Web Application Vulnerabilities

By K. JAGADEESH
connect2jagadeesh@zohomail.in

INDEX

INTRODUCTION

What is OWASP

OWASP stands for the Open Worldwide Application Security Project, a global nonprofit organization dedicated to improving the security of software across platforms such as web and mobile applications, IoT, infrastructure-as-code, and more.

OWASP's mission is to empower organizations and individuals to build, operate, and maintain trustworthy software through open-source projects, education, and community collaboration. Its vision is simple: "**No more insecure software**".

The OWASP Top 10

The OWASP Top 10 is the organization's flagship project—a regularly updated report ranking and describing the ten most critical web application security risks.

It serves as an industry standard for awareness, compliance, and secure development practices, cited in industry regulations and used by auditors to assess web security.

The Top 10 is crowdsourced from experts worldwide and acts as a guideline for developers, security professionals, and organizations to manage, mitigate, and remediate security flaws.

What is Web-Vulnerabilities

Web application vulnerabilities are security weaknesses or flaws in web applications that can be exploited by attackers to compromise the confidentiality, integrity, or availability of the application and its data. These vulnerabilities arise from design, implementation, or configuration errors and can allow unauthorized access, data theft, manipulation, or service disruption.

Impact

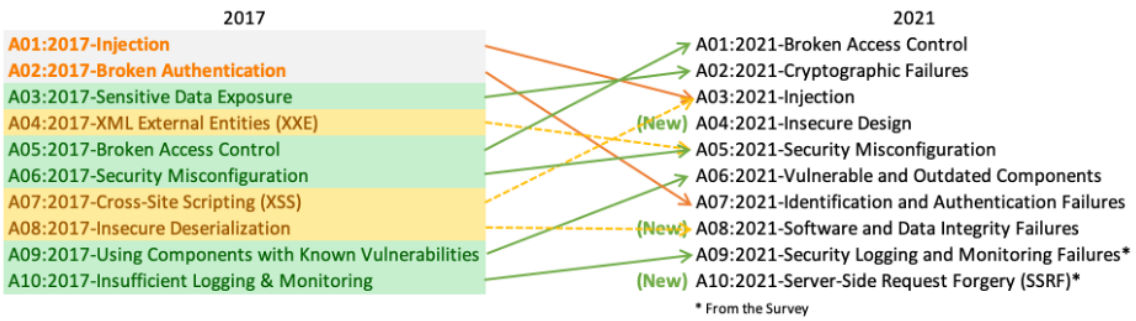
OWASP enables open participation and contribution, harnessing collective expertise to produce up-to-date resources for securing modern software environments.

Its efforts shape secure software development and maintenance, making its best practices essential for anyone working in cybersecurity or software engineering.

Want know more about OWASP visit: https://owasp.org/Top10/

The OWASP 2021 Top 10 Vulnerabilities

1. **Broken Access Control**: Weaknesses in authentication mechanisms let attackers bypass login controls, impersonate users, or escalate privileges.
2. **Cryptographic Failures**: refer to security weaknesses arising from the improper use, absence, or misconfiguration of cryptographic systems designed to protect sensitive data.
3. **Injection Attacks:** Attackers inject malicious commands or code through un-sanitized user inputs to manipulate or retrieve sensitive data. This can lead to data breaches or full system control.
4. **Insecure Design: Insecure design** refers to vulnerabilities which are inherent to the application's architecture. They are not vulnerabilities regarding bad implementations or configurations.
5. **Security Misconfiguration:** Security Misconfigurations are distinct from the other Top 10 vulnerabilities because they occur when security could have been appropriately configured but was not.
6. **Vulnerable and Outdated Components:** Third-party libraries, frameworks, or software dependencies used in web applications that contain known security vulnerabilities.
7. **Identification and Authentication Failures:** vulnerabilities in a system's mechanisms that verify users' identities and authenticate them before granting access.
8. **Software and Data Integrity Failures:** This vulnerability arises from code or infrastructure that uses software or data without using any kind of integrity checks.
9. **Security Logging and Monitoring Failures:** refer to security weaknesses that arise when a system or application does not properly record, monitor, or analyze security-relevant events.
10. **Server-Side Request Forgery (SSRF):** This type of vulnerability occurs when an attacker can coerce a web application into sending requests on their behalf to arbitrary destinations while having control of the contents of the request itself.



Understanding The Top 10

1. Broken Access Control (BAC)

Access control is the mechanism by which an application enforces policies on which users (or services) can access specific resources or perform certain functions. Broken access control occurs when these enforcement mechanisms are flawed or inconsistent, allowing unauthorized users to act outside their intended permissions.

Broken access control can manifest through several types of vulnerabilities:

- **Insecure Direct Object Reference (IDOR) :** Attackers manipulate resource identifiers in URLs or requests (e.g., changing a user ID in a profile URL) to access data or functions they shouldn't.
- **Horizontal Privilege Escalation:** A user accesses resources or data belonging to other users with the same privilege level (e.g., viewing or modifying another user's profile).
- **Vertical Privilege Escalation:** A less privileged user gains access to functions or data reserved for higher privilege users, such as admins.
- **Context-Dependent Privilege Escalation:** Users gain unintended privileges because of a specific context or state of the application (e.g., modifying another customer's shopping cart by changing the cart ID).
- **Session Management Flaws:** Attackers hijack or misuse session tokens to impersonate legitimate users and bypass authorization checks.

Risks & Impacts

The impacts of broken access control can be severe, including unauthorized data disclosure, account takeover, financial losses, legal penalties, and reputational damage for organizations. Attackers exploiting such flaws can view or modify sensitive information and perform administrative tasks, potentially compromising the entire system.

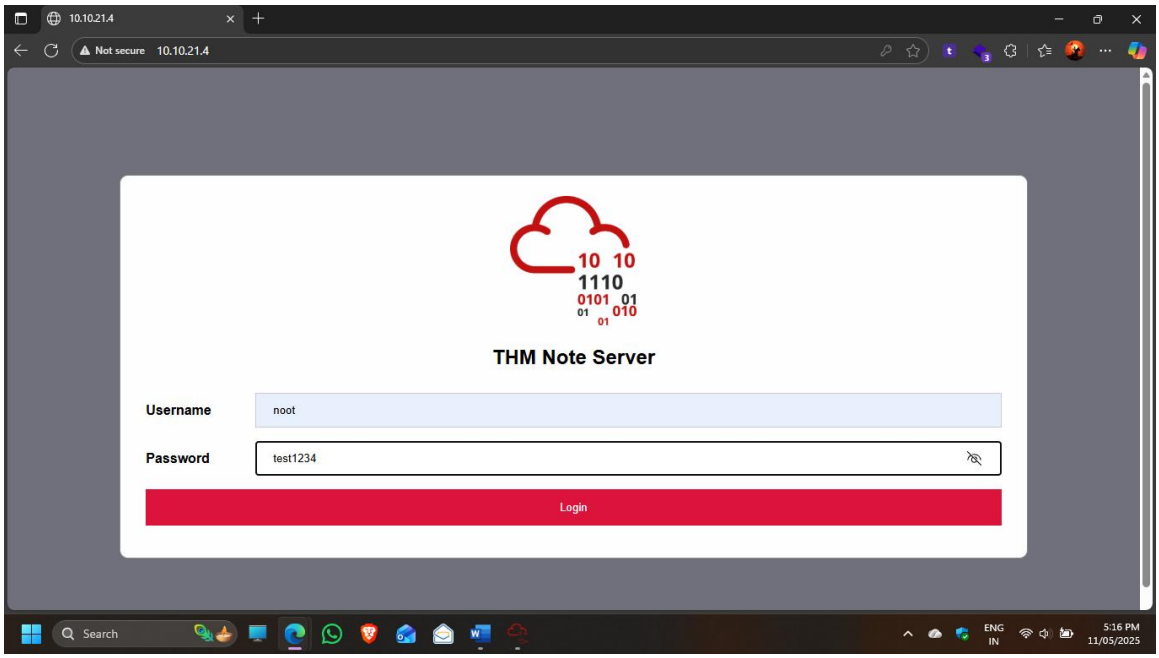
Common Attack Scenarios

- Changing URL parameters to view or modify another user's data without permission.
- Directly accessing admin-only pages without proper authorization checks.
- Exploiting weaknesses in access policies to escalate privileges within the system

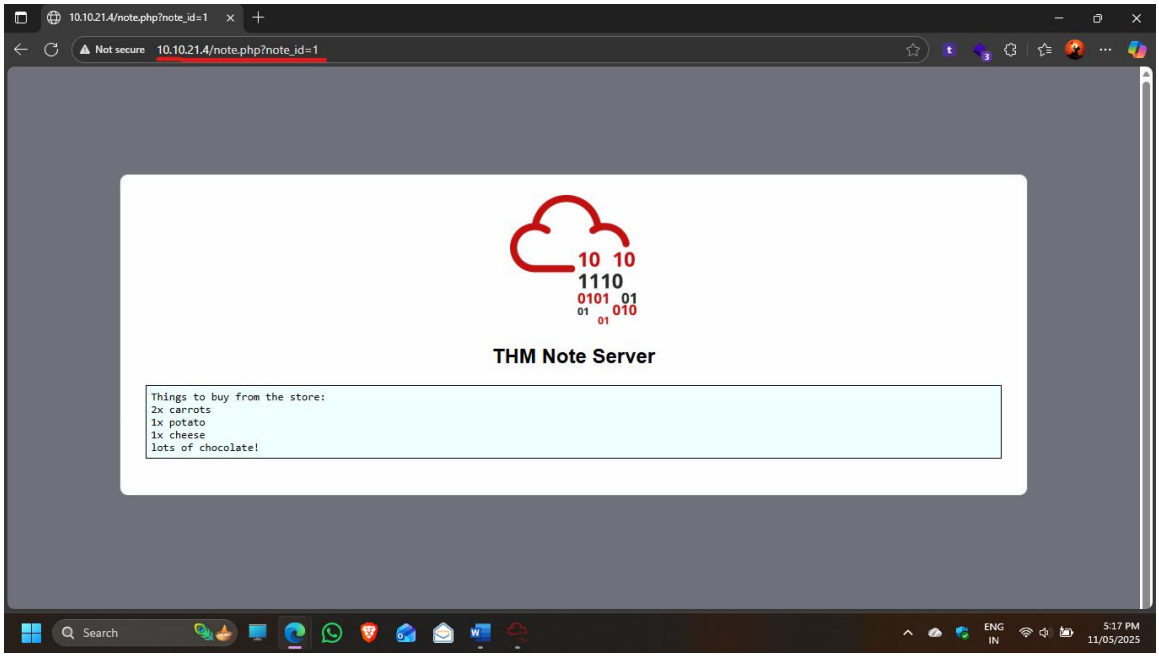
Broken access control is a major security weakness where authorization enforcement is improperly done, leading to unauthorized access and actions. It is considered the top security risk for web applications by OWASP and demands diligent mitigation.

IDOR Example

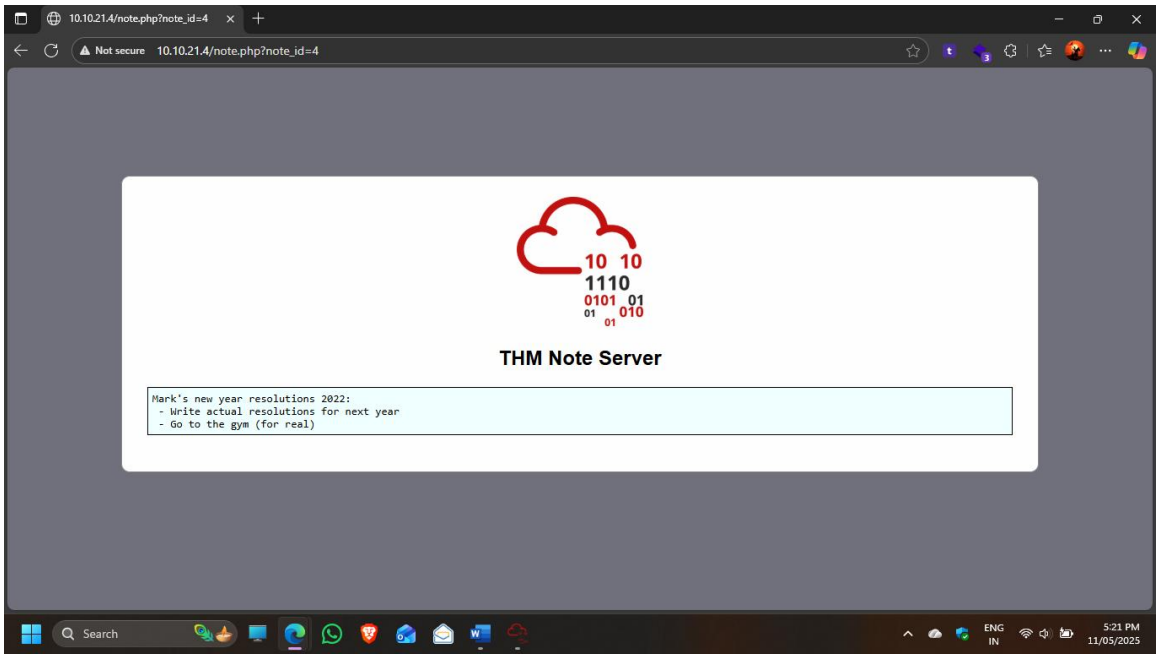
Step 1 : Login with our own Credentials.



Step 2: We can see or user ID and we are able to edit the value in that URL.

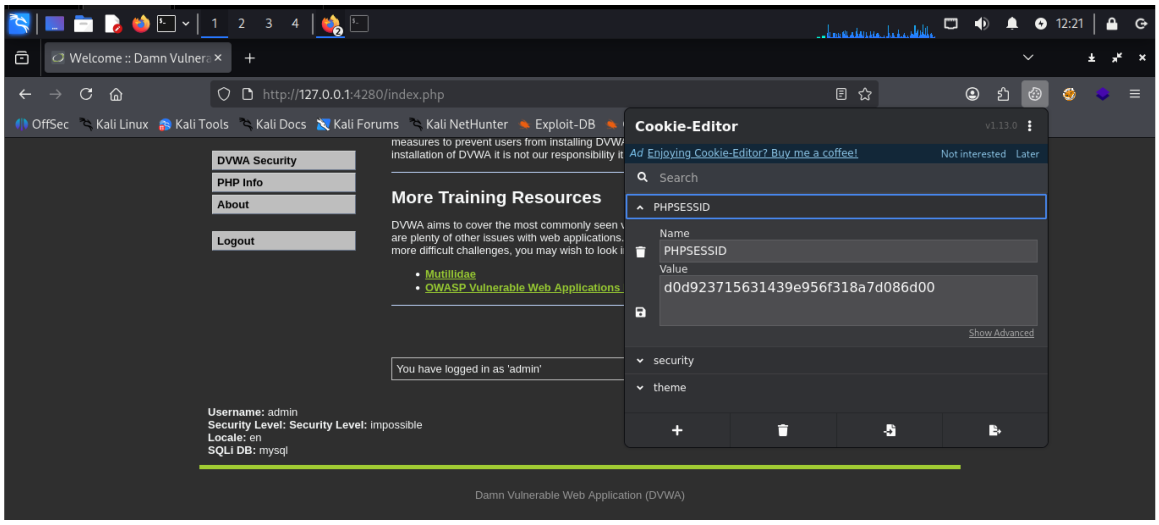


Step 3：When we change the Input value of the note_id variable in that url we are able to access the other users Note without any credential of that users.

