Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2022, 9(7):59-65

Research Article ISSN: 2394 - 658X

Secure Coding: A Guide for Software Developers

Krishna Mohan Pitchikala

ABSTRACT
Software security is a major concern in today's fast-changing digital world. With more cyber threats emerging, it's crucial for software developers to use safe coding practices to reduce vulnerabilities and protect the integrity, confidentiality, and availability of software applications. This paper explains how software developers can write secure codes by looking at ways of handling OWASP top 10 threats which are among the greatest security risks for web applications. It outlines each vulnerability and provides actual examples as well as remediation steps for every issue. With such an approach, developers will be able to enhance protection levels on their programs thus lowering chances of attacks on information systems making the internet safer. Following industry standards like those set by OWASP significantly strengthens security measures. Considering the constantly changing nature of technology in our world today, this guide is essential reading not only for software engineers who want to learn skills to build strong applications but also for anyone interested in understanding various types of cyber-attacks and how to prevent them. It provides valuable knowledge on safeguarding against cyber threats, making it a crucial resource for building robust and secure applications in our fast-evolving digital environment.

Key words: Secure Coding, Software Developers, Software security

1. INTRODUCTION

The Open Web Application Security Project (OWASP) is a global non-profit organization aimed at enhancing software security by means of open-source community-driven projects, educational materials and training. In 2001, OWASP was founded to address the growing need for securing web applications. The organization's most important contribution is its OWASP Top 10 list, which highlights the top ten web application vulnerabilities. This list is updated regularly and includes the most serious risks to web applications identified by OWASP. For developers, security professionals, and businesses, this report offers invaluable information about common weak points that hackers often target during attacks because they are widespread vulnerabilities across many systems [1].
The latest version of OWASP Top 10 was published in 2021 [2]. Updates are usually made every three or four years with new threats being considered alongside advancements in web technologies. Therefore, following the OWASP Top 10 at all times is crucial for maintaining web application security. It helps in identifying flaws so that necessary measures can be implemented quickly thereby ensuring ongoing protection against breaches not only once but continuously. This also promotes good programming habits among developers, reducing potential financial losses from reputational damage caused due to leakage of sensitive data. Moreover, adhering to these guidelines significantly lowers the risk of data loss from hacking, creating a safer online environment for everyone. Adopting to the best practices as those outlined in the OWASP Top 10, are essential right from the design phase to protect against emerging threats and to comply with industry regulations.

2. TOP 10 WEB APPLICATION SECURITY RISKS

BROKEN ACCESS CONTROL:

Access control makes sure users can only do what they are allowed to do. Broken Access Control is a security flaw where unauthorized users can access data or functions, they shouldn't. This can lead to unauthorized access, changes, or deletion of data, and users doing things they shouldn't be able to. It can be exploited through complex attacks like using Mimikatz to steal credentials or simple ones like URL manipulation. Common issues include granting too much access by not following principle of least privilege or bypassing checks by altering URLs or using attack tools. [3, 4]

2.1 Example:

One simple way to exploit access control flaws is called forced browsing. For example, on a website like example.com, users and admins must log in to access certain pages. However, if attackers know the URLs, they can directly enter:

http://example.com/user_page

or

http://example.com/admin_page

A secure website should redirect them to the login page. If it doesn't, this is broken access control. Even basic attacks like this can be dangerous if user data isn't properly protected. For instance, accessing:

http://example.com/password_list.txt

could reveal users' plain text passwords

2.2 Mitigation:

Access control works best in trusted server-side code or server-less APIs where attackers can't modify the checks or metadata. To prevent broken access control, follow these practices:

- Deny access by default, except for public resources.
- Ensure access controls enforce record ownership.
- Log access control failures and alert admins when needed.
- Rate limit API and controller access to reduce automated attacks.
- Invalidate session identifiers on logout and use short-lived JWT tokens.

Developers and QA staff should also include access control tests in their unit and integration tests [1]

3. CRYPTOGRAPHIC FAILURES

Cryptographic failure happens when sensitive information, like passwords, personal details, or financial data, is either not protected at all or is protected using weak methods. Sensitive data is anything an organization wants to keep private, and its protection level can vary based on laws and industry standards. Not protecting data means not following proper security measures, like encrypting bank account numbers but then sending them over an unencrypted connection. Insufficient cryptography refers to using outdated or weak encryption methods that can be easily cracked, so it's important to use strong, current encryption methods and update them regularly to stay secure [5, 6].

