Task-3

Hiya Sharma
21BCY10078

Date- 28/8/2023

Vulnerabilities other than top 10 Owasp-

1.Insecure Deserialization-

Insecure deserialization is a security vulnerability that occurs when an application does not properly

validate or sanitize data during the process of deserialization.

Attackers can exploit insecure deserialization vulnerabilities to execute malicious code, perform unauthorized actions, or even gain remote control of a system.

Serialization is the process that converts an object to a format that can later be restored. Deserialization is the opposite process, which takes data from a file, stream or network and rebuilds it into an object. Serialized objects can be structured in text, such as JSON, XML or YAML.

Serialization and deserialization are safe, common processes in web applications. However, an attacker can abuse the deserialization process if it's left insecure. Attackers could, for example, inject hostile serialized objects into a web app, where the victim's computer would initialize deserialization of the hostile data. Attackers could then change the angle of attack, making insecure deserialization the initial entry point to a victim's computer.

How to detect insecure deserialization

It is hard to detect attacks caused by insecure deserialization because the process of deserialization uses common code libraries found in web development. Some ways to identify insecure deserialization include the following:

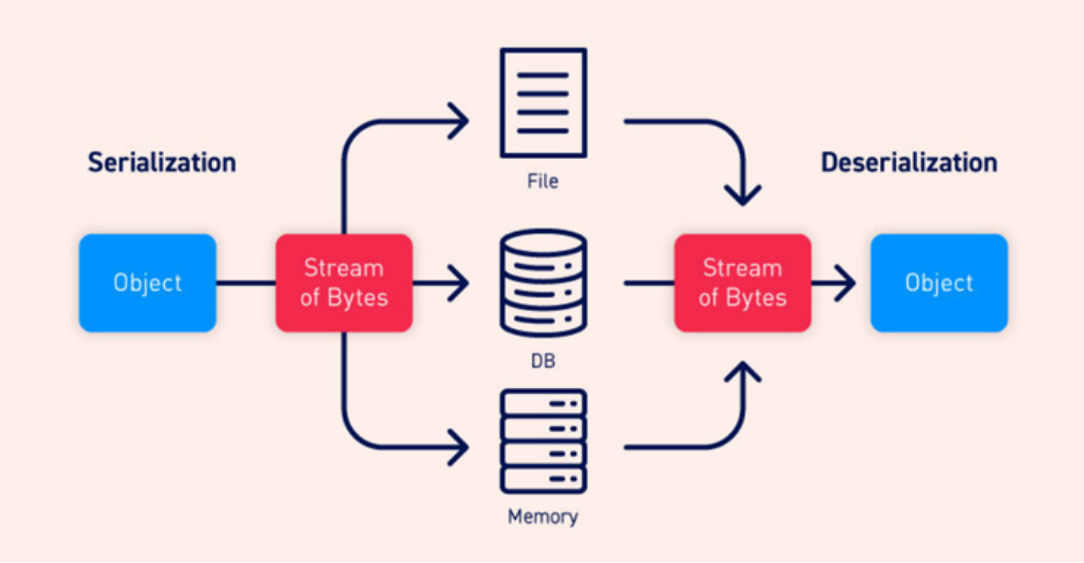
- Check deserializations to see if the data is correctly handled as user input instead of trusted internal data.
- Check deserializations to ensure the data is what it is supposed to be before it is used.

- Use a monitoring tool for deserializations and set notifications for common vulnerable components.
- Run regular security scans.

How to avoid insecure deserialization

Use the following best practices to avoid insecure deserialization:

- Monitor the deserialization process.
- Encrypt serialization processes.
- Do not accept serialized objects from unknown or untrusted sources.
- Run the deserialization code with limited access permissions.
- Use a firewall that detects insecure deserialization.



2.Denial Of Service attack-

A **Denial-of-Service (DoS) attack** is an attack meant to shut down a machine or network, making it inaccessible to its intended users. DoS attacks accomplish this by flooding the target with traffic, or sending it information that triggers a crash. In both instances, the DoS attack deprives legitimate users (i.e. employees, members, or account holders) of the service or resource they expected.

Victims of DoS attacks often target web servers of high-profile organizations such as banking, commerce, and media companies, or government and trade organizations. Though DoS attacks do not typically result in the theft or loss of significant information or other assets, they can cost the victim a great deal of time and money to handle.

