
- using a docker container

Why use Coraza? Web Application Firewalls are usually the first line of defense against HTTP attacks on web applications and servers. The Coraza WAF is widely used for providing this security, especially for cloud applications, along with the original OWASP ModSecurity WAF.

How to use Coraza The best way to start is to create a Coraza WAF instance and then add rules to this WAF, following the Coraza Quick Start tutorial.

There are multiple ways of running Coraza, and the one chosen will depend on an individual organization's deployment:

- Coraza SPOA connector runs the Coraza WAF as a backing service for HAProxy
- Coraza Caddy Module provides Web Application Firewall capabilities for Caddy
- the Coraza Proxy WASM filter can be loaded directly from Envoy or used as an Istio plugin
- Coraza as a C library, used for applications written in C rather than golang

References

- OWASP Coraza
- OWASP CRS
- OWASP ModSecurity
- Secure Cloud Architecture cheat sheet

9.3 ModSecurity Web Application Firewall

ModSecurity is an open source Web Application Firewall (WAF) widely deployed on web servers that has been in continuous development and widespread use since 2002.

In 2024 it became an OWASP Production project, supported by the existing leadership and contributors.

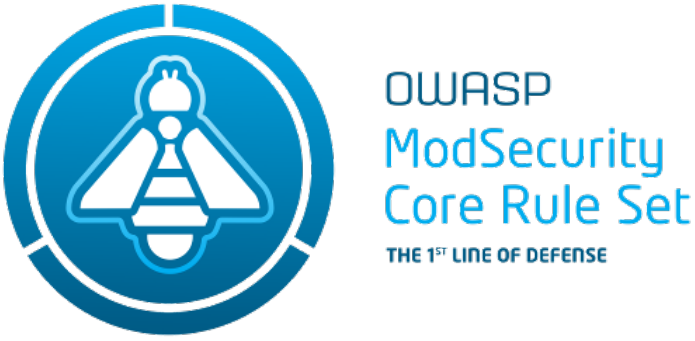
What is ModSecurity? In January 2024 the ModSecurity Web Application Firewall project was adopted by OWASP, previously TrustWave had been the custodian of this project. ModSecurity itself has a long history as an open source project, the first release was in November 2002, and is widely used as a web application firewall for cloud applications and on-premises web servers.

The ModSecurity WAF needs to be configured in operational deployments, and this can be done using the OWASP CRS.

Why use ModSecurity? Web Application Firewalls are often the first line of defense against HTTP attacks on web applications and servers. The ModSecurity WAF is widely used for this purpose along with the Coraza WAF, also provided by OWASP.

How to use ModSecurity ModSecurity is a Web Application Firewall, which scans the incoming and outgoing HTTP traffic to a web server. The ModSecurity WAF is deployed as a proxy server in front of a web application, or deployed within the web server itself, to provide protection against HTTP attacks.

The rules applied to the HTTP traffic are provided as configuration to ModSecurity, and these rules allow many different actions to be applied such as blocking traffic, redirecting requests, and many more. See the documentation for deploying and running ModSecurity, along with the documentation on configuring ModSecurity with the CRS.



9.4 OWASP CRS

The OWASP CRS project, formerly known as Core Rule Set, is a set of generic attack detection rules for use with ModSecurity compatible web application firewalls such as OWASP Coraza. CRS is an OWASP Flagship tool project and can be downloaded for either Apache or IIS/Nginx web servers.

What is the CRS? The CRS are attack detection rules for use with ModSecurity, Coraza and other ModSecurity compatible web application firewalls. The CRS aims to protect web applications from a wide range of attacks with a minimum of false alerts. The CRS provides protection against many common attack categories, including those in the OWASP Top Ten.

Why use it? If an organization is using a Coraza, ModSecurity or compatible Web Application Firewall (WAF) then it is very likely that the CRS is already in use by this WAF. The CRS provides the policy for the Coraza / Modsecurity engine so that traffic to a web application is inspected for various attacks and malicious traffic is blocked.