3.1 Example:

An example of cryptographic failure is when a company stores users' passwords in plain text instead of hashing them securely. If an attacker gains access to the database, they can easily see and use these passwords. This situation could have been avoided by using a strong hashing algorithm, which ensures that passwords are stored in a way that makes it extremely difficult for attackers to recover the original passwords even if they access the database

Another example of cryptographic failure is using an outdated encryption method like MD5 to protect credit card numbers. MD5 is no longer secure and can be cracked quickly with modern computers. If a hacker intercepts the encrypted data, they can easily decrypt the credit card numbers and use them fraudulently. This could be prevented by using a stronger, more up-to-date encryption method like AES.

3.2 Mitigation:

A best practice is to ensure that data is encrypted at rest, encrypted in transit and is end to end encrypted. At a minimum, do the following to mitigate cryptographic failures:

- Encrypt all sensitive data stored on disk.
- Disable caching for responses that contain sensitive data.
- Use strong, up-to-date algorithms, protocols, and keys, and manage keys properly.
- Encrypt all data in transit using secure protocols like TLS with forward secrecy (FS) and enforce encryption with HTTP Strict Transport Security (HSTS).
- Store passwords with strong adaptive and salted hashing functions like Argon2, scrypt, bcrypt, or PBKDF2.
- Avoid using outdated cryptographic functions and padding schemes like MD5, SHA1, and PKCS number 1 v1.5.
- Use authenticated encryption, not just encryption.
- Independently verify the effectiveness of your configuration and settings.

4. INJECTION

An injection vulnerability happens when an attacker tricks a system into running harmful code by inserting it into regular input, like a form field or URL. This can occur in various parts of a website or application, such as databases, operating systems, or user interfaces. When the system runs this harmful code, it can lead to unauthorized access, data theft, or other malicious actions. Simply put, injection vulnerabilities allow attackers to make a system do things it wasn't meant to do by sneaking in bad instructions. An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized.
- Dynamic queries or non-parameterized calls are used without context-aware escaping.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract sensitive records.
- Hostile data is directly used or concatenated in dynamic queries, commands, or stored procedures.

Common types of injection include SQL, NoSQL, OS command, ORM, LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is similar across all interpreters. The best way to detect if applications are vulnerable to injections is through source code review. Automated testing of all parameters, headers, URLs, cookies, JSON, SOAP, and XML data inputs is highly recommended. Organizations can use static (SAST), dynamic (DAST), and interactive (IAST) application security testing tools in the CI/CD pipeline to identify injection flaws before production deployment. [7, 8]

4.1 Example:

Let's see a simple example of an SQL Injection Attack. An attacker takes advantage of a vulnerability in a login form t bypass authentication. Normally, the code checks if the username and password match a record in the database:

SELECT * FROM users WHERE username = 'admin' AND password = 'password123';

Now, the attacker modifies the input, attacker enters OR '1'='1 in the query making the password field invaluable. The resulting query is

SELECT * FROM users WHERE username = 'admin' OR '1'='1' AND password = 'anything';

The OR '1'='1' part of the query is always true. This modifies the query to always return a valid user, bypassing the need for a correct password. The attacker gains access to the system without knowing the actual password. They can view or modify sensitive data, potentially compromising the entire application.

4.2 Mitigation:

Preventing injection requires keeping data separate from commands and queries. Here are some best practices to help organizations protect their applications from code injection attacks:

- Use Parameterized Queries as they are simple to learn and use, and they prevent SQL injection. They require you to define the SQL code first and then pass each parameter separately, keeping the code and data distinct.
- Encode HTML outputs to ensure that malicious input is displayed safely without being executed, preventing scripts from running.
- For any dynamic queries, escape special characters using the specific syntax for that interpreter. SQL structures like table and column names cannot be escaped, so user-supplied names are risky.
- Use LIMIT and other SQL controls as these prevent mass disclosure of records in case of SQL injection.
- Use static type systems to establish security checks at compile time, reducing runtime overhead and preventing type-related vulnerabilities.
- Use software composition analysis (SCA) tools to automatically check and fix potential injection vulnerabilities in the source code.
- Avoid JavaScript code serialization packages as it can lead to unsafe character handling. Developers should use safer methods to handle input data instead.

5. INSECURE DESIGN

Insecure design refers to weaknesses in a system due to missing or ineffective security controls. It's different from insecure implementation because it involves flaws in the design itself, not errors in coding. An insecure design can't be fixed by perfect coding since the necessary security measures were never included. A common cause of insecure design is failing to assess the business risks and determine the required security level for the software or system [9, 10, 11].

5.1 Example:

An example scenario is when an e-commerce website lets users log in with their username and password but doesn't limit the number of login attempts. Without rate limiting, an attacker can try many username and password combinations until they get in. This design flaw makes the system vulnerable to unauthorized access.

