Interview Questions

Interview Questions:-

1. What is OAuth? What are Grant types and their mechanism, what is scope? how to exploit and mitigate it? (back-end; server-to-server mechanism)

A. *OAuth* stands for Open Authorization and it is an authorization framework that is being used for 10 years now by the websites and the web applications. It enables them to access limited data of an user to another application/3rd party Application. It enables the users to login to the client applications seamlessly without even exposing their credentials.

B. ***OAuth*** works by sharing limited access of the user data and it does by defining the interaction between the 3 parties client application, a resource owner and an OAuth service. The interaction happens through these OAuth flows: (i) "Authorization_Code" (ii) "Implicit". There are 4 but mainly these 2 are used.

C. ***Scope*** - The client has to specify or tell the OAuth server that what kind of subset of user data the client application is requesting and for that it uses the *scope* parameter to define them. i.e., `scope=contacts`

D. ***Authorization Code Grant Type*** - In this grant type the user (the resource owner) uses the client application to login through the OAuth service by making some redirects throw browser-based HTTP requests. Now the user will asked for their consent for the request, once the user has given the consent to the OAuth service provider (authorization server) that server will grant the "Authorization_code" to the client application, now the application will use that "code" to request for "access_token" to the authorization server and the server will provide the same to the client application. Now using both the code and token the client can make call to the API to fetch the data of the users to be used.

E. ***Implicit Grant Type*** - It is same like the Authorization code grant type and the flow is also same but this is not secure as the "auth_code" because upon the user's consent given by the OAuth service, the service will directly gives the "access_token" without even requesting the code. This is not happening like over a secure-back-channel. Its not like the "auth_code" grant like server-server interaction, it is browser-based redirects and used by single-page application and native desktop application. It cannot store "client_secret" on the back-end.

F. ***Vulnerabilities to exploit OAuth:-*** (i) From Client Side Application (ii) From OAuth Service provider/server

1. Improper implementation of Implicit Grant Type :- (i) Bypassing implicit grant type flow.
2. Flawed CSRF Protection :- (i) We can perform CSRF attack with the stolen code and deliver it to any user, once the user clicks it will load and complete the OAuth flow process

3. Leaking the Authorization code and Access Token through :- **(i) Hijacking the redirect_uri** - We can hijack the "redirect_uri" parameter in request putting our controlled server and deliver it to victim to steal their "auth_code". **(ii) Using flawed validation of redirect_uri** - Bypassing flawed redirect_uri validation by appending extra values to the redirect_uri with default host i.e., `https://default-host.net$@abc.evil.net&@def.evil.net#@ghi.evil.net/`, using double/duplicate redirect_uris and using localhost **(iii) Stealing codes and access tokens via a proxy page** - using vulnerable subdomains, getting a path using directory traversal and open-redirects, XSS and HTML Injection.
4. Flawed Scope Verification :- **(i) Scope upgrade: authorization code flow** - If an attacker finds a malicious application which receives the "auth_code", the attacker can steal and use the code to do a POST request changing/upgrading the scope of the request to get the "access_token" and if the server that not validate the initial request it can generate the "access_token" through that upgraded scope. **(ii) Scope upgrade: implicit flow** - The "access_token" is already their the attacker needs to use malicious website to steal it and use it with modified "scope" parameter.
5. Unverified User Registration - When authenticating users via OAuth, the client application makes the implicit assumption that the information stored by the OAuth provider is correct. An attacker can exploit this by registering an account with the OAuth provider using the same details as a target user, such as a known email address. Client applications may then allow the attacker to sign in as the victim via this fraudulent account with the OAuth provider.

G. *To mitigate* against OAuth attacks both the Client Application and OAuth Service Provider has to take apply robust security implementation

On OAuth Provider Side :-

- Client application needs to register valid whitelists of "redirect_uris" so that the OAuth provider can valid them using the same.
- Only the exact and complete match should be allowed skip using pattern matching.
- The user's session should be hashed and the server must have to enforce the "state" parameter to avoid attacks like CSRF.
- The "client_id" and the "scope" also has to validated by the server to ensure the same client is exchanging the request without tempering the scope.