How to use it The use of the CRS assumes that a ModSecurity, Coraza or compatible WAF has been installed. Refer to the Coraza tutorial or the ModSecurity on how to do this.

To get started with CRS refer to the CRS installation instructions.

The OWASP Spotlight series provides an overview of how to use this CRS: 'Project 3 - Core Rule Set (CRS) - 1st Line of Defense'.

References

- OWASP CRS
- OWASP ModSecurity
- OWASP Coraza

10. Metrics

Metrics are important in an organization for various reasons, and in software security they can be used to:

- measure the effectiveness of security controls
- determine security posture
- provide justification for security programs
- and others

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

Strategy and Metrics

The software security program is foundational to the strategic planning an organizations security posture. Metrics keep track of the security activities within the plan and provide the information for gap analysis.

The Software Assurance Maturity Model (SAMM) provides descriptions and definitions for the Strategy and Metrics business practices within the Governance business function. It provides two streams for achieving organizational maturity:

- Create and Promote which concerns the risks identified with the organization and what level of risk is acceptable
- Measure and Improve which describes monitoring the security strategy through metrics

The categories of metrics suggested by SAMM are :

- Effort metrics: the effort spent on security
- Result metrics: the results of security efforts
- Environment metrics: the environment where security efforts take place

There are other metrics, perhaps specific to an individual organization, that can also be collected and acted on.

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:

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

The Software Assurance Maturity Model (SAMM) project provides an effective and measurable way for an organization to analyze their secure development lifecycle, and identify any gaps or improvements. SAMM is one of the OWASP's flagship projects, and can be downloaded from the SAMM project site.

What is SAMM? SAMM is regarded as the prime maturity model for software assurance. SAMM provides an effective and measurable way for all types of organizations to analyze and improve their software security posture. SAMM supports the complete secure software development lifecycle and is technology and process agnostic.

The SAMM model is hierarchical. At the highest level SAMM defines five business functions; activities that software development must fulfill to some degree:

- Governance
- Design
- Implementation
- Verification
- Operations

Each business function in turn has three security practices, which are areas of security-related activities that build assurance for the related business function.

Security practices have activities, grouped in logical flows and divided into two streams (A and B). Streams cover different aspects of a practice and have their own objectives, aligning and linking the activities in the practice over the different maturity levels.

For each security practice, SAMM defines three maturity levels which generalize to foundational, mature and advanced. Each level has a successively more sophisticated objective with specific activities, and more strict success metrics.

Why use it? The structure and setup of the SAMM model support:

- assessment of the organization's current software security posture
- definition of the organization's targets
- definition of an implementation roadmap to get there
- prescriptive advice on how to implement particular activities

These give the security activities expected at each maturity level, and provide input to the gap analysis.

How to use it The OWASP Spotlight series provides an overview of using the SAMM: 'Project 9 - Software Assurance Maturity Model (SAMM)'.

Security gap analysis can benefit from an assessment which measures the quality of the software assurance maturity process. The SAMM Assessment tools include spreadsheets and online tools such as SAMMwise and SAMMY.

The SAMM model describes these fundamentals of software security, which it calls Business Functions. 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

Business Function	Business Practices		
Design	Threat Assessment	Security Requirements	Secure 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 which provide different objectives for the same practice.

References

- OWASP Software Assurance Maturity Model (SAMM)
- SAMMY management tool
- OWASP SAMM project

11.1.2 Application Security Verification Standard

The Application Security Verification Standard (ASVS) is a long established OWASP flagship project, and is widely used to identify gaps in security as well as the verification of web applications.

It can be downloaded from the OWASP project page in various languages and formats: PDF, Word, CSV, XML and JSON. Having said that, the recommended way to consume the ASVS is to access the github markdown pages directly - this will ensure that the latest version is used.