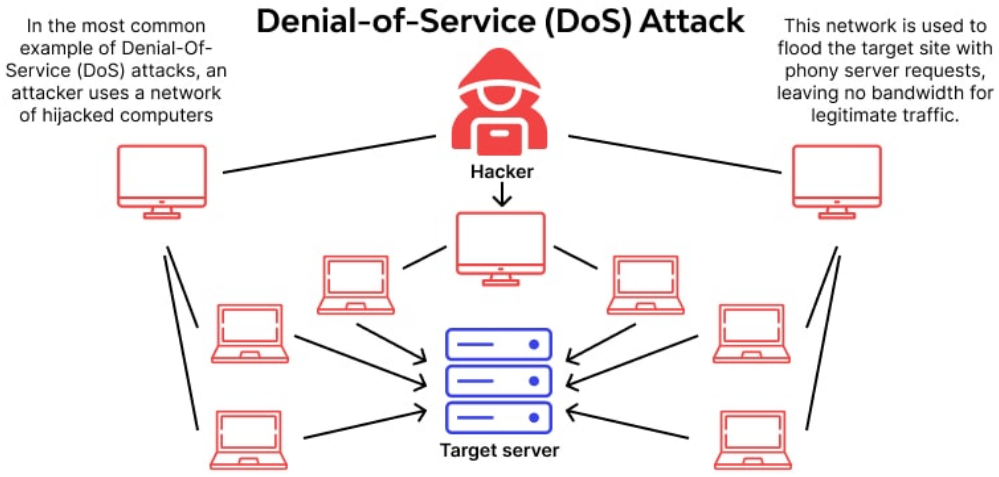
How to detect Denial of Service attack-

- Traffic Analysis:
  - Monitor incoming network traffic for unusual patterns or spikes.
  - Use network monitoring tools and intrusion detection systems (IDS).
- Rate Limiting and Thresholds:
  - Implement limits on requests or connections.
  - Configure alerts for exceeding predefined thresholds.
- Anomaly Detection:
  - Utilize systems that establish baselines for normal network behavior.
  - Detect deviations from these baselines as potential attacks.
- Resource Monitoring:
  - Monitor server resource utilization (CPU, memory, bandwidth).
  - Set up alerts for high resource usage.
- Application Layer Monitoring:
  - Analyze logs for unusual activities.
  - Employ Web Application Firewalls (WAFs) for filtering.
- Traffic Filtering and DDoS Mitigation Services:
  - Use CDNs or specialized DDoS protection providers.
  - Filter out malicious traffic before reaching your infrastructure.
- Packet Inspection and Flow Analysis:

    - Inspect network packets and analyze flow data.
    - Identify traffic patterns associated with attacks.
- Behavioral Analysis:
    - Employ behavioral analysis to detect abnormal traffic behavior.
    - Identify patterns inconsistent with normal use.
- Regular Security Audits and Testing:
    - Conduct security audits and penetration testing.
    - Identify vulnerabilities that may be exploited.
- Incident Response Plan:
    - Develop a response plan outlining actions during a DoS attack.
    - Ensure a coordinated and effective response.

How to avoid Denial of service attack

- Use firewalls, load balancers, and DDoS mitigation services.
- Implement rate limiting and traffic shaping.
- Secure web applications with WAFs and input validation.
- Maintain redundancy and failover systems.
- Educate staff, update systems, and employ intrusion detection.
- Develop an incident response plan and collaborate with ISPs.



3. XML External Entity (XXE) Attack:

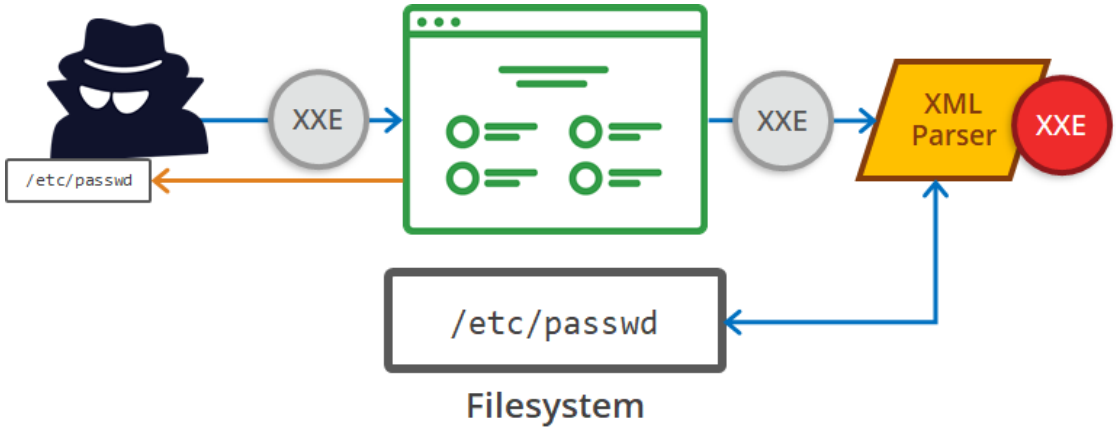
XML External Entity (XXE) Attack is a type of security vulnerability that targets applications or systems parsing XML data, potentially leading to data exposure, server-side request forgery, and even remote code execution. It occurs when an attacker injects malicious XML content into an application's XML parser, which is then processed in a way that allows the attacker to exploit the system.
Attackers leverage external entities in XML documents to manipulate the parser, often leading to unintended and harmful actions. Prevention measures include disabling external entity processing, using secure parsers, validating input, and implementing access controls, all of which are essential for protecting against this form of attack and ensuring the security of applications that handle XML data.

The impact of an XXE Attack can vary based on the application's configuration and context. Potential consequences include:

- Data Exposure: Reading sensitive files or documents.
- Server-Side Request Forgery (SSRF): Forcing the application to make unauthorized requests to internal resources.
- Remote Code Execution: Running arbitrary code on the server, leading to complete compromise.

To defend against XXE Attacks, organizations should adopt the following preventive measures:

- Disable External Entities: Disable the processing of external entities in XML parsers to prevent attackers from referencing external resources.
- Use Secure Parsers: Employ secure and modern XML parsers that have built-in protection against XXE attacks.
- Input Validation: Validate and sanitize XML input, especially when it originates from untrusted sources, to filter out malicious content.
- Avoid Processing Sensitive Data: Avoid parsing XML input that contains sensitive information, especially if it comes from untrusted sources.
- Access Controls: Implement robust access controls to prevent unauthorized access to internal resources.
- Regular Updates: Keep all software components and libraries up to date to benefit from security patches and improvements.