On Client Application Side :-

- Client must enforce and ensure that the "state" parameter is mandatory.
- "client_id" is secret so must have to kept private.
- Avoid leaking of the authorization_code to the referrer link.
- The "redirect_uri" parameter should not only to be sent to `/authorization` endpoint but also to the `/token` endpoint. This will verify "redirect_uri".

2. What is SOP, CSP and CORs?

A. *SOP* - It stands for Same Origin Policy and it prevents against the JavaScript of one websites from loading any content from another website. Through SOP website that share same protocol, host and port can only interact with each other

It is possible to relax SOP using the `document.domain` property, this allows websites to read or access content from other website with different origin only if it is part of your FQDN (Fully Qualified Domain Name). For example subdomain.domain.com wants to read file from maindomain.com than that has to be mentioned with the `document.domain` and will be able to load any content.

B. *CORs* - It stands for Cross Origin Resource Sharing is a browser mechanism and it came into idea to extend and relax the SOP. It helped with the controlled access to load content using JS from one origin to another only when it is allowed. ***It also opens the possibility for cross-domain attacks if misconfigured or poorly implemented.***

**It is not a security mechanism so attacks like XSS and CSRF attacks can be performed.*

In order to allow the JS to load any content from other website which is cross-origin-resource sharing the server has to set the `"Access-Control-Allow-Origin"` to the specific `"origin"`, `"null"` or wildcard `"*"`. To allow the credentials (cookies or Authorization Bearer Tokens) the server has to set the "Access-Control-Allow-Credential" to `"true"`.

Attacks we can perform on poorly misconfigured CORs are:-

- Basic Origin reflection.
- Using "null" for origin.
- Exploiting XSS using CORs Trust relation through "Sub-Domain"

Mitigation:-

- Proper implementation and configuration of cross-origin requests should be applied like if sensitive data is being transferred it should allow the specified origin through the header.
- Avoid whitelisting the "null" value.
- Allow only trusted site.

C. *CSP* - It stands for Content Policy Security and it is a browser security mechanism that helps to mitigate attacks like XSS, ClickJacking and other. It works by restricting the pages to load any script or content or the resources and also restricting whether a page can be framed.

* CSP can be implemented through header by server :- `Content-Security-Policy` containing the policy value.

1. Mitigating XSS attacks using CSP

`script-src 'self'`

`script src 'https://www.maindomain.com'`

2. Mitigating dangling markup attacks using CSP - If XSS attack is bypassed using "img-src".

`img-src 'self'`

`img-src 'https://www.maindoamin.com'`

3. Protecting against clickjacking using CSP

`frame-ancestor 'self'`

`frame-ancestor 'https://www.maindomain.com'`

* Applying `CSP` is better than `X-Frame-Option` as it is more flexible and it validates each frame in the parent frame in the hierarchy as the `X-Frame-Option` only validated the top-level frames.

---

3. What is DOM and DOM XSS? What is Source and Sink?

DOM - It stands for Document Object Model and is the hierarchal representation of the HTML elements on the page that can be controlled by JavaScript by manipulating the elements and objects on the DOM.

DOM XSS - arises when the JavaScript takes the data from attacker-controllable source like the URL and will pass it through the sink which supports dynamic code execution like eval() or innerHTML. This let it to execute any malicious JavaScript attacking the victim's session on sensitive data.

Source - Source is the JavaScript property that accepts data that is potentially attacker-controlled.

Sink - Sink is the JavaScript function or DOM Object that will take the data from source and will execute it.

---

4. What is XSS? Types of XSS, how to exploit and mitigate it?

XSS which is also known as Cross Site Scripting is a vulnerability where an application is vulnerable to JavaScript Injection attack and attacker can inject any malicious JavaScript code and it will be executed letting the attacker steal user's sensitive data or do further advance exploitation.

