Application Security

A course notebook based on Derek Fisher's Udemy Course:
Application Security – The Complete Guide.

Table of Contents

1. SDLC Software/System Development Life Cycle

- Requirement Analysis
- Design
- Implementation
- Testing
- Evolution/Operation

1.1. Requirement Analysis

- A high-level view of requirements and goals
- Extracts requirements or requirement analysis
- The client has an idea of what they want - not how
- Scope defined and agreed with
- Prioritizations of requirements
- Slotting of resources (proper amount of developers, architects, etc.)

1.2. Design

- Describe features and operations
  - Screen layout
  - Business rules
  - Process diagrams
  - Pseudocode and documentation
- Prototype work
- Detailed design
  - Technology choices

    - System architecture

1.3. Implementation

1.3.1. Input

- Requirements
- Business process
- Business rules
- Software design
- Specification

1.3.2. Output

- Deliverable code

1.4. Testing

- Static analysis: Code testing
- Dynamic analysis: Running software testing
- Unit testing: Verify the functionality of specific code
- Integration testing: Verify the interfaces between components
- Interface testing: Testing data passed between units
- System testing: Testing a completely integrated system

1.5. Evolution/Operation

- Patch

- Build
- Test
- Production

2. SLDC Security Perspective

2.1. Requirement Analysis & Design

- Security frameworks
- Standards & guidelines

2.2. Design & Implementation

- Threat modeling

2.3. Testing

- Static analysis
- Dynamic analysis
- Interactive testing
- Protection tools

2.4. Evolution/Operation

- Vulnerability management techniques
- Risk rating

3. Security Definitions

The simple definition of security is *the state of being protected or safe from harm, things done to make people or places safe, or the area in place (such as an airport) where people are checked to make sure they are not carrying weapons or other illegal materials*.

But in terms of software, **security is anything you do to protect an asset that is vulnerable to some attack, failure, or error**.

- An asset may be valuable because it **holds** its value
- An asset may be valuable because it **produces** its value
- An asset may be valuable because it **provides access** to value

An **attack is any intentional action that can reduce the value of an asset**. For instance, a car (the asset), if for some reason someone decides to break a glass or do any damage to the asset, it will lose part or all of its value.

Failures and errors are unintentional actions that can reduce the value of an asset. For example, a 3D printer presents a hardware failure, preventing it from generating the value.

Attacks, failures, and errors are actions that we have collectively refer to as **threats**. Thus, security is anything you do to protect an asset that is vulnerable to some threat.

4. Security Goals

4.1. The "Anything"

Security, and more specifically Cybersecurity, can be understood as a set of goals. These goals are specifically defined by how we measure an asset's value.

What is the asset's actual value? How does value define our security goals? The goal of security is to protect an asset's value from threats.

- ☐ Determining what assets we want to protect
  - ☐ Learn how the asset works and interacts with other things
    - ☐ Determine how our asset's value is reduced directly and indirectly
      - ☐ Take steps to mitigate the threats

4.2. Cybersecurity-Specific Goals

We must consider the unique nature of its assets and capabilities when considering security goals. It is important to understand the **CIA: Confidentiality, Integrity, Availability**.

The CIA is the foundation of everything in regards to cybersecurity.

4.2.1. CIA

4.2.1.1. Confidentiality

The information is only available to those who **should have** access to it. Do not share access (or let access) just in case. Lock by default, make sure only authorized people/interfaces have access to the information.

When we protect something that provides access value, we are maintaining its confidentiality.

4.2.1.2. Integrity

The data is known to be correct and trusted. In other words, if I send information from one endpoint A to another endpoint B, I expect that what I send from endpoint A is what I receive on endpoint B. What this means is that the information in transit was not intervened, altered, tampered with, etc.

Examples of this are when you download software from websites or use a CDN. These assets come with a fingerprint (hash/checksum) that helps you determine that what you’re downloading is exactly what you expect.

When we protect something that holds value, we are maintaining its integrity.

4.2.1.3. Availability

The information is available for use by legitimate users when it is needed. In other words, let’s say I went to google.com and made a search, I expect Google to load and show the results; otherwise, it would be an availability issue.

When we protect something that produces value, we are maintaining its availability.

4.3. Real World Example

We will be using a rocket as an example. We will launch a rocket to space, and we want to think about the CIA.

4.3.1. List Assets

- Crew
- Rocket

- Fuel
- Cargo

4.3.2. List Vulnerabilities

- Problems with the heat shield
- Re-entry
- Equipment
- Rocket structure

4.3.3. List threats

- Space debris
- Atmosphere
- Air from ground control

4.3.4. Secure It

Think about the assets, vulnerabilities, and threads; go through the list and describe the protection actions that are appropriate for each asset.

For example, for the crew, we want to have considerations for the wellbeing of the people.

If it's the heat shield, we want to make sure that we do some tests on it to make sure it's going to survive reentry.

If it's about space debris, we want to make sure that we have the right detection tools in place to prevent the rocket from being blindsided.

We have well-defined goals and security mechanisms, but some mechanisms are better than others because they fit the **security principles**. The security principles aid in selecting or designing the correct mechanisms to implement our goals.

Complexity is the enemy of security, so the more complex your system is, the more difficult it is to secure.

4.4. Security Principles

- **The economy of the mechanism:** Keep it simple, make sure that the stuff you're building can be easily understood in a short time.
- **Fail-safe defaults:** If the system fails, it will fail to a safe default state. For instance, if the authentication of an application fails, the default does not allow *access*.
- **Complete mediation:** Verify that every call, request, execution, process, etc. is being made by an authorized user and is being verified.
- **Open design:** No security through obscurity.
- **Separation of privilege:** You want to make sure that, for critical workflows, you don't have a single person or single account, or single user that can initiate that workflow.
- **Least privilege:** You should only have the rights to a system that allows you to do your work and no more.
- **Least common mechanism:** Stay away from shared components. You want to have 1 source of truth for maintainability. It's easier to fix a security issue in a module that is used by multiple components rather than components implementing their logic.
- **Psychological acceptability:** You want to make security usable enough that somebody is not going to try to circumvent it. Don't make security difficult.
- **Work factor:** Make sure the effort that somebody has to put into breaking your security should be higher than the asset they are trying to get.
- **Compromise recording:** Audit everything. You want to make sure that you have the information on all your workflows, where data is going, who is accessing what, etc.

4.5. The Pyramid

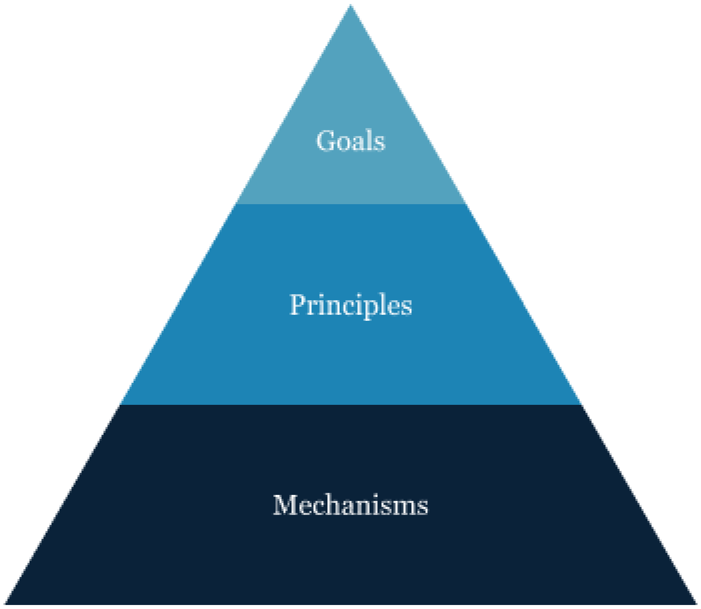


Fig. 4.1. The security pyramid.

Our goal is the CIA, the principles are what we saw in the last section, and the mechanisms are what this course is about (therefore the entirety of this document).

5. OWASP

5.1. OWASP Top 10

Top 10 is the most mature project of the Open Web Application Security Project (OWASP), or at least the most well-known project.

5.1.1. A1:2017-Injection

Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization

5.1.2. A2:2017-Broken Authentication

Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.

5.1.3. A3:2017-Sensitive Data Exposure

Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII (Personal Identifiable Information). Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.

5.1.4. A4:2017-XML External Entities (XXE)

Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.

5.1.5. A5:2017-Broken Access Control

Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

5.1.6. A6:2017-Security Misconfiguration

Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad-hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched/upgraded in a timely fashion.

5.1.7. A7:2017-Cross-Site Scripting XSS

XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface websites, or redirect the user to malicious sites.

5.1.8. A8:2017-Insecure Deserialization

Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.

5.1.9. A9:2017-Components with Known Vulnerabilities

Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.

5.1.10. A10:2017-Insufficient Logging & Monitoring

Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

5.2. WebGoat

WebGoat is a deliberately insecure application that allows interested developers just like you to test vulnerabilities commonly found in Java-based applications that use common and popular open-source components.

More information: https://owasp.org/www-project-webgoat/

5.3. Other OWASP Projects

5.3.1. Flagship

The OWASP Flagship designation is given to projects that have demonstrated strategic value to OWASP and application security as a whole.

5.3.2. Labs

OWASP Labs projects represent projects that have produced a deliverable of value.

5.3.3. Incubator

OWASP Incubator projects represent the experimental playground where projects are still being fleshed out, ideas are still being proven, and development is still underway.

5.3.4. Low Activity

These projects had no releases in at least a year. However, they have shown to be valuable.

5.4. Resources

- Watch the AppSec tutorial series to get started: https://owasp-academy.teachable.com/p/owasp-appsec-tutorials
- OWASP Top 10: The classic guidelines: https://owasp.org/www-project-top-ten/
- OWASP Cheat Sheets to get into stuff without getting annoyed: https://cheatsheetseries.owasp.org/

- OWASP ASVS: Verification checklist for testing applications: https://owasp.org/www-project-application-security-verification-standard/
- Secure Coding Practices Reference Guide: Technology-agnostic set of general software security coding practices: https://owasp.org/www-project-secure-coding-practices-quick-reference-guide/migrated_content
- OWASP Testing Guide: What/how to test web and/or mobile applications: https://owasp.org/www-project-web-security-testing-guide/
- Code Review Guidelines: https://owasp.org/www-project-code-review-guide/

6. SANS

The SANS Institute (officially the Escal Institute of Advanced Technologies) is a private U.S. for-profit company founded in 1989 that specializes in information security, cybersecurity training, and selling certificates.

SANS stands for **SysAdmin**, **Audit**, **Network**, and **Security**.

6.1. Top 25

The SANS Top 25 can be interpreted as the 25 most dangerous errors/failures.

6.1.1. Category I: Insecure Interaction Between Components

These weaknesses are related to insecure ways in which data is sent and received between separate components, modules, programs, processes, threads, or systems.

Examples:

- CWE-89 Improper Neutralization of Special Elements used in an SQL Command (‘SQL Injection’)
- CWE-78 Improper Neutralization of Special Elements used in an OS Command (‘OS Command Injection’)
- CWE-79 Improper Neutralization of Input During Web Page Generation (‘Cross-Site Scripting’)
- CWE-434 Unrestricted Upload of File with Dangerous Type
- CWE-352 Cross-Site Request Forgery (CSRF)
- CWE-601 URL Redirection to Untrusted Site (‘Open Redirect’)

6.1.2. Category II: Risky Resource Management

The weaknesses in this category are related to ways in which software does not properly manage the creation, usage, transfer, or destruction of important resources.

Examples:

- CWE-120 Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’)
- CWE-22 Improper Limitation of a Pathname to a Restricted Directory (‘Path Traversal’)
- CWE-494 Download of Code Without Integrity Check
- CWE-829 Inclusion of Functionality from Untrusted Control Sphere
- CWE-676 Use of Potentially Dangerous Function
- CWE-131 Incorrect Calculation of Buffer Size
- CWE-134 Uncontrolled Format String
- CWE-190 Integer Overflow or Wraparound

6.1.3. Category III: Porous Defenses

The weaknesses in this category are related to defensive techniques that are often misused, abused, or just plain ignored.

Examples:

- CWE-306 Missing Authentication for Critical Function
- CWE-862 Missing Authorization
- CWE-798 Use of Hard-coded Credentials
- CWE-311 Missing Encryption of Sensitive Data
- CWE-807 Reliance on Untrusted Inputs in a Security Decision
- CWE-250 Execution with Unnecessary Privileges
- CWE-863 Incorrect Authorization
- CWE-732 Incorrect Permission Assignment for Critical Resource
- CWE-327 Use of a Broken or Risky Cryptographic Algorithm

- CWE-307 Improper Restriction of Excessive Authentication Attempts
- CWE-759 Use of a One-Way Hash without a Salt

In developing their Top 25 list, CWE/SANS included a comparison to the OWASP Top 10, a clear statement of the importance of OWASP's list while also recognizing distinct differences between the two.

Most clearly defined is that the OWASP Top 10 deals strictly with vulnerabilities found in web applications where the **Top 25 deals with weaknesses found in desktop and server applications as well**.

Further contrast is seen in how the list is compiled. OWASP gives more credence to the **risk** each vulnerability presents, as opposed to the CWE/SANS Top 25 that included the **prevalence** of each weakness.

This factor is what gives cross-site scripting the edge in the Top 25 as it is ranked number 1 while OWASP has it ranked at number 2.

7. Threat Actions

- **Confidentiality:** Concept of preventing the disclosure of information to unauthorized parties.
- **Integrity:** Refers to protecting the data from unauthorized alteration.
- **Availability:** Access to systems by authorized personnel can be expressed as the system's availability.
- **Authentication:** The process of determining the identity of a user.
- **Authorization:** The process of applying access control rules to a user process, determining whether or not a particular user process can access an object.
- **Accounting (Audit):** This is a means of measuring activity.
- **Non-Repudiation:** This is the concept of preventing a subject from denying a previous action with an object in a system.
- **Least Privilege:** The subject should have only the necessary rights and privileges to perform its current task with no additional rights and privileges.
- **Separation of Duties:** Ensures that for any given task, more than one individual needs to be involved.
- **Defense in Depth:** Also known by the terms *layered security* (or defense) and *diversity defense*.
- **Fail-Safe:** When a system experiences a failure, it should fail to a safe state. For example, doors should not open when there's a power failure.
- **Fail Secure:** The default state is locked or secured. So a fail-secure lock locks the door when the power is removed.
- **Single Point of Failure:** Any aspect of a system that, if it fails, the entire system fails.

8. Threat Actors

8.1. Script Kiddies

- Low skill
- Looking for easy and simple attacks
- Motivated by revenge or fame

8.2. Hacktivist

- Moderate to high skill
- Looking to make an example of an organization
- Motivated by activism

8.3. Hackers

- High skilled
- Looking to understand how things work
- Motivation varies

8.4. Cyber Criminals

- High skilled
- Looking for financial exploits
- Motivated money
    - Ransomware
    - Cryptojacking

8.5. Advanced Persistent Threat

- Very high-skilled, deep pockets
- Looking to commit cyber attacks to weaken a political adversary
- Driven largely by national interest

9. Identifying Vulnerabilities

9.1. CVE Common Vulnerabilities and Exposure

A list of common identifiers for publicly known cyber security vulnerabilities.

- One identifier for one vulnerability with a standardized description
- A dictionary rather than a database
- The way to interoperability and better security coverage
- A basis for evaluation among services, tools, and databases
- Industry-endorsed via the CVE Numbering Authorities, CVE Board, and numerous products and services that include CVE

9.2. CVSS Common Vulnerability Scoring System

The CVSS provides a way to capture the principal characteristics of a vulnerability and produce a numerical score reflecting the severity. The numerical score can then be translated into a qualitative representation (such as low, medium, high, and critical) to help organizations properly assess and prioritize their vulnerability management processes.

- Calculating a score: https://www.first.org/cvss/calculator/3.0
- Example CVE with a CVSS score: https://nvd.nist.gov/vuln/detail/CVE-2017-14977

9.3. CWE Common Weakness Enumeration

CWE is a community-developed list of common software security weaknesses. It serves as a common language, measuring stick for software security tools, and as a baseline for weakness identification, mitigation, and prevention efforts.

At its core, the CWE is a list of software weakness types:

1. **Research:** This view is intended to facilitate research into weaknesses, including their inter-dependencies and their role in vulnerability.
2. **Development:** This view organizes weaknesses around concepts that are frequently used or encountered in software development.
3. **Architecture:** This view organizes weaknesses according to common architectural security tactics.