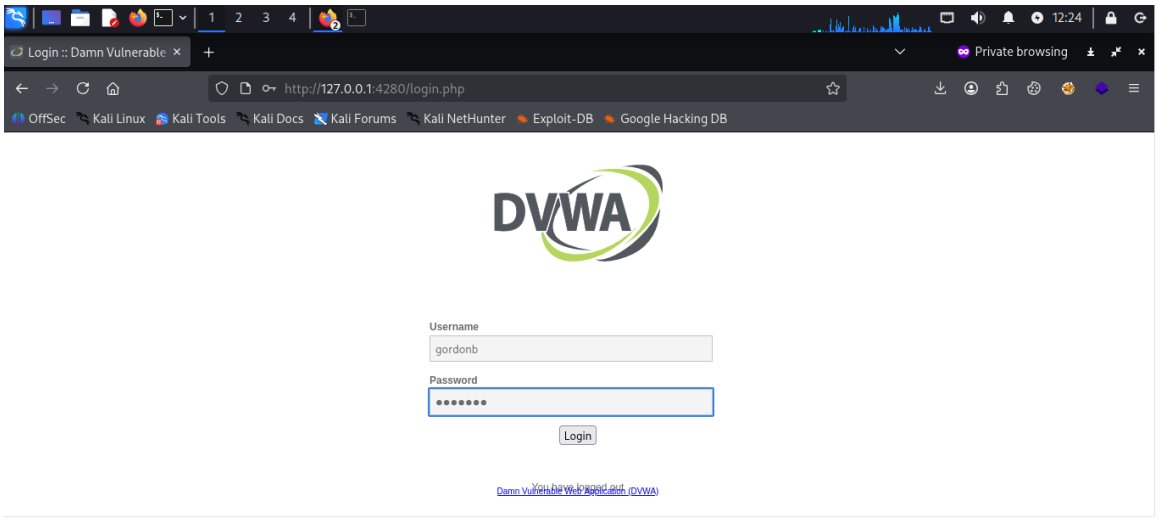


Using Session Token

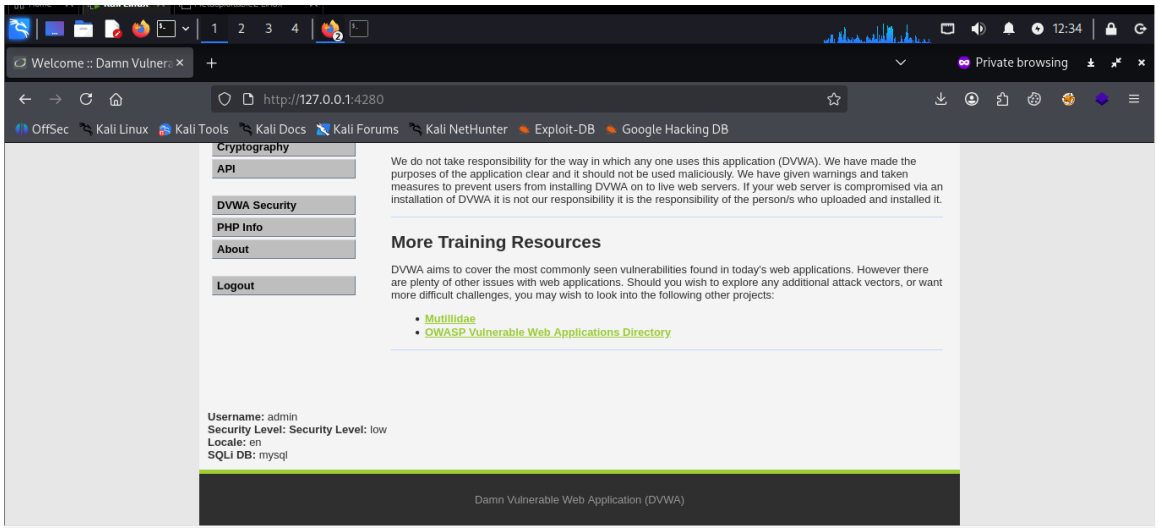
Step 1：Find the Users session ID. Here “admin” is our target and our session ID is “d0d923715631439e956f318a7d086d00”. Copy the session ID.



Step 2: Login with our personnel account. From this account we are going to check whether the Web-App vulnerable to BAC or not. Here “gordonb” would be our personnel account.



Step 3: Just paste the Session ID of “admin” user in the place of “gordonb” session ID and save it. Then refresh the page



In this case we are able to access the other users Data without their concern and even any authentication by Server. This is called IDOR.

Preventive Measures

- Implementing robust authorization checks on every request, not trusting client-side controls.
- Following the principle of least privilege, granting users only the minimum access needed.
- Validating and enforcing authorization dynamically at the business logic and data access layers.
- Regular security testing, including penetration testing and code reviews, to detect access control vulnerabilities.
- Proper session management to prevent token hijacking and misuse.

2. Cryptographic Failure

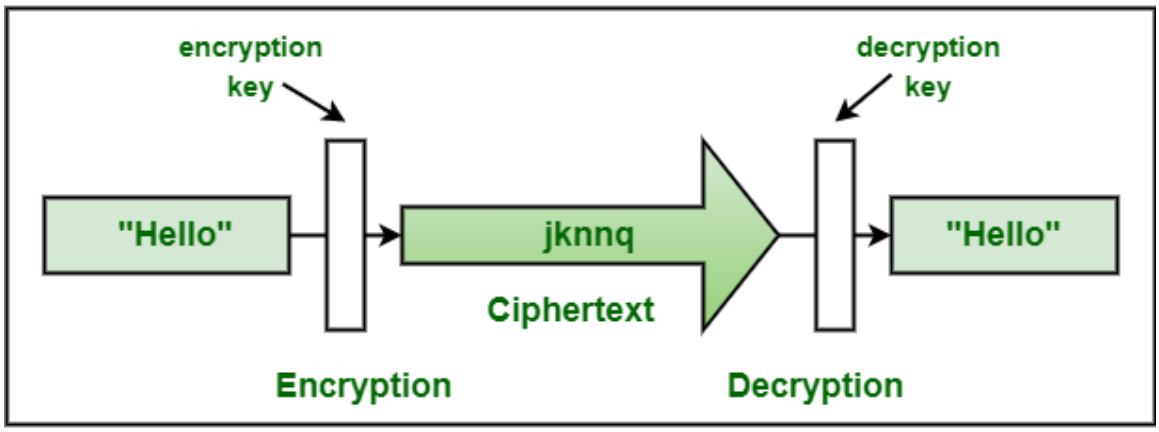
Cryptographic failure, as defined by OWASP, refers to security vulnerabilities that arise from the incorrect use, weak implementation, or absence of cryptographic protection, leading to exposure of sensitive data. This category replaced the broader “Sensitive Data Exposure” in the OWASP Top 10 of 2021 to emphasize failures in cryptographic mechanisms rather than just the result of exposed data.

These failures are often caused by:

- No encryption or weak encryption (storing or transmitting data in clear text)
- Using outdated or broken cryptographic algorithms (e.g., MD5, SHA-1, DES, RC4)
- Poor cryptographic key management (hard-coded keys, reused keys, lack of access control)
- Insecure random number generation for keys or initialization vectors
- Misconfigured SSL/TLS protocols and certificates (self-signed, expired, unsupported versions)
- Weak or fast password hashing without salts (allowing brute-force and rainbow table attacks)
- Improper implementation or design flaws in crypto systems
- Side-channel attacks exploiting timing, power consumption, or other leaks.

Importance

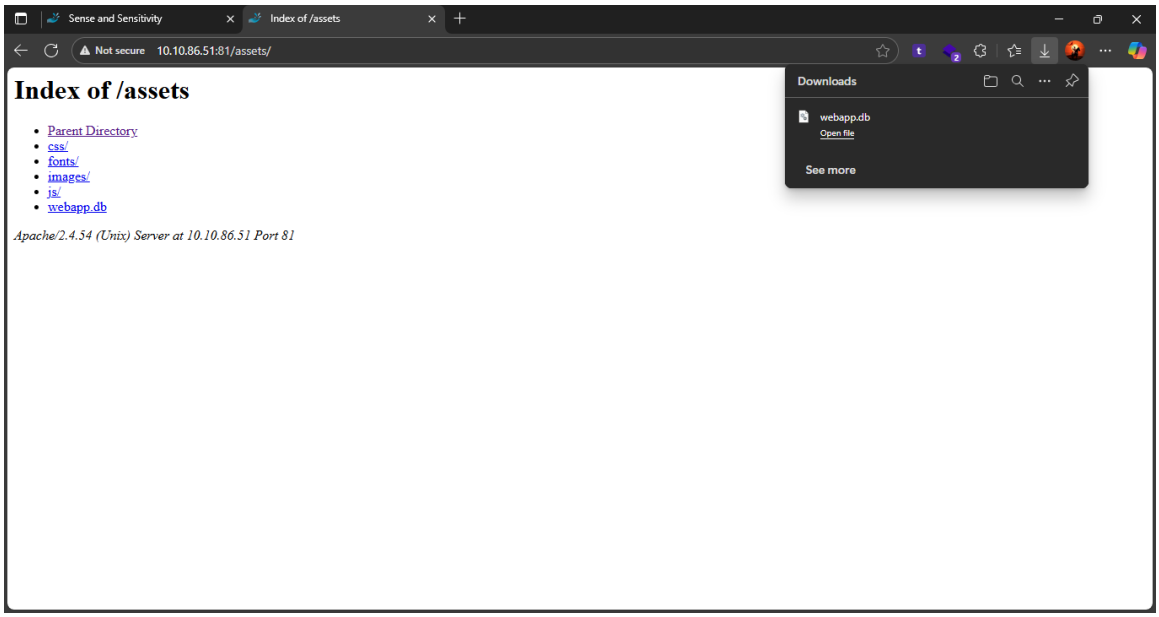
Cryptographic protection is often the last line of defense. Most breaches do not stem from breaking strong ciphers but from missing, weak, or incorrectly used cryptography. This category encompasses failures that allow attackers to access or manipulate sensitive data even if other security controls are bypassed.



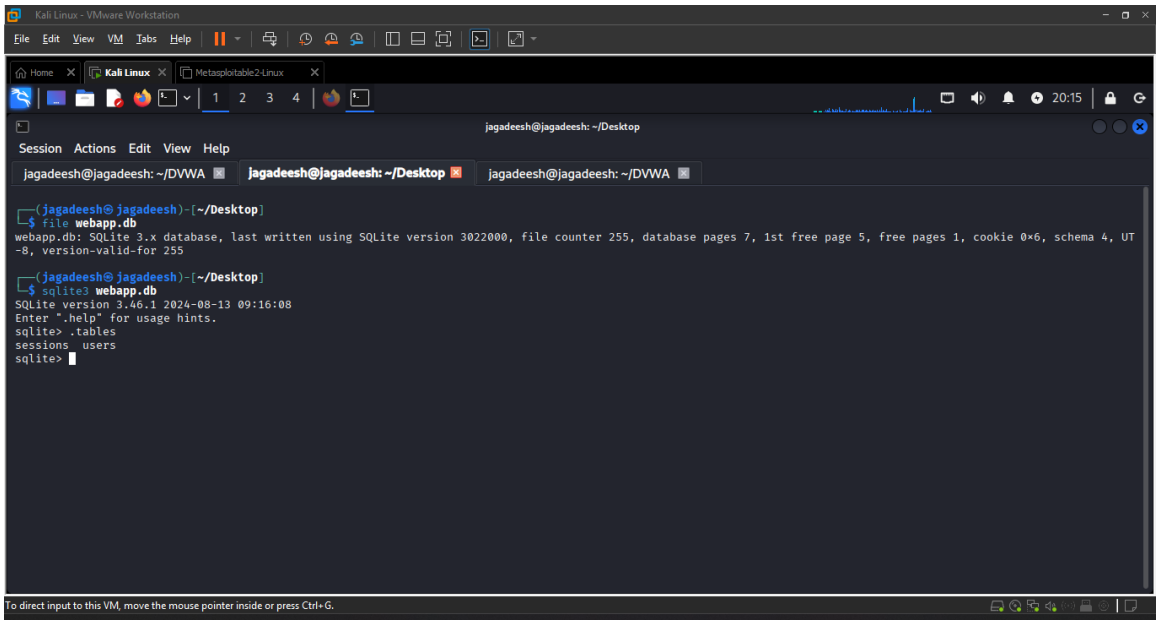
Cryptography

Example

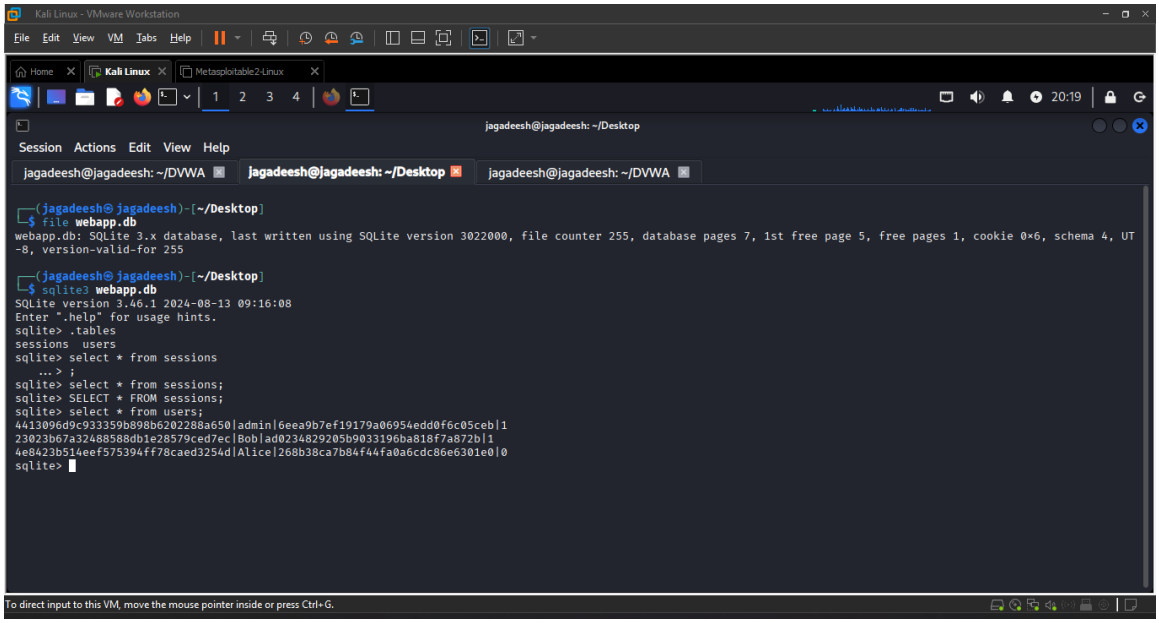
Step 1: Let say we've found a directory or a file which may contains Important files and Passwords. Here "/assets" would be like that. And we downloaded that in this case. In other cases, we can see them on web page.



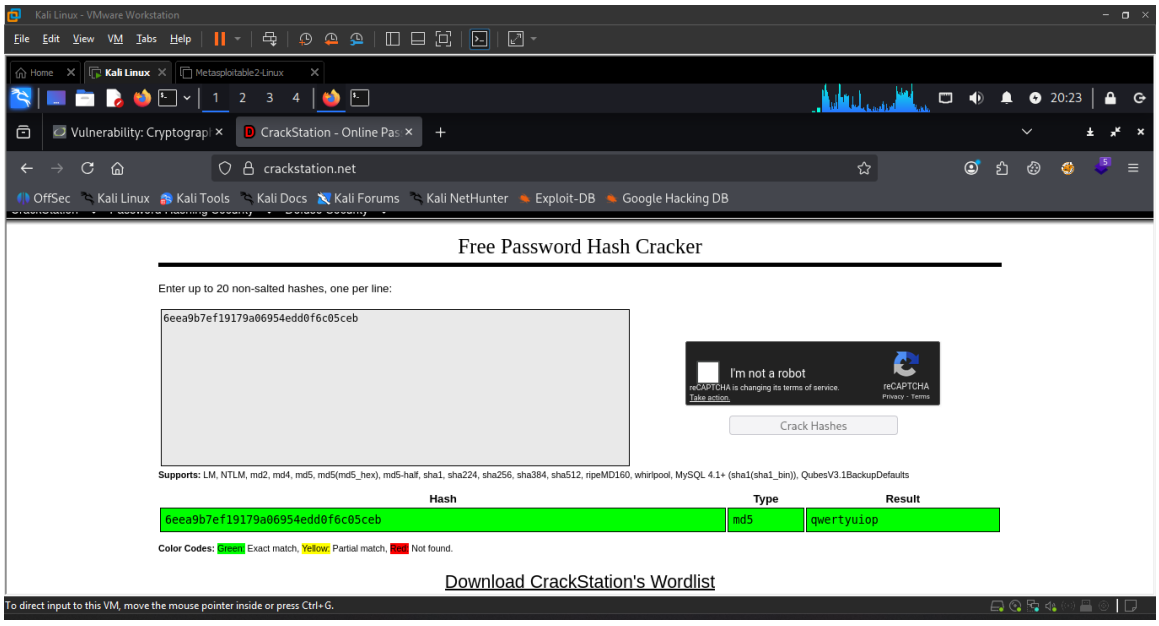
Step 2: We found that the "*.db" file is an sqlite3.* format. So, we can access by using sqlite3. And we found 2 tables in that "sessions" & "users".



Step 3: By opening the table users, we found 3 users admin, Bob and Alice. And we found their password hashes. Let's try to crack those password hashes.



Step 4: By applying a simple hash cracking technique, we found the actual Password of the user "admin".



In this case we are able to find the password hashes easily and can crack those hashes seamlessly, which is very easy to crack.

Common Vulnerabilities & CWEs

OWASP associates many Common Weakness Enumerations (CWEs) under Cryptographic Failures, such as:

- CWE-259 Hard-coded passwords
- CWE-327 Risky cryptographic algorithms
- CWE-331 Insufficient entropy (weak randomness)
- CWE-321 Hard-coded keys
- CWE-329 Non-random IV in CBC mode
- CWE-759/760 Weak password hashing.

Prevention Best Practices

- Use strong, modern encryption algorithms such as AES-256 for data at rest and TLS 1.2/1.3 for data in transit.
- Avoid deprecated or broken algorithms and protocols, disabling SSL 3.0, TLS 1.0, and any weak cipher suites.
- Manage keys securely using dedicated key management services; avoid hard-coding or storing keys insecurely.
- Implement proper password hashing with adaptive salted algorithms like Argon2, bcrypt, or PBKDF2.
- Ensure TLS certificates are properly validated, trusted, and not expired.
- Use cryptographically secure random number generators (CSPRNGs) for generating keys or nonces.
- Limit and classify sensitive data, encrypting only what is necessary and avoiding logging sensitive information.

Cryptographic failures are critical security issues resulting from the improper use, weak implementation, or absence of cryptographic measures leading to sensitive data exposure. Addressing these failures is essential for protecting user data and maintaining trust and regulatory compliance.

3. Injection Attacks

Injection attacks occur when untrusted user input is sent to an interpreter as part of a command or query, successfully misleading the interpreter into executing unintended commands or accessing data without proper authorization. This lack of input validation, sanitization, or escaping enables attackers to change the intended logic of commands or queries, often yielding unauthorized database access, remote code execution, or manipulation of application behavior.