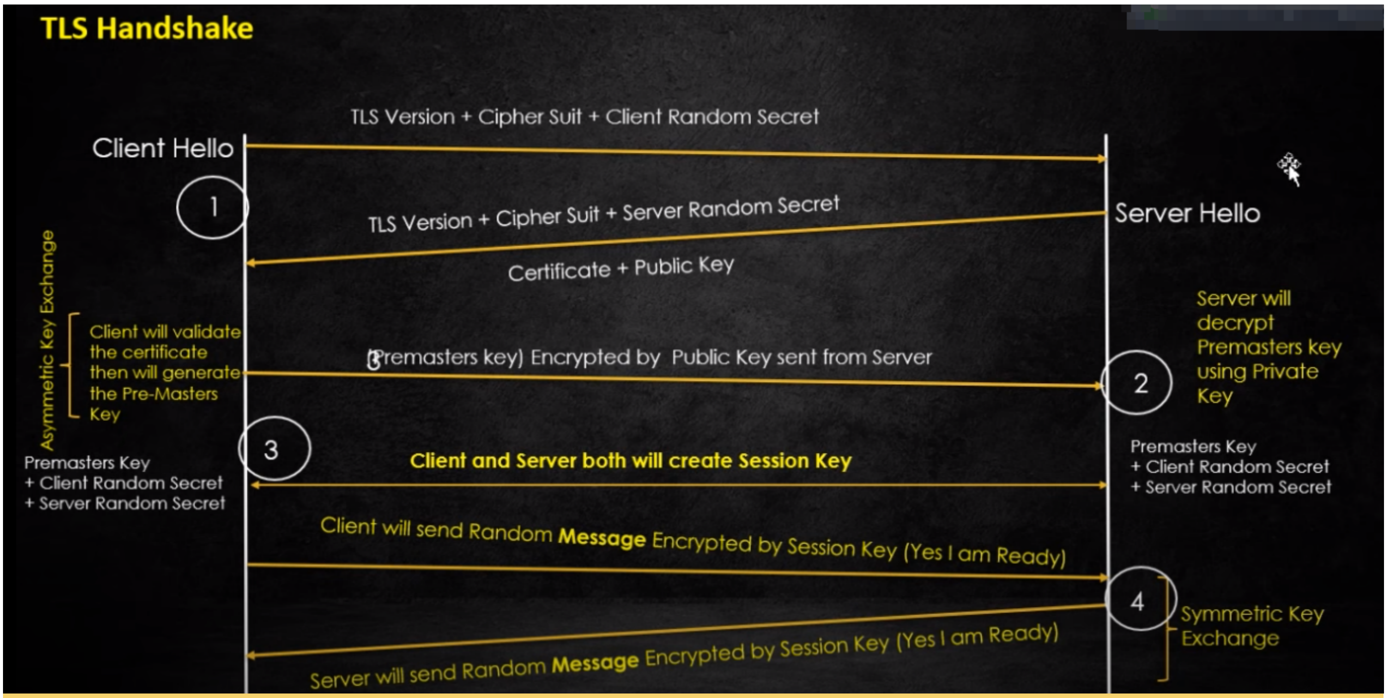
There mainly 3 types of XSS :-

- Reflected XSS
- Stored XSS
- DOM XSS

To mitigate XSS a developer can implement these security measures :-

- Input validation/sensitization during arrival of the input (Filtration).
- Encoding data on output.
- Using security response header.
- Using whitelisting instead of blacklisting.
- Using CSP.

5. TLS/SSL Handshake explanation.


TLS Handshake
Client Hello
TLS Version + Cipher Suit + Client Random Secret
1
Server Hello
TLS Version + Cipher Suit + Server Random Secret
Certificate + Public Key
Asymmetric Key Exchange
Client will validate the certificate then will generate the Pre-Masters Key
(Premasters key) Encrypted by Public Key sent from Server
2
Server will decrypt Premasters key using Private Key
3
Client and Server both will create Session Key
Premasters Key
+ Client Random Secret
+ Server Random Secret
Premasters Key
+ Client Random Secret
+ Server Random Secret
Client will send Random Message Encrypted by Session Key (Yes I am Ready)
4
Symmetric Key Exchange
Server will send Random Message Encrypted by Session Key (Yes I am Ready)

6. What is CSRF? How to exploit? How to fix CSRF?

In ***CSRF attack*** an attacker can induce or force a victim on clicking an attacker controllable URL which will perform action on behalf of the victim user like changing or resetting passwords or email addresses of the victim. The delivery mechanism is like similar to Reflected XSS where attacker will inject a malicious HTML on the website that they control and induce the victim to visit.