9.4. NVE National Vulnerability Database

The NVD is the U.S. government repository of standards-based vulnerability management data represented using the Security Content Automation Protocol (SCAP). This data enabled the automation of vulnerability management, security measurement, and compliance.

Includes databases of security checklist references, security-related software flaws, misconfigurations, product names, and impact metrics.

Common Platform Enumeration (CPE) is a structured naming scheme for information technology systems, software, and packages. Based upon the generic syntax for Uniform Resource Identifiers (URI).

9.5. ASVS Application Security Verification Standard

Testing and verification require a clear set of goals and objectives. It also requires proper guidance on how to get those goals/objectives.

The ASVS defines three security verification levels, with each increasing the depth.

9.5.1. Levels

9.5.1.1. Level 1

This level is for low assurance requirements. It is completely penetration testable.

9.5.1.2. Level 2

This level is for applications that contain sensitive data, which requires protection and is the recommended level for most applications.

9.5.1.3. Level 3

This level is for the most critical applications which perform high-value transactions, contain sensitive medical data, or any application that requires the highest level of trust.

9.5.2. Domains

1. Architecture, Design and Threat Modeling Requirements
2. Authentication Verification Requirements
3. Session Management Verification Requirements
4. Access Control Verification Requirements
5. Validation, Sanitization, and Encoding Verification Requirements
6. Stored Cryptography Verification Requirements
7. Error Handling and Logging Verification Requirements

8. Data Protection Verification Requirements
9. Communications Verification Requirements
10. Malicious Code Verification Requirements
11. Business Logic Verification Requirements
12. File and Resources Verification Requirements
13. API and Web Service Verification Requirements
14. Configuration Verification Requirements

9.5.3. How to Apply ASVS

In penetration testing:

- Developers don't always open attacks on their applications
- With the ASVS, a set of guidelines on what will be tested helps define the scope
    - The scope will also include time investment
    - A risk-based approach to the scope
- The final report will show how the application performed against the **standards** and the **level**.

As a secure SDLC:

- Creating requirements from the base verification statements
    - **ASVS verification statement:** Verify that the input validation routines are enforced on the server-side.
    - **Secure requirement:** Software will validate the input on the server-side.

9.5.4. Phases

Design	ASVS is used to create secure/security requirements.
Development	Secure development.
Peer review	ASVS is used to peer-review during security code reviews.
Security tests	ASVS is used to define test plans during automated testing.

9.6. SKF Security Knowledge Framework

Developed by the OWASP, it is intended to be a tool that is used as a guide for building and verifying software security. Education is the first step in the SSDLC.

"The OWASP SKF is an expert security system web application that uses the OWASP Application Security Standard and other resources. It can be used to support developers in pre-development (security by design) as well as after code is released (OWASP ASVS Levels 1-3)."

9.6.1. Why?

- Security by design
- Information is hard to find
- Examples lack security details
- Security is hard,
- Together we can create secure web applications
- Defensive coding approach
- SKF is the first step in SDLC

9.6.2. How?

- Security requirements for development and third-party vendor applications
- Security knowledge reference (code examples/knowledge base items)

- Security is part of the design with the pre-development functionality in SKF
- Post-development functionality security in SKF for verification with the OWASP ASVS

9.6.2.1. Pre-development Stage

Here we detect threats beforehand and provide secure development patterns as well as providing feedback and solutions on how to handle threats.

9.6.2.2. Post-development Stage

Using a checklist, we guide developers through a process where we hard their application infrastructure and functions by providing feedback and solutions.

9.7. Code Review

Like threat modeling, you want to have the appropriate members involved in the review.

- Developers
- Architects
- Security Subject Matter Expert (SME)
- Depending on the portion of the application, you might need to include the SME for that topic (authentication, database logic, user experience, etc.)

9.7.1. Scope and Aid

1. Code reviews should take into consideration the threat model and high-risk transactions in the application.
2. A completed threat model will highlight the areas of concern.
3. Any time code is added/updated in those high-risk areas, a code review should include a security component.

4. When changes are required to the threat model due to findings during the review, the threat model should be updated.

9.7.2. The Risk

When considering the risk of code under review, consider some common criteria for establishing the risk of a particular code module. The higher the risk, the more thorough the review should be.

Ease of exposure	● Is the code change in a piece of code directly exposed to the internet? ● Does an insider use the interface directly?
Value of loss	● How much could be lost if the module has a vulnerability introduced? ● Does the module contain some critical password hashing mechanism or a simple change to the HTML border on some internal test tool?
Regulatory controls	● If the piece of code implements business logic associated with a standard with which must be compiled, then these modules can be considered high risk as the penalties for non-conformity can be high.

9.7.3. Understanding the Application

- Application features and business logic
- Context/sensitive data
- The code (language, feature, nuance of language)
- User roles and access rights (anonymous access)
- Application type (mobile, desktop, web)
- Design and architecture of the application

- Company standards, guidelines, and requirements that apply

The reviewer will need certain information about the development to be effective. Things like design documents, business requirements, functional specifications, test results, etc.

If the reviewer is not part of the development team, they need to talk with the developers and the lead architect for the application and get a sense of the application. Does not have to be a long meeting, it would be a whiteboard session for the development team to share some basic information about the key security considerations and controls.

9.7.4. Gathering Information

- Walkthrough of the actual running application.
- A brief overview of the code structure of the codebase and any libraries.
- Knowing the architecture of the application goes a long way in understanding the applicable security threats.
    - Tech stack, development, users, and data.
- All the required information of the proposed design including flow charts, sequence diagrams, class diagrams, and requirement documents to understand the objective of the proposed design should be used as a reference during the review.

9.7.5. Code Flow

Vulnerabilities:

- Presence of backdoor, placements of checks, insecure data binding, etc.

What to ask:

- Are there backdoor/unexposed business logic classes?
- Are there unused configurations related to the business logic?

- Proper mapping of user privileges and methods/actions allowed to them?
- Are security checks placed before processing inputs?
- Check if unexposed variables are present in form objects that get bound to user inputs. If present, check if they have default values and are initialized before binding.

9.7.6. Authentication and Access Control

Vulnerabilities:

- Insecure authentication, insecure session management, weak password handling, etc.

What to ask:

- Is the placement of the authentication and authorization check correct?
- Does authentication/authorization fail securely?
- Is the check applied on all the required fields and folders within the webroot directory?
- Is there any default configuration like all-access?
- Does the design handle sessions securely?
- Is the password complexity check enforced on the passwords?
- Are the passwords stored in an encrypted format or disclosed to the user/written to a file/log/console?

9.7.7. Data Access Mechanisms

Vulnerabilities:

- Presence of sensitive data in configuration/code, support for insecure data sources.

What to ask:

- Are database credentials stored in an encrypted format?

- Does the design support weak data stores like flat files?

9.7.8. Centralized Validation

Vulnerabilities:

- Weakness in any existing security control

What to ask:

- Does the centralized validation get applied to all requests and all the inputs and check/block all the special characters?
- Is there a special kind of request being excluded from validation?
- Does the design maintain an exclusion list for parameters or featured from being validated?

9.7.9. Architecture

Vulnerabilities:

- Insecure data handling/transmission, elevated privilege

What to ask:

- Is the data sent on an encrypted channel? (Like HTTPS)
- Does the design involve session sharing between components/modules?
  - Is the session validated correctly on both ends? (Client/server)
- Does the design use any elevated OS privileges for external connections/commands?

9.7.10. Configuration

Vulnerabilities:

- Known flaws in third parties, common security controls

What to ask:

- Is there any known flaw in APIs/technology used?
- Are all security settings enabled in the design?

9.8. Risk Rating

9.8.1. Identify a Risk

The first step is to identify a security risk that needs to be rated. The tester needs to gather information about the threat agent involved, the attack that will be used, the vulnerability involved, and the impact of a successful exploit on the business.

9.8.2. Estimate the Likelihood

Once a tester has identified a potential risk and wants to figure out how serious it is, the first step is to estimate the likelihood. At the highest level, this is a rough measure of how likely this particular vulnerability is to be uncovered and exploited by an attacker.

Here you are using the **Threat Agent Factors** and **Vulnerability Factors**.

9.8.2.1. Threat Agent Factors

The goal here is to estimate the likelihood of a successful attack by a group of threat agents. Use the worst-case threat agent.

- **Skill level:** How technically skilled is this group of threat agents?
- **Motive:** How motivated is this group to find and exploit the vulnerability?
- **Opportunity:** What resources and opportunities are required for this group of threat agents to find and exploit this vulnerability?
- **Size:** How large is this group of threat agents?

9.8.2.2. Vulnerability Factors

The goal here is to estimate the likelihood of the particular vulnerability involved being discovered and exploited.

- **Ease of discovery:** How easy is it for this group of threat agents to discover this vulnerability?
- **Ease of exploitation:** How easy is it for this group of threat agents to exploit this vulnerability?
- **Awareness:** How well-known is this vulnerability to this group of threat agents?
- **Intrusion detection:** How likely is an exploit to be detected.

9.8.3. Estimate Impact

When considering the impact of a successful attack, it's important to realize that there are two kinds of impact. The first is the **technical impact** on the application, the data it uses, and the functionality it provides. The second is the **business impact** on the organization operating the application.

9.8.3.1. Technical Impact

The technical impact can be broken down into factors aligned with the traditional security areas of concern (CIA + accountability). The goal is to estimate the magnitude of **the impact on the system** if the vulnerability was exploited.

- **Loss of confidentiality:** How much data could be disclosed and how sensitive it is?
- **Loss of integrity:** How much data could be corrupted and how damaged is it?
- **Loss of availability:** How much service could be lost and how vital is it?
- **Loss of accountability:** Are the threat agents' actions traceable to an individual?

9.8.3.2. Business Impact

Stems from the technical impact but requires a deep understanding of **what is important to the organization running the application**. In general, you should be aiming to support your risks with business impact, particularly if your audience is executive level. The business risk is what justifies investment in fixing security problems.

- **Financial damage:** How much financial damage will result from an exploit?
- **Reputation damage:** Would an exploit result in reputation damage that would harm the business?
- **Non-compliance:** How much exposure does non-compliance introduce?
- **Privacy violation:** How much PII could be disclosed?

9.8.4. Severity of the Risk

In this step, the likelihood estimate and the impact estimate are put together to calculate the overall security for this risk. This is done by figuring out whether the likelihood is low, medium, or high and then do the same for the impact.

- **Informal:** In many environments, there is nothing wrong with reviewing the factors and simply capturing the answers. The tester should think through the factors and identify the key *driving* factors that are controlling the result.
- **Repeatable:** If it is necessary to defend the rating or make them repeatable then it is necessary to go through a more formal process of rating the factors and calculating the result.

Example:

Vulnerability Factors			
Easy of Discovery	Ease of Exploit	Awareness	Intrusion Detection
3	6	9	2
Threat Agent Factors			
Skill Level	Motive	Opportunity	Size
5	2	7	1
Overall likelihood: 4.375 (avg)			

Technical Impact			
Confidentiality Loss	Integrity Loss	Availability Loss	Accountability Loss
9	7	5	8
Overall technical impact: 7.25 (high)			
Business Impact			
Financial Damage	Reputation Damage	Non-Compliance	Privacy Violation
1	2	1	5
Overall business impact: 2.25 (low)			

Likelihood/impact levels: 0-2 low, 3-5 medium, 6-9 high.

9.8.5. Deciding What to Fix

After the risks to the application have been classified, there will be a prioritized list of what to fix. As a general rule, the most severe risks should be fixed first. It simply doesn't help the overall risk profile to fix less important risks even if they are easy or cheap to fix.

Remember that not all risks are worth fixing and some loss is not only expected but justifiable based upon the cost of fixing the issue. For example, if it would cost $100,000 to implement controls to step $2,000 of fraud per year, it would take 50 years of ROI to stamp out the loss. But remember, there may be reputation damage from the fraud that could cost the organization much more.

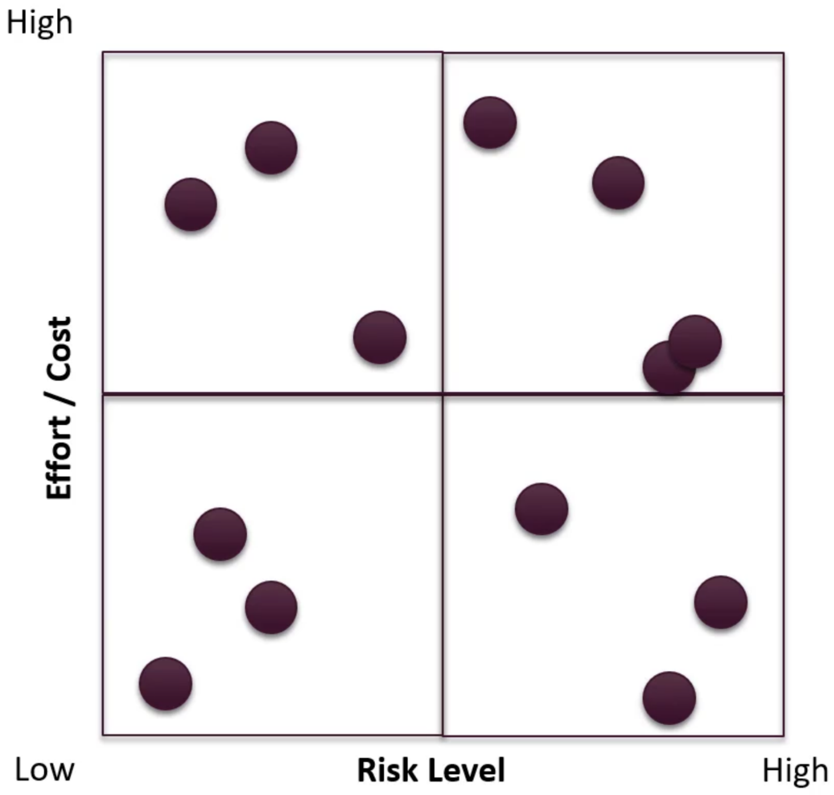


Fig. 9.1. Risk level vs. effort/cost

9.8.6. Example

My health records is a SaaS application that provides a user the ability to view their medical records from multiple sources (like an HIE).

Functional Description:

- The user must authenticate using a username and password.
- The application only allows view rights, not updating of medical information.
- The user can access the application from multiple devices as long as it has a browser.

- DocumentViewer allows users to search medical records using date ranges, keywords, and encounter locations.
- The user can print documents.

Online risk calculator: https://security-net.biz/files/owaspriskcalc.html

9.9. Threat Modeling

Threat modeling is a structured approach to identify, quantify, and address the security threats and risks associated with an application. Is an investigative technique used to identify application risks/hazards that are technical (and even implementation-specific)?

9.9.1. Definitions

9.9.1.1. Abuser

Those who would misuse intentionally or unintentionally an element of the system under consideration.

- Hacktivists
- Cyber-criminals
- Advanced persistent threats

9.9.1.2. Asset

Anything we deem to have a value that the system must protect from abusers.

- Money
- Precious metals
- Houses, cards, etc.
- Data (PHI, PII, PCI, etc.)

9.9.1.3. Hazard

A potential source of harm or danger. A harmless state with the potential to change into a threat.

- Tree hazards include dead or dying trees, dead parts of live trees, or unstable alive trees that are within a distance of people of property (a target). Hazard trees have the potential to cause property damage, personal injury, or fatality in the event of a failure.

9.9.1.4. Threat

A means by which an abuser might compromise an asset that has potential for success. Threats can include everything from hackers and malware, from earthquakes to wars. The intention is not a factor when considering threats, thus the mechanical failure of a typical platter hard drive is a threat as is a coordinated attack by an abuser.

- A threat is a specific type of hazard involving an abuser potentially harming an asset in a harmful state

9.9.1.5. Risk

The potential of loss, damage, or destruction of an asset as the result of a threat exploiting a vulnerability. Based on the previous example, the mechanical failure of a hard drive is probably more likely than an attack by an abuser but the overall impact might be significantly lower thus making it less risky.

9.9.2. Identify, Quantify, and Address

Do not trust your gut feelings. As humans, we are naturally included to make immediate decisions based upon a feeling.

Common phrases that distract (these should still be identified in the thread model):

- Well, the attacker would need to be on the internal network.
- That is only exploitable under this specific condition.

What to do:

- If you have questions you can't answer right away, don't get blocked. Concentrate on what you do know, document what you don't, and find out later.
- Document any assumptions you make and follow up on them later.
- Don't assume you have a secure environment. In most cases, you don't have control over it.
- Don't assume that compute, network or storage resources are reliable. Can an application successfully survive the loss of storage, network outages, etc., and stay healthy?
- When threat modeling, don't assume your environment is correctly configured.