Common Types of Injection Attacks

- **SQL Injection (SQLi):** The most well-known type, where attackers inject malicious SQL queries into database queries. This can expose sensitive data such as passwords, credit card information, or personally identifiable information by altering the SQL command's behavior.
- **Cross-Site Scripting (XSS):** Although sometimes classified separately, XSS is a form of injection where attackers inject malicious scripts into web pages viewed by other users. This can lead to theft of session tokens, cookies, or manipulation of website content.
- **LDAP Injection:** Exploits web applications that build LDAP queries from user input, allowing attackers to dynamically alter LDAP queries and gain unauthorized access or modify directory structures.
- **Command Injection:** Allows attackers to execute arbitrary commands on the host operating system through a vulnerable application by injecting operating system command syntax.
- **CRLF Injection:** Involves injecting carriage return and line feed characters to manipulate HTTP headers and responses, potentially leading to web cache poisoning or session fixation attacks.
- **Host Header Injection:** Exploits vulnerability in processing the Host HTTP header to conduct attacks like web cache poisoning or redirect victims to malicious URLs.

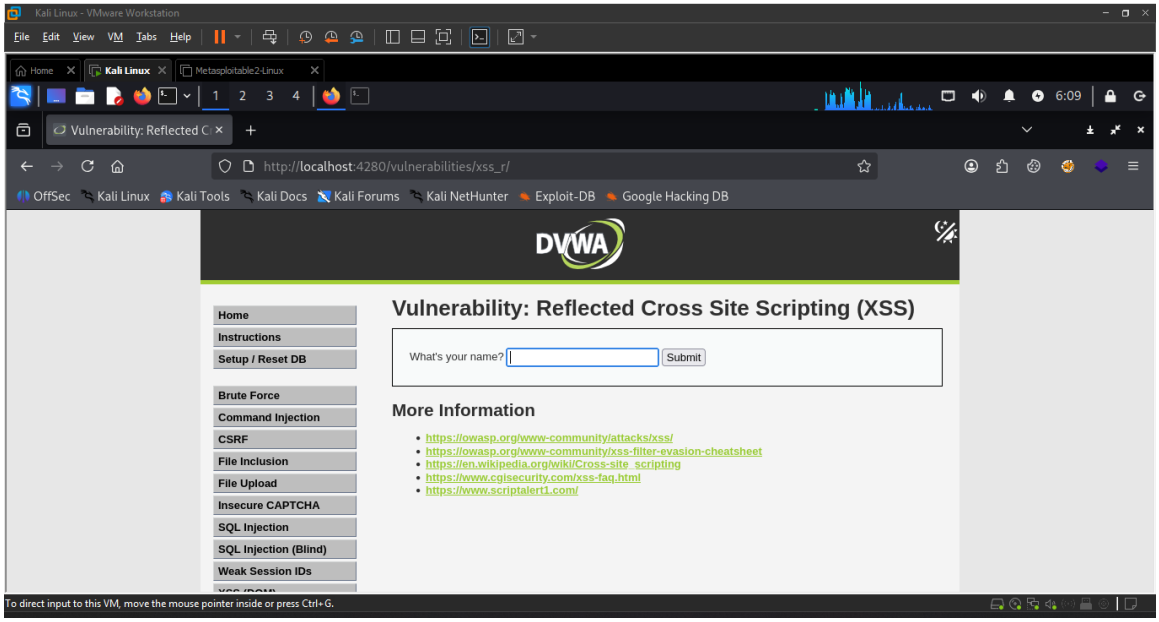
Impact of Injection Attacks

Successful injection attacks can lead to:

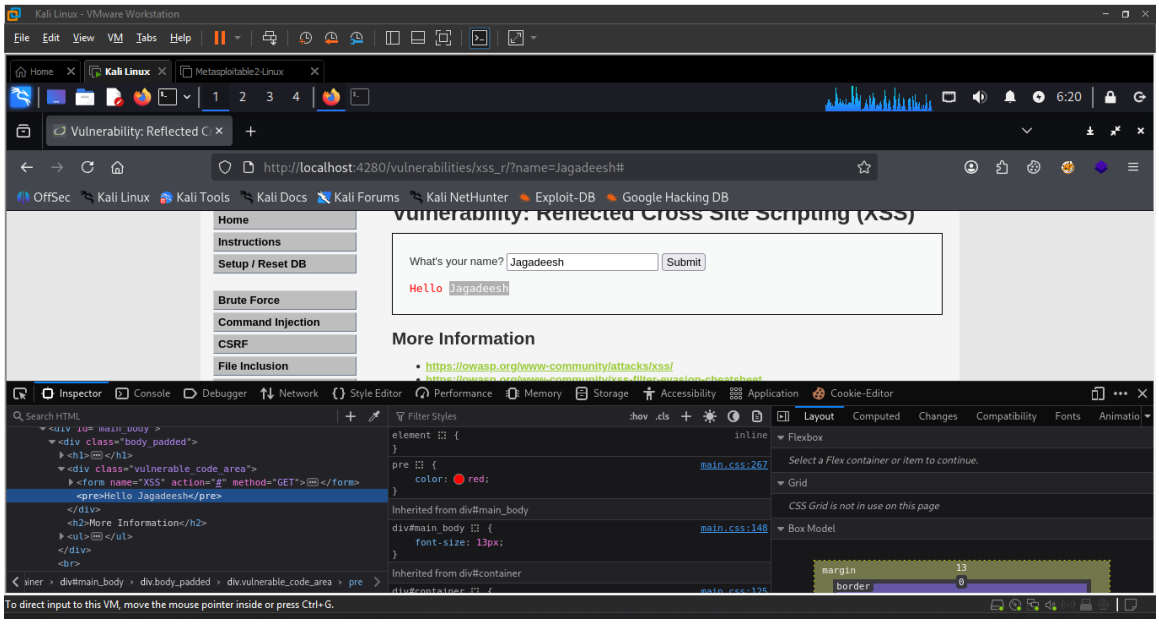
- Unauthorized data viewing, editing, or deletion.
- Authentication bypass and privilege escalation.
- Data loss, corruption, or breach.
- Complete system compromise.
- Execution of arbitrary commands leading to control over backend systems.

XSS Injection

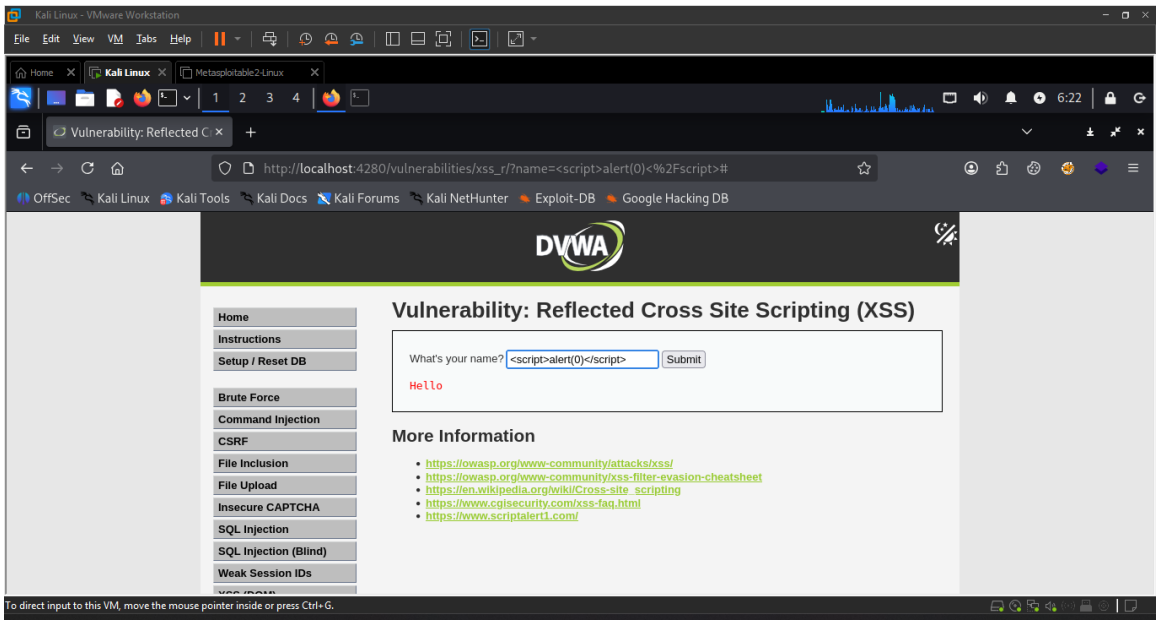
Step 1: Let's find a user input field like search box in the target website. The user input fields may like URL query parameters, Form fields, Message boards, HTTP headers etc.

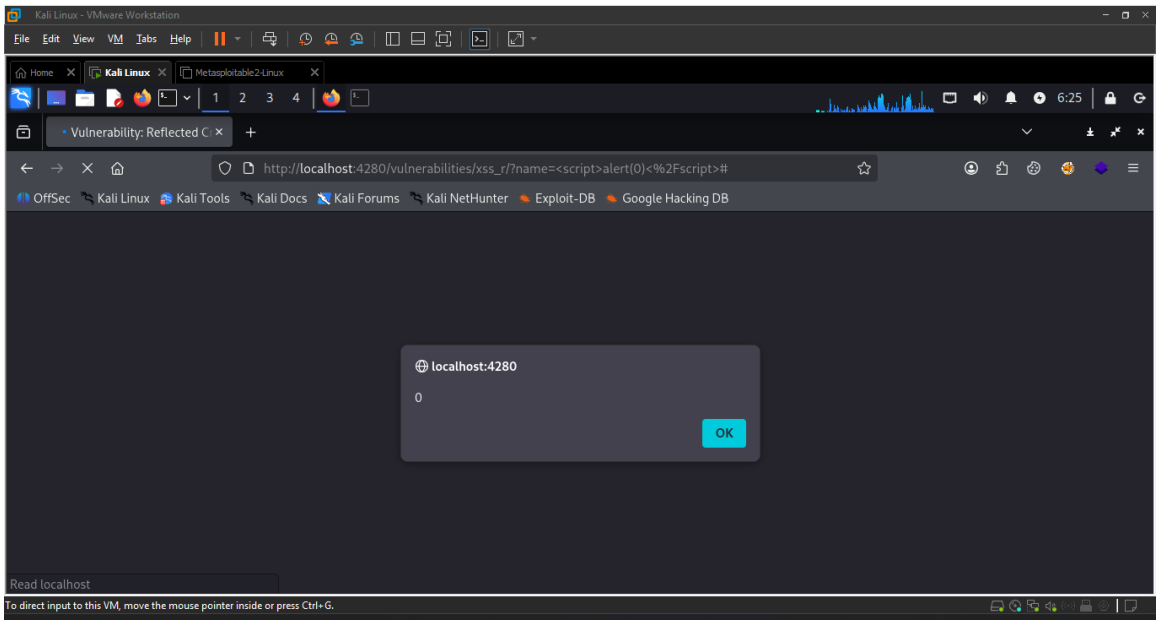


Step 2: Make sure that the input given to that page must be reflect in that webpage. So, that we sure about that our input can change the backend script of that webpage.

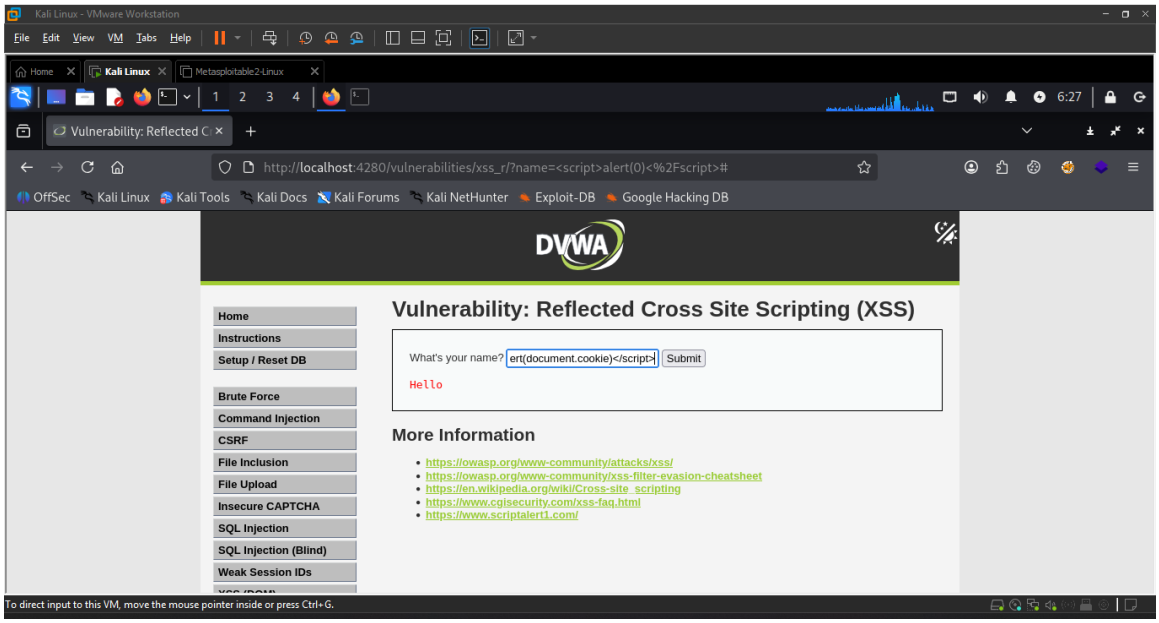


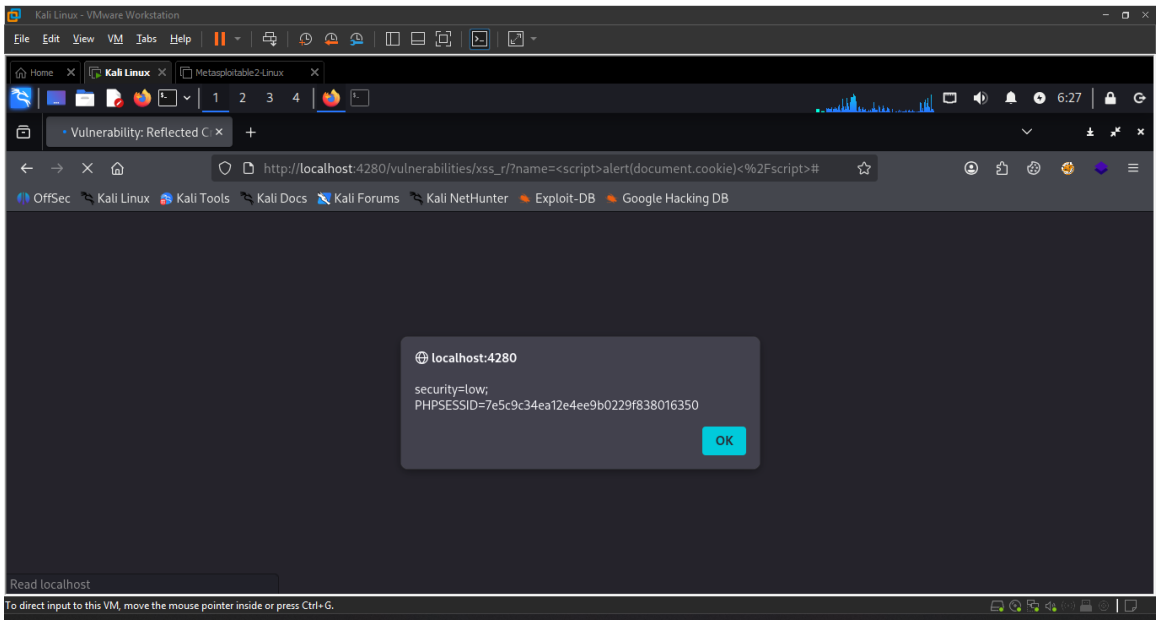
Step 3: Lets go with the very basic JS query “<script>alert(0)</script>”. If it executed in the console successfully the web-app have been vulnerable to XSS.





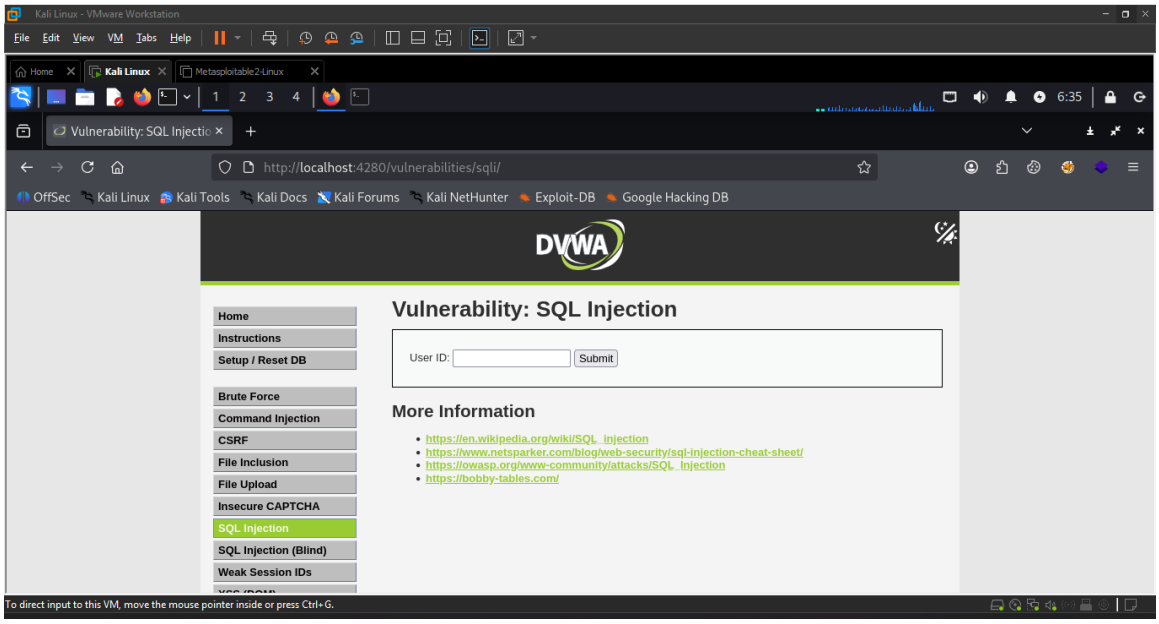
By using this technique, we can also get the session token, cookies and more of the user of that current account.