There 3 requirements for this :-

- A relevant action - an action that will do CRUD operation.

- Session Handling Cookies - which is to target the same victim with same session.
- No unpredictable request parameter - Suppose there is form that also takes existing password there this attack will fail.

To mitigate CSRF :-

- Developer show use CSRF Token with high entropy, it should tied to the user's session and server has to also store in order to validate the user witch each request.
- Using the SameSite attribute within the cookie like `"strict"` and `"Lax"` **(i)** If a cookie is set with the SameSite=Strict attribute, browsers will not send it in any cross-site requests. (ii) Lax SameSite restrictions mean that browsers will send the cookie in cross-site requests, but only if the request uses the GET method.
- Referrer-Based Validation - Here the header has to be validated that the request is coming from application.

---

7. What are the HTTP cookie attributes name and explain?

- ***HttpOnly*** attribute focus is to prevent access to cookie values via JavaScript, mitigation against Cross-site scripting (XSS) attacks.
- ***Secure flag*** - A cookie with the Secure attribute is only sent to the server with an encrypted request over the HTTPS protocol.
- ***SameSite*** attribute lets servers specify whether/when cookies are sent with cross-site requests.

---

8. List all Security Headers and explain them?

- CSP - will try to prevent from attacks like XSS
- HSTS - prevents an attacker from downgrading the HTTPS connection to an HTTP connection.
- X-Frame-Options - will help to mitigate ClickJacking attacks
- CORs - will prevent from cross origin attacks.
- X-content-Type-Options -will help browser to protect the MIME type sniffing
- XSS protection

---

9. TCP VS UDP explain!

TCP	**UDP**
It is connection oriented protocol.	It not connection oriented but connectionless protocol.

It is comparatively slower than UDP.	It is faster than TCP.
Packet re-transmission is guaranteed.	No guarantee of package delivery.
It is reliable.	It is not reliable.
Handshake.	No-Handshake.
Used in HTTP(S),FTP,SSH etc.	Used in DNS,VoIP etc.

10. What is SSRF? How to exploit and mitigate it?

Server-Side Request Forgery vulnerability allows an attacker to to induce/force an web application to make unauthorized requests to internal resources or external services that is an unintentional behaviour. It can lead to data leakage and remote code execution.

Exploiting SSRF:

To exploit SSRF, an attacker tricks a web application into making requests to unintended, potentially sensitive resources. Here's a simplified example:

1. An attacker finds a vulnerable web application that allows user-controlled input to specify a URL, which the application fetches and displays.
2. The attacker submits a malicious URL as input, pointing to an internal server or a service like "http://localhost/admin."
3. The web application makes the request to the malicious URL as if it were a legitimate external request.
4. The attacker can observe the response, potentially gaining access to sensitive information or internal resources.

To prevent SSRF attacks:

- Input Validation and Whitelisting: Ensure that any user-provided URLs or input are properly validated and restricted. Use whitelists to specify which domains or resources the application is allowed to access.
- Use Trusted Libraries: use trusted HTTP client libraries and ensure they are configured securely. Some libraries may have features that can mitigate SSRF vulnerabilities.
- Avoid User-Controlled URLs: Avoid situations where user input can directly specify URLs for the application to fetch. If necessary, use a safe domain-validated URL parsing library that checks if the provided URL is valid.
- Disable unused URL schemas
- Authentication on internal services

10. What is XXE? How to exploit and mitigate it?

Vulnerability Description:

XXE vulnerabilities occurs when an application fails processes XML input without properly restricting access to external resources. Attackers can use this to exploit XXE by injecting malicious XML content that references external entities such as files or network resources.