5.2 Mitigation:

- Use a secure development process with security experts to design and evaluate security and privacy controls.
- Create and use a library of secure design patterns and ready-to-use components.

- Apply threat modeling for important areas like authentication, access control, business logic, and key workflows.
- Include security requirements and controls in user stories.
- Add plausibility checks at each level of your application, from frontend to backend.
- Write unit and integration tests to ensure critical workflows are secure. Develop use-cases and misuse-cases for each part of your application.
- Limit resource consumption per user or service.

6. SECURITY MISCONFIGURATION

Security misconfiguration happens when an application is not set up correctly, making it vulnerable. This includes:

- Missing security hardening or misconfigured permissions.
- Enabled unnecessary features like extra ports or accounts.
- Default accounts and passwords are unchanged.
- Error messages reveal too much information.
- Latest security features are not used after upgrades.
- Security settings in servers, frameworks, libraries, and databases are not secure.
- Security headers are missing or improperly set.
- Software is outdated or has known vulnerabilities.

Without a consistent process for setting up security, systems are at higher risk [12, 13].

6.1 Example:

A company sets up a new database to store customer information but forgets to secure it properly. They leave the database accessible over the internet without any restrictions, use a weak or no password for the admin user, and don't encrypt the data. This makes it easy for attackers to find the database, guess the password, and access all the customer information. The attackers can view, change, or delete data and intercept any information being transmitted. To avoid this, the company should restrict database access, use strong passwords, encrypt data, and regularly update security settings.

6.2 Mitigation:

To protect your systems from security misconfigurations, follow these steps:

- Immediately change default usernames and passwords to strong, unique ones.
- Use firewalls and VPNs to restrict who can access your systems and only give users the permissions they absolutely need.
- Disable any services or features you don't need.
- Regularly update your software and apply security patches.
- Regularly review and change default settings to secure ones.
- Disable directory listings and set proper file permissions.
- Use encryption (like TLS/SSL) to protect data being transmitted over the internet.
- Enable logging to track access and changes and regularly check logs for any suspicious activity.
- Use automated tools to scan for vulnerabilities.
- Regularly conduct security assessments and penetration testing.
- Have an automated process to verify the effectiveness of configurations and settings.
- Customize error messages to avoid revealing system details.

7. VULNERABLE AND OUTDATED COMPONENTS

The software application might be at risk if:

- You don’t know the versions of all the components you use, both on the client and server side, including nested dependencies.
- The software, including the OS, servers, databases, applications, APIs, and libraries, is outdated, unsupported, or has known vulnerabilities.
- You don’t regularly scan for vulnerabilities or follow security updates for your software.
- You don’t promptly fix or update your platform, frameworks, and dependencies, waiting for monthly or quarterly patches, which leaves your system exposed for longer periods.
- Developers don’t test if updated or patched libraries work well with your software.
- You don’t secure the configurations of your software components properly [14, 15]

7.1 Example:

A small business uses an old version of a web framework (version 1.5) to run their website, while the latest version is 3.0. The old version has known security flaws that hackers can easily exploit, and it's no longer supported by the developers. This means the business is vulnerable to attacks that can lead to unauthorized access, data theft, or loss of control over the website. To fix this, the business should upgrade to the latest version, test it to make sure it works with their existing code, and regularly check for and apply updates to keep the website secure.

7.2 Mitigation:

- Ensure all software components, including the OS, frameworks, libraries, and applications, are regularly updated to the latest versions.
- Implement a patch management process to apply patches and updates promptly based on risk assessment.
- Schedule regular patching cycles and ensure critical patches are applied immediately.
- Use automated tools to receive alerts on new vulnerabilities and available patches.
- Regularly scan for vulnerabilities using security scanning tools.
- Continuously monitor sources like the Common Vulnerabilities and Exposures (CVE) database and the National Vulnerability Database (NVD) for known vulnerabilities.
- Remove or replace outdated and unsupported components with modern, maintained alternatives.
- Decommission unused or unnecessary software components to reduce the attack surface.
- Obtain software components only from official and reputable sources.
- Prefer signed packages and secure download links to avoid malicious modifications.
- Establish and enforce security policies that require regular updates, vulnerability scans, and prompt patching.