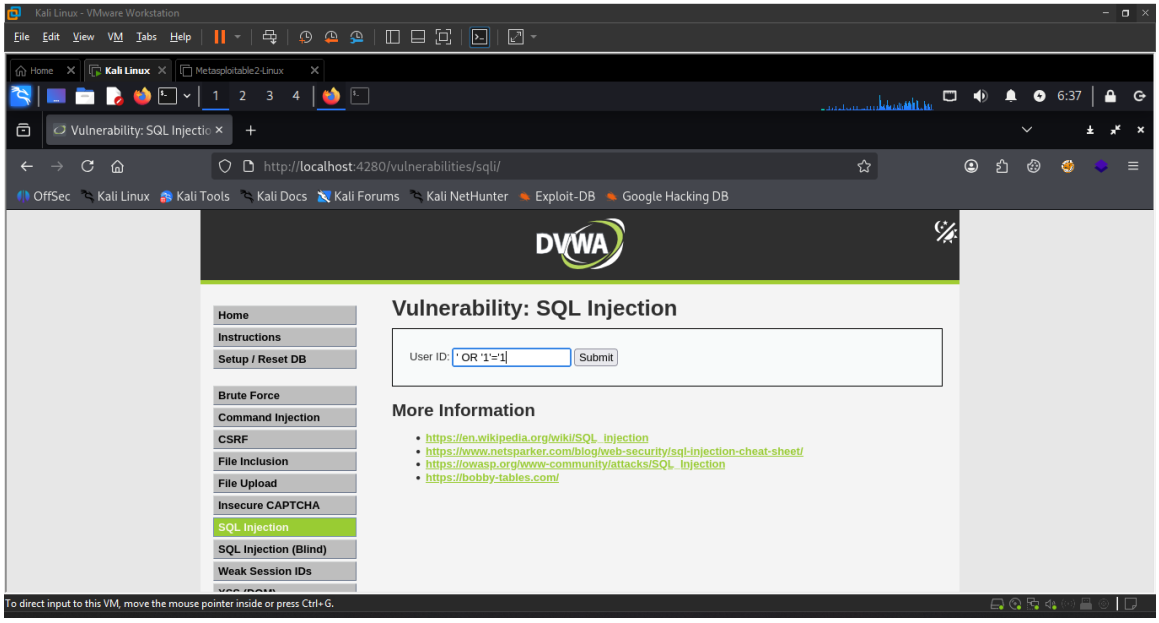


SQL Injection

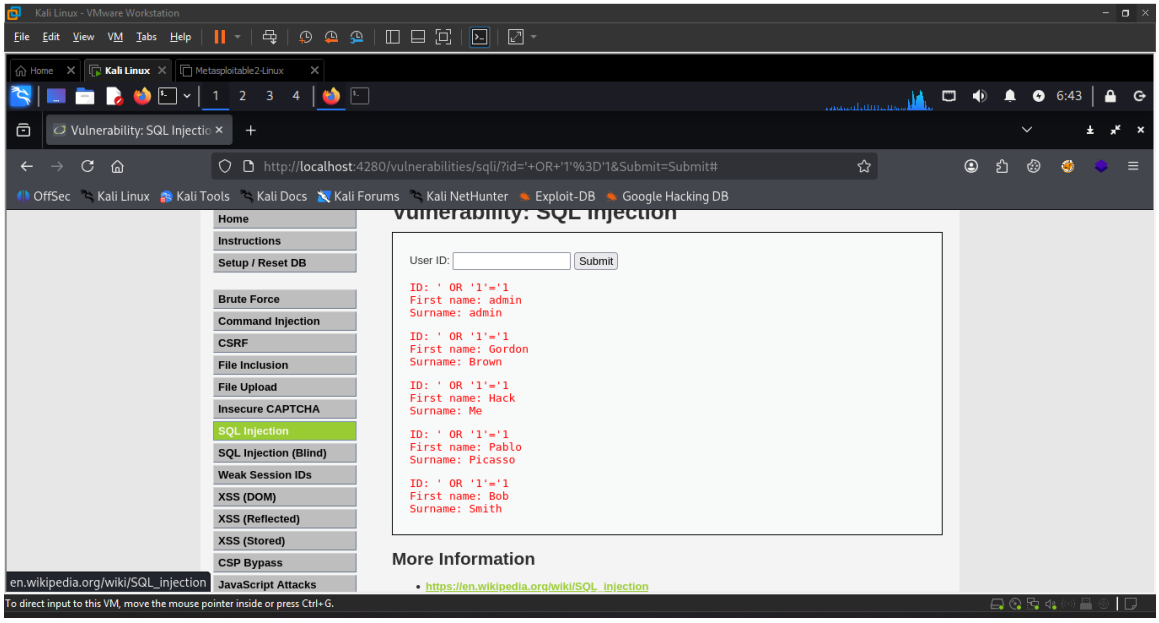
Step 1: To find this vulnerability we need a user input filed like XSS injection attack. Before we do this, we must find the appropriate SQL service that were used in the development of the web page.



Step 2: We have different types of Database query languages. So, we should find that before proceeding. Here I'm using mySQL. Here "' OR '1'='1 ". This payload refers always '1' which is nothing but 'TRUE'.



Step 3: Submit the query that we want to inject. And reverify whether the query worked or not. If not try different queries.



Here, we found the users Personal details like First Name and Last Name with a simple SQL query from the database, which is very dangerous to users' privacy.

Why Injection Attacks Are Still Prevalent

Injection vulnerabilities persist notably due to legacy applications, improper input validation, and reliance on unsafe coding practices. Attackers exploit subtle weaknesses in how inputs are handled and parsed by interpreters, making injection attacks a critical part of the OWASP Top 10 web application security risks.

Mitigation Strategies

To protect applications from injection attacks, it is essential to:

- Use parameterized queries and prepared statements rather than string concatenation for database queries.
- Thoroughly validate, sanitize, and escape all user inputs.
- Employ proper output encoding especially for data rendered in web pages (to prevent XSS).
- Utilize security frameworks that enforce safe coding practices.
- Regularly test applications using static and dynamic analysis tools to detect injection flaws.

4. Insecure Design

Insecure design refers to situations where security controls or considerations are missing, misapplied, or inadequately planned during the requirements, architecture, or design phase of developing software systems. Unlike insecure implementation errors (bugs, misconfigurations), these issues result from not considering security from the ground up, often manifesting as business logic flaws, insufficient segregation of tenants, and lack of plausibility checks.

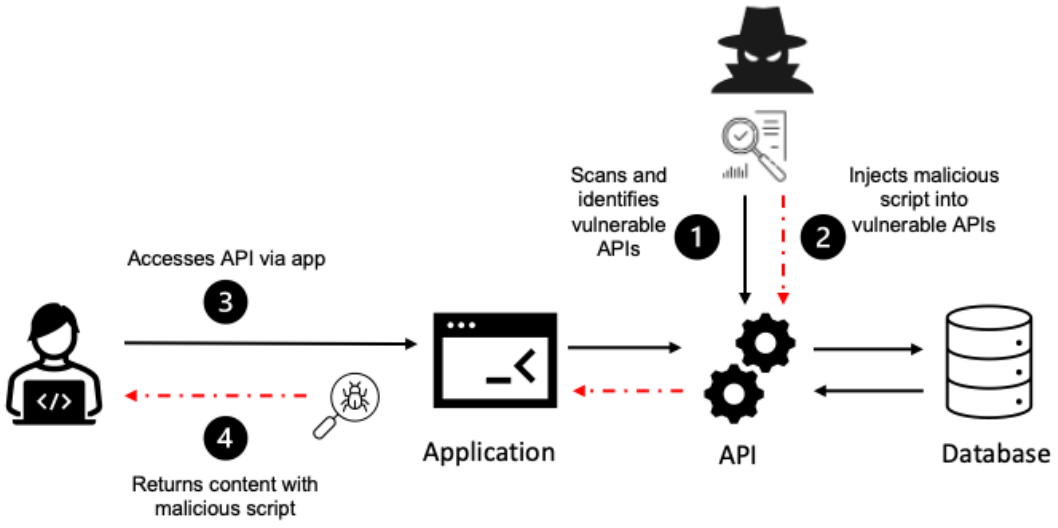
Common Examples

- Lack of input validation or sanitization, allowing injection or XSS attacks.
- Missing or weak authentication and authorization mechanisms.
- Improper session or resource management (e.g., sessions never expire, shared sessions).
- Failure to segregate user data, leading to privacy breaches (multi-tenant mis design).
- Application logic that permits actions out of order or without appropriate restrictions
- Insecure data storage design, such as storing secrets or sensitive data without encryption.

Why Insecure Design Matters

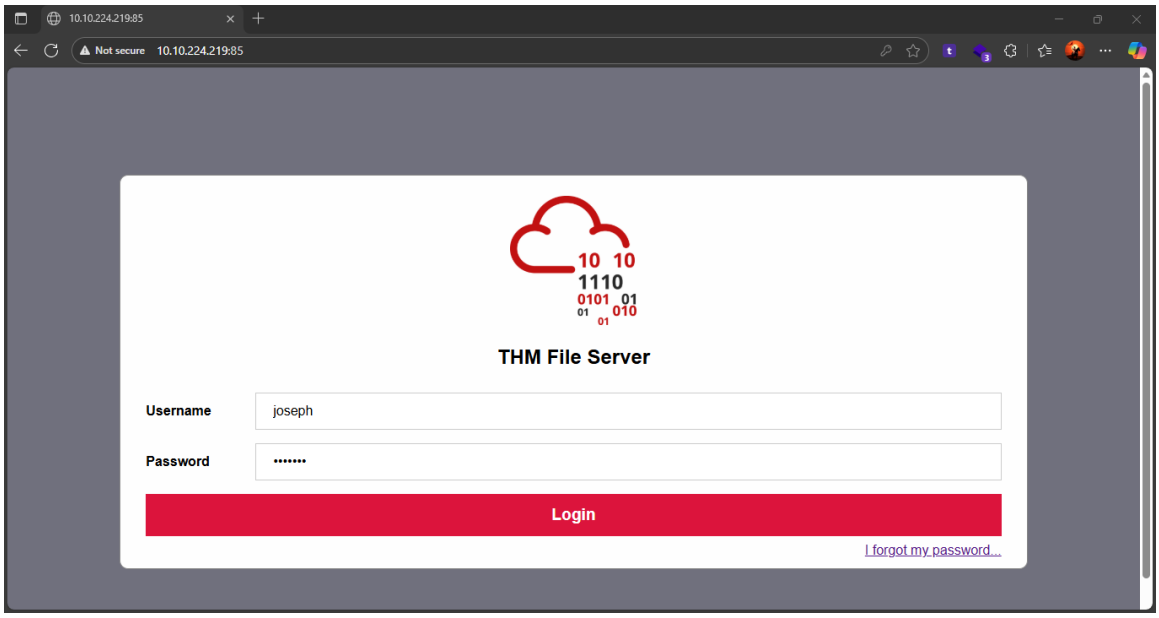
A secure implementation cannot compensate for bad design since missing security controls leave fundamental attack surfaces exposed. Such weaknesses can:

- Expose sensitive data to unauthorized users
- Allow privilege escalation or horizontal/vertical attacks
- Lead to denial of service or bypass of business rules

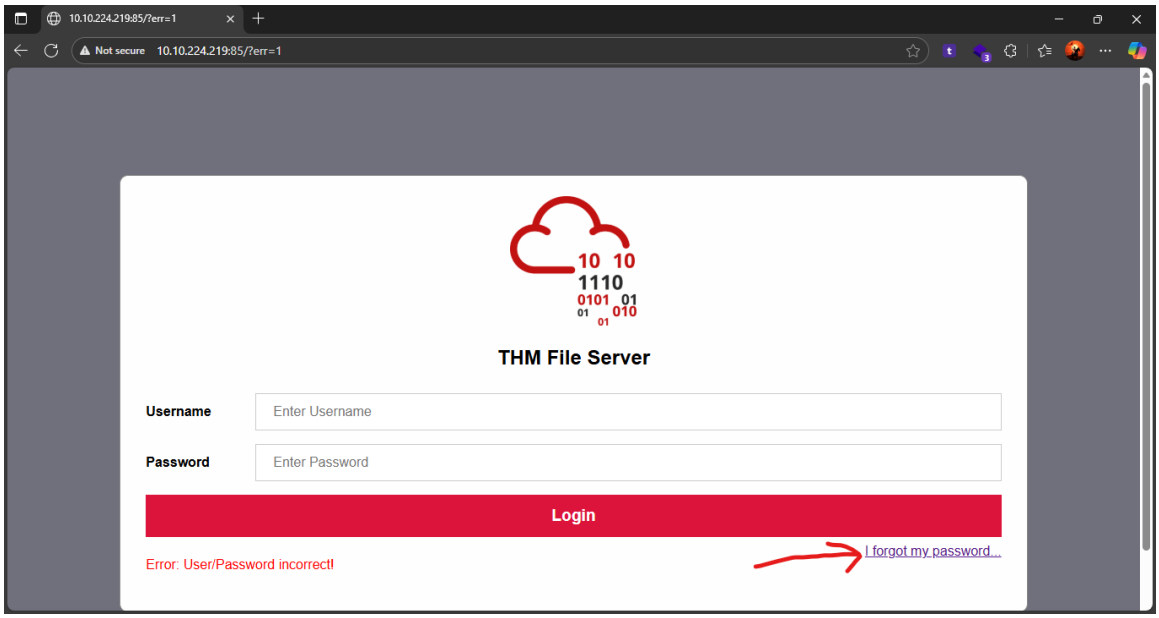


Weak Authorization

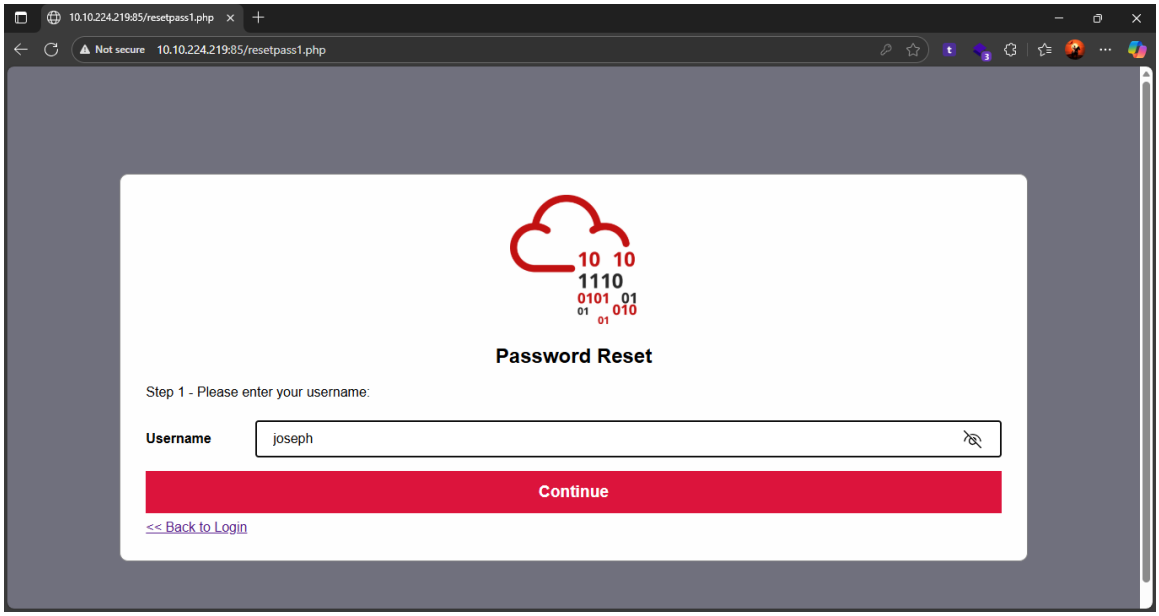
Step 1: We have a web page and don't have valid password for the user "joseph".