Exploitation:

Exploiting XXE vulnerabilities often involves crafting malicious XML documents that reference external entities, which can have several adverse consequences:

- File Disclosure: Attackers can read sensitive files on the server by referencing them as external entities in the XML document.
- Server-Side Request Forgery (SSRF): Attackers can use the application to make requests to internal network resources or external services, potentially facilitating an SSRF attack.
- Denial of Service: Maliciously crafted XML input can lead to resource exhaustion and a denial-of-service condition.
- Remote Code Execution: In severe cases, XXE can lead to remote code execution, allowing attackers to execute arbitrary code on the server.

Mitigation:

All XXE vulnerabilities arise because the application's XML parsing library supports potentially dangerous XML features that the application does not need or intend to use. The easiest and most effective way to prevent XXE attacks is to disable those features.

- Input Validation
- Use Modern Parsers
- Disable External Entity Resolution: We can disable the resolution of external entities entirely. If we don't need external entity references, we can configure the parser not to resolve them.
- Content Security Policies: Implement and enforce a Content Security Policy restricts the sources from which an application can load external content, including XML files.
- Firewall and WAF
- Regular Updates

1. ***XML : XML*** is a language designed for storing and transporting data.
2. ***XML Entities : XML entities*** are a way of representing an item of data within an XML document. For example, the entities < and > represent the characters < and >. These are metacharacters used to denote XML tags, and so must generally be represented using their entities when they appear within data.

3. ***DTD : document type definition (DTD)*** contains declarations that can define the structure of an XML document, the types of data values it can contain and other items.

---

12. What is LOG4J and how to exploit and prevent?

CVE-2021-44228 affects the Java logging package log4j and allows remote code execution on hosts engaging with software that utilizes this log4j version. log4j version 2.16.0 is available and patches this vulnerability (JNDI is fully disabled, support for Message Lookups is removed, and the new DoS vulnerability CVE-2021-45046 is not present)

Vulnerability Description:
The vulnerability is related to how Log4j processes user-supplied data in log messages. Log4j has a feature that allows embedding variables in log messages, and this feature can be abused by an attacker to include malicious code. It is primarily due to the use of the JNDI (Java Naming and Directory Interface) lookup mechanism, which can be triggered through the log messages.

Exploitation:
To exploit this vulnerability, an attacker would craft a specially formatted log message that includes a JNDI lookup to a malicious server controlled by the attacker. When the application processes this log message, it would initiate a JNDI lookup, which could result in the execution of arbitrary code or other malicious actions on the target system. Essentially, an attacker can gain remote code execution on a vulnerable system.

Prevention:
To prevent this vulnerability, you should take the following steps:

- Update Log4j
- Monitor for Exploitation
- Filter Inputs
- Firewalls and Intrusion Detection Systems
- Vendor Patching

---

13. What is 2nd order SQL Injection?

A 2nd order SQL Injection occurs when attacker's malicious input is stored by the application but that malicious SQL payload is executed at a later point in time or by a different user. It is a less common but more subtle form of SQL injection.

---

14. What is the safe method of Password storage on the server database for the security of users?

Salting: Generate a unique random salt for each user. A salt is a random value that is combined with the user's password before hashing. This ensures that even if two users have the same password, their hashes will be different due to the unique salt. Store the salt alongside the hashed password in the database.

Hashing: Use a strong cryptographic hashing algorithm Argon2id. Hash the user's password before storing it in the database. The hash is a one-way transformation, making it computationally infeasible for an attacker to reverse the process and obtain the original password.

---

15. What is Prototype Pollution, and what exploits could it lead to with both client / server-side variants?

Prototype pollution affects JavaScript-based applications, both on the client-side and server-side. It occurs when an attacker manipulates the prototype of an object and introduces unintended or malicious properties and behaviours. Prototype pollution can have serious consequences lead to various exploits, including:

Client-Side Exploits:

1. **Cross-Site Scripting (XSS) Amplification:** An attacker can manipulate the prototype of JavaScript objects to inject or modify functions and properties, leading to the execution of malicious code. This can amplify the impact of an XSS vulnerability and potentially compromise user data or accounts.
2. **Denial of Service (DoS):** An attacker can overload the application by manipulating the prototype, causing excessive resource consumption or crashes.
3. **Information Disclosure:** By polluting object prototypes, an attacker can potentially access sensitive information stored within objects or leak internal application data.