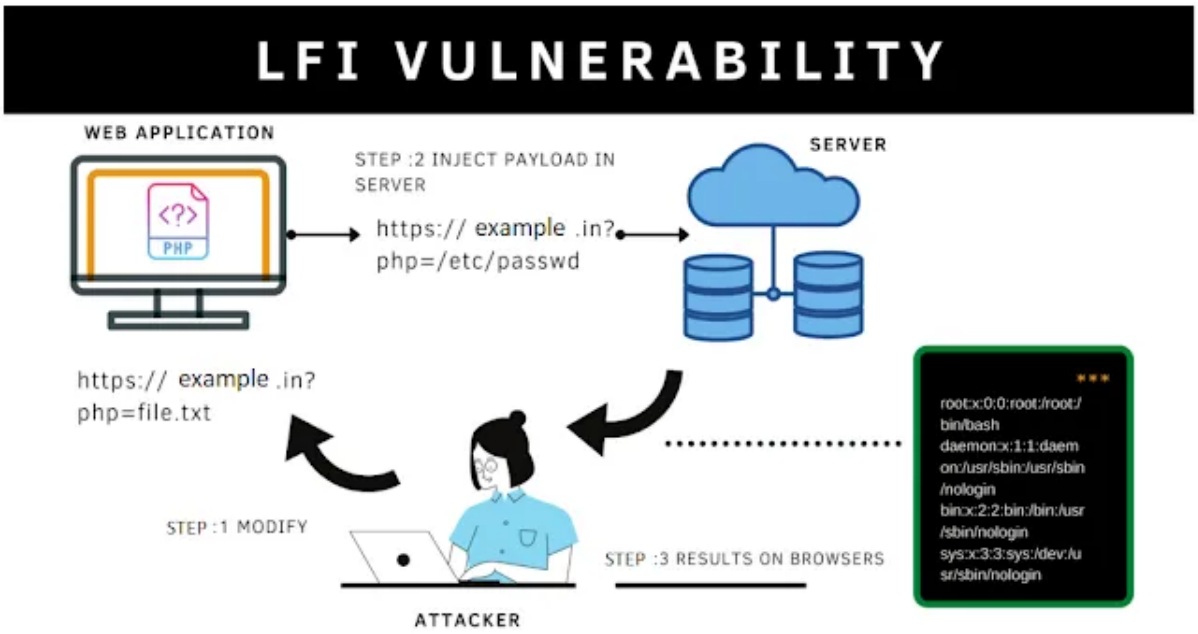


4. Local File Inclusion (LFI)-

Local File Inclusion (LFI) is a cybersecurity vulnerability that occurs when an application allows an attacker to include files on the server through the web browser. This vulnerability is typically found in web applications that dynamically retrieve and include files based on user input. When exploited, an attacker can traverse the server's file system, read sensitive files, and even execute malicious code.
When user input is not properly validated or sanitized, attackers can manipulate it to access sensitive files or execute malicious code on the server. LFI can lead to data breaches and compromise the entire system. Preventive measures include input validation, access controls, least privilege, and the use of Web Application Firewalls (WAFs) to block malicious requests. Regular security assessments and code reviews are crucial for identifying and addressing LFI vulnerabilities.

how to detect Local File Inclusion (LFI) vulnerabilities:

- Manual Testing: Manually provide input to the application, including traversal sequences, and observe file inclusion behavior.
- Patterns: Look for common LFI patterns in URLs and parameters.
- Error Messages: Analyze error messages and responses for clues, as LFI often triggers revealing errors.
- Input Fuzzing: Use input fuzzing with payloads designed for directory traversal and file access.
- Directory Listings: Check if directory listings are exposed, indicating potential LFI.
- Access Logs: Examine access logs for unusual access attempts or patterns.
- Security Tools: Use web vulnerability scanners like Burp Suite or OWASP ZAP.
- Response Analysis: Carefully analyze application responses for unexpected file content.
- Content Comparison: Compare retrieved content with known server files.
- Special Characters: Test special characters and URL encoding to bypass input validation.
- Parameter Manipulation: Experiment with different parameter values and paths.
- Brute Force: If necessary, employ a brute force approach with various file paths.
- Code Review: Review the application's code for user input used in file inclusion operations.



5.Cross-Site WebSocket Hijacking (CSWSH):