9.9.3. Benefits

- A better understanding of the architecture
- Create reusable architecture models
- Inputs intro:
  - Penetration testing
  - Scanning
  - Code reviews
  - Risk management

9.9.4. Types

9.9.4.1. Manual

- More favorable in terms of quality and custom-ability.
- Just needs a whiteboard and a group of experts on the product and security
- Not scalable

- Best done in groups of a manageable size (6-8 people), including
  - Implementation expert (architect)
  - Solutions designer
  - Implementation team
  - **Security SME**

9.9.4.1.1. Audience

- **Your team:** The threat model becomes a reference for understanding the security of your solution, and therefore, is like a system-level tech design.
- **Pen testers:** This is a map to potentially hacking the application.
- **Other teams:** Others may rely on your components to understand their security. Threat models should reference related threat models.
- **Clients:** Your clients may ask to see if you are considering security. You would most likely hand over a high-level and not a raw threat model.

9.9.4.1.2. Mindset

- What are you building?
- What can go wrong?
- What should you do about those things that can go wrong?
- Did you do a decent job analysis?

Threat models are never complete. This is a living artifact expected to change and grow over time. A complex system is never truly complete.

9.9.4.1.3. Sample Board

Who	What	Why	How	Impact	Counter measure
Russian Mob	Credit Info	Financial Gain	Phishing	Limited to targets	2-FA
Hollywood					

9.9.4.2. Tool-Assisted

- Scalable
- Consensus on the tool is not always possible. Some may use Wiki, PPT, Visio, architecture tools, or a TM tool.
- Can lead to a “check the box” mentality. This means that the people involved may feel that there’s nothing else that needs to be done once the items are completed, which is not always the case.

9.9.4.3. OCTAVE

Operational Critical Threat Asset and Vuln Evaluation.

- Focuses on the non-technical risk that results from breached data assets.
- Assets are identified and classified. This helps define the scope.
- The drawback is that as systems grow, re-doing the identification and classification can be difficult/time-consuming.

9.9.4.4. PASTA (Business Impact)

Process for Attack Simulation and Threat Analysis.

- Takes an attacker view and then develops threat management, enumeration, and scoring process. This can then be elevated to key decision-makers to determine what risk to tackle as opposed to developing requirements at the SDLC level.

9.9.4.5. STRIDE (Technical Impact)

- Stands for **Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege**.
- Used in MS-TM and MS SDL (Security Development Lifecycle) and one that is generally easy to follow if doing a manual threat model.
- Builds a DFD used to identify system entities, events, and boundaries then applies a general set of known threats.

9.9.4.5.1. Spoofing

One person or program successfully masquerades as another by falsifying data, thereby, gaining an illegitimate advantage.

Example: Threat action aimed to illegally access and use another user's credentials, such as username and password.

Security control: Authentication.

9.9.4.5.2. Tampering

Intentional modification of products in a way that would make them harmful to the consumer.

Example: Threat action aimed to maliciously change or modify persistent data, such as databases and the alteration in transit between two computers over an open network, such as the internet.

Security control: Integrity checks using hashing algorithms/checksum.

9.9.4.5.3. Repudiation

State of affairs where the author of a statement will not be able to successfully challenge the authorship of the statement or validity of an associated contract.

Example: Threat action to perform illegal operations in a system that cannot trace the prohibited operations.

Security control: Non-repudiation using encryption, digital signatures, and notarization.

9.9.4.5.4. Information Disclosure

The intentional or unintentional release of secure or private/confidential information to an untrusted environment.

Example: Threat action to read a file that one was not granted access to, or to read data in transit.

Security control: Confidentiality through encryption.

9.9.4.5.5. Denial of Service

A cyber-attack where the perpetrator seeks to make a machine or network resource unavailable to its intended users by temporarily or indefinitely disrupting the services of a host.

Example: Threat aimed to deny access to valid users, such as by making a web server temporarily unavailable or unusable.

Security control: Availability.

9.9.4.5.6. Elevation of Privileges

Is the act of exploiting a bug, design flag, or configuration oversight in an operating system or software application to gain elevated access to resources that are normally protected from an application or user.

Example: Threat aimed to gain privileged access to resources for gaining unauthorized access to information or to compromise a system.

Security control: Authorization.

9.10. Handling Risks

- **Accept:** Document the risk, acknowledge it, and assign ownership.
- **Avoid:** Place other controls that will reduce or eliminate the risk.
- **Mitigate:** Fix the issue that exposes you to the risk.
- **Transfer:** If you’re practically unable to deal with a risk, you may contractually obligate someone else to accept the risk.

10. Defense

"An approach to cybersecurity in which a series of defensive mechanisms are layered to protect valuable data and information. If one mechanism fails, another steps up immediately to thwart an attack."

– ForcePoint

- Don't rely on defense-in-depth to always protect your app
- Systems fail, they can be circumvented by the weakest link
- Your app may not always be behind those defenses

10.1. Defense-in-Depth

The principle of defense-in-depth is that layered security mechanisms increase the security of the system as a whole.

In theory, in defense-in-depth, you never make a tradeoff between performance and security, but in reality, you do. To properly calculate those choices, you need the full context of your stack.

10.2. Content Security Policy (CSP)

CSP is an added layer of security that helps to detect and mitigate certain types of attacks including CSS and injection.

To enable CSP you need to configure your web server to return the CSP HTTP header. Browsers that don't support it still work with servers that implement it.

CSP makes it possible for server administrators to reduce or eliminate the vectors by which XSS can occur by specifying the domains that the browser should consider to be valid sources of executable scripts.

A CSP compatible browser will then only execute scripts loaded in source files received from those domains in the allowlist, ignoring all other scripts (including inline scripts and event-handling HTML attributes).

As an ultimate form of protection, sites that want to never allow scripts to be executed can opt to globally disable script execution.

- A policy is described using a series of policy directives, each of which describes the policy for a certain resource type or policy area. The policy should include a `default-src` directive which is a fallback for other resource types when they don’t have policies of their own.
- A policy needs to include a `default-src` or `script-src` directive to prevent inline scripts from running, as well as blocking the use of `eval()`.
- A policy needs to include a `default-src` or `style-src` directive to restrict inline styles from being applied from an `<style>` element or a `style` attribute.

10.3. Security Model

Security models are used to understand the systems and processes developed to enforce security principles. Three key elements play a role in systems concerning model implementation:

1. People
2. Processes
3. Technology

Addressing a single element of the three may provide benefits, but more effectiveness can be achieved through addressing multiple elements.

How security models are used in an OS design:

- Security Policy
    - □ NIST
    - □ ISO

    - ☐ FIPS
- Security Model
    - ☐ Biba
    - ☐ Bell LaPadulla
    - ☐ State Machine
- Programming Code
    - ☐ SQL
    - ☐ C#
    - ☐ Python
- Operating System
    - ☐ Windows
    - ☐ Linux
    - ☐ macOS

10.3.1. Access Control Model

10.3.1.1. Access Control List (ACL)

It is a list of permissions attached to an object. An ACL specifies which users or system processes are granted access to objects, as well as what operations are allowed on given objects.

10.3.1.2. Bell-LaPadula Model

A formal state transition model of computer security policy that describes a set of access control rules which use security labels on objects and clearances for subjects. Security labels range from the most sensitive (e.g. top secret) down to the least sensitive (e.g. unclassified or public).

10.3.1.3. Role-based Access Control (RBAC)

It is a policy-neutral access control mechanism defined around roles and privileges. The components of BRAC such as role-permissions, user-role, and role-role relationships make it simple to perform user assignments.

10.3.1.4. Attribute-based Access Control (ABAC)

Also known as *policy-based access control*, defines an access control paradigm whereby access rights are granted to users through the use of policies that combine attributes.

The policies can use any type of attribute (user attributes, resources attributes, objects, environment attributes, etc.). This model supports boolean logic, in which rules contain IF, THEN statements about who is making the request, the resource, and the action.

10.3.2. Multi-level Security Model

10.3.2.1. Brewer-Nash Model

Also known as the **Chinese Wall**, this technology can be employed to prevent access to data by conflicting groups. People can be trained not to compromise the separation of information. Policies can be put in place to ensure that the technology and the actions of personnel are properly engaged to prevent compromise.

10.3.2.2. Data Flow Diagrams

Specifically designed to document the storage, movement, and processing of data in a system. They are constructed on a series of levels (highest to lowest):

- **Level 0:** High-level contextual view of the data flow through the system.
- **Level 1:** Created by expanding elements of the level 0 diagram.
- **Level 2:** Level 1 exploded diagram or lowest-level diagram of a system.

10.3.2.3. Use Case Models

Requirements from the behavioral perspective describe how the system utilizes data. Use cases are constructed to demonstrate how the system processes data for each of its defined functions.

10.3.2.4. Assurance Models

The level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its lifecycle, and that the software functions in an intended manner.

11. Attacks

The anatomy of an attack:

- **Vulnerability:** Adobe Flash CVE-2016-0960
- **Exploit:** The code is written to take advantage of the vulnerability
- **Payload:** Ransomware, Trojan, RAT, keylogger...

11.1. Injection

Anytime user input changes the intended behavior of the system. There are many types of injection attacks, to mention a few:

- Queries:
  - SQL queries
  - LDAP queries
  - XPath queries
- OS commands
- HTTP redirects

It happens when you:

- Trust user input
- Don’t run input sanitization
- Don’t separate untrusted data

11.1.1. SQL Injection

The Structured Query Language (SQL) can be manipulated to perform tasks that were not the original intent of the developer. This is called SQL Injection.

SQL injection (also called SQLi) is one of the most common web hacking techniques. A SQL injection attack consists of insertion or “injection” of malicious code via the SQL query input from the client to the application. If not dealt with correctly, SQL injection can seriously impact data integrity and security.

SQL injections can occur when unfiltered data from the client (such as input from a search field) gets into the SQL interpreter of the application itself. If an application fails to either correctly sanitize user input (using prepared statements or similar) or filter the input against special characters, hackers can manipulate the underlying SQL statement to their advantage.

For example, if the input is not filtered for SQL metacharacters like `--` (which comments out the rest of the line) or `;` (which ends a SQL query), SQL injection may happen.

There are many types of SQLi, to mention a few:

- **String SQLi:** Take advantage of the way the SQL parser handles the queries. You can append behavior to the query when the unprepared value is a string (or delimited by single quotes).
- **Numeric SQLi:** Just like String SQLi, this attack takes advantage of the parser but unlike String SQLi, you don't need to "close" the unprepared value, but to add a replacement value and add behavior.
- **Blind SQLi:** Asks the database **`true`** or **`false`** questions and determines the answer based on the application's response. This attack is often used when the web application is configured to show generic error messages but has not mitigated the code that is vulnerable to SQL injection.

11.1.1.1. Data Manipulation Language (DML)

As implied by the name, data manipulation language deals with the manipulation of data. Many of the most common SQL statements, including **`SELECT`**, **`INSERT`**, **`UPDATE`**, and **`DELETE`**, may be categorized as DML statements.

DML statements may be used for requesting records (**`SELECT`**), adding records (**`INSERT`**), deleting records (**`DELETE`**), and modifying existing records (**`UPDATE`**).

If an attacker succeeds in “injecting” DML statements into a SQL database, he can violate confidentiality (using **SELECT** statements), integrity (using **UPDATE** statements), and availability (using **DELETE** or **UPDATE** statements) of a system.

11.1.1.2. Data Definition Language (DDL)

DDL includes commands for defining data structures. DDL commands are commonly used to define a database’s schema. The schema refers to the overall structure or organization of the database and in SQL databases, includes objects such as tables, indexes, views, relationships, triggers, and more.

If an attacker successfully “injects” DDL SQL commands into a database, he can violate the integrity (using **ALTER** and **DROP** statements) and availability (using **DROP** statements) of a system.

11.1.1.3. Data Control Language (DCL)

DCL is used to implement access control logic in a database. DCL can be used to revoke and grant user privileges on database objects such as tables, views, and functions.

If an attacker successfully “injects” DCL SQL commands into a database, he can violate the confidentiality (using **GRANT** commands) and availability (using **REVOKE** commands) of a system. For example, the attacker could grant himself admin privileges on the database or revoke the privileges of the true administrator.

11.1.1.4. Example

Consider a web application that allows users to retrieve user information simply by inputting a username into a form field. The input from the user is sent to the server and gets inserted into a SQL query which then is processed by a SQL interpreter.

```
"SELECT * FROM users WHERE name = '" + userName + "'";
```

The variable userName holds the input from the client and “injects” it into the query. If the input were **Smith** the query would then become:

```
"SELECT * FROM users WHERE name = 'Smith'";
```

It would retrieve all data for the user with the name Smith.

SQL injection can be used for far more than reading the data of a single user. The following are just a few examples of data a hacker could input in a form field (or anywhere user input is accepted) in an attempt to exploit a SQL injection vulnerability:

Input	`Smith' OR '1' = '1`
Result	`SELECT * FROM users WHERE name = 'Smith' OR TRUE;`
Description	All entries from the **`users`** table.

Input	`Smith' OR 1 = 1; --`
Result	`SELECT * FROM users WHERE name = 'Smith' OR TRUE;--';`
Description	All entries from the **`users`** table.

Input	`Smith'; DROP TABLE users; TRUNCATE audit_log; --`
Result	`SELECT * FROM users WHERE name = 'Smith'; DROP TABLE users; TRUNCATE audit_log; --;`
Description	Chains multiple SQL-Commands to both **`DROP`** the users table and delete all entries from the **`audit_log`** table.

11.1.1.5. Consequences

A successful SQL injection exploit can:

- Read and modify sensitive data from the database
- Execute administrative operations on the database
- Shutdown auditing or the DBMS
- Truncate tables and logs
- Add users
- Recover the content of a given file present on the DBMS file system

- Issue commands to the operating system

SQL injection attacks allow attackers to

- Spoof identity
- Tamper with existing data
- Cause repudiation issues such as voiding transactions or changing balances
- Allow the complete disclosure of all data on the system
- Destroy the data or make it otherwise unavailable
- Become administrator of the database server

11.1.1.6. Severity

The severity of SQL injection attacks is limited by

- Attacker’s skill and imagination
- Defense in depth countermeasures
    - Input validation
    - Least privilege
- Database technology

11.1.1.7. Impact

An attacker could:

- Steal all data from the database
- Access Personally Identifiable Information (PII)/Protected Health Information (PHI)/Payment Card Information (PCI) data
- Take over the backend server or entire network
- Remove data

11.1.1.8. Prevention

- Parameterized queries

- Prepared statements
- Heavy type-checks
- Parameter sanitization (allow lists) should be used in conjunction with parameterized queries
- Use secure server configuration
    - Accounts with least privileges as possible
    - Firewalls
    - Implement allow lists, deny by default

11.1.2. OS/Command Injection

An OS command injection is a web security vulnerability that enables the execution of unauthorized operating system commands. An OS command injection vulnerability arises when a web application sends unsanitized, unfiltered system commands to be executed.

11.1.3. XPath Injection

XPath Injection is an attack technique used to exploit applications that construct XPath (XML Path Language) queries from user-supplied input to query or navigate XML documents.

11.1.4. LDAP Injection

Lightweight Directory Access Protocol (LDAP) is an open-standard protocol for both querying and manipulating X.500 directory services (like Active Directory).

LDAP injection is an attack technique used to exploit websites that construct LDAP statements from user-supplied input. The LDAP protocol runs over internet transport protocols such as TCP. Web applications may use user-supplied input to create custom LDAP statements for dynamic web page requests.

When an attacker can modify an LDAP statement, the process will run with the same permissions as the component that executed the command (e.g. a database server, web server, application server, etc.).

11.1.4.1. Example

```
https://www.example.com/default.aspx?user=*
```

In the example above, we send the * character in the user parameter which will result in the filter variable in the code to be initialized:

```
with(samAccountName=*)
```

The resulting LDAP statement will make the server return all the objects that contain the `samAccountName` attribute. In addition, the attacker can specify other attributes to search for and the page will return an object matching the query.

11.1.5. Testing for Input

- Input unexpected data into all parameters.
- Put simple SQL injections like `'` into every field.
- Check all parameters.
- Use automated code review tools like static analysis when possible.

11.1.6. Defenses

- Never trust user input
- Scrub your inputs
- Whitelist your expected parameters
- Use prepared statements

11.2. Insecure Deserialization

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object to be able to recreate it when needed.