Server-Side Exploits:

1. **Command Injection:** In server-side JavaScript environments, such as Node.js, prototype pollution can lead to command injection attacks when server-side code processes data derived from client-side requests.
2. **Request Forgery:** Attackers can manipulate server-side JavaScript objects, leading to request forgery attacks or other malicious interactions with external services or APIs.
3. **Authentication Bypass:** If prototype pollution affects authentication mechanisms, attackers may manipulate user roles, permissions, or session handling, leading to unauthorized access.

To mitigate prototype pollution vulnerabilities, consider the following practices:

- **Input Validation:** Always validate and sanitize user input and other data that could be manipulated by attackers.
- **Whitelist Object Properties:** Define a whitelist of allowed properties and reject any objects with unrecognized properties.
- **Use Trusted Libraries:** When working with third-party libraries or modules, ensure they follow best practices to prevent prototype pollution.
- **Regular Updates:** Keep your JavaScript runtime and libraries up-to-date, as newer versions may include security patches.
- **Security Testing:** Perform security testing, including static analysis and dynamic scanning, to identify and fix prototype pollution vulnerabilities in your code.
- **Content Security Policy (CSP):** Implement a CSP to mitigate the impact of client-side exploits like XSS.
- **Access Control:** Enforce proper access controls to restrict the privileges and actions of users or external entities within your application.

16. What are Parameterized Queries?

Parameterized queries contain placeholders for the input data, which is then escaped and passed on by the drivers. Instead of directly passing the data into the SQL query, we use placeholders and then fill them with PHP functions.

Consider the following modified code:

```
$username = $_POST['username'];
  $password = $_POST['password'];

  $query = "SELECT * FROM logins WHERE username=? AND password = ?" ;
  $stmt = mysqli_prepare($conn, $query);
  mysqli_stmt_bind_param($stmt, 'ss', $username, $password);
  mysqli_stmt_execute($stmt);
  $result = mysqli_stmt_get_result($stmt);

  $row = mysqli_fetch_array($result);
  mysqli_stmt_close($stmt);
```

The query is modified to contain two placeholders, marked with ? where the username and password will be placed. We then bind the username and password to the query using the mysqli_stmt_bind_param() function. This will safely escape any quotes and place the values in the query.

17. What is De-serialization? How to mitigate?

Serialization is the process of converting data structures or objects into a format that can be easily stored, transmitted, or reconstructed. Deserialization, on the other hand, is the process of taking serialized data and recreating the original data structure or object. Serialization and deserialization are commonly used in programming for various purposes, including data storage, network communication, and object persistence.

Insecure deserialization is when user-controllable data is deserialized by a website. This potentially enables an attacker to manipulate serialized objects in order to pass harmful data into the application code.

---

18. What are the ways we can bypass and upload file?

File upload vulnerabilities occurs when a web application allows users to upload files (e.g., images, documents) without proper validation and controls. These vulnerabilities can be exploited by attackers to upload and execute malicious files on the server, potentially compromising the application, its users, or the underlying server.

Here are the common ways file upload vulnerabilities can be exploited and some mitigation strategies:

1. Bypassing File Type Checks:

Content Spoofing: Attackers can manipulate file extensions or content to make the server interpret a malicious file as something benign (e.g., a malicious PHP script disguised as a harmless image file). Mitigation: Validate the file type by both examining the file extension and checking the file's content using techniques like file signatures (magic bytes).

2. Malicious File Execution:

File Inclusion: If the application allows the uploaded file to be included or executed on the server, attackers can upload malicious files and exploit them through various techniques like Local File Inclusion (LFI) or Remote File Inclusion (RFI). Mitigation: Do not allow uploaded files to be directly executed on the server and use safe storage locations.

3. Overwriting Sensitive Files:

Attackers may try to upload files with common names to overwrite existing files on the server, such as configuration files, .htaccess, or even the application's source code. Mitigation: Prevent overwriting of critical files and store uploaded files in a secure, isolated directory.