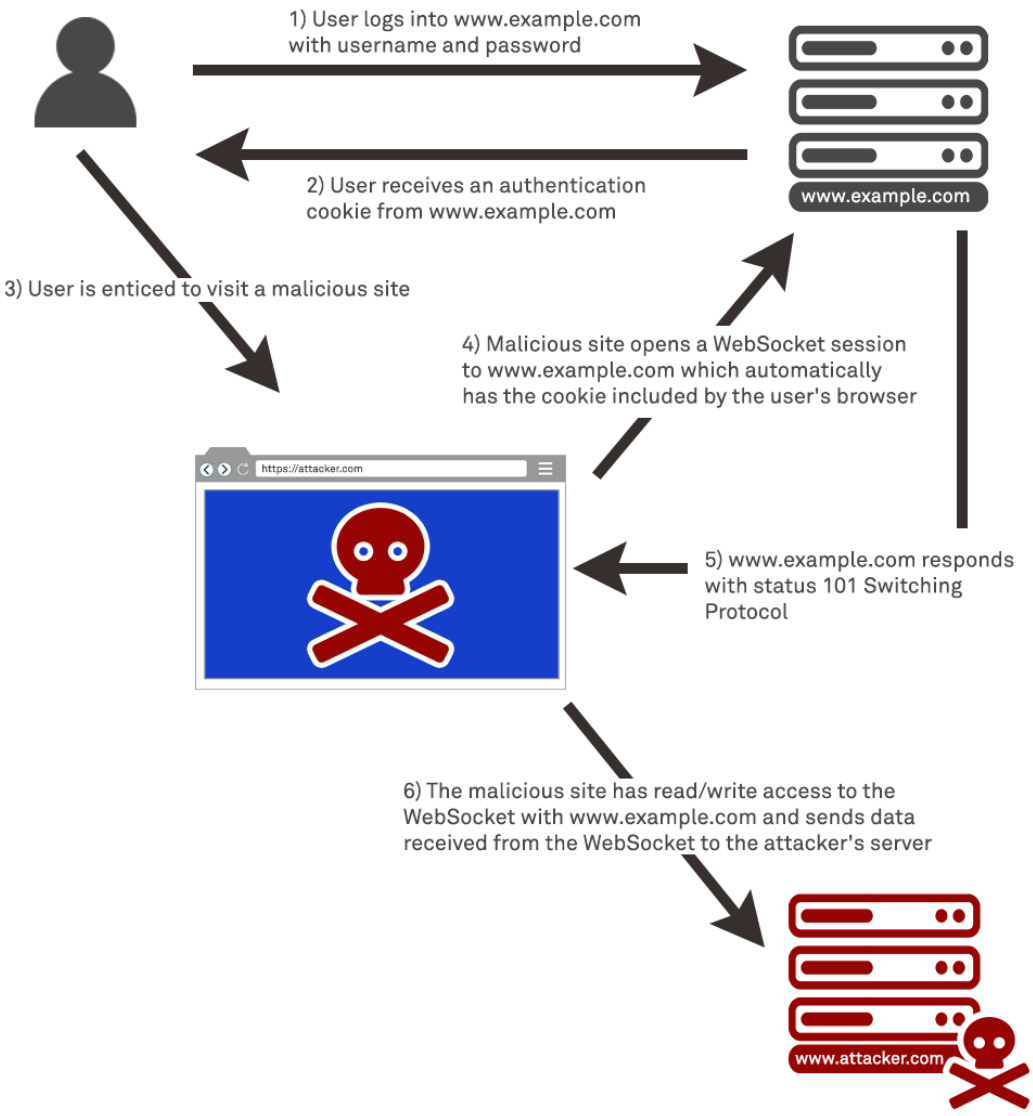
Cross-Site WebSocket Hijacking (CSWSH) is a type of security vulnerability that occurs when a WebSocket connection is established between a user's browser and a remote server. In this type of attack, an attacker can hijack a WebSocket connection and send arbitrary data to the server, potentially leading to unauthorized actions or data theft.
The attack works by tricking the user's browser into establishing a WebSocket connection with a malicious website, which then sends data to the server on behalf of the user. This can be used to bypass authentication mechanisms and perform actions on the user's behalf, such as updating their profile or making unauthorized purchases.

Some potential impacts of CSRF attacks are:

- Unauthorized actions: An attacker can use a CSRF attack to force a user to perform unauthorized actions on the vulnerable web application, such as changing their password, making unauthorized purchases, or deleting or modifying data.
- Data breaches: An attacker can use a CSRF attack to override sensitive information from the vulnerable web application, such as login credentials.
- Financial loss: An attacker can use a CSRF attack to force a user to make unauthorized financial transactions, resulting in financial loss for the victim.
- Reputation damage: A successful attack exploiting a CSRF vulnerability can lead to loss of customer trust and reputational damage for the organization.
- Regulatory violations: A successful attack exploiting a CSRF vulnerability can lead to violations of regulatory requirements, such as data protection laws.

to prevent CSWSH attacks:

- Validate the origin of WebSocket connections: Verify that the WebSocket connection is being established from a trusted origin. This can be done by checking the Origin header in the WebSocket handshake request.
- Use cookies with the SameSite attribute: Use cookies with the SameSite attribute set to strict or lax. This can help prevent cross-site request forgery (CSRF) attacks, which can be used to initiate a CSWSH attack.
- Use authentication and authorization: Implement strong authentication and authorization mechanisms to prevent unauthorized access to sensitive data or actions.
- Use secure WebSocket connections: Use secure WebSocket connections (wss://) to encrypt the data transmitted between the client and server, and prevent eavesdropping or tampering.



6. DOM-based Cross-Site Scripting (DOM XSS):

The Document Object Model (DOM) is a programming interface that defines how to create, modify or erase elements in an HTML or XML document.
DOM-based XSS is a cross-site scripting vulnerability that enables attackers to inject a malicious payload into a web page by manipulating the client's browser environment. Since these attacks rely on the Document Object Model, they are orchestrated on the client-side after loading the page. In such attacks, the HTML source code and the response to the attack remain unchanged, so the malicious input is not included in the server response. Since the malicious payload is stored within the client's browser environment, the attack cannot be detected using traditional traffic analysis tools.