8. IDENTIFICATION AND AUTHENTICATION FAILURES

Verifying user identity, authentication, and session management is crucial to prevent authentication attacks. An application might have weaknesses if it:

- Allows brute force attacks or automated attacks like credential stuffing.
- Accepts default or weak passwords like "Password1", "admin/admin" and has weak or ineffective password recovery processes.
- Lacks effective multi-factor authentication.
- Stores passwords in plain text or uses weak methods to hash passwords.
- Application session timeouts are not set correctly
- Exposing session IDs in the URL, reusing them after login, and not invalidating them after logout or inactivity [16, 17]

8.1 Example:

Let’s say a user named John Doe tries to log into his online banking account using his username, johndoe123@gmail.com, and the password, password123. He accidentally mistypes his password, and the system doesn't recognize it. Instead of a generic error message, the system says, "Incorrect password. Hint: Your favorite pet's name."

This hint is meant to help John remember his password, but it's too easy for others to guess. Anyone who knows John or follows his social media, where he often posts about his dog Max, could figure it out. This makes it easier for someone with bad intentions to guess John's password and access his account, creating a security risk.

8.2 Mitigation:

- Use multi-factor authentication whenever possible to prevent automated attacks, such as brute force and the reuse of stolen credentials.
- When logging in and out, make sure session cookies are cleared and session IDs are changed to prevent attacks.
- Never use default credentials, especially for admin accounts.
- Check new or changed passwords against a list of the 10,000 worst passwords to ensure they are not weak.
- Follow guidelines for password length, complexity, and rotation from NIST 800-63b or other modern standards.
- Make registration, password recovery, and API processes secure by using the same messages for all outcomes to prevent attackers from figuring out if an account exists.
- Limit or delay repeated failed login attempts but avoid causing a denial of service.
- Log all failures and alert administrators about potential attacks.
- Use a secure session manager that generates a new random session ID after login, keeps session IDs out of URLs, securely stores them, and invalidates them after logout, idle periods, or timeouts.
- For password resets, use secure processes with validation tokens sent via email and ensure reset links expire after one use.
- Implement CAPTCHA to differentiate between humans and bots to prevent automated attacks.

9. SOFTWARE AND DATA INTEGRITY FAILURES

Software and data integrity failures happen when the code and infrastructure don't protect against integrity attacks. For example, an app might use plugins, libraries, or modules from untrusted sources or content distribution networks (CDNs). An insecure CI/CD pipeline can allow unauthorized access, malicious code, or even system takeover. Many applications now automatically download updates, but if these updates aren't properly verified, attackers could distribute their own malicious updates. Another example is insecure deserialization, where objects or data are converted into a format that attackers can see and manipulate [18, 19].

9.1 Example:

Let's say a user downloads an app on their mobile phone from an untrusted source and gives it access. If the app contains malicious code, it can be used by attackers to steal private information. Additionally, if the app automatically downloads updates without verifying them, this app could bring harmful software to the phone, putting the user's data and privacy at risk.

9.2 Mitigation:

- Use digital signatures to verify that software or data is from a trusted source and hasn't been altered.
- Ensure libraries and dependencies are from trusted repositories. For higher security, consider hosting an internal vetted repository.
- Use a software supply chain security tool, like OWASP Dependency Check, to verify components are free of known vulnerabilities.
- Have a review process for code and configuration changes to prevent malicious code from entering your software pipeline.
- Secure your CI/CD pipeline with proper segregation, configuration, and access control to maintain code integrity during build and deployment.
- Avoid sending unsigned or unencrypted serialized data to untrusted clients without integrity checks or digital signatures to detect tampering.
- Always use and develop apps with strong security practices.
- Regularly audit your code before it goes to production.
- Frequently perform penetration testing to ensure high-security levels.

10. SECURITY LOGGING AND MONITORING FAILURES

The OWASP Top 10 2021 emphasizes the need for proper logging and monitoring to detect and respond to breaches. Problems arise when important events like logins and high-value transactions aren't logged, warnings and errors are unclear, logs aren't monitored for suspicious activity, and logs are only stored locally. Without effective alerting thresholds and response processes, and if penetration tests don't trigger alerts, the application can't respond to attacks in real-time. Additionally, showing logging and alerting events to users or attackers can lead to information leaks. [20]

10.1 Example:

The operator of a children's health plan website didn't know their site was hacked because there was no monitoring or logging system. An outside source informed them that someone had accessed and altered sensitive medical records of over 3.5 million kids. Investigators found that developers had left major security flaws unfixed. This means data might have been leaking since early 2013, over seven years ago, due to the lack of monitoring and logging.