4. Path Traversal:

Attackers can craft file names that exploit path traversal vulnerabilities to access and overwrite sensitive files on the server. Mitigation: Implement input validation to block directory traversal attempts and sanitize user input.

5. Malicious File Content:

Attackers can embed malicious code within legitimate file types (e.g., JavaScript or PHP code inside an image) and upload these files to compromise the application. Mitigation: Scan the file content for malicious code, sanitize user input, and implement Content Security Policies (CSP) to restrict what can be executed in a web context.

6. Server-Side Request Forgery (SSRF):

If the application processes or fetches URLs from the uploaded file, it can lead to SSRF vulnerabilities, allowing attackers to access internal resources. Mitigation: Whitelist allowed domains, restrict external requests, and properly validate and sanitize URLs.

To mitigate file upload vulnerabilities, it's crucial to follow these best practices:

- Implement strict input validation and validation checks on uploaded files, including file type, size, and content.
- Store uploaded files in a dedicated directory, separate from application files.
- Avoid directly executing uploaded files on the server.
- Sanitize user inputs and prevent directory traversal attempts.
- Use security mechanisms like Web Application Firewalls (WAFs) to filter out malicious requests.
- Regularly update and patch the application and its dependencies.

---

19. What JWTs? What are the attacks on JWTs? How to mitigate?

JWT, or JSON Web Token, is a compact and self-contained means of representing information between two parties as a JSON object. JWTs are commonly used for authentication and authorization in web applications and APIs. They consist of three parts: a header, a payload, and a signature. The header typically specifies the type of token and the signing algorithm, the payload contains the claims, and the signature is used to verify the authenticity of the token.

Attacks on JWTs can occur when security measures are not implemented correctly. Common JWT-related attacks include:

1. JWT Tampering: Attackers can modify the contents of the payload (claims) or the header of a JWT to impersonate other users, gain unauthorized access, or modify session data.
2. JWT Token Expiry Manipulation: If the expiration (exp) claim is not properly validated, attackers can modify the token's expiry time to extend their session or access resources for an extended period.

3. JWT Token Leakage: If the token is not properly secured, attackers may be able to steal it from the client, eavesdrop on the network, or extract it from logs, leading to unauthorized access.
4. RS256 to HS256 Attack: Some applications may switch from RSA (RS256) to HMAC (HS256) for JWT signatures. Attackers may attempt to manipulate the signing algorithm from RS256 to HS256 to generate fake tokens with HMAC signatures.

To bypass these attacks and mitigate JWT-related vulnerabilities, consider the following best practices:

- Use Strong and Unique Keys: Choose strong and unique keys for signing JWTs. Use long, random, and secret keys that are not easily guessable or discoverable.
- Validate the JWT Signature: Always verify the JWT's signature to ensure that it hasn't been tampered with. Use the appropriate public key or secret key depending on the signing algorithm (e.g., RS256, HS256).
- Check the Token Expiry (exp): Validate the "exp" claim to ensure that the token has not expired. Reject expired tokens to prevent session hijacking.
- Implement Proper Access Controls: Ensure that your application enforces proper access controls to determine whether a user with a valid JWT should be allowed to access a specific resource.
- Prevent Token Leakage: Protect JWTs from leakage by using secure HTTP cookies with the "HttpOnly" and "Secure" flags, encrypting the token, and securing client-side storage mechanisms like Local Storage.
- Rate Limiting: Implement rate limiting to prevent attackers from using a large number of stolen or forged JWTs in rapid succession.
- Implement Session Management: For session tokens, consider implementing session management mechanisms that allow you to revoke tokens and perform session clean-up.

---

25. What is Parameter Pollution? How to exploit and mitigate it?

Parameter pollution, also known as "HTTP parameter pollution," is a security vulnerability that occurs when an attacker manipulates the parameters of an HTTP request to exploit a web application. It can lead to various security issues, including unauthorized access, data leakage, and application malfunction. Parameter pollution typically happens in web applications that use query strings, form data, or other parameters as inputs to control application behaviour.