Serialization may be used in applications for:

- Remote and inter-process communication (RPC/IPC)
- Wire protocols, web services, message brokers
- Caching/Persistence
- Databases, cache servers, file systems
- HTTP cookies, HTML form parameters, API authentication tokens

The reverse process of serialization is called **deserialization**. Takes the data structured in some format and rebuilds it into an object. Today, the most popular data for serializing data is JSON. Before that it was XML.

Many programming languages offer a native capability for serializing objects. These native formats usually offer more features than JSON or XML, including customizability of the serialization process.

Unfortunately, the features of these native deserialization mechanisms can be repurposed for malicious effects when operating on untrusted data. Attacks against deserializers have been found to allow denial-of-service, access control, and remote code execution attacks.

11.2.1. Prevention

The only safe architectural pattern is not to accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types.

Other options:

- Implementing **integrity checks** such as digital signatures on any serialized objects to prevent hostile object creation or data tampering.
- Enforcing **strict type constraints** during deserialization before object creation as the code typically expects a definable set of classes.
- Isolating and running code that **deserializes in low privilege environments** when possible.
- **Logging deserialization exceptions and failures**, such as where the incoming type is not the expected type, or the deserialization throws exceptions.
- **Restricting or monitoring incoming and outgoing network connections** from containers or servers that deserialize.
- **Monitoring deserialization**, alerting if a user deserializes constantly.

11.3. Insecure Logging

The exploitation of insufficient logging and monitoring is the bedrock of nearly every major incident. Attackers rely on the lack of monitoring and timely response to achieve their goals without being detected.

Most successful attacks start with vulnerability probing. Allowing such probes to continue can raise the likelihood of a successful exploit by nearly 100%.

In 2016, identifying a breach took an average of 191 days, plenty of time for damage to be inflicted.

Insufficient logging, detection, monitoring, and active response occurs any time:

- **Auditable events**, such as logins, failed logins, and high-value transactions are not logged.
- Warnings and errors generate **no**, **inadequate**, or **unclear log messages**.
- Logs of applications and APIs are **not monitored** for suspicious activity.
- Logs are **only stored locally**.
- Appropriate alerting thresholds and response **escalation processes are not in place** or effective.

- Penetration testing and scans by DAST tools (such as OWASP ZAP) **do not trigger alerts**.
- The application is unable to detect, escalate, or alert for active attacks in real-time or near real-time.

11.3.1. Good Practices

As per the risk of the data stored or processed by the application:

- Ensure all login, access control failures, and server-side input **validation failures can be logged** with sufficient user context to identify suspicious or malicious accounts, and held for sufficient time to allow delayed forensic analysis.
- Ensure that **logs are generated in a format that can be easily consumed by centralized log management solutions**.
- Ensure **high-value transactions have an audit trail** with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
- Establish effective monitoring and alerting such that **suspicious activities are detected** and responded to in a timely fashion.
- Establish or adopt an **incident response and recovery plan**.
- SIEM (Security Information and Event Management).

11.4. Security Misconfiguration

Security misconfiguration is the absence of security settings in:

- Application
- Framework
- Database
- Web server
- Platform

Also the lack of:

- Patching
- Secure settings for parsers
- Outdated security configuration
- Default settings/passwords

Debug messages allow attackers to understand the flow of a program, thus, better planning for an attack. Debug should never be turned on in production environments.

Directory listing leakage lets you enumerate the HTTP responses from a server to get information.

- Default resource within an unintended directory (e.g. web server root), like `index.html`, `home.html`.
- 403 Forbidden error message (susceptible to enumeration attack) HTTP status code.
- 404 Not Found error message (susceptible to enumeration attack).
- Listing showing the contents of the directory, world-readable (directory indexing vulnerability).

11.4.1. Defenses

There are a lot of “secure configurations”, too numerous to list here. The strategy is to understand what you are implementing.

- Remove default settings
- Ensure that explicit security settings are enabled
- Keep in mind the basics:
    - Confidentiality, Integrity, Availability
    - Least privilege

11.5. Sensitive Data Exposure

Sensitive data exposure is the breach of data that should have been, otherwise, protected. Attackers steal or modify data accessible to them due to weak protection mechanisms. Protection mechanisms in this area include encryption algorithms as well as hashing functions.

11.5.1. Protected Health Information

- Names
- Dates
- Phone/Fax numbers
- Email
- Social Security Number (SSN)
- Medical Record Number (MRN)
- Account numbers
- Images
- Biometric data (finger, retinal, voiceprints, etc

11.5.2. Personally Identifiable Information

- Names
- Address
- Passport
- Vehicle information
- Driver’s license
- Credit card numbers
- Digital identity
- Birthplace
- Generic information
- Login/credentials

11.5.3. Sensitive Financial Information

- Credit/debit card numbers
- Security codes
- Account numbers
- Loan agreements
- Loan details
- Tax ID
- POS transactions

11.5.4. Protecting

11.5.4.1. Protecting Data at Rest

- Application layer
- Database layer
- Filesystem level
- Media/device level

11.5.4.2. Protecting Data Access (Access Controls)

- Account/data access based on role/privilege

11.5.4.3. Protecting Data in Motion

- Transport layer encryption
- Digital certificates (client and server-side)
- Payload encryption

11.5.4.4. Protecting Data in Use (Memory Leak)

- Secure compiler settings
- Code quality/cleanup issues
- No insecure direct object references

11.6. XML External Entities (XXE)

Attackers can exploit vulnerable XML processors if they can upload XML or include hostile content in an XML document, exploiting vulnerable code, dependencies, or integrations.

By default, many older XML processors allow the specification of an external entity, a URI that is dereferenced and evaluated during XML processing.

These flaws can be used to extract data, execute a remote request from the server, scan internal systems, perform a Denial-of-Service (DoS) attack, as well execute other attacks.

Applications and in particular XML-based web services or downstream integrations might be vulnerable to attacks if:

- The application **accepts XML directly** or **XML uploads**, especially from untrusted sources, or inserts untrusted data into XML documents, which is then **parsed by an XML processor**.
- The application **uses Security Assertion Markup Language** (SAML) for identity processing within federated security (single access to multiple systems across different enterprises) or single sign-on (SSO, single access to different systems within a single organization) purposes. SAML uses XML for identity assertions and may be vulnerable.
- If the application **uses SOAP before version 1.2**, it is likely susceptible to XXE attacks if XML entities are being passed to the SOAP framework.
- Being vulnerable to XXE attacks likely means that the application is vulnerable to DoS attacks including the **Billion Laughs Attack**.

11.6.1. Protecting

Developer training is essential to identify and mitigate XXE. Besides that, preventing XXE requires:

- Whenever possible, use less complex data formats such as JSON, and **avoid serialization of sensitive data**.
- **Patch or upgrade all XML processors** and libraries in use by the application or on the underlying operating system.
- **Update SOAP to 1.2 or higher**.

- **Disable XML external entity processing** in all XML parsers in the application.
- **Implement positive (or allow-lists) server-side input validation**, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes.
- Verify that XML or XSL file upload functionality **validates the incoming XML using XSD** validation or similar.
- **Static Application Security Testing (SAST) tools** can help detect XXE in source code, although manual code review is the best alternative in large, complex applications with many integrations.

11.7. Broken Access Control

Access control enforces policy such that users cannot act outside of their intended permissions. Failures typically lead to unauthorized information disclosure, modification or destruction of all data, or performing a business function outside the limits of the user.

Access control weaknesses are common due to the lack of automated detection, and lack of effective functional testing by application developers.

The technical impact is attackers acting as users or administrators, or users using privileged functions, or creating, accessing, updating, or destroying every record.

11.7.1. Authorization

Authorization is the process where requests to access a resource should be granted or denied. It should be noted that authorization is not equivalent to authentication –as these terms and their definitions are frequently confused.

- **Authentication:** Prove and validate identity.

- **Authorization:** Execution of rules that determine what functionality and data the user (or Principal) may access, ensuring the proper allocation of access rights after the authentication is successful.

Having a license doesn't mean you are granted access to a military base. You have authentication, but not authorization.

11.7.2. Common Vulnerabilities

- Bypassing access control checks by modifying the URL, internal application state, or the HTML page, or simply using a custom API attack tool.
- Allowing the primary key to be changed to another's user record, permitting viewing or editing someone else's account.
- Elevation of privilege by acting as a user without being logged in, or acting as an admin when logged in as a standard user.
- Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT), access control token or a cookie, a hidden field manipulated to elevate privileges, or abusing JWT invalidation.
- CORS misconfiguration allows unauthorized API access.
- Force browsing to authenticated pages as an unauthenticated user or privileged pages as a standard user.
- Accessing API endpoints with missing access controls for `POST`, `PUT`, and `DELETE`.

11.7.3. Prevention

- Apart from public resources, deny by default.
- Implement access control mechanisms once and reuse them throughout the application, including minimizing CORS usage.
- Model access controls should enforce record ownership, rather than accepting that the user can create, read, update, or delete any record.
- Disable web server directory listing and ensure file metadata (e.g. git) and backup files are not present within web roots.

- Log access control failures, alert admins when appropriate (e.g. repeated failures).
- Rate-limit APIs and controller access to minimize the harm from automated attack tooling.
- JWT tokens should be invalidated on the server after logout.

11.8. Broken Authentication

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, session tokens, or to exploit other implementation flaws to assume other users' identities.

Developers frequently build custom authentication and session management schemes but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, password management, timeout, remember me, secret questions, account updates, etc.

Finding such flaws can sometimes be difficult, as each implementation is unique. The bottom line is building authentication and session management schemes is hard.

11.8.1. Identifying

- **Missing authentication:** Shared and/or bookmarked URL circumvented authentication.
- **Poor credential strength:** Well-known credentials (ID/password leaks).
- **Poor credential handling/management/storage:** Usernames, passwords, emails, etc. as query strings or stored in plain text.
- **Poor account recovery practices:** For example, using "secret questions" with answers that can be easily socially engineered.
- **Unlimited attempts.**

11.8.2. Attacks

- **Password guessing attack:** Via social engineering. For instance *“John from IT needs your help…”*.
- **Dictionary attack:** Dictionary words that are hashed and tested against an application.
- **Brute force attack:** Guessing or targetted hashes.
- **Username enumeration:** Guessable patterns of usernames or login failure messages that reveal too much information.
- **Phishing:** Trick users into providing their credentials to an imposter, look-alike site.

11.8.3. Account Recovery Risks

- **Social engineering:** Emailing a password reset form without using something like two-factor flows.
- **Easily guessable security answers:** Questions like *“what school did you attend”* can be easily found/guessed/socially engineered.
- **Password sent through insecure channels:** For example, plain text password sent via email.
- **Password change not required:** Once you’ve been given a new password, it should be changed on the next login.

11.9. Cross-Site Scripting (XSS)

XSS is a security vulnerability typically found in web applications that enables attackers to inject client-side scripts into web pages viewed by other users.

A cross-site scripting vulnerability may be used by attackers to bypass access controls such as the same-origin policy.

This subversion is possible because the web application fails to properly validate input from the web browser (i.e. client) and/or fails to properly escape that input in the response.

11.9.1. Reflected XSS

The reflected XSS attack is the appending of a malicious script to a command or URL, and the results are reflected to the victim.

An attacker can craft a special URL that points to this web page and also inserts some of the attacker's content into the page. This content could consist of a form that requires a user for credentials (passwords, credit card numbers, etc.) and passes those values back to the attacker.

11.9.2. Stored XSS

The stored (or persisted) XSS vulnerability is a more devastating variant of a cross-site scripting flaw. It occurs when the data provided by the attacker is saved by the server, and then permanently displayed on "normal" pages returned to other users in the course of regular browsing, without proper HTML escaping.

A classic example of this is with online message boards where users are allowed to post HTML formatted messages for other users to read.

11.10. Using Vulnerable Components

The term *components* in this section refers to application frameworks, libraries, or other software modules integrated into an application as *dependencies*. Such components are usually written by third parties but this is not exclusive.

This section references using these components when they may have malicious code or security weaknesses within them, therefore becoming a vulnerable component.

Dependency is a broad software engineering term used to refer to when a piece of software relies on another one.

11.10.1. Dependency Types

- A **direct dependency** is a functionality exported by a library, API, or any software component that is referred directly to by the program itself.
- A **transitive dependency** is a functional dependency that holds by transitivity among various software components.

11.10.2. Defences

Most applications include either commercial products or open-source software (OOS) within their software bundles.

11.10.2.1. Commercial

For commercial products, most major vendors such as Oracle, Google, and IBM provide security bulletins to distribution lists for notification purposes. Make sure you are signed up for these.

11.10.2.2. Open Source

Solutions like Dependency Check or Bundler Audit are OOS and may help you to automatically scan for vulnerable packages.

Sign up for regular security bulletins from the National Vulnerability Database.

11.10.2.3. General

- Do not give extreme trust to any 3rd-party component.
- Always verify package size and checksum.
- Download components directly from the vendor website, never a secondary.

- Challenge the vendor to provide evidence of security vulnerability scanning.
- Use well-known vendors that are maintained.
- If possible, scan the code yourself

11.10.3. Patching & Notifications

11.10.3.1. Notifications

- Have a means for receiving notifications on potentially vulnerable software.
- Many vendors like Microsoft already offer a notification service, however, other services and feeds exist.
- Receiving notifications is only one part of the solution. You must also be able to:
    - Know where to patch (what systems or software are vulnerable)
    - Have the ability to test the new patch
    - Have the means to deliver the patch
    - Ability to notify those impacted by the changes to the system (users, customers, etc.)

11.10.3.2. Patching

Recall the SDLC and adapt it to:

- **Development:** Make sure the patch works in development.
- **Pre-prod (staging):** Test the patch in a production-equivalent environment to make sure it will work when the production environment is updated.
- **Production:** Release the new version to the public.

12. Session Management

A web session is a sequence of network HTTP request and response transactions associated with the same user. Modern and complex web applications require the retaining of information or status about each user for the duration of multiple requests.

HTTP is a stateless protocol (RFC2616 Section 5), where each request and response pair is independent of other web interactions. Therefore, to introduce the concept of a session, it is required to implement session management capabilities that link both the authentication and authorization (or access control) modules commonly available in web applications.

The session ID or token binds the user authentication credentials (in the form of a user session) to the user HTTP traffic and the appropriate access controls enforced by the web application.

12.1. Poor Session Lifecycle Management

- Guessable session IDs
- Not destroying sessions when the session is over
- Reusing sessions
- Exposing sessions

12.2. Defenses

- Generate unique session IDs
  - Use a large range of possible values, consider GUIDs
  - Generate them in a random (un-guessable) order
- Do not reuse session IDs
- Distinguish between “session” identifiers generated before and after user authentication

    - Always generate a new identifier after login and logout
- **Do not expose session IDs** in application URLs

12.3. Best Practices

- Automatically end/expire sessions after periods of user inactivity
- Do not allow infinitely-lived sessions (even with user activity)
- Always offer users a logout option
    - Do not rely on expiry alone
    - Do not assume users know to close the browser
- Tie audit events to the user session
    - Use an audit-trail-safe session alias
    - Audit IP addresses and `X-Forwarded-For`

12.4. Session Cookies

- **Mark session cookies as `Secure`:** Instructs browser to send cookies with HTTPS requests only preventing eavesdropping (i.e. session sniffing).
- **Mark session cookies as `HttpOnly`:** Prevents client code (scripts) from accessing the cookie, preventing XSS attacks.
- **Avoid setting values for `Max-Age` and `expires`:** This ensures the browser will remove cookies when it's closed. Setting either value causes the browser to treat the cookie as a persistent cookie.

As OWASP says	Instead, use
Authentication, authorization, and session management are hard.	Identity federation.
	Standard authentication protocols.
Rolling your solution is perilous.	Hardened web frameworks.

12.5. Web Sessions

A session is a conversational state between the client and server-side and consists of multiple requests and responses between them.

Since HTTP and web server are both stateless, the only way to maintain a session is when some unique information about the session (the session ID) is transferred between the server and the client in every request and response.

Sessions provide the ability to establish variables –such as access rights and localization settings– which apply to every interaction a user has with the application during the session.

Web applications can create sessions to keep track of anonymous users after the very first user request. An example would be maintaining the user language preference.

Additionally, web applications will make use of sessions once the user has been authenticated. This ensures the ability to identify the user on any subsequent request as well as being able to apply security access controls, authorized access to the user's private data, and increase the usability of the application.