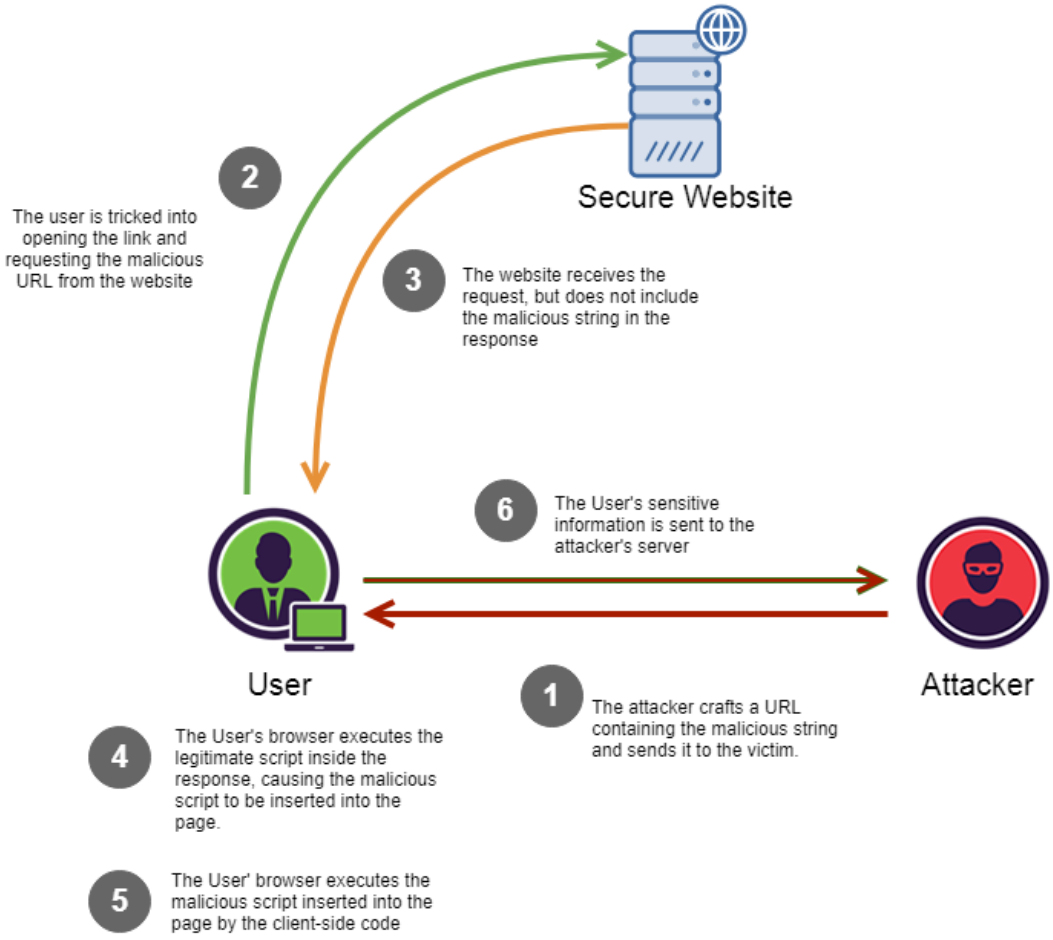
to detect DOM-based Cross-Site Scripting (DOM XSS) vulnerabilities:

- Static Analysis: Use specialized tools for static code analysis to identify potential DOM XSS issues.
- Dynamic Testing: Manually interact with the application and observe behavior for unexpected DOM modifications.
- Browser Developer Tools: Inspect the DOM and JavaScript execution in browser developer tools.
- JavaScript Code Review: Manually review JavaScript code for places where user input affects the DOM.
- Auditing Libraries: Check if the application uses libraries or frameworks with built-in DOM XSS protection.
- Security Tools: Employ web vulnerability scanners capable of detecting DOM XSS.
- Payload Testing: Inject test payloads designed for DOM XSS.
- Network Traffic Monitoring: Monitor traffic for unusual data flows or external domain communications.
- Content Security Policy (CSP): Implement and test CSP headers to restrict inline script execution.
- Input Validation: Apply input validation and sanitization practices to filter out malicious input.
- Bug Bounty and Crowdsource Testing: Engage security researchers to identify vulnerabilities.
- Real-Time Monitoring: Use real-time monitoring to detect unusual client-side behavior.
- Security Training: Train development and QA teams to recognize and address DOM XSS vulnerabilities.

To prevent DOM-based Cross-Site Scripting (DOM XSS) vulnerabilities:

- Input Validation: Validate and sanitize user inputs to block malicious code from entering the application.
- Content Security Policy (CSP): Implement a strong CSP to restrict script execution sources and prevent inline scripts.
- Escape Output: Encode user-generated content properly before rendering it in the DOM to neutralize potential XSS payloads.

- Use Trusted Libraries: Rely on trusted JavaScript libraries and frameworks that offer built-in XSS protection.
- Security Training: Educate developers on secure coding practices to minimize the risk of introducing vulnerabilities.
- Regular Security Testing: Perform automated and manual security testing to identify and fix vulnerabilities proactively.
- Updates and Patching: Keep all software components, including libraries and frameworks, up-to-date with the latest security patches.
- Monitoring: Continuously monitor for unusual client-side behavior or security incidents.