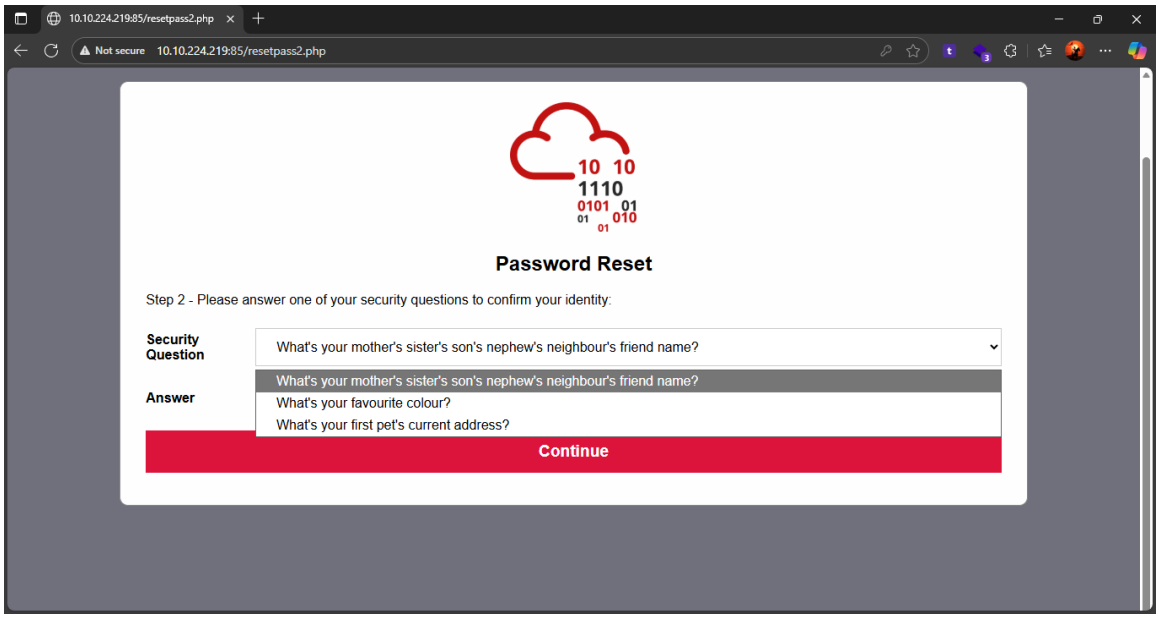
When we try different well-known passwords, we can't get the correct one. Now we try to get reset the password by using "Forgot password" option given in the bottom.



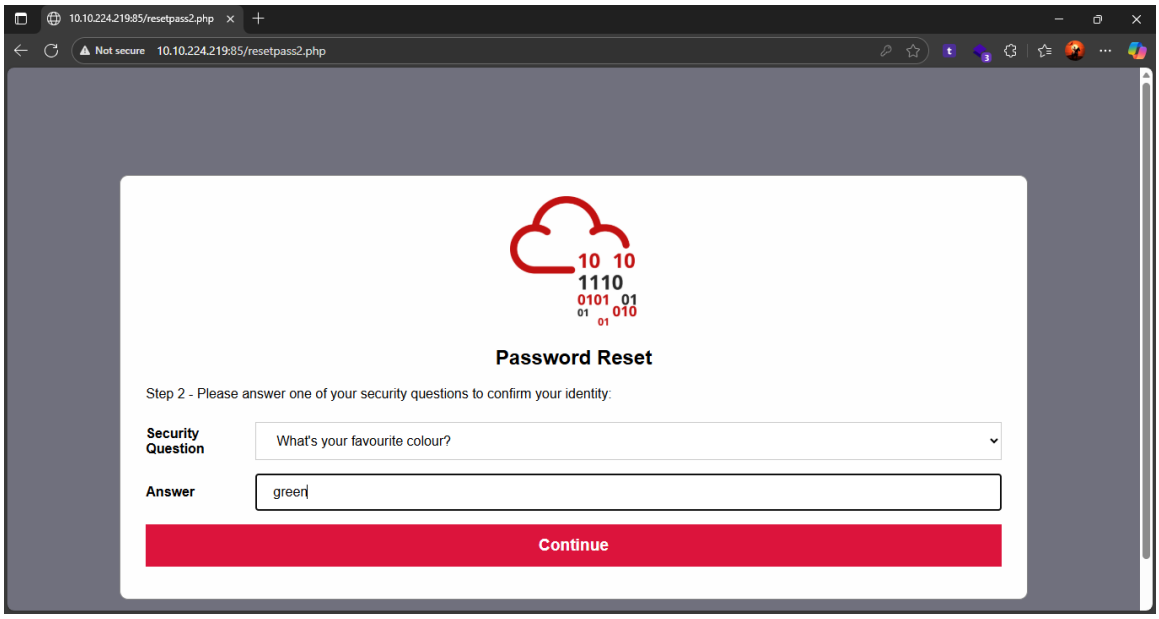
Step 2: In the process of resetting the password for the user “joseph” continue by selecting the user’s name.



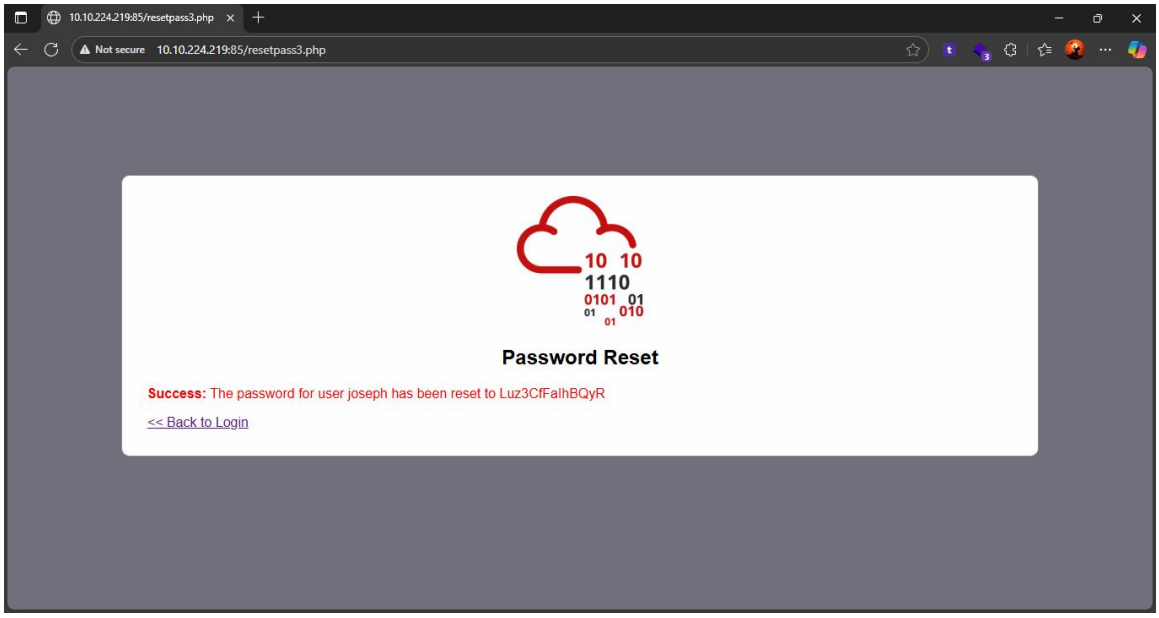
By continuing in this process, we got the authorization process with “Security Question”. For the 1st and 3rd question we can have many options, so we can’t type all the possible names.



Step 3: But for the 2nd question we have finite number of possible values which are enough to bypass the authentication process. And we make that by trying new values and finally found that the answer was “green”.



Then we got the Password



Prevention and Best Practices

OWASP recommends integrating security early and continuously in the development lifecycle:

- Use threat modeling during design phases to assess and mitigate potential flows and attacks.
- Maintain a library of secure design patterns and reference architectures.
- Apply the principle of least privilege, ensuring users/components get only minimal necessary permissions.
- Implement defense in depth—multiple layers of security across architectural tiers.
- Segregate duties, use plausibility checks, fail securely, and separate tenant data by design.
- Integrate plausible security controls into user stories and test all critical business logic against both use-cases and misuse-cases.
- Regularly validate and revisit threat models and design patterns throughout development.

Adopting a secure development lifecycle, with input from AppSec experts, is critical to minimizing insecure design risks. Threat modeling should guide requirements, architecture, and refinement sessions, making security a foundational element of every application component.

In essence, insecure design in OWASP is about the absence or weakness of controls because the threat was never considered or mitigated during planning. Secure design demands proactive, continuous attention—right from requirements gathering to final deployment and beyond.

5. Security Misconfiguration

Security misconfiguration is a critical vulnerability category in the OWASP Top 10 list that occurs when security settings are not properly defined, maintained, or enforced across an application, its components, or infrastructure. This vulnerability can exist at any level of the application stack, including network services, servers, platforms, frameworks, databases, and cloud services.

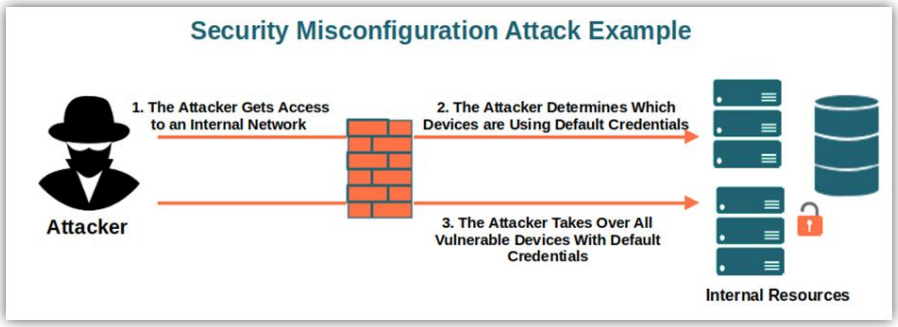
Description and Causes

Security misconfiguration arises due to missing or improper security hardening. Common scenarios include the following:

- Use of default configurations, accounts, and passwords that remain unchanged.
- Enabling unnecessary features, ports, services, or privileges that increase the attack surface.
- Inadequate or improperly set permissions on files, directories, cloud storage, or APIs.
- Exposure of sensitive information through overly informative error messages or stack traces.
- Failure to apply security patches, updates, or disable insecure legacy features.
- Lack of secure communication protocols such as TLS/SSL, or misconfigured headers and security directives.
- Misconfigured session management or weak encryption settings.
- Default settings in application servers, application frameworks, or libraries not set to secure values.

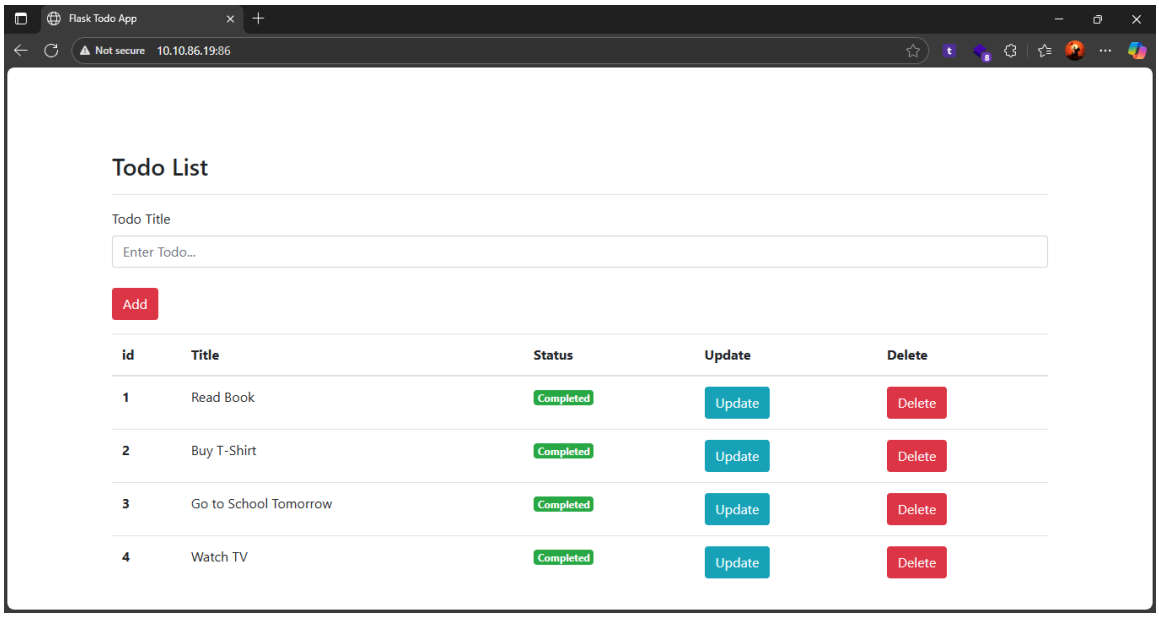
Examples of Security Misconfigurations

- Leaving administration consoles accessible with default credentials.
- Directory listing enabled on servers, allowing attackers to browse and download sensitive files.
- Debugging features left enabled in production releases, exposing internal application details.
- Cloud storage buckets or services set with permissions open to the internet or unauthorized users.
- Storing sensitive data like passwords or API keys in clear text or inadequately encrypted.

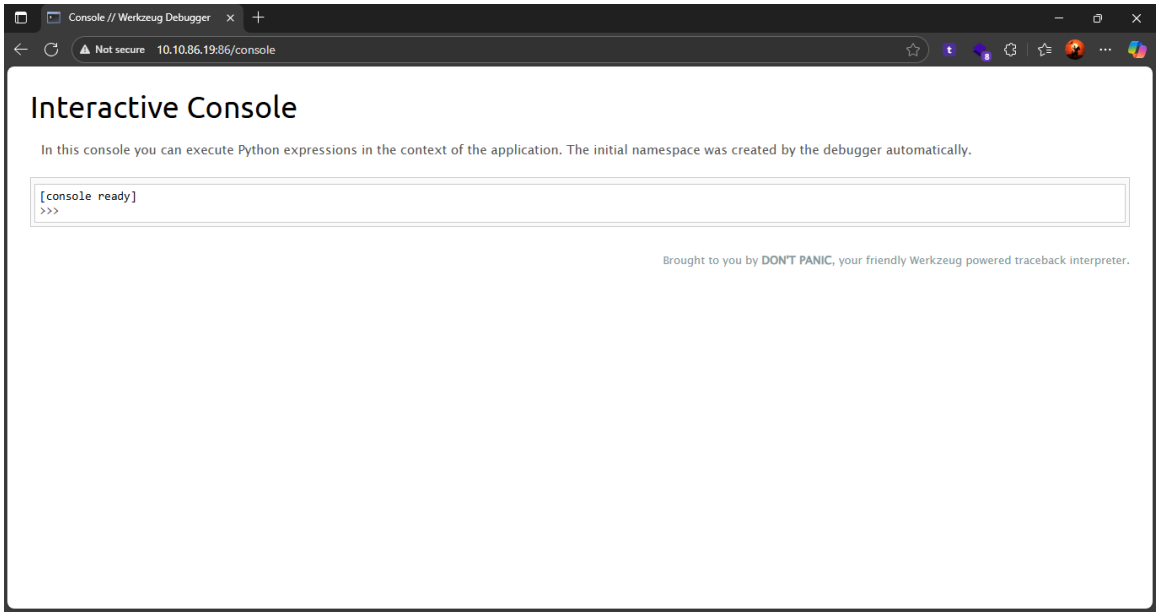


Security Misconfiguration Example

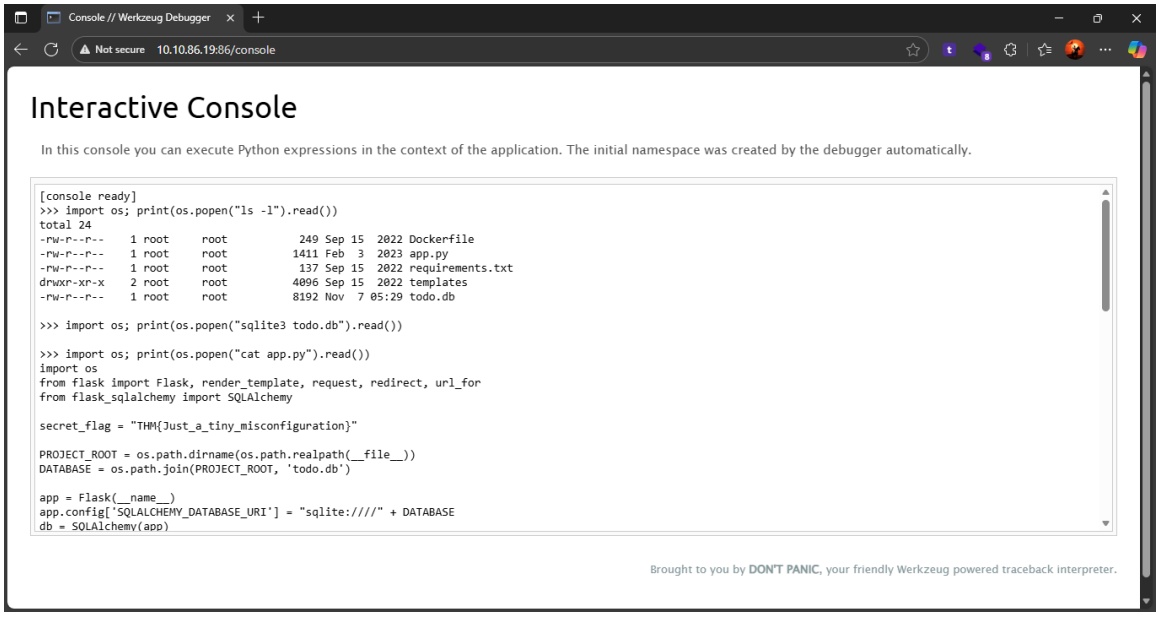
Step 1: Let's say we've a website built with Python and Flask.