Therefore, current web applications provide session capabilities both **pre- and post-authentication**.

Once an authenticated session has been established, the session ID (or token) is temporarily equivalent to the strongest authentication method by using the application. Such as usernames and passwords, passphrases, one-time passwords (OTP), client-based certificates, smartcards, or biometrics (such as fingerprints or eye retina).

12.5.1. User Authentication

This is a common method for users to provide authentication credentials from the login page and then the authentication information is sent back and forth between the server and client to maintain the session.

12.5.2. HTML Hidden Fields

A unique hidden field in the HTML content and when the user starts navigating we can set its unique value to the user to keep track of the session.

12.5.3. URL Rewriting

A session identifier parameter is appended to every request and response to keep track of the session.

12.5.4. Cookies

Cookies are small pieces of information that are sent by the web server in the response headers and the browser stores them. When the client makes further requests, it adds the cookie to the request header to keep track of the session.

12.6. Federation

A federation identity in information technology is the means of linking a person's electronic identity and attributes, stored across multiple distinct **identity management** systems.

Federation identity is related to single sign-on (SSO), in which a user's single authentication ticket is trusted across multiple IT systems or even organizations.

The "federation" of identity describes the technologies, standards, and use-cases which serve to enable the portability of identity information across otherwise autonomous security domains.

12.6.1. Technologies

- SAML (Security Assertion Markup Language)
- OAuth
- OpenID
- Security tokens (Simple Web Tokens, JSON Web Tokens, and SAML Assertions)
- Web Service Specification and Windows Identity Foundation

13. JSON Web Tokens

JSON Web Tokens (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

Although JWTs can be encrypted to also provide secrecy between parties, signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties.

When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

13.1. Use Cases

13.1.1. Authorization

This is the most common scenario for using JWT. Once a user is logged in, the subsequent requests will include the JWT allowing the user to access routes, services, and resources that are allowed with that token.

13.1.2. Information Exchange

JWTs are a good way of securely transmitting information between parties. Since they are signed tokens, they become handy to confirm senders are who they say they are. Also, they are hashed, making it possible for integrity validation.

13.2. How it Works

In authentication, when the user successfully logs in using their credentials, a JWT will be returned and must be saved locally instead of the traditional approach of creating a session in the server and returning a cookie.

Whenever the user wants to access a protected route, it should send the JWT, typically in the **`Authorization`** header using the Bearer schema.

This is a stateless authentication mechanism as the user state is never saved in the server memory. The protected routes will check for a valid JWT in the Authorization header and if there is, the user is allowed to proceed. As JWTs are self-contained, all the necessary information is there reducing the need of going back and forth to the database.

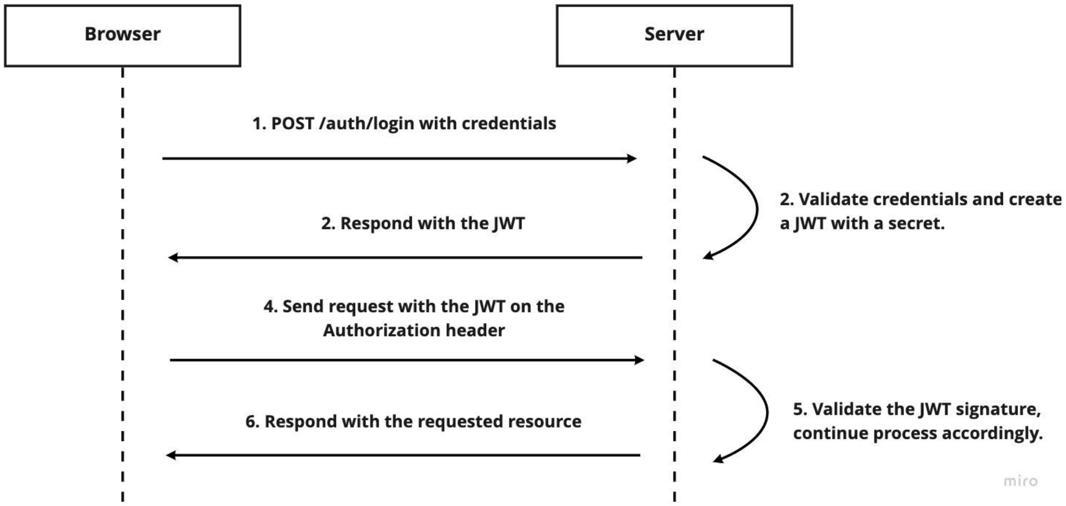

Browser
Server
1. POST /auth/login with credentials
2. Validate credentials and create a JWT with a secret.
2. Respond with the JWT
4. Send request with the JWT on the Authorization header
5. Validate the JWT signature, continue process accordingly.
6. Respond with the requested resource
miro

Fig. 13.1. JWT client/server sequence

13.3. Structure

In its compact form, JWTs consist of three parts separated by dots:

```
{header}.{payload}.{signature}
```

13.3.1. Header

The header typically consists of two parts: the token type (JWT) and the hashing algorithm being used (HMAC SHA256 or RSA).

Example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

13.3.2. Payload

Contains the statements about the entity (also known as *claims*) –e.g. a user– and additional data.

The payload is Base64-encoded to form the second part of the token. For signed tokens this information, though protected against manipulation, is readable by anyone. Do not put secret information in the payload or header unless it’s encrypted.

Example:

```
{
  "sub": "1234657890",
  "name": "John Doe",
  "iat": 1633729587
}
```

There are three types of claims: registered, public, and private.

13.3.2.1. Registered Claims

These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. Some of them are:

- Issuer (**iss**)
- Issued at (**iat**)
- Expiration time (**exp**)
- Subject (**sub**)
- Audience (**aud**)

A full reference on registered claims can be found in the RFC:
https://datatracker.ietf.org/doc/html/rfc7519#section-4.1

13.3.2.2. Public Claims

These can be defined at will. To avoid collisions, they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision-resistant namespace.

13.3.2.3. Private Claims

These are custom claims created to share information between parties that agree on using them and are neither registered nor public claims.

13.3.3. Signature

Consisting of the Base64-encoded header and payload, along with a secret, and signed with the algorithm specified in the header.

