OWASP Top 10 Vulnerabilities: Injection and Insecure Design

Daniela Tomás
up202004946@edu.fc.up.pt

Diogo Nunes
up202007895@edu.fc.up.pt

João Veloso
up202005801@edu.fc.up.pt

***Abstract*—This document explores the critical topic of cybersecurity, specifically focusing on the difficult challenges presented by injection vulnerabilities and insecure design flaws in web applications.**

***Index Terms*—cybersecurity, injection, insecure design, exploitation, countermeasures**

I. Introduction

Insecure design flaws [1] and code injection [2] are vulnerabilities that can seriously jeopardize the confidentiality, availability, and integrity of web applications. Safeguarding digital assets and maintaining a secure online environment require both understanding the tactics used by attackers and putting effective countermeasures in place.

This document explores exploitation techniques employed by attackers, providing concrete examples to illustrate the potential risks. Moreover, it outlines essential countermeasure examples, emphasizing the significance of input validation, parameterized queries, secure coding practices, proper access controls, and regular security audits. The document also explores state-of-the-art trends and technologies, including advancements in web application firewalls, machine learning-based threat detection, and secure coding frameworks, which play an important role in mitigating these vulnerabilities. By understanding these threats and implementing proactive security measures, organizations can fortify their web applications against potential cyberattacks, ensuring a resilient and secure digital ecosystem.

II. A03:2021 - Injection

Injections occur when an application sends untrusted data to an interpreter. This can trick the interpreter into executing unintended commands or accessing data without the required authorization.

Injection flaws are very prevalent, particularly in legacy code, and are often found in SQL queries, LDAP queries, XPath queries, OS commands, program arguments, etc.

Injection flaws are normally discovered when there is the ability to examine code directly, but they can also be found via testing, albeit with more difficulty. There are also tools to help discover them, such as fuzzers and scanners.

A. Exploitation Techniques and Examples

1) SQL Injection: Attackers insert malicious SQL statements into input fields, manipulating the database.

For example, in a simple login form where the query is a simple SELECT query to the "users" table, an attacker could simply type `admin'#`, effectively cutting out the part of the query that utilizes the password and gaining admin privileges [3].

xkcd [4] has a relevant comic about SQL injection (Fig. 1).

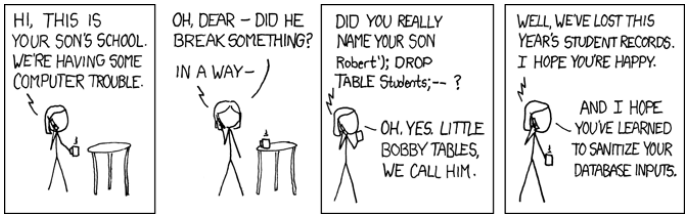

Fig. 1. xkcd about SQL injection.

2) Command Injection: Command Injection is a security vulnerability that occurs when an attacker can manipulate the input of a system command. By injecting malicious commands, attackers can execute arbitrary code on the underlying operating system. This type of vulnerability is particularly dangerous when user input is directly incorporated into system commands without proper validation or sanitization.

For example, in a simple web application that allows a user to ping a server by inserting a hostname or IP, the user can provide the following input: `8.8.8.8; ls -la`, which will lead to the ls command being run after the ping [5].

3) LDAP Injection: Attackers exploit this vulnerability to manipulate the queries, potentially gaining unauthorized access, retrieving sensitive information, or performing unintended operations on the LDAP server.

For instance, in an application that uses LDAP to authenticate users against a directory server. The application constructs an LDAP query to validate user credentials. An attacker, instead of providing a valid username and password, enters the following input for the username: `*)(uid=*))(|(uid=*`. In this case, the attacker's input manipulates the query to match all users, bypassing the authentication process. The attacker can potentially gain unauthorized access to the application [6].

B. Countermeasures

First of all, we need to understand the injection attack that we are dealing with. So knowing how these attacks work and their potential repercussions is a good start to creating

a countermeasure. After we understand the attack, a code review must be performed. This means that the code must be examined for vulnerabilities. We should look for any user input or external data that has been incorporated directly into the code through queries or commands without proper validation or sanitization. This input validation or sanitization must ensure that any data coming from non-trusted sources is validated and sanitized before it is used in the application. A good example of sanitizing user input are the parameterized statements. These statements, used in database-related injection attacks like SQL injection, are preferable to concatenating user input into SQL queries.

We can also use APIs that are trusted and have well-documented interfaces. Using these types of APIs avoids the use of the interpreter entirely or provides a parameterized interface.

Regularly conducting security testing is good because we are able to identify and mitigate any weaknesses related to injection vulnerabilities. Also implementing security headers to protect against various types of attacks and implementing proper access controls and authorization mechanisms to ensure that users can only access and modify data or perform actions they are authorized to.

III. A04:2021 - Insecure Design

Insecure design vulnerabilities result from inadequate system architecture or design choices that provide security flaws. Attackers take advantage of these flaws to compromise the system's availability, confidentiality, or integrity. Injection vulnerabilities can result from insecure design practices.

A. Exploitation Techniques and Examples

Insecure design can present itself in a variety of ways, and attackers are fast to exploit flaws.