Step 2: In that website we found a page "/console" which we can directly operate without any authentication. Here we get a Python interpreter that connected to the webserver. From here we can able to access the webserver with ease.



Step 3: With a simple python script “import os; print(os.popen("COMMAND").read())” we can directly communicate with the Operating System of the Web server.



And we can retrieve the sensitive data from the Web server.

Impact

Security misconfiguration can lead to unauthorized access to sensitive data, full system compromise, data breaches, and exploitation of known vulnerabilities in outdated components. The risk is heightened in complex systems where consistent and repeatable secure configurations are not enforced or automated.

Prevention Recommendations

- Implement repeatable and automated security configuration and hardening processes across all environments.
- Disable or remove unused features, services, and default accounts.
- Apply the principle of least privilege to permissions and access controls.
- Regularly patch and update all software components and frameworks.
- Ensure secure communication protocols are enforced and security headers are correctly set.
- Limit the amount of information revealed in error messages or server responses.
- Review and audit cloud service permissions and configurations to minimize exposed data.
- Segment the application architecture to isolate critical components and reduce attack vectors.

6. Vulnerable & Outdated Components

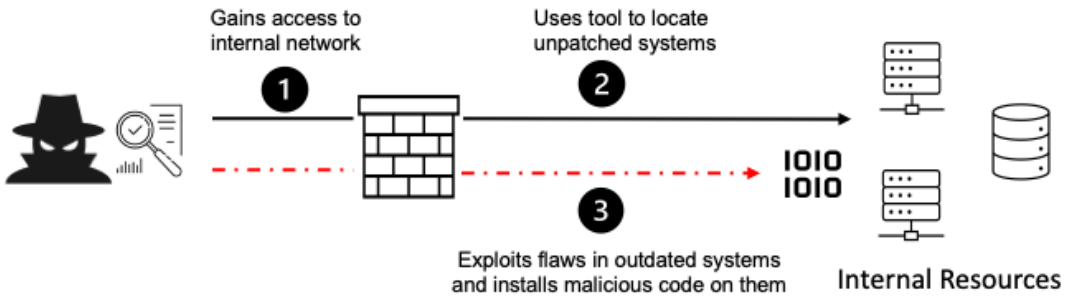
Vulnerable and outdated components are third-party software libraries, frameworks, or dependencies integrated into web applications that have known security vulnerabilities or are no longer maintained with security patches.

Description:

- These components include client-side and server-side software elements, such as open-source libraries (Java, Python, PHP, JavaScript), middleware, APIs, plugins, and entire frameworks.
- They become vulnerable when their versions are unknown, older, deprecated, or no longer receive updates or security patches from developers.
- Examples are unpatched libraries susceptible to cross-site scripting (XSS), SQL injection, remote code execution (RCE), or more complex attacks like server-side request forgery (SSRF).
- Such components may also harbor misconfigurations that increase risk.

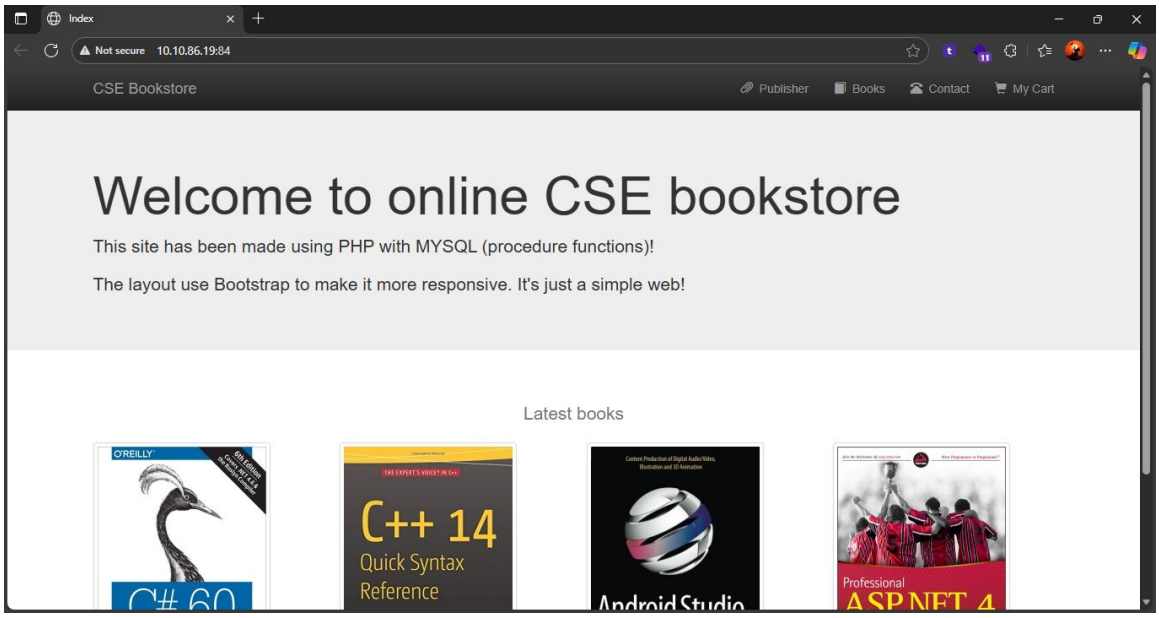
Risks and Exploits

- Attackers frequently scan for known vulnerable versions with published Common Vulnerabilities and Exposures (CVEs), exploiting them to gain unauthorized access, perform remote code execution, or breach data confidentiality.
- Exploitation can lead to data breaches, financial loss, and reputation damage.
- Notoriously, vulnerabilities in Apache Struts led to the Equifax data breach of 2017, and the Log4j "Log4Shell" exploit is another significant example.
- Other risks include cascading system failures, bypassing authentication, and introducing persistent threats.

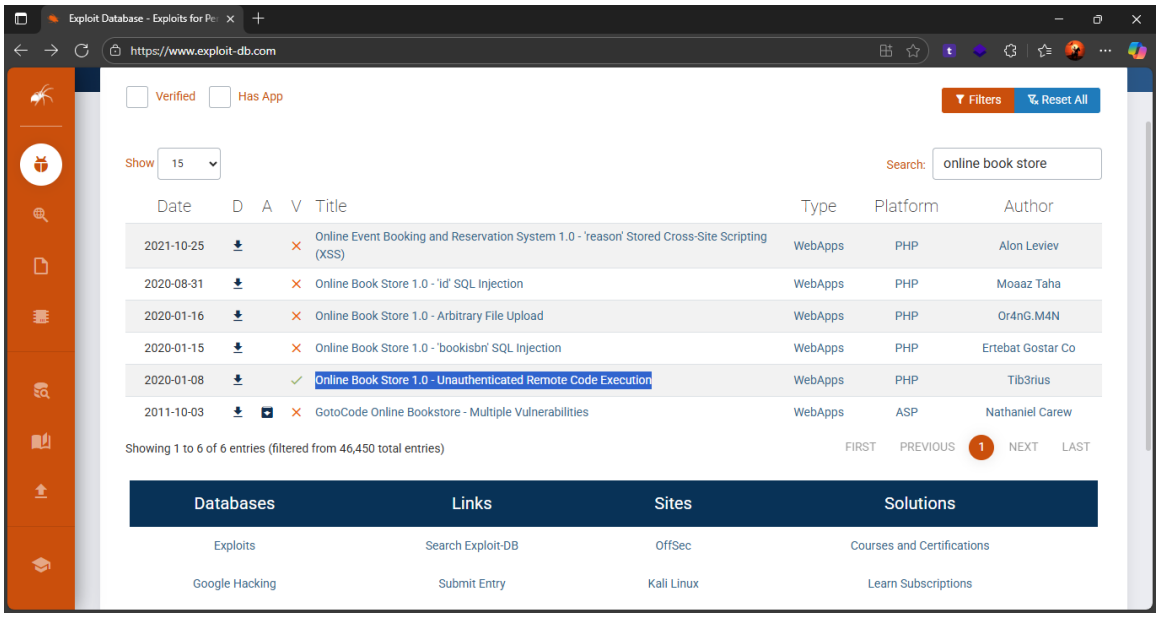


Example

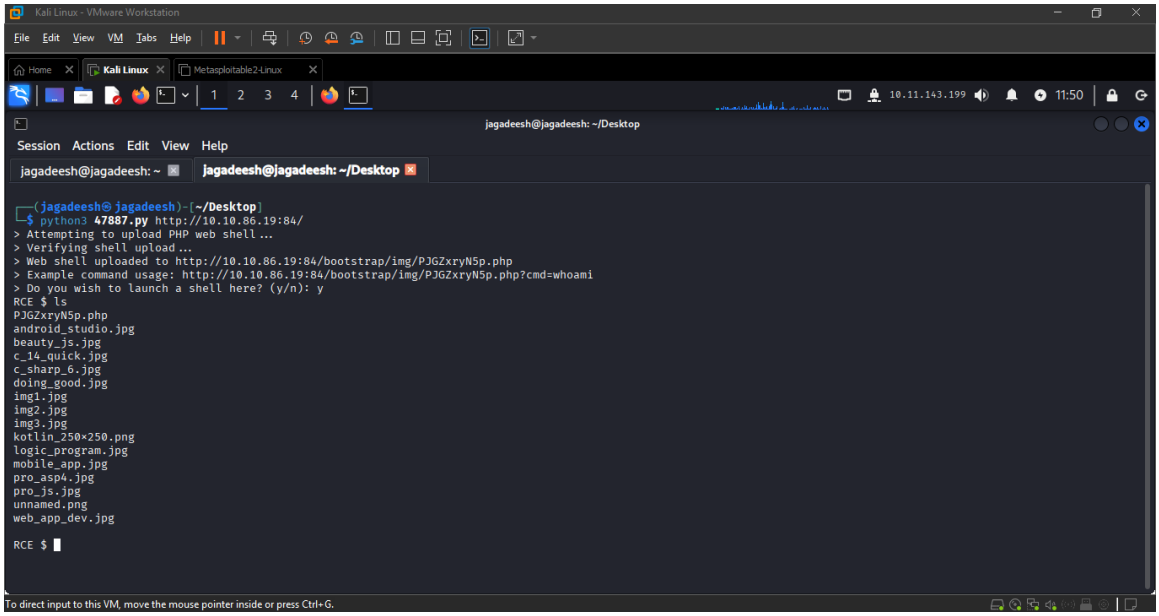
Step 1: Let's we have a website that built on pre build dependency called "CSE Bookstore".



Step 2: Goto https://www.exploit-db.com/ and search for "Online book store". Then we found an RCE exploit.



Step 3: By executing that exploit that exploit against the target webpage, we get a Shell that connected to that web server. Here we can fetch the all info of the web server.



Prevention and Mitigation

- Maintain an up-to-date inventory of all components and their versions.
- Employ automated tools for continuous vulnerability scanning and dynamic security testing.
- Prioritize patching and updating components based on real exploit risks.
- Use secure development practices, including evaluating third-party components before integration.
- Adhere to compliance requirements and policies ensuring timely updates and patch management.

Vulnerable and outdated components represent a significant supply chain risk in modern software development that requires awareness, constant monitoring, and proactive management to prevent serious security incidents.

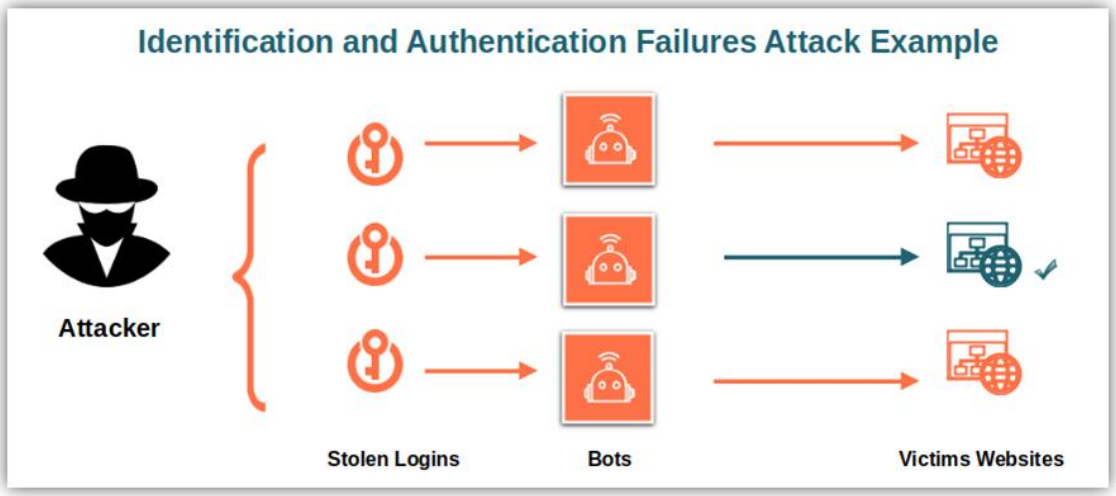
7. Identification and Authentication Failure

Identification and Authentication Failures, formerly known as Broken Authentication risks focusing on vulnerabilities related to user identity verification processes that allow unauthorized access. These failures occur when an application improperly manages user identification or authentication, allowing to bypass login mechanisms or steal credentials.

Key aspects include:

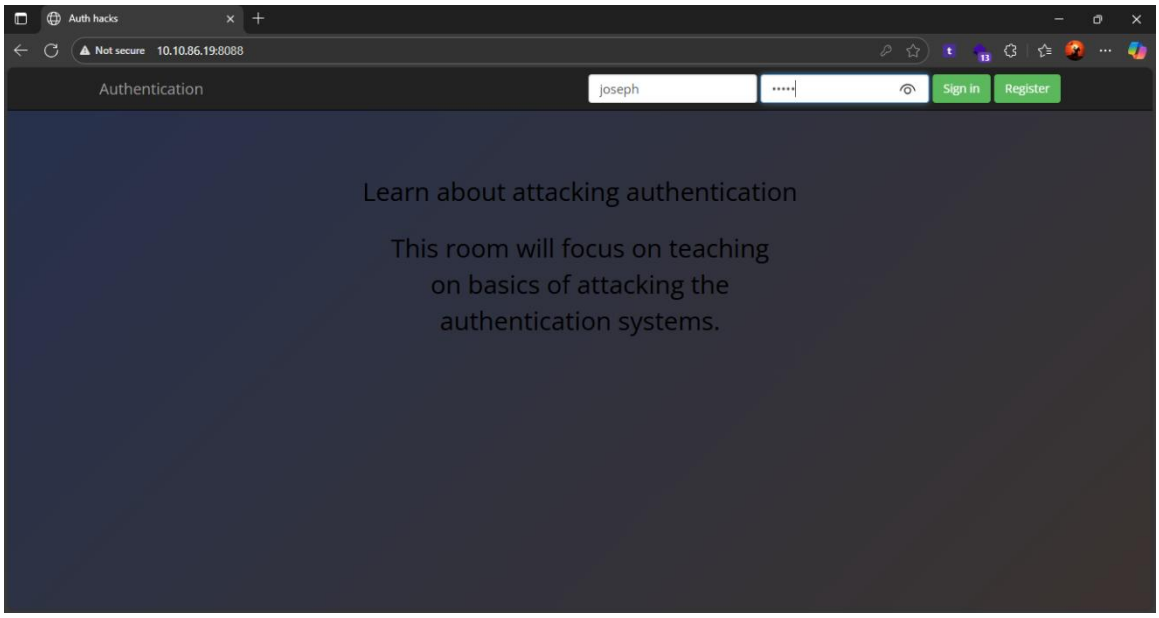
- Weak, reused, or guessable passwords that make it easy for attackers to crack accounts.
- Absence or ineffective implementation of multi-factor authentication (MFA), which is an essential extra security layer.
- Vulnerabilities in password recovery processes that use weak or easily exploitable methods such as knowledge-based answers.
- Exposing session identifiers insecurely, such as in URLs, or failing to invalidate sessions properly after logout or inactivity, leading to session hijacking.
- Lack of rate limiting on authentication attempts, enabling brute-force or credential stuffing attacks.
- Use of plain text or weakly hashed password storage and ineffective credential recovery mechanisms.
- Failure to rotate or expire sessions or reuse session identifiers after login.

Typical attack scenarios include credential stuffing with lists of known credentials, brute-force attacks, session fixation, and exploitation of weak password reset URLs or tokens.