7. Directory Traversal-

Directory traversal (also known as file path traversal) is a web security vulnerability that allows an attacker to read arbitrary files on the server that is running an application. This might include application code and data, credentials for back-end systems, and sensitive operating system files. In some cases, an attacker might be able to write to arbitrary files on the server, allowing them to modify application data or behaviour, and ultimately take full control of the server.

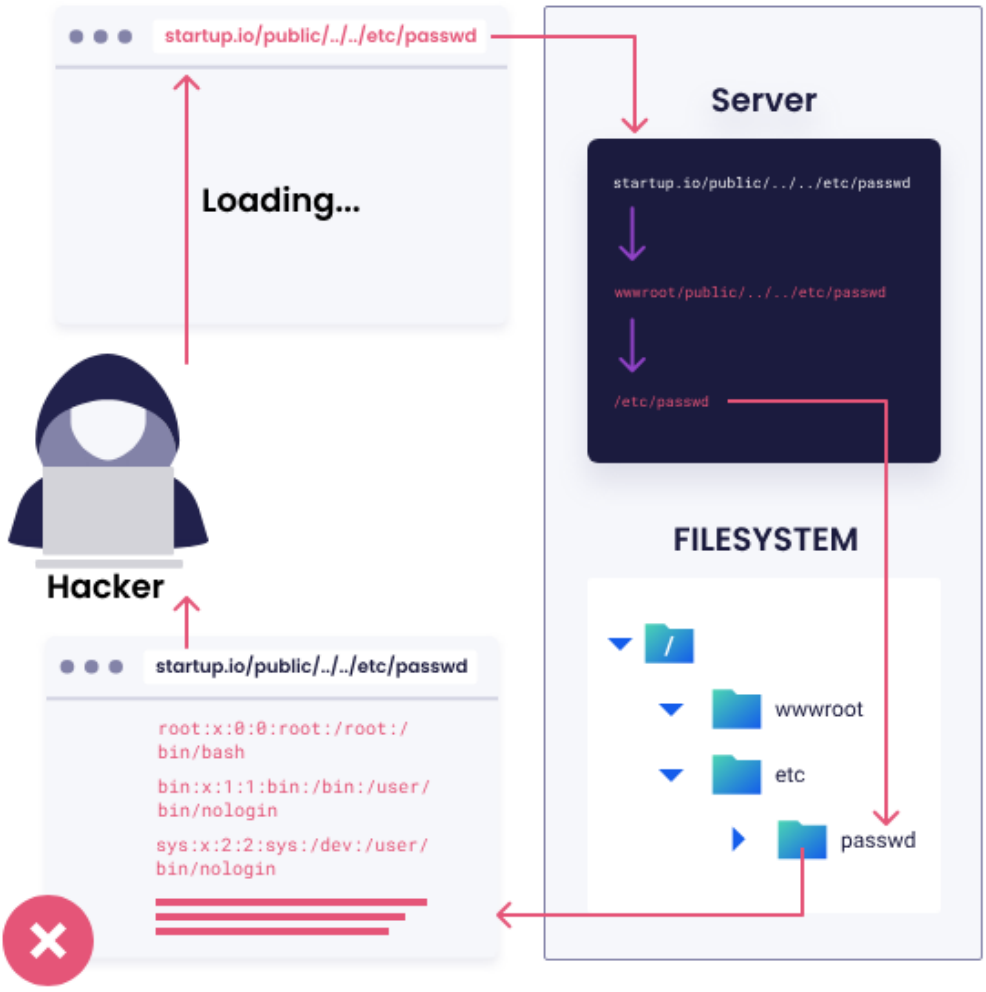
To prevent directory traversal vulnerabilities in web applications, follow these short points:

- Input Validation: Thoroughly validate and sanitize user input to ensure it cannot be manipulated to access unintended files or directories.
- Use Whitelists: Implement whitelists for file and directory access, allowing only known, safe inputs.
- Security Headers: Set up security headers, like Content Security Policy (CSP), to restrict content loading from unauthorized sources.
- File System Isolation: Isolate the web application from sensitive system files and directories, ensuring restricted access.
- File Path Encoding: Encode file paths before using them to prevent malicious input from being interpreted as valid paths.
- Access Controls: Implement strict access controls and user permissions to limit access to authorized resources.
- Patch and Update: Keep the server and application software up to date with the latest security patches and updates.
- Security Testing: Regularly perform security testing and code reviews to identify and address vulnerabilities proactively.

What an attacker can do if your website is vulnerable?

With a system vulnerable to directory traversal, an attacker can make use of this vulnerability to step out of the root directory and access other parts of the file system. This might give the attacker the ability to view restricted files, which could provide the attacker with more information required to further compromise the system.

Depending on how the website access is set up, the attacker will execute commands by impersonating himself as the user which is associated with “the website”. Therefore it all depends on what the website user has been given access to in the system.



8.Content Spoofing-

Content spoofing, also referred to as content injection, “arbitrary text injection” or virtual defacement, is an attack targeting a user made possible by an injection vulnerability in a web application. When an application does not properly handle user-supplied data, an attacker can supply content to a web application, typically via a parameter value, that is reflected back to the user. This presents the user with a modified page under the context of the trusted domain. This attack is typically used as, or in conjunction with, social engineering because the attack is exploiting a code-based vulnerability and a user’s trust. As a side note, this attack is widely misunderstood as a kind of bug that brings no impact.

Detecting content spoofing involves monitoring for suspicious behavior and inconsistencies in web content. In short:

- Use web application security tools to scan for known vulnerabilities.
- Implement security headers like Content Security Policy (CSP) to control content loading.