Exploiting Parameter Pollution:

1. The exploitation of parameter pollution often involves manipulating the parameters in a way that tricks the application into performing unintended actions or returning sensitive data. Here are a few examples of how it can be exploited:
2. Access Control Bypass: By manipulating parameters, an attacker can access resources or perform actions they are not authorized to, such as viewing other users' data or administrative functions.

3. Data Leakage: Parameter pollution may lead to the exposure of sensitive data, such as database contents, configuration files, or logs.
4. Denial of Service (DoS): Attackers can overload the application by making it process large or complex requests, causing resource exhaustion and application downtime.
5. Function Execution: In some cases, parameter pollution can trick the application into executing unintended functions or code.

To mitigate parameter pollution vulnerabilities, consider the following best practices:

- Input Validation: Implement strict input validation to ensure that parameter values are within expected bounds and adhere to defined data types. Reject any input that doesn't conform to expected formats.
- Whitelist Parameters: Maintain a whitelist of allowed parameters and their expected values. Reject any parameters that are not on the whitelist.
- Avoid Mixing Data Sources: Keep data sources and user-controlled input separate. Don't use input parameters to directly access or manipulate sensitive data or resources.
- Proper Access Controls: Enforce proper access controls to ensure that users can only access resources or perform actions they are authorized for.
- Parameter Sanitization: Use input sanitization techniques to remove or neutralize potentially harmful characters or sequences.
- Use Security Tokens: Implement security tokens or anti-CSRF tokens to ensure that requests are originating from the intended users.
- Rate Limiting: Implement rate limiting to prevent abuse and DoS attacks by limiting the number of requests a user can make in a given time frame.
- Access Logs and Monitoring: Maintain detailed access logs and monitoring systems to detect unusual or malicious requests and behavior.

24. What is Race Condition Vulnerability?

It is a business logic flaws and occurs when websites fails process multiple requests at the same time/fraction of time without any security implementation which could lead to collisions of requests that causes unintended behaviour in the application. A race condition attack uses carefully timed requests to cause intentional collisions and exploit this unintended behaviour for malicious purposes.

Shell

In the simplest possible terms, shells are what we use when interfacing with a Command Line environment (CLI). The common bash or sh programs in Linux are examples of shells, as are cmd.exe and Powershell on Windows.

- **Reverse shells** are when the target is forced/induced to execute code that connects *back* to attacker's computer. We would use tools to set up a *listener* which would be used to receive the connection. Reverse shells are a good way to bypass firewall rules that may prevent us from connecting to arbitrary ports on the target; however, the drawback is that, when receiving a shell from a machine across the internet, we would need to configure our own network to accept the shell.
- **Bind shells** are when the code executed on the target and is used to start a listener attached to a shell directly on the target. This would then be opened up to the internet, meaning we can connect to the port that the code has opened and obtain remote code execution that way. This has the advantage of not requiring any configuration on our own network, but may be prevented by firewalls protecting the target.

20. What is the difference between encoding, encryption, and hashing?

21. What is the use of Burp Collaborator? At what layer BurpSuites works on?

22. If in the XXE attack you are not able to read any file and not able to fetch any URL then what will you do?

23. What is a Billion laugh attack?

How to do OTP Bypass and Rate limit bypass?

What is Sub-Domain Takeover?

What is SQL Injection? What are the types of SQL Injection? How to exploit and mitigate it?

How to read/write files using SQL Injection? How to gain RCE using SQL Injection and what are the condition to gain it?

What is SAML? How to exploit and mitigate it?

Describe IDOR and explain how mitigating it is different from other access control vulnerabilities.

Name some ways TLS / SSL can be misconfigured.

What is DOM Clobbering and how can it be used to bypass (some) HTML sanitizers, resulting in XSS?

Describe three “403 Forbidden” bypass techniques.

Describe two output encoding techniques and the context in which they should be used to mitigate Cross-site Scripting.

In what ways could an open redirect be exploited?

What are JWKs and JKUs and how does their usage differ in JWTs?