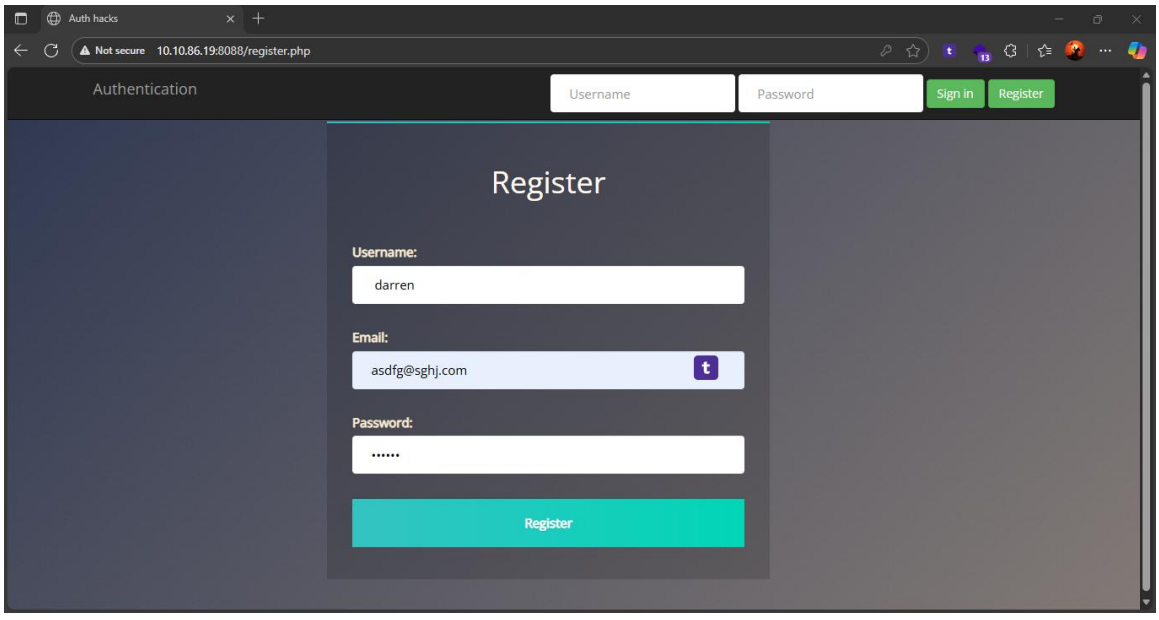


Identification Failure Example

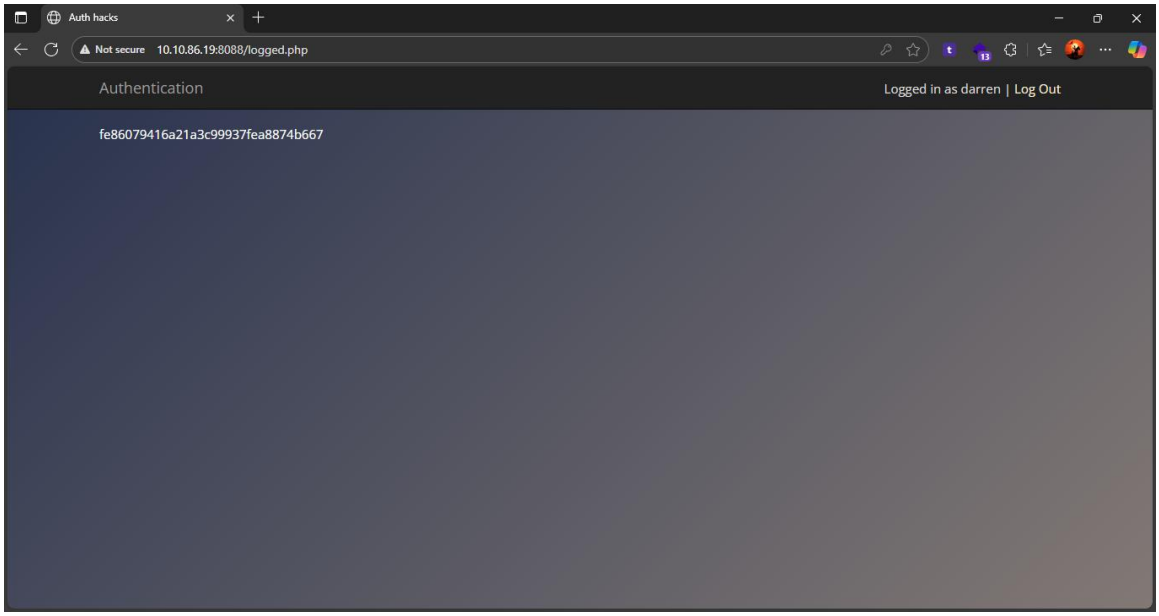
Step 1: Let's try to login to an account named "darren" with default password. But failed to login because of incorrect password.



Step 2 : Then go to the registration page and create a user with name " darren" but with a space before.



Step 3: Now, login with our new credentials we can directly login with the “darren” account without any authorization.



Mitigation involves:

- Implementing strong, unique password policies and encouraging or enforcing the use of MFA.
- Properly securing session tokens and invalidating them after logout or timeout.
- Avoiding password rotation policies that encourage weak passwords per modern guidance like NIST 800-63.
- Enforcing rate limiting on login attempts.
- Ensuring secure storage of credentials using strong cryptographic hashing.
- Using secure password recovery methods without predictable or reusable tokens.

Identification and Authentication Failures lead to unauthorized access, data breaches, and compliance violations that can have severe financial and reputational consequences for organizations.

8. Software and Data Integrity Failure

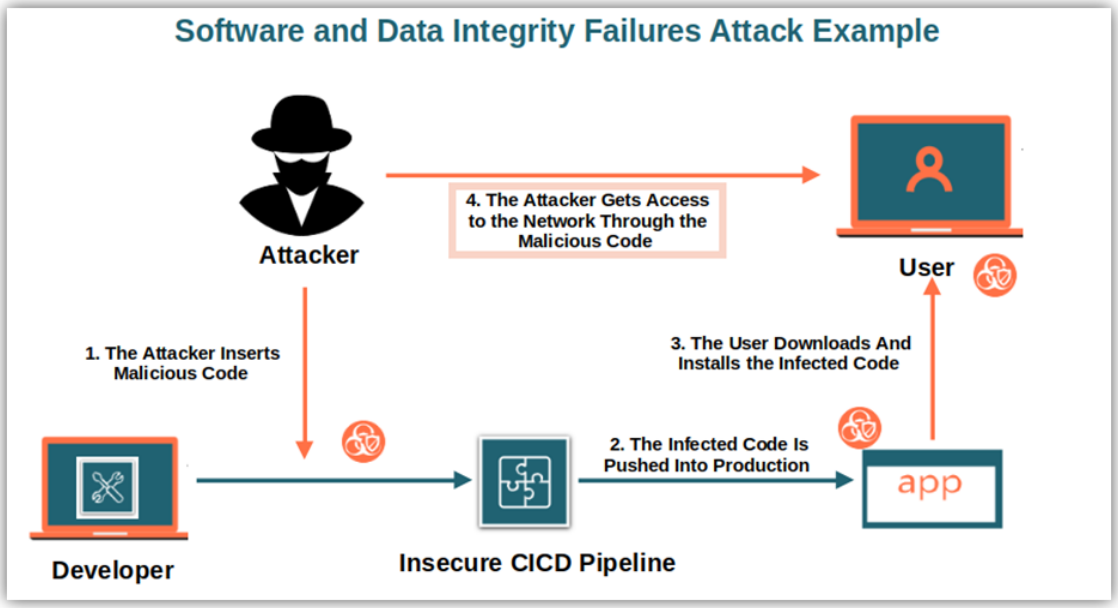
Security and data integrity failures in the context of the OWASP Top 10 refer to vulnerabilities where applications fail to properly protect critical software solutions, processes, and data against unauthorized changes, tampering, or supply chain threats.

What Are Security and Data Integrity Failures?

Software and data integrity failures occur when systems, processes, or applications inadvertently trust code, data, or updates without proper validation, authenticity checks, or integrity protection. These failures allow attackers to manipulate application behavior, introduce malicious code, or compromise sensitive operations by modifying code, data, or the supply chain itself.

Common Causes and Scenarios

- Applications or systems download code or updates from untrusted sources without verifying authenticity (e.g., missing digital signatures).
- Unprotected or misconfigured Continuous Integration/Continuous Deployment (CI/CD) pipelines allowing unauthorized access or tampering.
- Using open-source or third-party dependencies from unverified repositories.
- Deserialization of untrusted data, resulting in code injection or unauthorized actions.
- Acceptance and processing of data from untrusted sources without sufficient checks.
- Auto-update mechanisms that do not validate update integrity.



Software Integrity Example

Suppose you have a website that uses third-party libraries that are stored in some external servers that are out of your control. While this may sound a bit strange, this is actually a somewhat common practice.

Take as an example jQuery, a commonly used JavaScript library. If you want, you can include jQuery in your website directly from their servers without actually downloading it by including the following line in the HTML code of your website:

<script src="https://code.jquery.com/jquery-3.6.1.min.js"></script>

When a user navigates to your website, its browser will read its HTML code and download jQuery from the specified external source.

The problem is that if an attacker somehow hacks into the jQuery official repository, they could change the contents of https://code.jquery.com/jquery-3.6.1.min.js to inject malicious code. As a result, anyone visiting your website would now pull the malicious code and execute it into their browsers unknowingly. This is a software integrity failure as your website makes no checks against the third-party library to see if it has changed.

Modern browsers allow you to specify a hash along the library's URL so that the library code is executed only if the hash of the downloaded file matches the expected value. This security mechanism is called Sub resource Integrity (SRI).

The correct way to insert the library in your HTML code would be to use SRI and include an integrity hash so that if somehow an attacker is able to modify the library, any client navigating through your website won't execute the modified version. Here's how that should look in HTML:

<script src= “https://code.jquery.com/jquery-3.6.1.min.js”
integrity=sha256o88AwQnZB+VDvE9tvIXrMQaPlFFSUTR+nldQm1LuPXQ="
crossorigin="anonymous"></script>

Consequences of Security and Data Integrity Failures

- Attackers can modify or delete data, compromising confidentiality, integrity, and availability of information.
- Unauthorized code or malware can be distributed under the guise of legitimate updates.
- Financial, reputational, and operational damage for organizations.

Mitigation Strategies

- Always use digital signatures and cryptographic hashes to verify code and data sources.
- Secure CI/CD pipelines with strong authentication and authorization.
- Regularly audit dependencies and restrict the use of untrusted or unverified third-party code.
- Validate and sanitize all input, with particular scrutiny on data deserialization.
- Implement layered security checks and protected trust boundaries within the software lifecycle.

Security and data integrity failures are about more than simple misconfigurations—they involve the full software supply chain, trust boundaries, and the protection of update mechanisms, pipelines, and data exchanges. Vigilance, validation, and continuous monitoring are essential to prevent attackers from exploiting these weaknesses in modern web applications.

9. Security Logging and Monitoring Failures

Security logging and monitoring failures refer to weaknesses in how applications and systems record, observe, and respond to events that could signal malicious activity or breaches. This vulnerability is included in the OWASP Top 10 because effective logging and monitoring are necessary for rapid breach detection, investigation, and recovery. Without them, threats can go unnoticed for long periods, making it nearly impossible to contain attacks or understand their scope.

What Are Logging and Monitoring Failures?

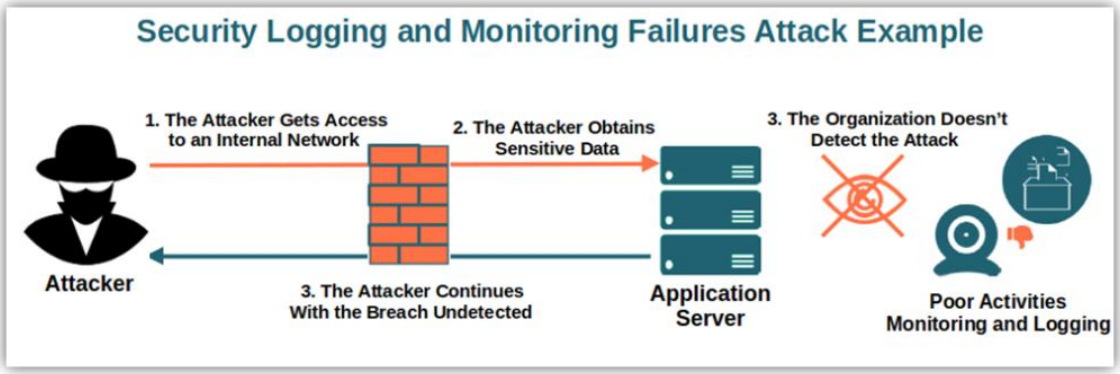
- Logging and monitoring failures happen when important security events (e.g., failed logins, privilege changes, access to sensitive information) are either not captured, stored, or reviewed effectively.
- Common issues include incomplete or absent logs, logs not containing key details (timestamps, user IDs, IP addresses), failing to monitor them in real-time, insecure storage, and not retaining logs long enough for thorough investigations.

Impact of Logging and Monitoring Failures

- Breaches may last longer, cause more damage, and result in regulatory penalties if logging and monitoring are inadequate.
- Organizations may only discover attacks after significant impact, risking both data loss and reputation damage.
- Inadequate monitoring undermines automated security tools, leading to missed errors and vulnerabilities even in protected environments.

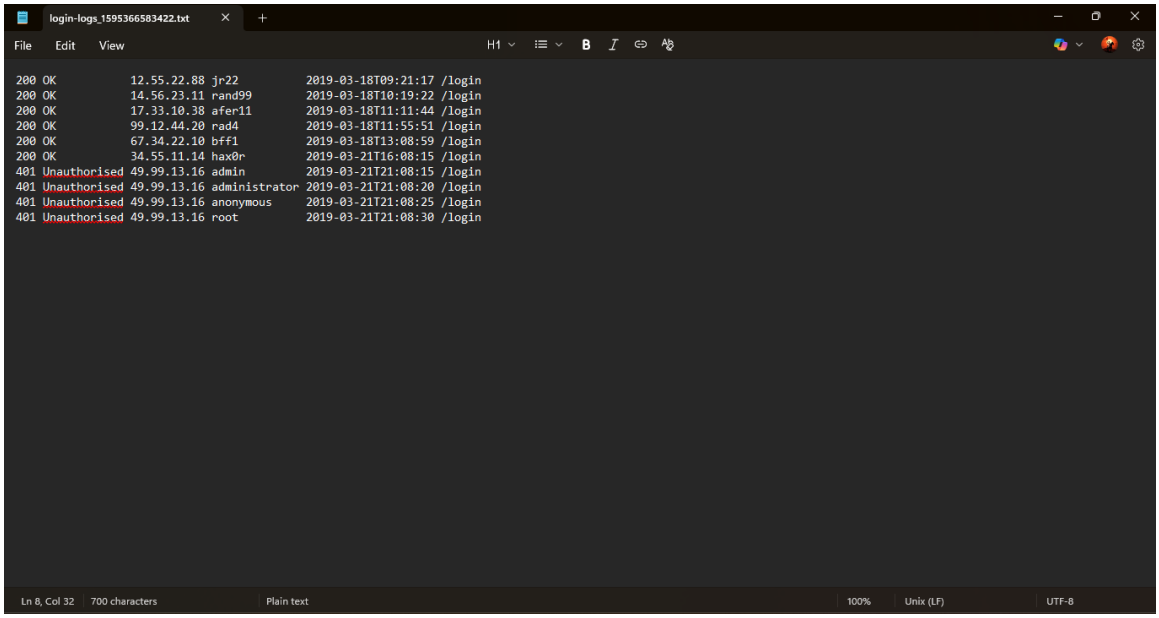
Common Technical Issues

- Logs often lack standardized structure, making it hard to correlate activities across systems during an investigation.
- Many organizations store logs only locally instead of centralizing them, creating analysis blind spots in multi-stage or distributed attacks.



Scenario

Let’s analyze a log file.



In this file the IP address “49.99.13.16” tries to login with multiple users. It may be a Password Spraying attack. But what if the IDS and other SIEM tools failed to detect this attack. And in some cases, the Logs are also be failed to record.

Recommendations and Detection

- Ensure logging and alerting for auditable events such as authentication failures, authorization issues, and high-value transactions.
- Logs should be reviewed regularly, stored securely, retained for long enough to allow for post-incident investigation, and protected against tampering.
- Monitoring should ideally be real-time, correlating events across systems and producing actionable alerts; periodic reviews and analysis should detect suspicious activity.
- Integrate logging with incident response workflows so alerts can trigger investigation and escalation as needed.
- Technical mapping to Common Weakness Enumerations (CWEs): logs not sanitized (CWE-117), missing key actions (CWE-223), logs with sensitive info (CWE-532), incomplete or absent logging mechanisms (CWE-778).

10.Server-Side Request Forgery

SSRF occurs when an application fetches or manipulates a remote resource (often a URL) based on user input without sufficiently validating it. Attackers can supply a malicious address to trick the server into making requests to internal services, privilege-protected endpoints, or even files on the host system. This can lead to exposure or modification of sensitive data and can facilitate lateral movement within protected networks.