10.2 Mitigation:

- Ensure that login failures, access control failures, and server-side input validation failures are logged with enough information to identify suspicious or malicious accounts. Retain these logs long enough to allow for thorough forensic analysis later.
- Create logs in a format that can be easily processed by log management tools.
- Properly encode log data to prevent injection attacks against logging and monitoring systems.
- Set up an audit trail with integrity controls for high-value transactions to prevent tampering or deletion (e.g., using append-only database tables).
- Ensure that suspicious activities are quickly detected through effective monitoring and alerting by DevSecOps teams.
- Adopt an incident response and recovery plan, such as the one outlined in NIST 800-61r2 or later, or a Reportable Event Readiness Plan from the National Institute of Standards and Technology (NIST).
- Prevent public access to logs and alerts to avoid information leaks to users or attackers.

11. SERVER-SIDE REQUEST FORGERY

A server-side request forgery (SSRF) is a security flaw where an attacker tricks a server into sending requests to unauthorized places, leading to data leaks or unauthorized actions. This happens when web applications fetch remote resources without checking the user-supplied URL. Attackers can then direct requests to unexpected destinations, even those protected by firewalls. SSRF incidents are increasing because modern web apps often fetch URLs, and the complexity of cloud services makes the issue more severe. Notably, SSRF is listed in the OWASP Top 10 2021 and was involved in breaches at Capital One and Microsoft Exchange [21, 22]

11.1 Example:

Let's say a web app allows users to enter a URL to fetch and display an image. The application takes the URL as input, retrieves the image from that URL, and shows it on the website. However, this can be exploited by an attacker. Instead of providing a URL for an image, the attacker could enter a URL that points to internal services or sensitive data. For example, the attacker might enter a URL that accesses internal server information or private files, leading to unauthorized access and data exposure.

11.2 Mitigation:

- Use an allow list to restrict the URL schema, port, and destination
- Ensure all client-supplied input data is sanitized and validated.
- Ensure the response data matches expectations and never deliver the raw response body to the client.
- Do not rely on deny lists or regular expressions for SSRF mitigation, as attackers can bypass them using payload lists, tools, and alternate IP encodings.
- Log all accepted and blocked network flows on firewalls to monitor for potential security issues.
- Maintain URL consistency to prevent attacks like DNS rebinding and "time of check, time of use" (TOCTOU) race conditions

12. CONCLUSION

To sum up, it's essential to follow secure coding practices when developing strong software that can withstand today's digital threats. Understanding and fixing the vulnerabilities listed in the OWASP Top 10 can significantly improve application security. This guide offers insights into each OWASP vulnerability and practical steps to protect your code. By following these methods, you can ensure data protection, maintain user trust, and meet industry standards. Continuous learning and adapting to secure coding practices are crucial as threats evolve. Sticking to these principles will help developers contribute to a safer and more reliable digital world.

REFERENCES

[1]. https://owasp.org/about/
[2]. https://owasp.org/Top10/
[3]. https://www.immuniweb.com/blog/OWASP-broken-access-control.html
[4]. https://owasp.org/Top10/A01_2021-Broken_Access_Control/
[5]. https://www.pullrequest.com/blog/what-are-cryptographic-failures-and-how-to-prevent-giant-leaks/
[6]. https://owasp.org/Top10/A02_2021-Cryptographic_Failures/
[7]. https://brightsec.com/blog/sql-injection-attack/
[8]. https://owasp.org/Top10/A03_2021-Injection/
[9]. https://brightsec.com/blog/code-injection/#attack-prevention
[10]. https://www.horangi.com/blog/real-life-examples-of-web-vulnerabilities#:~:text=A04%3A2021%20Insecure%20Design&text=It%20is%20different%20from%20insecure,vulnerable%20to%20attacks%20and%20exploits.
[11]. https://owasp.org/Top10/A04_2021-Insecure_Design/
[12]. https://brightsec.com/blog/security-misconfiguration/#:~:text=What%20Is%20Security%20Misconfiguration%3F,application%20stack%2C%20cloud%20or%20network.
[13]. https://owasp.org/Top10/A05_2021-Security_Misconfiguration/
[14]. https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/
[15]. https://medium.com/@shivam_bathla/a06-2021-vulnerable-and-outdated-components-a5d96017049c
[16]. https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/
[17]. https://cyolo.io/blog/identification-and-authentication-failures-and-how-to-prevent-them
[18]. https://medium.com/@shivendra_anand/software-and-data-integrity-failures-a08-2021-5a67756ddc90
[19]. https://owasp.org/Top10/A08_2021-Software_and_Data_Integrity_Failures/
[20]. https://owasp.org/Top10/A09_2021-Security_Logging_and_Monitoring_Failures/
[21]. https://www.acunetix.com/blog/articles/server-side-request-forgery-vulnerability/
[22]. https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/