- Monitor user interactions and traffic for signs of unusual behavior or unauthorized content alterations.
- Conduct regular security testing and code reviews to identify and address spoofing vulnerabilities proactively.

To prevent content spoofing in web applications:

- Implement Content Security Policy (CSP): Define strict policies that control which domains are allowed to load content, scripts, and other resources on your website.
- Validate User Input: Thoroughly validate and sanitize user-generated content to prevent malicious code injection.
- Implement Cross-Site Scripting (XSS) Protection: Protect against XSS attacks, which can lead to content spoofing, by encoding and validating input and output.
- Use Secure Authentication: Ensure proper authentication mechanisms to prevent unauthorized users from accessing and manipulating content.
- Regular Security Testing: Conduct security assessments, including penetration testing and code reviews, to identify and fix vulnerabilities that could lead to content spoofing.
- Stay Informed: Keep up with security best practices and stay informed about emerging threats to adapt your defenses accordingly.

9.Insecure API Endpoints:

Application programming interfaces (API) that connect enterprise applications and data to the Internet are subject to the same vulnerabilities as regular web applications and need to be addressed with at least the same rigor.

Insecure API endpoints refer to vulnerabilities in the interfaces that allow external systems, applications, or users to interact with an application's data or functionality. These vulnerabilities can lead to unauthorized access, data exposure, and other security risks.

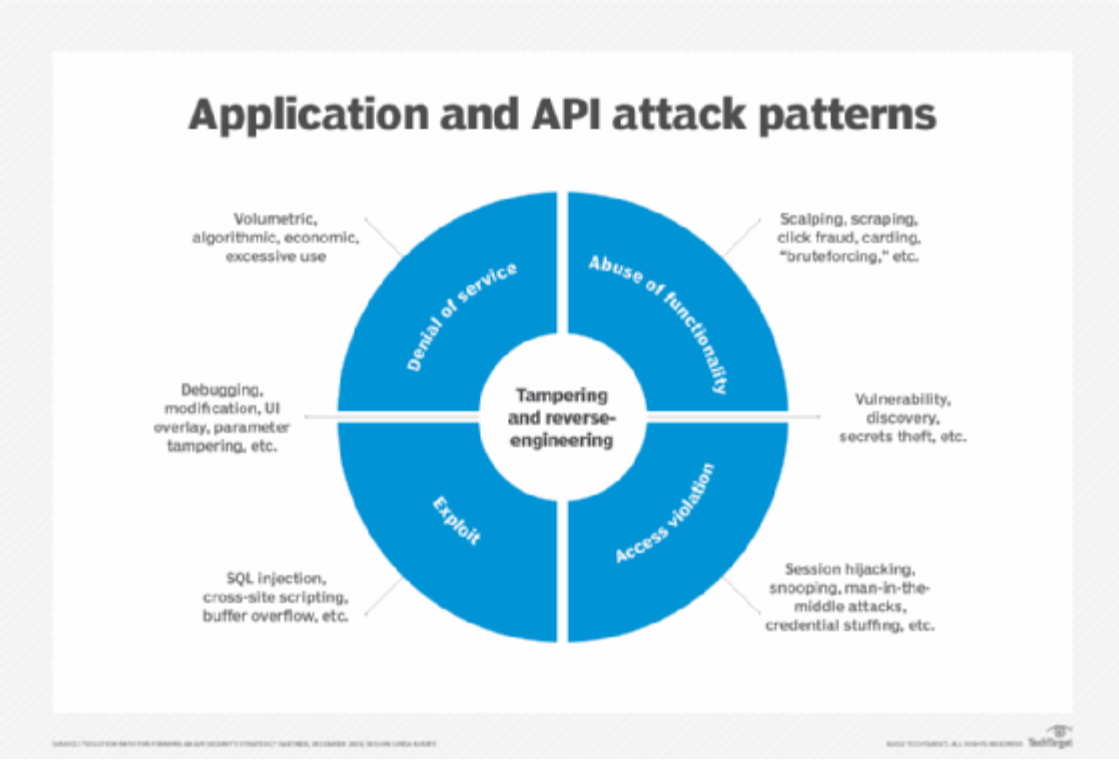
To prevent insecure API endpoints:

- Authentication and Authorization: Implement strong authentication and authorization mechanisms to ensure only authorized users or systems can access the APIs.
- Input Validation: Thoroughly validate and sanitize incoming data to prevent malicious input or SQL injection.
- Rate Limiting: Implement rate limiting to prevent abuse or excessive use of APIs, protecting against DDoS attacks and unauthorized access.
- Encrypted Communication: Use secure protocols (e.g., HTTPS) to encrypt data transmitted between the client and the API server.
- Security Tokens: Utilize tokens or API keys for access control and validation.
- API Versioning: Implement versioning to ensure backward compatibility while allowing updates to the API without disrupting existing users.

- Audit Trails: Maintain detailed logs and audit trails to track API usage and detect unauthorized access.
- Error Handling: Implement secure error handling to avoid exposing sensitive information through error messages.
- Security Testing: Conduct security assessments and penetration testing to identify and remediate vulnerabilities in API endpoints.
- Access Controls: Enforce strict access controls to restrict access to sensitive resources and data.
- Regular Updates: Keep API endpoints and dependencies up to date with security patches and updates.