For example, if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(base64Encode(header) + '.' + base64Encode(payload), secret)
```

The signature is used to verify the message wasn't altered along the way and in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who says it is.

14. OAuth

An open standard for access delegation, commonly used as a way for internet users to grant websites or applications access to their information on other websites but without giving them the passwords.

OAuth is a delegated authorization framework for REST APIs, enabling applications to obtain limited access (**scope**) to a user's data.

This mechanism is used by companies such as Amazon, Google, Facebook, Microsoft, and Twitter to permit their users to share information about their accounts with third-party applications or websites.

OAuth decouples authentication from authorization and supports multiple use cases addressing different device capabilities. It supports server-to-server, browser-based, mobile/native, and console/TV applications.

Designed specifically to work over HTTP, OAuth essentially allows access tokens to be issued to third-party clients by an authorization server, with the approval of the resource owner. The third party then uses the access token to access the protected resources hosted by the resource server.

14.1. Actors

- **Resource owner:** Owns the data in the resource server. For example, as a user, I am the resource owner of my Facebook profile.
- **Resource server:** The server that stores (probably) and serves the resource data that the applications want to access.
- **Client:** The application that wants to access the resource data.
- **Authorization server:** The main OAuth engine.

14.2. Scopes

Scopes are what you see on the authorization screens when an app requests permissions. They are **bundles of permissions** asked for by the client when requesting a token. These are coded by the application developer when writing the application.

14.3. Tokens

Access tokens are the token the client uses to access the resource server. They are meant to be short-lived; think of them in hours and minutes and not days and months.

Since these tokens can be short-lived and scale-out, they can’t be revoked, you just have to wait for them to time out.

The **refresh token** is much longer-lived; think of them in days, months, and even years. This can be used to request new tokens and can be revoked to kill application access.

The OAuth specification doesn’t define what a token is. It can be in whatever format you want. Usually, though, you want these tokens to be JWT.

The tokens are retrieved from endpoints on the application server

- The **authorize endpoint** is where you go to get the consent and authorization from the user.
- The **token endpoint** provides the refresh token and access token.

You can use the token to access APIs. Once it expires, a new access token must be requested using the refresh token. If the refresh token is expired, a new authorization must be requested.

14.4. Response

When requesting a token (either by authorization or refresh token), the authorization server returns a **code** which is the authorization grant, and the **state** which is used to ensure it is not forged and it’s from the same request.

15. OpenID

Is an open standard and decentralized authentication protocol promoted by the non-profit OpenID Foundation.

It allows users to be authenticated (into an existing account) by co-operating sites (known as **relying parties** or **RP**) using a third-party service, eliminating the need for users to create new passwords and webmasters to provide their ad hoc login systems allowing users to log into multiple unrelated websites without having to have a separate identity and passwords for each.

You may choose to associate information with your OpenID that can be shared with the websites you visit, such as a name or email address.

With OpenID, your password is only given to your identity provider, and that provider then confirms your identity to the websites you visit. Other than your provider, no website ever sees your password.

The OpenID standard provides a framework for the communication that must take place between the identity provider and the **acceptor** (the RP). It does not rely on a central authority to authenticate a user's identity.

Neither services nor the OpenID standard may mandate a specific means by which to authenticate users, allowing for approaches ranging from the common (such as passwords) to the novel (such as smart cards or biometrics).

15.1. Authentication

The end-user interacts with an RP (such as a website) that provides an option to specify an OpenID for authentication.

The RP and the OpenID provider establish a shared secret that is saved by the RP. The RP redirects the end user's user-agent to the OpenID provider so the end-user can authenticate directly with the OpenID provider.

If the end-user accepts the OpenID provider's request to trust the RP, then the user-agent is redirected back to the RP.

15.2. OIDC OpenID Connect

It allows clients to verify the identity of the end-user based on the authentication performed by an authorization server as well as to obtain basic profile information about the end-user in an interoperable and REST-like manner.

Allows clients of all types including web-based applications to request and receive information about authenticated sessions and end-users.

OAuth is directly related to OIDC since it is an authentication layer built on top of OAuth 2.0. OAuth is also distinct from eXtensible Access Control Markup Language (XACML), which is an authorization policy standard.

OAuth can be used in conjunction with XACML where OAuth is used for ownership consent and access delegation whereas XACML is used to define the authorization policies (e.g. managers can view documents in their region).

The OIDC specification suite is extensible allowing participants to use optional features such as encryption of identity data, the discovery of OpenID Providers, and session management when it makes sense for them.

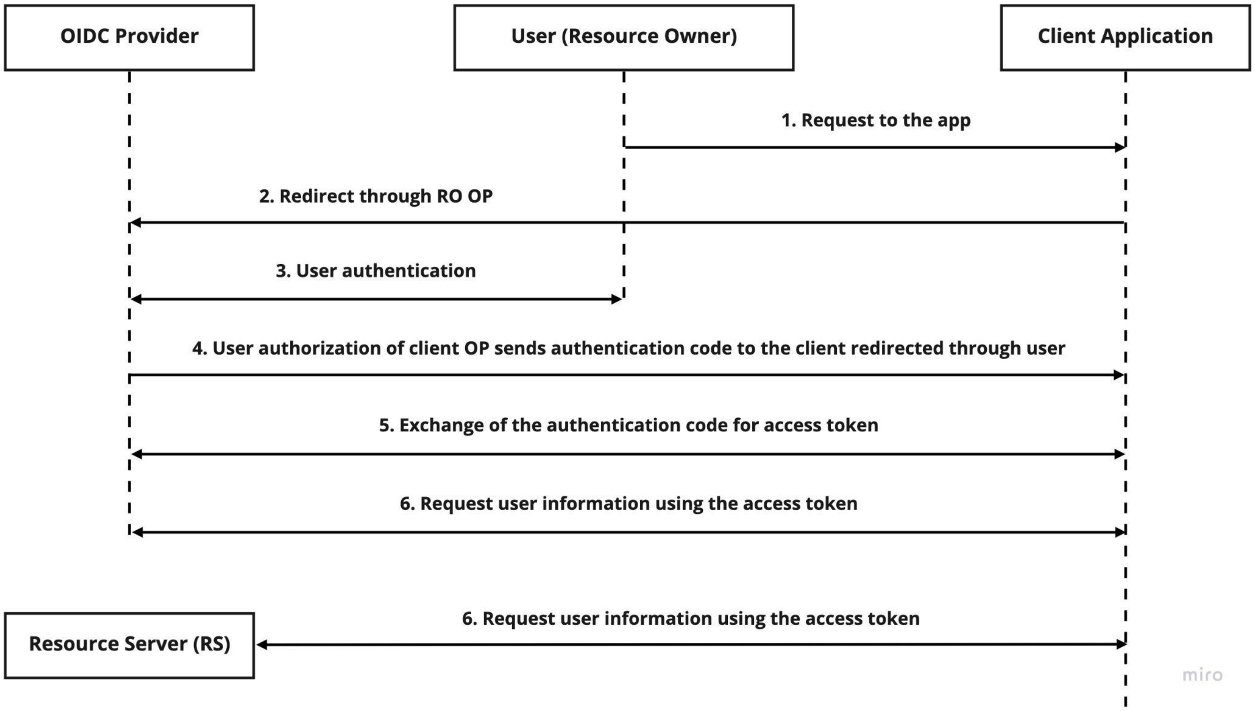

OIDC Provider
User (Resource Owner)
Client Application
1. Request to the app
2. Redirect through RO OP
3. User authentication
4. User authorization of client OP sends authentication code to the client redirected through user
5. Exchange of the authentication code for access token
6. Request user information using the access token
Resource Server (RS)
6. Request user information using the access token
miro

Fig. 15.1. OIDC, user, client application, and RS sequence

16. Encryption

16.1. Symmetric Encryption

Allows for encryption and decryption. The same key is used to encrypt and decrypt data, for example, to store data in a database.

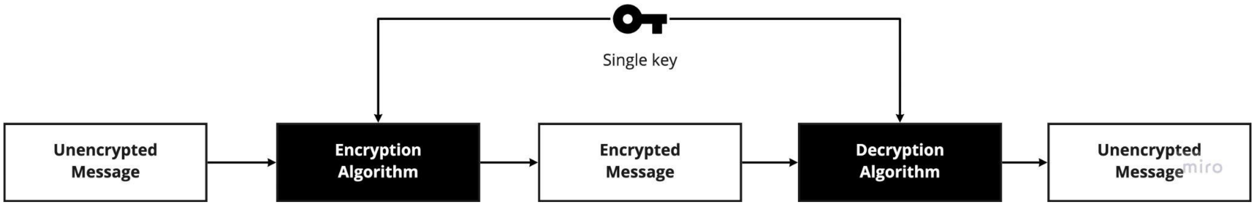


Fig. 16.1. Symmetric encryption flow

16.1.1. Algorithms

- **AES:** Advanced Encryption Standard, adopted by the U.S. government and is now used worldwide. It supersedes Data Encryption Standard (DES) which was published in 1977.
- **Blowfish:** Provides a good encryption rate in software and no effective cryptanalysis of it has been found to date.
- **DES:** Takes plain text in blocks of 64 bits and converts them to ciphertext using keys of 48 bits.
- **3DES:** A symmetric-key block cipher that applies the DES cipher algorithm three times to each data block.
- **RC4:** Rivest Cipher 4 also known as ARC4 or ARCFOUR, is a stream cipher that generates a pseudorandom stream of bits (a keystream). While it is remarkable for its simplicity and speed in software, multiple vulnerabilities have been discovered, rendering it insecure.

- **Caesar Cipher:** One of the earliest and simplest ciphers. It is a type of substitution cipher in which each letter in the plain text is shifted a certain number of places down the alphabet. For example, with a shift of 1, A would be B, B would be C, and so on.

16.2. Asymmetric Encryption

Allows for encryption and decryption, as well as repudiation. One key is used to encrypt and another one to decrypt the data. For example, to verify that a message came from an individual using their private key.

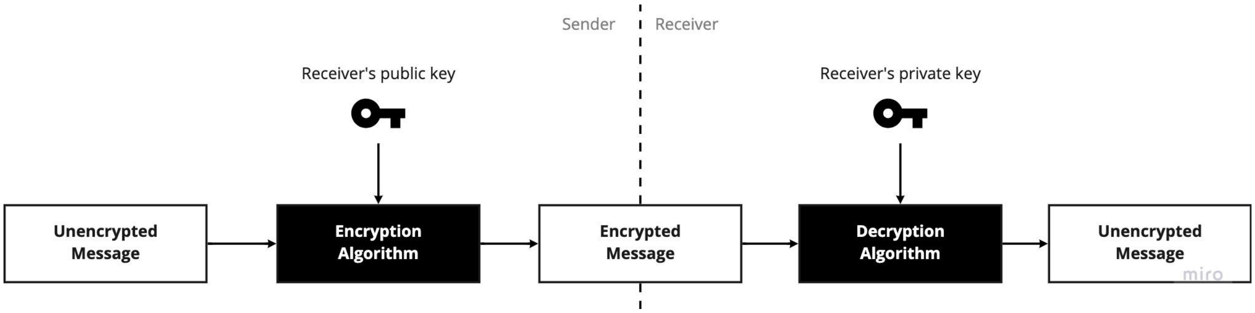


Fig. 16.2. Asymmetric encryption flow

- The keys are generated together
- The public keys are freely distributed
- The private keys are kept in secret and never handed out
- The private key is used for encryption, decryption, and signing

16.2.1. Algorithms

16.2.1.1. RSA

Rivest-Shamir-Adleman is the most widely used asymmetric encryption algorithm. Used for encryption and digital signatures.

In RSA cryptography, both the public and private keys can encrypt a message. The opposite key from the one used to encrypt the message is used to decrypt it.

It provides a method of assuring the confidentiality, integrity, authenticity, and non-reputability of electronic communications and data storage.

16.3. Symmetric vs. Asymmetric

The primary advantage of symmetric-key cryptography is the speed. Symmetric-key algorithms and the keys are shorter in length for the same security strength (256 bit vs. 2048 bit).

Since asymmetric-key cryptography requires fewer keys overall and symmetric-key cryptography is significantly faster, a hybrid approach is often used.

Asymmetric-key algorithms are used for the generation and verification of digital signatures and key establishments.

Symmetric-key algorithms are used for all other purposes (e.g. encryption) especially involving the protection of large amounts of data.

16.4. Key Management

The algorithms that encrypt data all the same, what makes it secure is the keys. As organizations use more encryption, they also end up with more keys and more variety of keys (up to millions).

Every day you generate more keys and they must be managed and controlled. If someone gets access to any of them, they get access to data. If keys are lost, you lose access to data.

Other factors that contributed to the pain were fragmented and isolated systems, lack of skilled staff, and inadequate management tools.

16.5. Use Cases

16.5.1. HTTPS

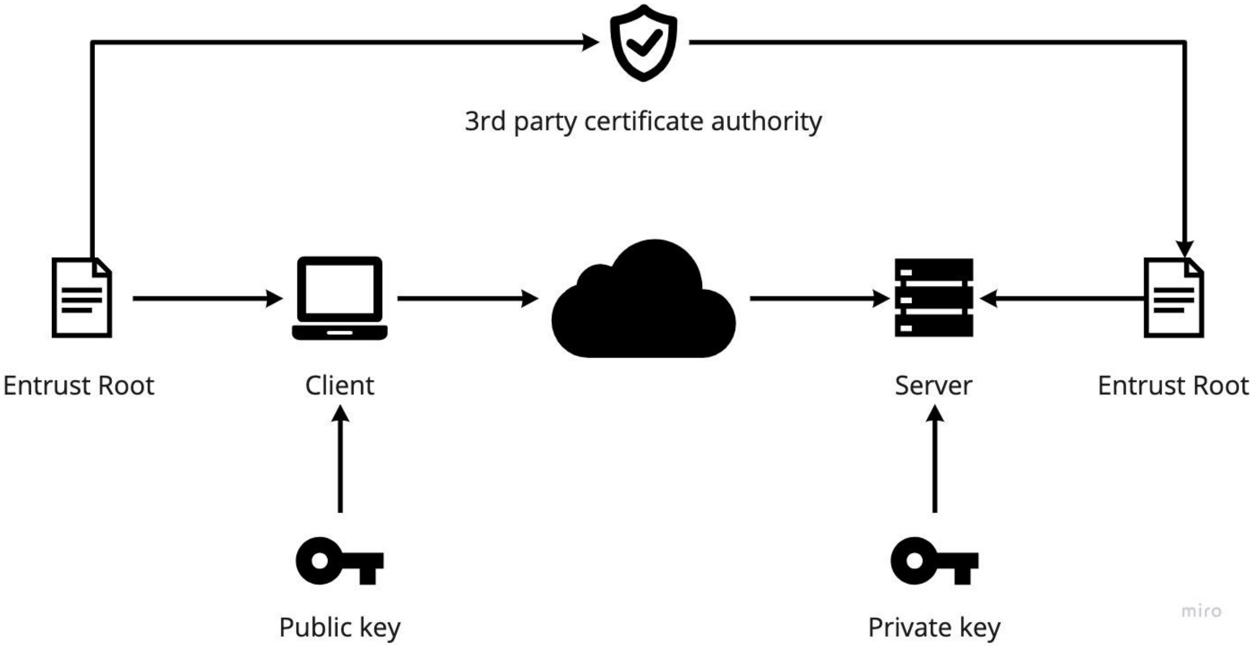
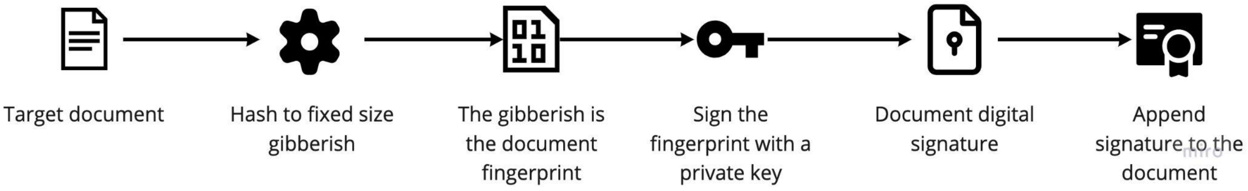


Fig. 16.3. HTTPS flow

16.5.2. Signing



Target document
Hash to fixed size gibberish
The gibberish is the document fingerprint
Sign the fingerprint with a private key
Document digital signature
Append signature to the document
miro

Fig. 16.4. Document signing flow

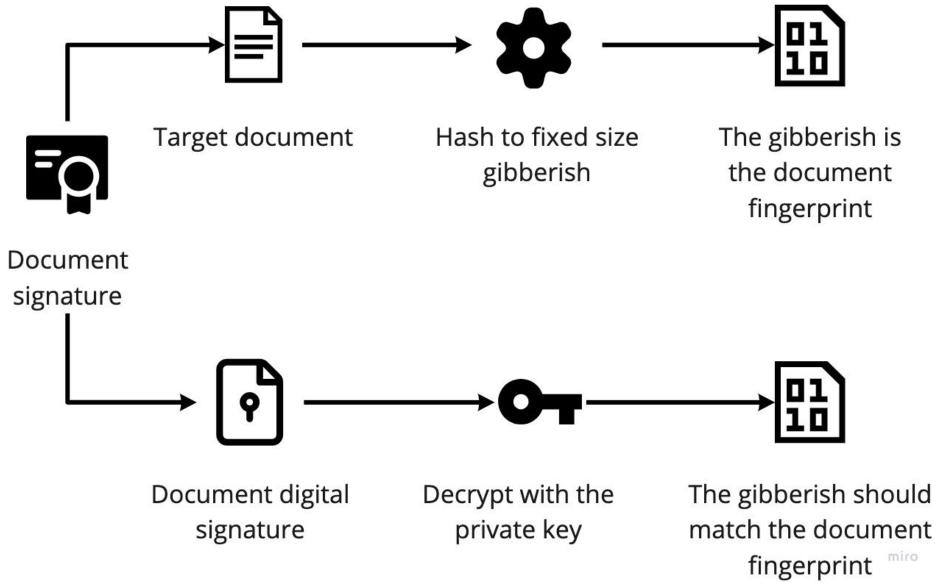

Target document
Hash to fixed size gibberish
The gibberish is the document fingerprint
Document signature
Document digital signature
Decrypt with the private key
The gibberish should match the document fingerprint
miro

Fig. 16.5. Signed document verification flow

17. Hashing

Hashes are one-way, not possible to reverse, and are collision-resistant, meaning that for a given input and its hash, it should be hard to find a different input with the same hash.

17.1. Hashing Functions

- **MD5:** Produces a 128-bit hash. Widely used, however not very secure, it can be used as a checksum to verify data integrity but only against unintentional corruption.
- **SHA-1:** Produces a 160-bit hash. Since 2005, it has been considered insecure against robust attacks. Since then, it has been deemed as insecure as MD5.
- **SHA-2:** Includes significant changes from SHA-1. The SHA-2 family consists of six hash functions with digests (hash values) that are 224, 256, 384, or 512 bit.
- **SHA-3:** The latest iteration of the SHA family with a varied output of 224, 256, 384, or 512 bit.

17.2. Salts

Hashing functions use a salt, which is random data of a fixed length concatenated to the input (unique for each input) used to make hashes unique and protect against brute force attacks.

17.3. Use Cases

- Passwords
- Checksum verification
- Blockchain

17.4. Attacks

17.4.1. Hash Collision

An attempt to find two input strings of a hash function that produce the same hash result. Because hash functions have an infinite input length and a predefined output length, there is inevitably going to be the possibility of two different inputs that produce the same output hash.

17.4.2. Birthday Attacks

This applies to finding collisions in hashing algorithms because it's much harder to find something that collides with a given hash than it is to find any two inputs that hash to the same value.

17.4.2.1. Example

A classroom of 30 students and 1 teacher; the teacher wishes to find pairs of students that have the same birthday.

- The teacher asks for everyone's birthday to find such pairs.
- If the teacher fixes the date to October 10, then the probability that at least one student is born on that day is about 7.9%
- The probability that at least one student has the same birthday as any other student is around 70%

17.4.2.2. Birthday Attacks and Digital Signatures

- A message ***m*** is typically signed by first computing ***H(m)*** where ***H*** is a cryptographic hash function and then using some secret key to sign ***H(m)***. Suppose Alice wants to trick Bob into signing a fraudulent contract.

- Alice prepares a fair contract ***m*** and a fraudulent one ***m'***. She then finds some positions where ***m*** can be changed without changing the meaning, such as inserting commas, empty lines, one versus two spaces after a sentence, replacing synonyms, etc. By combining these changes she can create a huge number of variations on ***m*** which are fair contracts.
- Similarly, Alice can also make some of these changes on ***m'*** to take it even closer towards ***m***, that is ***H(m) = H(m')***. Hence, Alice can now present the fair version ***m*** to Bob for signing. After Bob has signed it, Alice takes the signature and attaches it to the fraudulent contract. The signature proves that Bob has signed the fraudulent contract.

To avoid such an attack, the output of the hash function should be a very long sequence of bits such that birthday attacks now become computationally infeasible.

17.4.3. Brute Force

In cryptography, a brute force attack consists of an attacker trying many inputs with the hope of eventually guessing correctly.

17.4.4. Dictionary

A technique for defeating a cipher or authentication mechanism by trying to determine its decryption key or passphrase by trying hundreds (or even millions) of likely possibilities such as words in a dictionary.

17.4.5. Rainbow Table

Is a pre-computed table for reversing cryptographic hash functions, usually for cracking password hashes.

18. Digital Certificates

A digital certificate is a digital representation of identity and allows you to confirm with who, from, and to you are transferring data. A certificate binds an entity's unique distinguished name (DN) and other additional attributes that identify an entity with a public key associated with its corresponding private key.

In cryptography, a certificate authority or certification authority (CA) is an entity that issues digital certificates. A digital certificate certifies the ownership of a public key by the named subject of the certificate.

- Certificate Authorities are the foundation of Public Key Infrastructure (PKI)
- Offloads the trust to a third party
- RPs can rely on signatures or assertions that correspond to the certificate being presented

18.1. Public Key Infrastructure

PKI is a highly protected ecosystem that allows businesses to issue trusted digital certificates. There is a chain of trust in the issuing and root CAs and in many cases, the root certificate is *offline*. It is only used when cutting a certificate for issuing a CA. This protects the root.

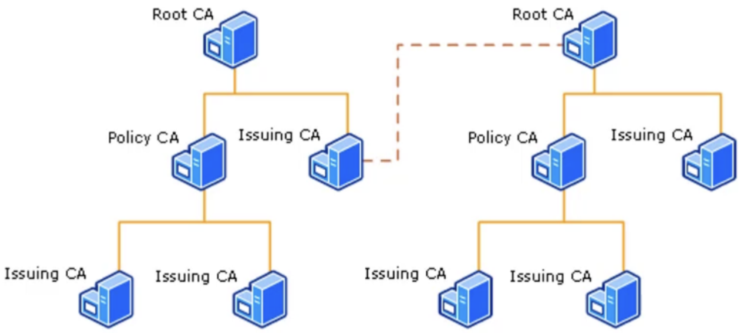


Fig. 18.1. PKI chain of trust

18.2. Certificate Signing Request

- The host creates a key pair (private and public keys)
- The host creates the CSR with information about the host:
  - DN
  - Location
  - Business name
  - Etc.