What is ASVS? The ASVS is an open standard that sets out the coverage and 'level of rigor' expected when it comes to performing web application security verification. The standard also provides a basis for testing any technical security controls that are relied on to protect against vulnerabilities in the application.

The ASVS is split into various sections:

- V1 Architecture, Design and Threat Modeling
- V2 Authentication
- V3 Session Management
- V4 Access Control
- V5 Validation, Sanitization and Encoding
- V6 Stored Cryptography
- V7 Error Handling and Logging
- V8 Data Protection
- V9 Communication
- V10 Malicious Code
- V11 Business Logic
- V12 Files and Resources
- V13 API and Web Service
- V14 Configuration

The ASVS defines three levels of security verification:

1. applications that only need low assurance levels; these applications are completely penetration testable
2. applications which contain sensitive data that require protection; the recommended level for most applications
3. the most critical applications that require the highest level of trust

Most applications will aim for Level 2, with only those applications that perform high value transactions, or contain sensitive medical data, aiming for the highest level of trust at level 3.

How to use it The ASVS is a list of verification requirements that is used by many organizations as a basis for the verification of their web applications. For this reason it can be used to identify gaps in the security of web applications. If the ASVS suggests using a control then that control should be considered for the application security, it may be not applicable but at least the control should have been considered at some point in the development process.

The OWASP Spotlight series provides an overview of the ASVS and its uses: 'Project 19 - OWASP Application Security Verification standard (ASVS)'.

The OWASP Cheat Sheets have been indexed specifically for each section of the ASVS, which can be used as documentation on controls for a given requirements category.

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 that can be used as a guide for security gap analysis.

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 industry-standard security controls for secure mobile applications. 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 The MASVS provides a list of expected security controls for mobile applications, and this can be used to identify missing or inadequate controls during the gap analysis. These controls can then be tested using the MAS Testing Guide.

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. This provides a starting point for a security gap evaluation for any existing controls.

References

- OWASP Mobile Application Security (MAS)
- MAS project
- MAS Testing Guide (MASTG)
- MAS Verification Standard (MASVS)
- OWASP Mobile Application Security cheat sheet

11.2 Bug Logging Tool

The OWASP Bug Logging Tool (BLT) is a community database of bugs found in an organization's web site or application. BLT is an OWASP Production tool project and has its own bug recording site.

What is BLT? BLT is a bug recording and bounty tool that allows external users to register and advise about bugs in an organization's web site or application. It allows an organization to run a bug bounty program without having to go through a commercial provider.

The BLT core project provides a development server docker image that can be used for the bug bounty program. The BLT-Flutter application provides an integrated method for reporters/users to report bugs. The BLT Extension is a Chrome extension that helps BLT reporters/users to take screenshots and add them to a BLT website.

Why use it? Bug bounty programs are an important path for reporting security bugs to an organization. These programs can be paid-for services provided by commercial companies, or they can be provided by the company / organization itself; and this is where BLT can help.

External reporters of bugs in web sites and applications are a valuable way of identifying security related bugs and issues; it provides a diverse range of individuals to hunt for bugs. BLT can provide the route for these security bugs to be responsibly disclosed to the organization.

How to use it BLT has its own bug recording site which can be used to disclose any type of bug in any web site. Ideally this is not used for security related bugs because these bugs need responsible disclosure. The organization should run its own BLT core site to accept submission of security related bugs, and encourage users/reporters to use the BLT app and chrome extension.

12. Appendices

12.1 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:

12.1.1 Container security

This 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, 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
- ensure `daemon.json` file individual ownership and group ownership is correctly set to root, if it is in use
- if `daemon.json` file is present its file permissions are correctly set to 644 or more restrictively

---

12.1.2 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.

12.1.3 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.

12.1.4 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

12.1.5 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>
```

12.1.6 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

---

12.1.7 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

12.1.8 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.

12.2 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:

12.2.1 Secure environment
12.2.2 System hardening
12.2.3 Open Source software

12.2.1 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.

12.2.2 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

12.2.3 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!!!*******