To detect insecure API endpoints:

- Conduct security testing, including scanning for known vulnerabilities.
- Monitor API traffic for suspicious patterns and unusual behavior.
- Analyze access logs and audit trails for unauthorized access attempts.
- Perform code reviews to identify and address security issues in the API code.
- Use web application security tools to assess API security regularly.



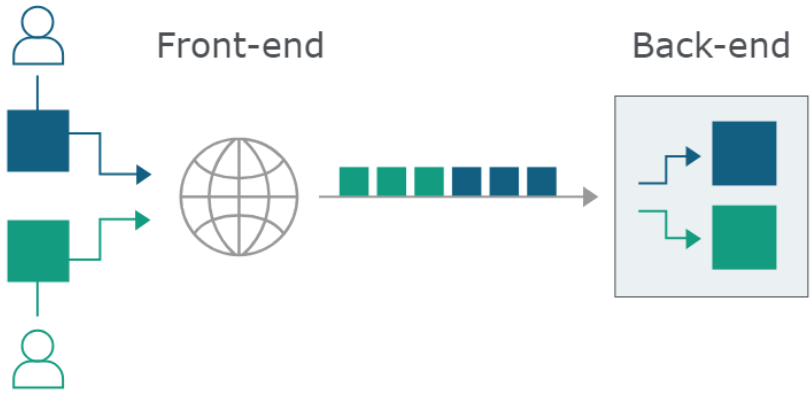
10.HTTP Request Smuggling:

HTTP request smuggling is a technique for interfering with the way a web site processes sequences of HTTP requests that are received from one or more users. Request smuggling vulnerabilities are often critical in nature, allowing an attacker to bypass security controls, gain unauthorized access to sensitive data, and directly compromise other application users. Request smuggling is primarily associated with HTTP/1 requests. However, websites that support HTTP/2 may be vulnerable, depending on their back-end architecture.

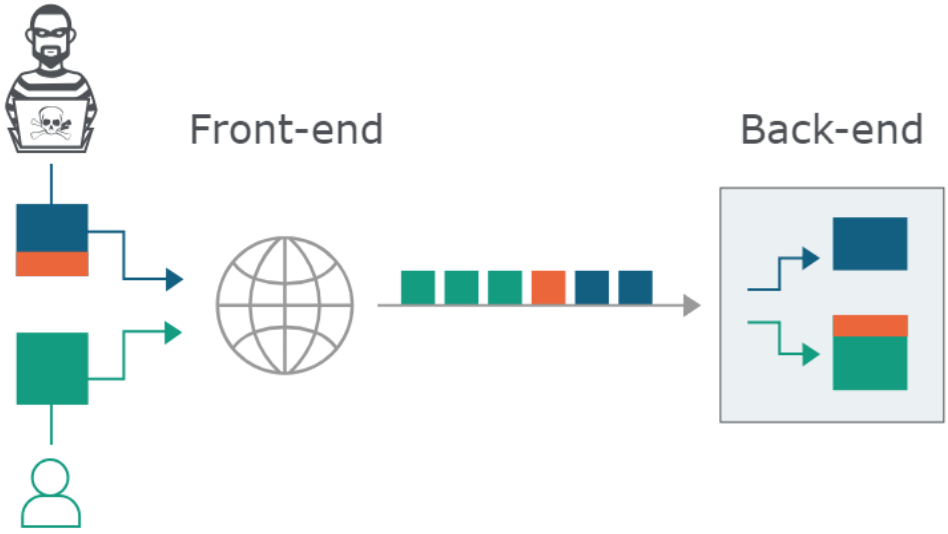
What happens in an HTTP request smuggling attack?

Today's web applications frequently employ chains of HTTP servers between users and the ultimate application logic. Users send requests to a front-end server (sometimes called a load balancer or reverse proxy) and this server forwards requests to one or more back-end servers. This type of architecture is increasingly common, and in some cases unavoidable, in modern cloud-based applications.

When the front-end server forwards HTTP requests to a back-end server, it typically sends several requests over the same back-end network connection, because this is much more efficient and performant. The protocol is very simple; HTTP requests are sent one after another, and the receiving server has to determine where one request ends and the next one begins:



In this situation, it is crucial that the front-end and back-end systems agree about the boundaries between requests. Otherwise, an attacker might be able to send an ambiguous request that gets interpreted differently by the front-end and back-end systems:



Here, the attacker causes part of their front-end request to be interpreted by the back-end server as the start of the next request. It is effectively prepended to the next request, and so can interfere with the way the application processes that request. This is a request smuggling attack, and it can have devastating results.

How to prevent HTTP request smuggling vulnerabilities

- Use HTTP/2 end to end and disable HTTP downgrading if possible. HTTP/2 uses a robust mechanism for determining the length of requests and, when used end to end, is inherently protected against request smuggling. If you can't avoid HTTP downgrading, make sure you validate the rewritten request against the HTTP/1.1 specification. For example, reject requests that contain newlines in the headers, colons in header names, and spaces in the request method.
- Make the front-end server normalize ambiguous requests and make the back-end server reject any that are still ambiguous, closing the TCP connection in the process.
- Never assume that requests won't have a body. This is the fundamental cause of both CL.0 and client-side desync vulnerabilities.
- Default to discarding the connection if server-level exceptions are triggered when handling requests.
- If you route traffic through a forward proxy, ensure that upstream HTTP/2 is enabled if possible.