How SSRF Attacks Work

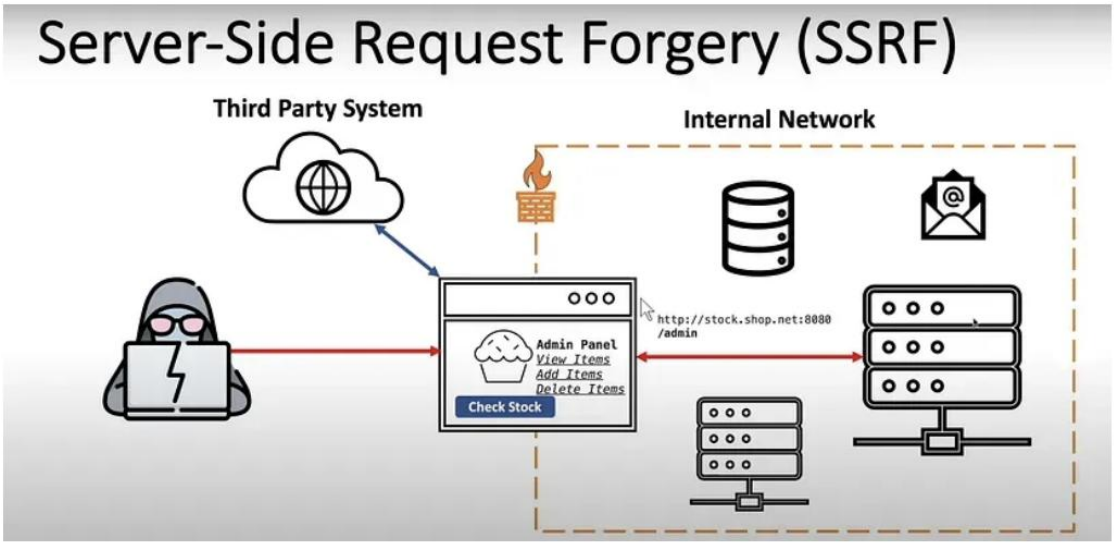
Developers often allow users to provide URLs for tasks such as importing data, loading images, or API calls. If the URL is not properly validated, attackers can:

- Read internal files using paths like file:///etc/passwd
- Access internal web services or databases only available on localhost or private network segments
- Query cloud service metadata endpoints, e.g., http://169.254.169.254/ on AWS, to obtain sensitive credentials
- Initiate internal port scans by sending requests to addresses within the local network, deducing open ports via response times or server feedback.

Security Impact

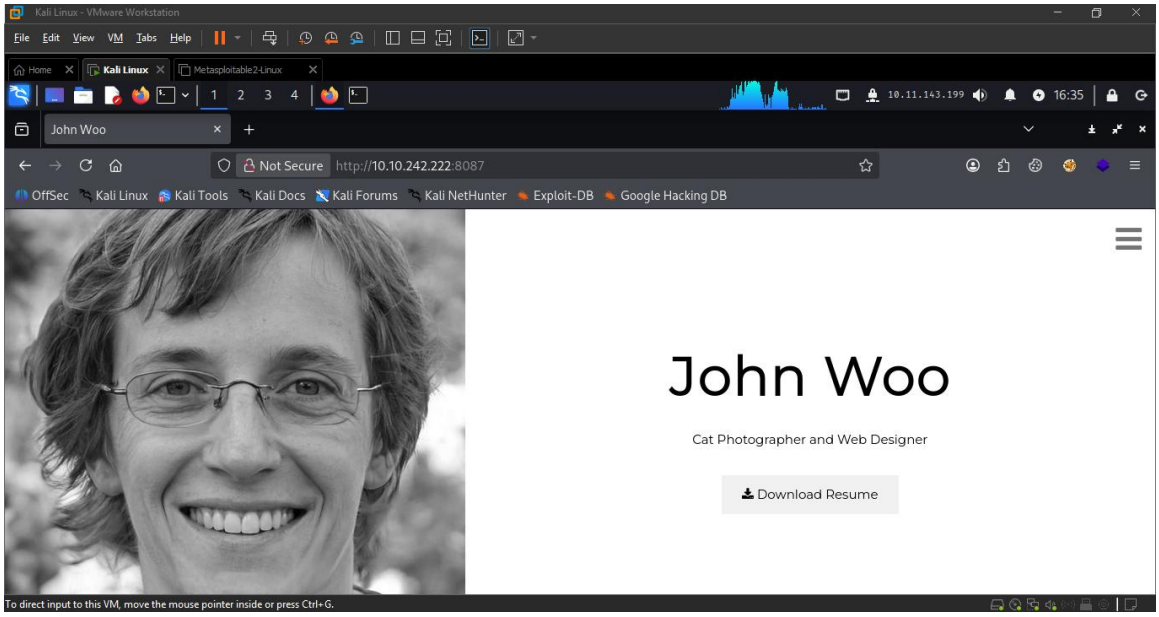
SSRF can:

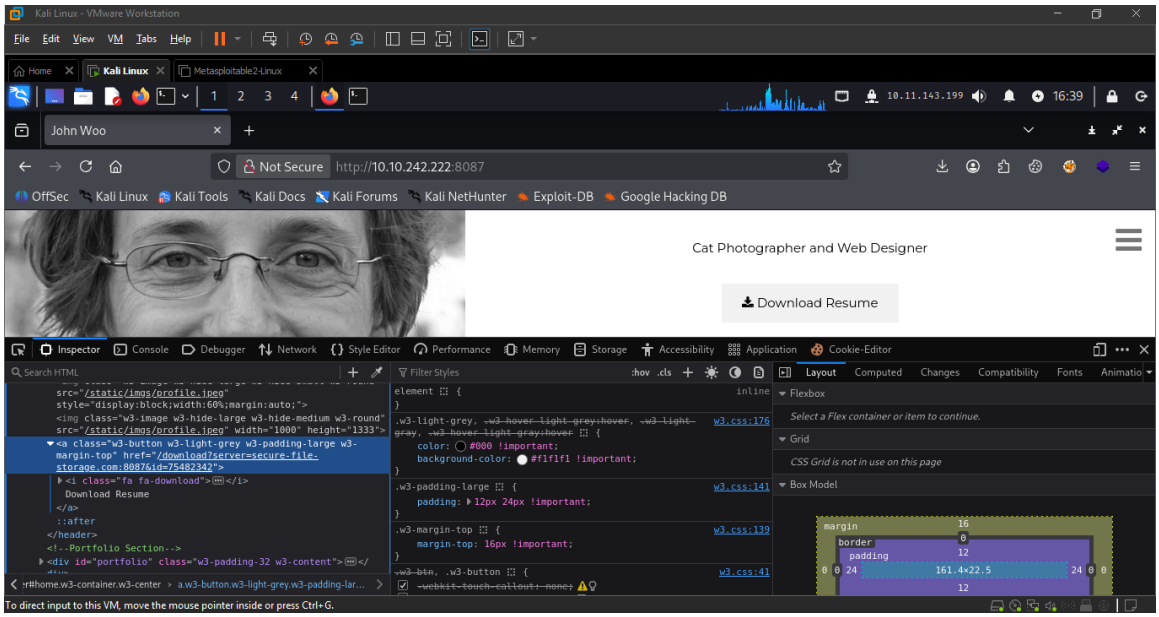
- Bypass firewalls and network segregation by leveraging trusted server privileges
- Steal cloud service credentials or configurations leading to wider compromise
- Perform denial-of-service or remote code execution if the SSRF exposes privileged interfaces
- Access internal admin interfaces, databases, or perform lateral scanning of the local network.



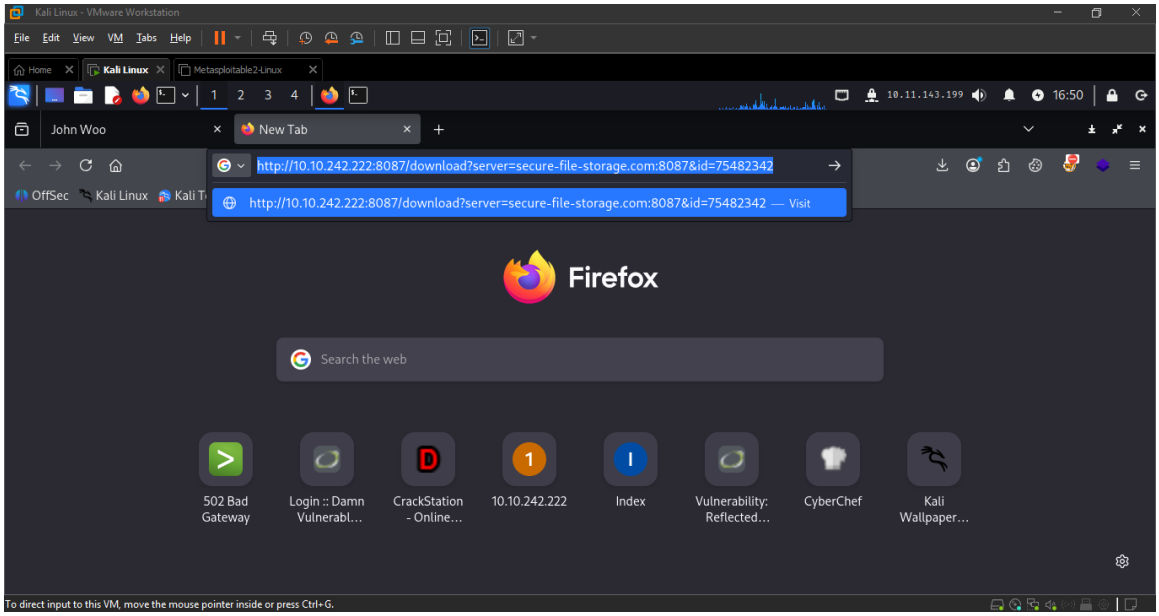
SSRF Example

Step 1: We have a “Download Resume” button that can download a Resume by clicking it. The web browser is redirecting to “http://10.10.242.222:8087//download?server=secure-file-storage.com:8087&id=75482342” to download the resume.

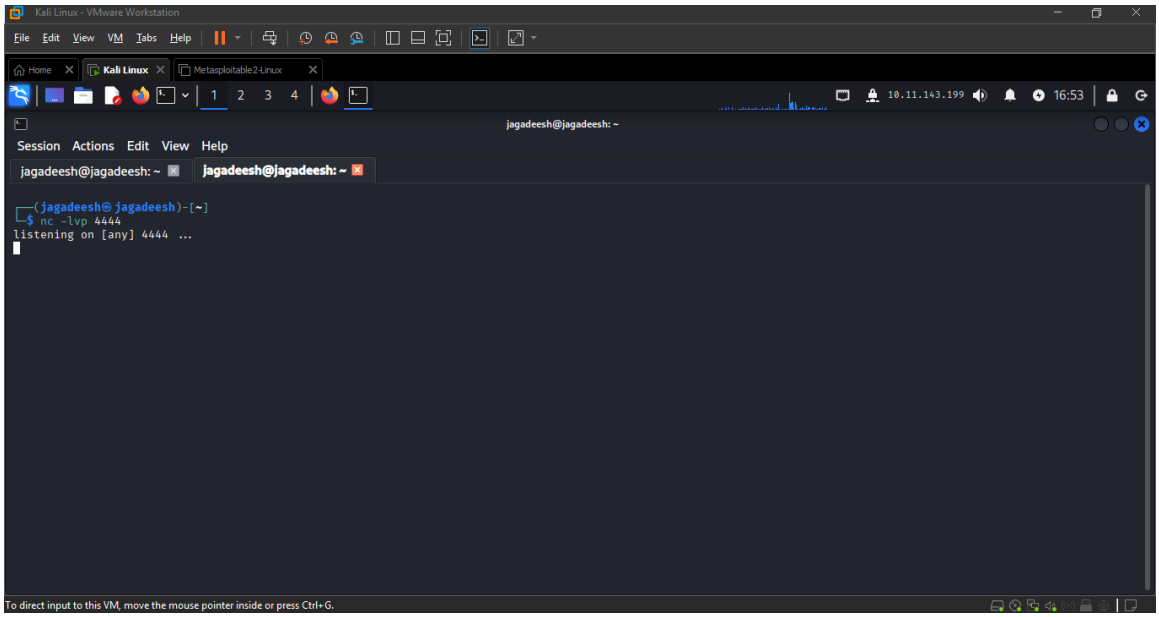




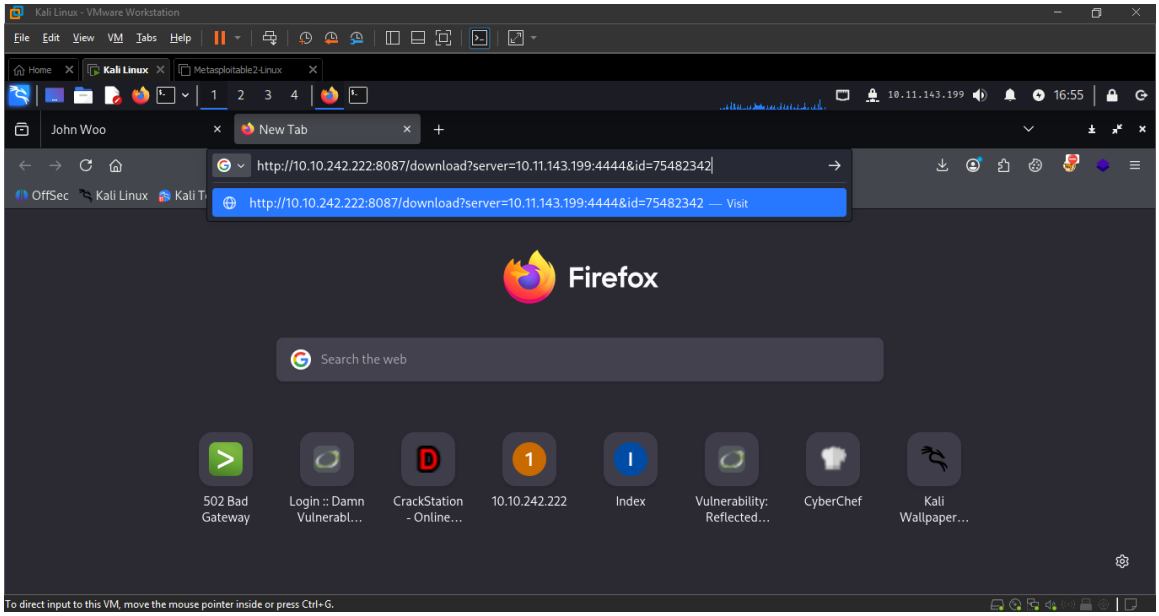
Step2 : Lets copy the link of the redirection link by right click on it. And paste it in another tab.



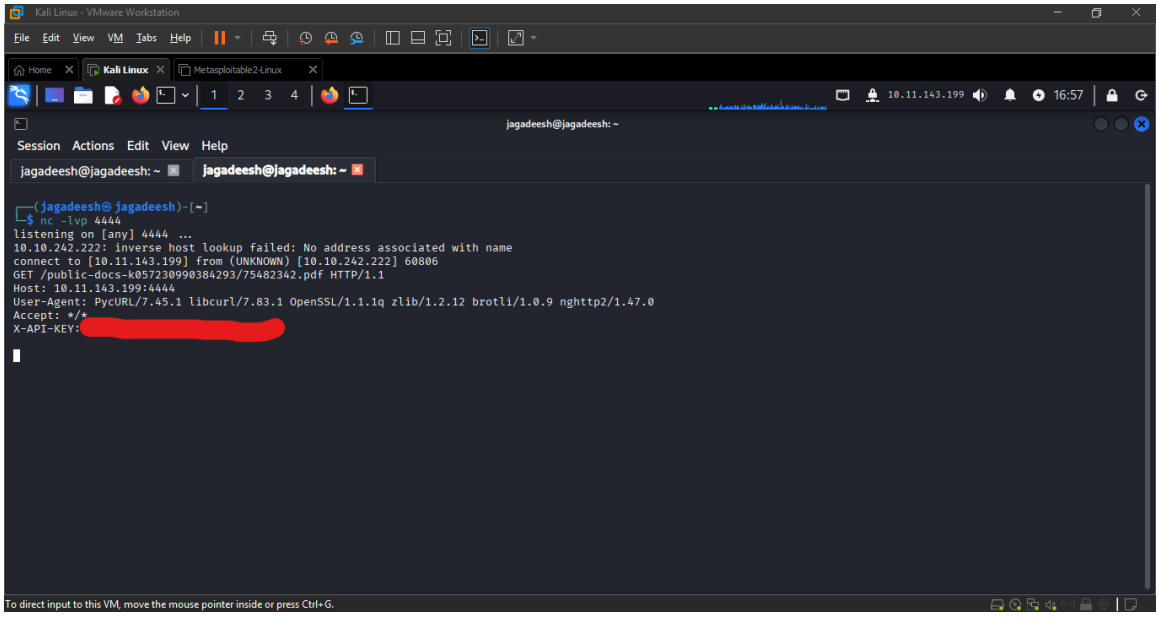
Step 3: Let's start a Listening server on our machine. To do that use "netcat". Go to the terminal and type command "nc -lvp 4444". Then enter, the nc server will start



Step 4: Now replace the redirection url with our IP address and port number that specified in the nc server. In this case it's 4444. Then press enter.



Step 5: Now open the NC server in terminal. There we can find the sensitive info of the web server and API token etc.



Other Example Scenarios

- An image upload feature lets users supply URLs. By submitting a URL like http://localhost/admin, the attacker can force the server to fetch protected pages or internal APIs.
- Entering URLs pointing to metadata endpoints or internal REST interfaces allows attackers to retrieve sensitive configuration or authentication data.
- Using inputs such as http://localhost:28017/ or file:///etc/passwd can expose local services or system files.

OWASP Recommendations for Prevention

- Validate and sanitize all user-supplied inputs for remote resource requests using strict allow lists, schemas, and patterns.
- Disable following redirects and restrict URL schemes and ports to only what is required.
- Prevent the server from accessing localhost or the internal network through direct means.
- Regularly audit server-side code and libraries managing HTTP or file requests to ensure robust controls and safe parsers

SSRF is a key concern in secure web and API development as highlighted by OWASP, demanding rigorous input validation, network segmentation, and safe coding practices to prevent attackers from exploiting this vulnerability and accessing protected internal assets.