1) Weak Authentication Design: To exploit the vulnerability, attackers can obtain unauthorized access to sensitive data and increase their privileges within a system by circumventing weak access control or authentication mechanisms.

For example, if an application has a weak or easily guessable password encryption scheme, attackers can crack passwords using brute force tactics or dictionary attacks and compromise user accounts [7].

2) Inadequate web server design: Web servers that cannot manage a huge amount of incoming HTTP requests might be overwhelmed with excessive requests, resulting in performance degradation or system failure.

E.G., flooding a website with a large amount of HTTP requests, using server resources, and perhaps slowing or making the website unavailable [8].

3) Detailed Error Messages: Providing detailed error messages to users or exposing sensitive information such as the internal state of the system in error messages.

For instance, an attacker can guess valid credentials if a login error message indicates if a username or password is wrong [9].

B. Countermeasures

Preventing insecure design is vital for developing secure applications. To protect applications or systems, there are multiple practices to be aware of.

Assuring that development teams are well-versed in security best practices and aware of common design flaws and their implications aids in the identification of possible vulnerabilities. It is also important to ensure that every feature and function is built with security in mind and validates security measures.

Another way to prevent insecure design is to create a library of secure design patterns and pre-built components. This not only saves time but also ensures that security precautions are used consistently.

A proactive approach to security is threat modeling. It entails methodically locating possible dangers and weaknesses in a program, especially in areas like authentication, access control, business logic, and crucial data flows. It may prevent security problems by attending to these worries throughout the planning and design phases.

Additionally, it is essential to make sure that error messages only include information that is relevant to the target audience.

Finally, it's critical to implement resource consumption limits by user or service. As a result, no one user or service can overload the system, which may otherwise be used to compromise system security.

IV. Discussion of the State-of-the-Art

The latest trends, technologies, and best practices related to countering injection and insecure design vulnerabilities include advancements in web application firewalls, machine learning-based threat detection, secure coding frameworks, automated testing, and a shift-left approach to security.

Injection vulnerabilities, such as SQL Injection and Command Injection, occur when untrusted data is improperly included in the code. The state-of-the-art includes parameterized queries, ORMs and Security Libraries, Web application firewalls, content security policies, static analysis and dynamic scanning tools, regular expression limitations, and API security to combat injection risks.

Insecure design encompasses security flaws within an application's architecture and development decisions. The state-of-the-art includes practices such as thread modeling, secure development frameworks, security by default, microservices and serverless architectures, Continuous Integration/Continuous Deployment (CI/CD), and a Secure API Design.

V. Conclusion

In the landscape of cybersecurity, addressing vulnerabilities such as injection attacks and insecure design flaws is vital. This document has explored the challenges posed by injection vulnerabilities, encompassing SQL, command, and LDAP injections, as well as the diverse aspects of insecure design, including inadequate authentication mechanisms, web server vulnerabilities, and error handling flaws.

Understanding the exploitation techniques deployed by attackers and implementing effective countermeasures are important steps towards ensuring the security and integrity of web applications. Additionally, a comprehensive understanding of the attack vectors is crucial for devising effective countermeasures.

Addressing insecure design demands a proactive approach, including secure architecture reviews, integration of security principles from the design phase, robust security training for development teams, and consistent adherence to secure coding practices. Access controls, security audits, and embracing the concept of security by design are indispensable in mitigating the risks associated with insecure design flaws.

In conclusion, by embracing a holistic approach that combines knowledge and vigilance, organizations can fortify their web applications, ensuring a resilient and secure digital ecosystem in the face of ever-changing cybersecurity challenges.

REFERENCES

[1] https://owasp.org/Top10/A04_2021-Insecure_Design/
[2] https://owasp.org/Top10/A03_2021-Injection/
[3] The MITRE Corporation, Common Weakness Enumeration, "CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')". https://cwe.mitre.org/data/definitions/89.html, 2023.
[4] https://xkcd.com/327/
[5] The MITRE Corporation, Common Weakness Enumeration, "CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')". https://cwe.mitre.org/data/definitions/78.html, 2023.
[6] The MITRE Corporation, Common Weakness Enumeration, "CWE-90: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')". https://cwe.mitre.org/data/definitions/90.html, 2023.
[7] The MITRE Corporation, Common Weakness Enumeration, "CWE-257: Storing Passwords in a Recoverable Format". https://cwe.mitre.org/data/definitions/257.html, 2023.
[8] The MITRE Corporation, Common Weakness Enumeration, "CWE-799: Improper Control of Interaction Frequency". https://cwe.mitre.org/data/definitions/799.html, 2023.
[9] The MITRE Corporation, Common Weakness Enumeration, "CWE-209: Generation of Error Message Containing Sensitive Information". https://cwe.mitre.org/data/definitions/209.html, 2023.
[10] https://owasp.org/www-project-top-ten/
[11] https://sectigostore.com/blog/what-is-owasp-what-are-the-owasp-top-10-vulnerabilities/
[12] Dafydd Stuttard and Marcus Pinto, The Web Application Hacker's Handbook, 2nd ed, 2011.