- The information about the host is hashed and signed creating a digital signature
- The public key, CSR, and signature are sent to the CA for validation
- The validated information and a signed certificate is produced with a CA private key

19. Password Handling

- Allow for long passphrases
- Mask user input fields (on forms)
- Always transmit over secure, encrypted channels
- Store securely using hashing
- Submit via POST body only
- Make sure they are never logged in log files

19.1. Storage

- Never store plain text passwords, only one-way hashes
- Use a cryptographically strong hash algorithm
- Use sufficient salt during hashing and don't reuse salt values
- Physically segregate stored hashes from the rest of the application data

Cracking a password in a year (size vs. cost). This also assumes a memory-hard function:

KDF	6-letter	8-letter	8-char	10-char	40-char text	80-char text
DES CRYPT	< $1	< $1	< $1	< $1	< $1	< $1
MD5	< $1	< $1	< $1	$1.1k	$1	$1.5T
MD5 CRYPT	< $1	< $1	$130	$1.1M	$1.4k	$\$1.5\times10^{15}$
PBKDF2 (100 ms)	< $1	< $1	$18k	$160M	$200k	$\$2.2\times10^{17}$
bcrypt (95 ms)	< $1	$4	$130k	$1.2B	$1.5M	$48B
scrypt (64 ms)	< $1	$150	$4.8M	$43B	$52M	$\$6\times10^{19}$
PBKDF2 (2 sec)	< $1	$29	$920k	$8.3B	$10M	$\$11\times10^{18}$
bcrypt (3 sec)	< $1	$130	$4.3M	$39B	$47M	$1.5T
scrypt (3.8 sec)	$900	$610k	$19B	$175T	$210B	$\$2.3\times10^{23}$

19.2. Entropy

Entropy is a measure of the disorder of a system. In a set of ordered entities, when increasing entropy, the entities get more disordered.

Password entropy is based on the character set used (which is expansible by using lowercase, uppercase, numbers, and symbols) and length. It predicts how difficult a given password would be to crack through guessing, brute force cracking, dictionary attacks, or other common methods.

Note: Encryption creates randomness which cannot be compressed as well. For maximum efficiency, you should compress before encryption.

19.3. Best Practices

- Minimum of 8 characters
- At least 1 character from at least 3 of these 4 groups:
    - Uppercase characters
    - Lowercase characters
    - Numeric characters
    - Special/symbolic characters
- Should not be a common dictionary word
- Should not be a permutation of the username
- Multi-word passphrase
- Avoid common patterns
- Add a second factor like one-time password generators or biometrics
- Prevent unlimited guessing with CAPTCHAs and temporary account locks

20. Frameworks and Process

20.1. What is PHI

- Names, phone numbers, email addresses
- All geographical identifiers smaller than a state, except for the initial three digits of a ZIP code
- Dates (other than a year) directly related to an individual
- SSN, account numbers, medical record numbers, health insurance beneficiary numbers
- Certificate/license numbers
- Vehicle identifiers and serial numbers including license plate numbers
- Device identifiers and serial numbers
- Web Uniform Resource Locators (URLs)
- Internet Protocol (IP) address numbers
- Biometric identifiers including finger, retinal, and voiceprints
- Full-face photographic images and any comparable images
- Any other unique identifying number, characteristic, or code

20.2. HIPAA Health Insurance Portability and Accountability Act

Signed in 1996, HIPAA aimed to:

- Modernize the flow of healthcare information
- Protect PII
- Address limitations on healthcare insurance coverage

20.2.1. Titles

20.2.1.1. Title I: HIPAA Health Insurance Reform

HIPAA Title I of the Health Insurance Portability and Accountability Act of 1996 protects health insurance coverage for workers and their families when they change or lose their jobs.

20.2.1.2. Title II: IPAA Administrative Simplification

The Administrative Simplification provisions of the Health Insurance Portability and Accountability Act of 1996 require the Department of Health and Human Services to establish national standards for electronic health care transactions and national identifiers for providers, health plans, and employers.

It also addresses the security and privacy of health data. Adopting these standards will improve the efficiency and effectiveness of the nation's health care system by encouraging the widespread use of electronic data interchange in health care.

Rule I – Privacy Rule: The HIPAA privacy rule **regulates the use and disclosure of protected health information (PHI)** held by *covered entities* (generally, health care clearinghouses employees, employer-sponsored health plans, health insurers, and medical service providers that engage in certain transactions).

For health information, privacy is defined as the right of an individual to keep their information from being disclosed. This is typically achieved through policy and procedure.

Privacy encompasses controlling who is authorized to access patient information; and under what conditions patient information may be accessed, used, and/or disclosed to a third party.

Rule 3 – Security Rule: This rule complements the privacy rule. While the privacy rule pertains to all PHI, including paper and electronic, the security rule **deals specifically with Electronic Protected Health Information (EPHI)**.

Security is defined as the mechanism to protect the privacy of health information. This includes the ability to control access to patient information as well as to safeguard patient information from unauthorized disclosure, alteration, loss, or destruction.

Security is typically accomplished through operational and technical controls within a covered entity. Since so much PHI is now stored and/or transmitted by computer systems, the HIPAA security rule was created to specifically address EPHI.

20.2.2. Violation Penalties

The minimum fine for willful violations of HIPAA Rules is $50,000. The maximum criminal penalty for a HIPAA violation by an individual is $250,000. Restitution may also need to be paid to the victims. In addition to the financial penalty, a jail term is likely for a criminal violation of HIPAA Rules.

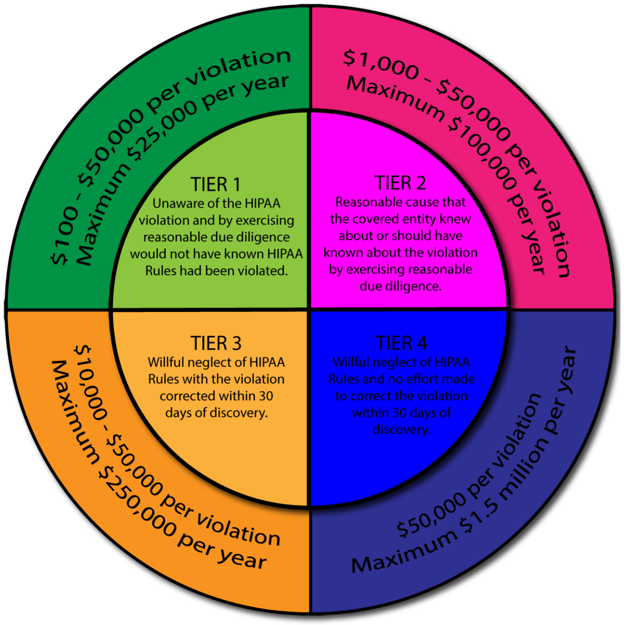

$100 - $50,000 per violation
Maximum $25,000 per year
TIER 1
Unaware of the HIPAA violation and by exercising reasonable due diligence would not have known HIPAA Rules had been violated.
$1,000 - $50,000 per violation
Maximum $100,000 per year
TIER 2
Reasonable cause that the covered entity knew about or should have known about the violation by exercising reasonable due diligence.
TIER 3
Willful neglect of HIPAA Rules with the violation corrected within 30 days of discovery.
$10,000 - $50,000 per violation
Maximum $250,000 per year
TIER 4
Willful neglect of HIPAA Rules and no effort made to correct the violation within 30 days of discovery.
$50,000 per violation
Maximum $1.5 million per year

Fig. 20.1. HIPAA violation penalties

20.3. HITECH Health Information Technology for Economic and Clinical Health

The HITECH act was created in 2009 to promote and expand the adoption of electronic health records (EHRs) by healthcare providers. It accelerated the adoption of EHR and provided incentives for the adoption through "Meaningful Use" and penalties for violations.

It is built on HIPAA, ensuring entities were complying with its rules, and that notifications were sent to affected individuals when health information was compromised.

Tougher penalties for HIPAA compliance failures were introduced to incentivize entities to comply with the HIPAA privacy and security rules.

Transitioning to electronic records was expensive, but this act introduced incentives to encourage hospitals and other healthcare providers to make the change.

Increased the EHR adoption from 3.2% in 2008 to 14.2% in 2015. By 2017 86% of office-based physicians had adopted an EHR and 96% of non-federal acute care hospitals had implemented certified health IT.

The act did not make compliance with HIPAA mandatory as that was already a requirement, but it did make sure that entities found not to comply could be issued with a substantial fine.

The HITECH act contains four subtitles (A-D):

- Subtitle A concerns the promotion of health information technology and is split into two parts:
  - Part 1 is concerned with improving healthcare quality, safety, and efficiency
  - Part 2 is concerned with the application of the use of health information technology standards and reports

- Subtitle d covers privacy and security of electronic health information and is also split into two parts:
    - Part 1 is concerned with improving the privacy and security of health IT and PHI
    - Part 2 covers the relationship between the HITECH act and other laws

20.4. HIPAA and HITECH

HIPAA and HITECH were brought together in 2013 through the HIPAA Omnibus Final Rule. The HITECH act was called for mandatory penalties for HIPAA-covered entities and business associates in cases where there was willful neglect of HIPAA rules.

With such high potential fines (up to $1.5M) HIPAA compliance could no longer be considered optional. The penalties could be higher than the cost of complying with HIPAA.

The HITECH act brought the new HIPAA Breach Notification Rule that required entities to issue notifications to affected individuals within sixty days of the discovery of a breach of unsecured protected health information.

Both acts address the security of electronic PHI (ePHI). However, before HITECH, patients were unable to find out to who their ePHI had been disclosed (both authorized and unauthorized where known). In 2011, a HITECH-required rule allowed patients to request access to reports.

21. PCI DSS Payment Card Industry Data Security Standard

The PCI DSS is information security for organizations that handle branded credit cards from the major card schemes. It is mandated by the card brands and administered by the **Payment Card Industry Security Standard Council (PCI SSC)**.

The standard was created to increase the controls around cardholder data to reduce credit card fraud.

The validation of compliance is performed annually either by an external QSA or by a firm-specific Internal Security Assessor that creates an RoC for organizations handling large volumes of transactions or by an SAQ for companies handling smaller volumes.

Compliance with PCI DSS is not required by federal law in the United States, however, the law of some U.S. states either refer to PCI DSS directly or make equivalent provisions.

21.1. QSA Qualified Security Assessor

A QSA is a certificate that has been provided by the PCI SSC. This certified person can audit merchants for PCI DSS compliance.

21.2. RoC Report on Compliance

An RoC is a form that has to be filled by all level 1 merchants undergoing a PCI DSS audit. The RoC form is used to verify that the merchant being audited is compliant with the PCI DSS standard.

21.3. SAQ Self-Assessment Questionnaire

The SAQ is a set of questionnaire documents that merchants are required to complete every year and submit to their transaction bank.

21.4. Control Objectives

The PCI DSS specifies twelve requirements for compliance, organized into six logical related groups called control objectives.

1. Build and maintain a secure network and systems
2. Protect cardholder data
3. Maintain a vulnerability management program
4. Implement strong access control measures
5. Regularly monitor and test networks
6. Maintain an information security policy

Objectives of PCI Security Requirements:

- Minimize attack surface
- Software protection mechanisms
- Secure software operations
- Secure software lifecycle management

22. DevOps

It is the breakdown of the barrier between development (responsible for writing production-ready code) and operations (responsible for delivering and maintaining the code deployed in a production environment).

22.1. CI Continuous Integration

A development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early.

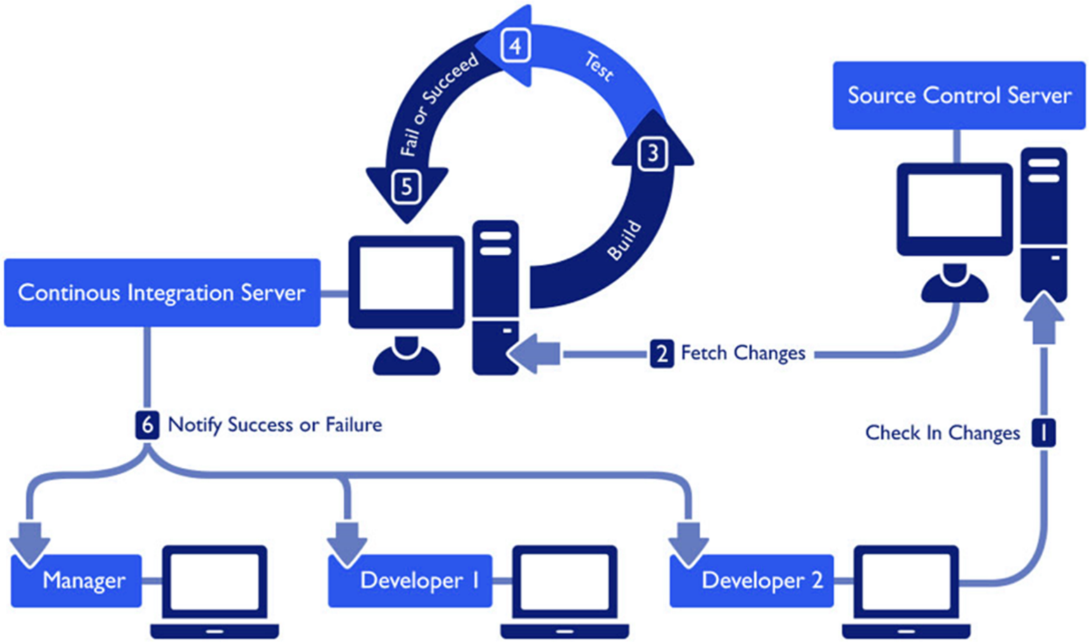


Fig. 22.1. Continuous integration cycle

Because you're integrating so frequently, there is significantly less back-tracking to discover where things went wrong, so you can spend more time building features.

"Continuous integration is cheap; not integrating continuously is expensive."

Not following a continuous approach means you will have longer periods between integrations. This makes it exponentially more difficult to find and fix problems. Such integration problems can easily knock a project off-schedule or cause it to fail altogether.

22.2. CD Continuous Delivery/Deployment

Continuous Delivery is the ability for a team to perform all integration tasks and make a production-ready package ready and available.

Continuous Deployment is the ability to get changes of all types into production or into the hands of users in a safe and timely manner.

The difference between continuous delivery and continuous deployment is that in continuous delivery the package is not pushed to a production environment without manual intervention.

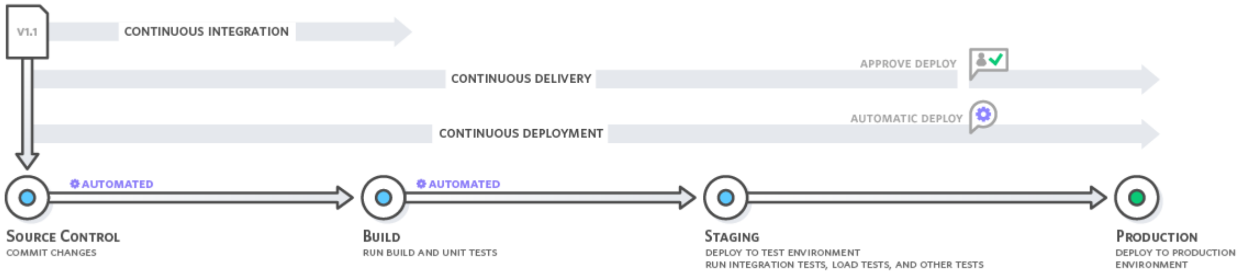


Fig. 22.2. Continuous Integration, Continuous Delivery, and Continuous Deployment

22.3. DevSecOps

How do we inject security?

- **Some of the basics from the SDLC still apply:** Threat modeling, abuse cases, code review, secure config, WAF, etc.
- **Tools can be slow, especially static analysis:** Try to reduce the size of the code being scanned. A large code base means long scan times.

- **Fast feedback loop from production:** Ensure that you have the means to deliver vulnerability or defect information quickly to the right environment team(s).
- **Patching:** Have a means for delivering patches quickly (following the DevOps principles).

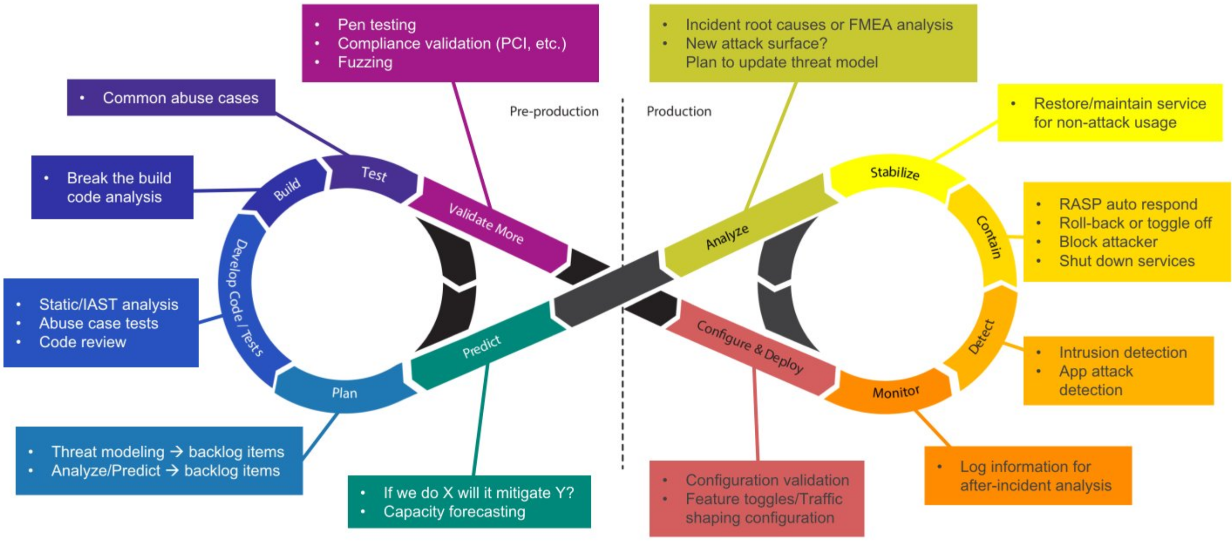


Fig. 22.3. DevSecOps lifecycle

Bottom line:

- Test early, test often
- Use the scanning tools available and scan smaller scopes
- Dynamic Application Security Testing (**DAST**), Run-time Application Security Protection (**RASP**), Interactive Application Security Testing (**IAST**), Web Application Firewall (**WAF**) work better in a DevOps environment than Static Application Security Testing (**SAST**)
- A constant feedback loop of communication
- Monitoring for vulnerabilities in the environment
- Monitoring external sources for vulnerabilities in third parties

23. Security Analysis

Most enterprises use many (even all) of the techniques outlined in this chapter. Nonetheless, no solution is a silver bullet and many are platform or language-dependent. This means that if you are using multiple platforms or languages you will need more than one tool (although very few enterprises are monolithic).

The results are different in each solution and there are many false positives.

A **false positive** is a finding in a tool or through some other technique that turns out not to be a viable finding. For example, tools that claim there is a password in clear text when it simply finds the word password.

A **false negative** is a vulnerability that can get past a scanning tool or other technique that is looking for vulnerabilities. For example, an SQLi vulnerability that is not identified by a scanning tool.

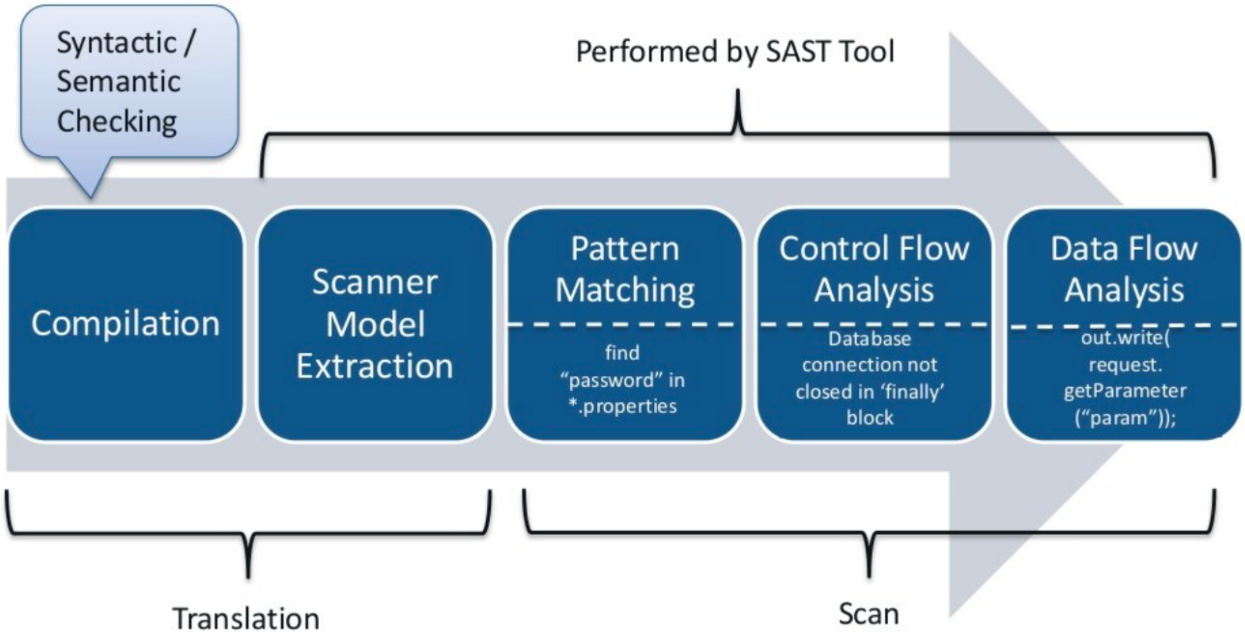
23.1. SAST Static Application Security Testing

Static analysis commonly refers to the running of tools that attempt to highlight possible vulnerabilities within the non-running source code by using techniques such as Taint Analysis and Data Flow Analysis.

Most static analysis tools are used as an aid for an analyst to help zero in on security-relevant portions of code so they can find flaws more efficiently, rather than a tool that simply finds flaws automatically.

Taint Analysis attempts to identify variables that have been tainted with user-controllable input and traces them to possible vulnerable functions also known as a “sink”. If the tainted variable gets passed to a sink without being sanitized it is flagged as a vulnerability.

Lexical Analysis converts source code syntax into tokens of information in an attempt to abstract the source code and make it easier to manipulate.



Syntactic / Semantic Checking
Performed by SAST Tool
Compilation
Scanner Model Extraction
Pattern Matching
find "password" in *.properties
Control Flow Analysis
Database connection not closed in 'finally' block
Data Flow Analysis
out.write(request. getParameter ("param"));
Translation
Scan

Fig. 23.1. High-level SAST process | Aspect Security

23.1.1. Strengths

- Helps in identifying the flaws in the code
- The testing is conducted by trained software engineers with good knowledge of coding
- It is a fast and easy way to find and fix the errors
- With automated tools, it becomes quite fast to scan and review software
- The use of automated tools provides mitigation recommendations
- With static testing it is possible to find errors at an early stage of the development lifecycle, thus, reducing the cost of fixing

23.1.2. Weaknesses

- It demands a great amount of time when done manually

- Automated tools work with a subset of programming languages
- Automated tools may provide false positives and false negatives
- Automated tools only scan the code
- Automated tools cannot pinpoint weak points that create troubles in runtime

23.2. DAST Dynamic Application Security Testing

DAST is a black-box security testing methodology in which an application is tested from the outside by examining it in its running state and trying to attack it just like an attacker would.

DAST scanners are, for the most part, technology-independent. This is because DAST scanners interact with an application from the outside-in and rely on HTTP as a common language across a myriad of programming languages, off the shelf and even custom-built frameworks.

23.2.1. Strengths

- Not as technology-dependent
- Can be run in production environments
- Not as many false positives as in SAST
- Can test software you don’t own
- Can be used to enhance penetration testing

23.2.2. Weaknesses

- Can’t locate the code responsible for the vulnerability
- Findings are later in the SDLC, although you can do DAST earlier
- Doesn’t locate code-specific security issues (like hard-coded passwords)
- Findings still need to be verified by an SME

23.3. IAST Interactive Application Security Testing

Assesses applications from within using software instrumentation. This technique allows IAST to combine the strengths of both SAST and DAST as well as providing access to code, HTTP traffic, library information, backend connections, and configuration information.

Some IAST products require the application to be attacked, while others can be used during normal quality assurance testing.

23.3.1. Strengths

- Agents: There is continuous monitoring that is always active
- Works well in the DevOps and DevSecOps model
- Lower cases of false positives since it can "see" active attacks and not potential ones
- Can have a targeted approach to defining the security scope

23.3.2. Weaknesses

- In the real world, agents are resisted because the owners of the systems are not always sure of what the agent is doing
- Instrumentation means possibly development and deployment work to take advantage of the benefits
- Many of them only work when they "see" something. In other words, you need to exercise a workflow for it to be picked up
- A steep learning curve for development and review of the results since it doesn't point to the line of code (see DAST)

23.4. RASP Runtime Application Security Protection

It is a security technology that uses runtime instrumentation to detect and block computer attacks by taking advantage of information from inside of the running software.

RASP technology can improve the security of the software by monitoring its inputs and blocking those that could allow attacks while protecting the runtime environment from unwanted changes and tampering.

RASP can prevent the exploitation and possibly take other actions, including terminating a user's session, shutting down the application, alerting the security personnel, and sending a warning to the user.

23.4.1. Strengths

- Can be configured to block or monitor
- Can block attacks as they happen
- Since it is similar to DAST and IAST (commonly referred to as a combination of both), the strengths of both can be considered here

23.4.2. Weaknesses

- Needs to see an attack as it happens (see IAST)
- Potential to block legitimate traffic
- Someone (or a group) needs to own the rules that define what is blocked
- Since it is similar to DAST and IAST (commonly referred to as a combination of both), the weaknesses of both can be considered here

23.5. WAF Web Application Firewall

Is an application firewall for HTTP-based applications. It applies a set of rules to HTTP conversations that generally cover common attacks such as XSS and SQLi.

It is deployed in front of web applications and analyses bi-directional traffic detecting and blocking anything malicious. This functionality can be implemented in software or hardware, running in an appliance device, or a typical server running a common operating system.

Note: WAFs can sometimes be considered an Application Security Manager (ASM).

Although the names for operating mode may differ, WAFs are deployed inline as:

- **Transparent bridge:** It inspected only the traffic that is configured for inspection while bridging the rest. Bridge mode deployment can be achieved with no changes to the network configuration of the upstream devices or web servers.
- **Reverse proxy:** Accepts traffic on the virtual IP address and proxies the traffic to the backend server behind the WAF.

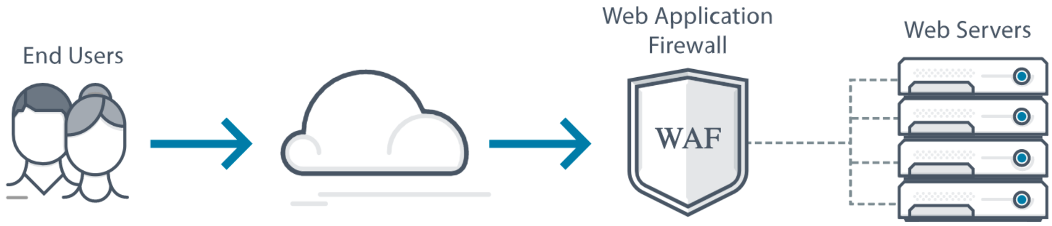


Fig. 23.2. WAF flow

23.5.1. Types

23.5.1.1. Network-based

- **Pros:** Low network latency since they are connected directly to the web servers.
- **Cons:** Higher cost and tougher management across large data centers

23.5.1.2. Host-based

- **Pros:** Affordable, no network latency.
- **Cons:** Agents, engineering cost/time, can create application complexity.

23.5.1.3. Cloud-based

- **Pros:** Cheapest, auto-updated/maintained, quick to deploy.
- **Cons:** High network latency, no ownership.

23.5.2. Strengths

- Can be in blocking or reporting mode
- Can be independent of the application
- Can block XSS, SQLi, cookie poisoning, non-validated input, DoS, web scraping, etc.

23.5.3. Weaknesses

- Potential performance issues
- Not solving the problems
- Can’t protect against every security issue

23.6. Penetration Testing

23.6.1. Types

- **White box:** Provides information about the system to the tester. Can include code, credentials, network maps, and other system information.

- **Black box:** Provides little to no system information. This resembles a typical attack where the information that can be gathered is generally only public information.
- **Grey box:** The in-between state. Some information but possibly limited to just essential information.
- **Internal:** A “red team” that is employed at the target company. This is a team/group that has other duties at the company but is engaged for some time to target a specific system/application.
- **External:** An external party that is engaged to test the system/application. The scope is defined and the party is given a timeframe for completion.

23.6.2. Strengths

- Findings are typically true and are actionable
- Can be scoped to specific areas and time
- Can be used in combination with other security methods
- Findings in threat model or scan tools can be verified

23.6.3. Weaknesses

- Not usually a full system test, it is very targeted
- Can be expensive and time-consuming
- Findings need to be secured, especially with a third party is involved in the testing process

24. Conclusions

24.1. Configuration Management

- All sensitive data should be encrypted at rest
- Catch all errors and display specific messages to but be careful
- Production data should only be in production
- Follow all guidelines for server, operating system, and database hardening

24.2. Account Management

- Database accounts should be least privileged, only read/write access to those tables and fields that are required
- Operating system accounts for the database should not be domain admin/root/system
- Change or remove all default accounts and passwords
- Pick more difficult account names
- Use 2-FA for all accounts, especially the privileged ones

24.3. Password Management

- Require strong passwords
- Use salt and strong hashes
- Don’t store unencrypted passwords
- Have a strong password recovery system

24.4. Hard-coding

- Don’t hard-code secrets in source code
- Put secrets in a config file and keep it in a secure location
- Use hardware security modules where available
- The secrets should be stored away from the data they secure
- Restrict and monitor access to files on the server

24.5. Upload and Download Management

- Restrict file types using server-side code
- Configure directory policies so files cannot be executed
- Restrict permissions on files so users can access only the files they own
- Mitigate directory traversal and arbitrary file retrieval issues to prevent users from downloading files from the operating system
- Use allow-lists, not deny-lists

24.6. Input Validation

- Don’t trust input coming from the outside boundary
- Clean on the first point of entry
- Sanitize per program’s context if reusing some user input from database/internal storage
- Mitigate multiple types of injection

24.7. Data in Motion

- Always use TLs
- Use best practices for encryption and strong ciphers

- Only use reputable certificate authorities
- Have a way to revoke certificates and keys

24.8. Security Planning

- **Involve** the security team when planning a big feature
- **Add** security features or controls to user stories when planning
- Create and maintain **threat models**
- Write and test **abuse/misuse** test cases
- Minimize attack surfaces (more points of interaction = more difficult to defend)
- Not everyone should have access to everything, even people or accounts you might think should have access
- Keep it simple. The simpler the design of the security, the easier it is to understand it and implement it correctly

24.9. Defense-in-Depth

- Don't rely on a single security method to protect everything
- Layer basic security practices to ensure the overall safety of an application
- Why is it important?
  - Failovers
  - Edge cases
  - Adding more friction to attackers
- Avoid security by obscurity
- Fix security issues correctly, don't just cover the surface or leaf cases (fix from root)
- Don't reinvent the wheel, use standards, and existing models

25. Tools

25.1. Dependency Check

- A .NET and Java-compatible tool that is used to scan libraries used as dependencies during the build process.
- Dependencies are matched against the NVD to determine whether the dependency being used is vulnerable.
- A report is generated and can be used to identify the dependencies as well as understand the mitigation strategies. In most cases, the mitigation is to use the most up-to-date level of software.

25.2. GitHub Security Alerts

GitHub tracks public vulnerabilities in packages from supported languages on MITRE's Common Vulnerabilities and Exposures list. Scans data in public commits and uses a combination of machine learning and human review to detect vulnerabilities that are not published in the CVE list.

When GitHub discovers or is notified of a new vulnerability, they identify public repositories (and private repositories that have opted-in to vulnerability detection) that use the affected version of the dependency and send a security alert.

- Works with both private and public repositories. Private repositories need to request alerts.
- Must use the GitHub Dependency Graph
- As of September 2021, it supports JavaScript, Ruby, Java, PHP, .NET, Python, and Go.
- Each security alert includes a severity level and a link to the affected file in your project.

25.2. From OWASP

- **Security Shepherd:** Web and mobile application security training platform. https://owasp.org/www-project-security-shepherd/
- **WebGoat:** Insecure web application with security lessons. https://owasp.org/www-project-webgoat/
- **OWASP Juice Shop:** Used in security training, awareness demos, CTFs, and as the guinea pig for security tools. https://owasp.org/www-project-juice-shop/
- **Vulnerable Web Applications Directory:** List of known vulnerable web applications for testing purposes. https://owasp.org/www-project-vulnerable-web-applications-directory/
- **OWASP ZAP (Zed Attack Proxy):** Free and open-source dynamic security scanner. https://owasp.org/www-project-zap/
- **OWTF (Offensive Web Testing Framework):** PenTesting framework for scanning sites. https://owasp.org/www-project-owtf/
- **OWASP WTE (Web Testing Environment):** Set of application security tools and documentation available in multiple formats such as VMs, Linux distribution packages, cloud-based installations, and ISO images. https://owasp.org/www-project-web-testing-environment/
- **AppSensor:** Guidance on how to detect/respond/defend within your application. https://owasp.org/www-project-appsensor/
- **CSRFGuard Project:** Securing web applications against Cross-Site Request Forgery (CSRF). https://owasp.org/www-project-csrfguard/
- **ESAPI:** Free, open-source, web application security control library that makes it easier for programmers to write lower-risk applications. https://owasp.org/www-project-enterprise-security-api/

- **Security Knowledge Framework (SKF):** A tool used as a guide for building and verifying secure software.
  https://owasp.org/www-project-security-knowledge-framework/
- **Dependency Check:** A utility that identifies project dependencies and checks if there are any known, publicly disclosed vulnerabilities.
  https://owasp.org/www-project-dependency-check/
- **Dependency Track:** Allows organizations to identify and reduce the risk from the use of third-party and open source components.
  https://owasp.org/www-project-dependency-track/
- **Defectdojo:** Vulnerability management tool that streamlines the testing process by offering templates, report generation, metrics, and baseline self-service tools.
  https://owasp.org/www-project-defectdojo/
- **WebScarab:** A framework for analyzing applications that communicate using the HTTP and HTTPS protocols.
  https://wiki.owasp.org/index.php/Category:OWASP_WebScarab_Project

25.3. Who's Who

- Charles Web Debugging Proxy:
  https://www.charlesproxy.com/
- Fiddler:
  https://www.telerik.com/fiddler
- Browser Developer Tools

25.4. From Microsoft

- Microsoft Threat Modeling Tool:
  https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling

25.5. DevOps vs. DevSecOps

DevOps Code Creating value and availability	DevSecOps Code Creating trust and confidence
Source code	
GitHub	gitrob
	Checkmarx
CI Server	
Jenkins	Splunk
Test and scan	
BlazeMeter	Contrast Security
	Gauntlt
Artifacts	
Nexus	Sonatype
	Nessus
	Tanium
Deploy	
Amazon Web Services	Chef InSpec
Monitoring	
New Relic	Splunk
	Metasploit
	FireEye

25.6. SAST

- Sonarqube

- Brakeman (Ruby on Rails)
- Fortify
- IBM AppScan
- SpotBugs (Java)

25.7. DAST

- WhiteHat Security
- Qualys
- Veracode
- OWASP ZAP
- Arachni

25.8. IAST

- Acunetix
- Checkmarx
- Synopsys

25.9. RASP

- Arxan
- Imperva
- Wallarm

Appendix

Definitions

Cryptanalysis: The process of finding weaknesses in cryptographic algorithms and using these weaknesses to decipher the ciphertext without knowing the secret key (instance deduction).

Misuse: A misuse case highlights something that should not happen (e.g. a negative scenario) and threats, hence identified, help in defining new requirements which are expressed as new use cases.

Abuse: An abuse use case is a type of complete interaction between a system and one or more actors where the results of the interaction are harmful to the system, one of the actors, or one of the stakeholders in the system.