**Week 7:**

**CSRF and IDOR Research**

**Cross-Site Request Forgery (CSRF)**

**DEFINITION:**

Cross-Site Request Forgery (CSRF) is an attack that forces authenticated users to submit a request to a Web application against which they are currently authenticated. CSRF attacks exploit the trust a Web application has in an authenticated user. A CSRF attack exploits a vulnerability in a Web application if it cannot differentiate between a request generated by an individual user and a request generated by a user without their consent.

**How does Cross-Site Request Forgery work?**

An attacker’s aim for carrying out a CSRF attack is to force the user to submit a state-changing request. Examples include:

* Submitting or deleting a record.
* Submitting a transaction.
* Purchasing a product.
* Changing a password.
* Sending a message.

Social engineering platforms are often used by attackers to launch a CSRF attack. This tricks the victim into clicking a URL that contains a maliciously crafted, unauthorized request for a particular Web application. The user’s browser then sends this maliciously crafted request to a targeted Web application. The request also includes any credentials related to the particular website (e.g., user session cookies). If the user is in an active session with a targeted Web application, the application treats this new request as an authorized request submitted by the user. Thus, the attacker succeeds in exploiting the Web application’s CSRF vulnerability.

A CSRF attack targets Web applications failing to differentiate between valid requests and forged requests controlled by an attacker. There are many ways for an attacker to try and exploit the CSRF vulnerability.

To give you an example, let’s say that Bob has an online banking account on samplebank.com. He regularly visits this site to conduct transactions with his friend Alice. Bob is unaware that samplebank.com is vulnerable to CSRF attacks. Meanwhile, an attacker aims to transfer $5,000 from Bob’s account by exploiting this vulnerability. To successfully launch this attack:

1. The attacker must build an exploit URL.
2. The attacker must also trick Bob into clicking the exploit URL.
3. Bob needs to have an active session with samplebank.com.

Let’s say that the online banking application is built using the GET method to submit a transfer request. As such, Bob’s request to transfer $500 to Alice (with account number 213367) might look like this:

*GET https://samplebank.com/onlinebanking/transfer?amount=500&accountNumber=213367 HTTP/1.1*

Aligning with the first requirement to successfully launch a CSRF attack, an attacker must craft a malicious URL to transfer $5,000 to account 425654:

https://samplebank.com/onlinebanking/transfer?amount=5000&accountNumber=425654

Using various social engineering attack methods, an attacker can trick Bob into loading the malicious URL. This can be achieved in various ways. For instance, including malicious HTML image elements onto forms, placing a malicious URL on pages that are often accessed by users while logged into the application, or sending a malicious URL through email.

The following is an example of a disguised URL:

*<img src  = “https://samplebank.com/onlinebanking/transfer?amount=5000&accountNumber=425654” width=“0” height= “0”>*

Consider the scenario that includes an image tag in an attacker-crafted email to Bob. Upon receiving it, Bob’s browser application opens this URL automatically—without human intervention. As a result, without Bob’s permission, a malicious request is sent to the online banking application. If Bob has an active session with samplebank.com, the application would treat this as an authorized amount transfer request coming from Bob. It would then transfer the amount to the account specified by an attacker.

There are some limitations. To carry out a successful CSRF attack, consider the following:

* The success of a CSRF attack depends on a user’s session with a vulnerable application. The attack will only be successful if the user is in an active session with the vulnerable application.
* An attacker must find a valid URL to maliciously craft. The URL needs to have a state-changing effect on the target application.
* An attacker also needs to find the right values for the URL parameters. Otherwise, the target application might reject the malicious request.

**How can an application prevent a Cross-Site Request Forgery attack?**

To defeat a CSRF attack, applications need a way to determine if the HTTP request is legitimately generated via the application’s user interface. The best way to achieve this is through a CSRF token. A CSRF token is a secure random token (e.g., synchronizer token or challenge token) that is used to prevent CSRF attacks. The token needs to be unique per user session and should be of large random value to make it difficult to guess.

A CSRF secure application assigns a unique CSRF token for every user session. These tokens are inserted within hidden parameters of HTML forms related to critical server-side operations. They are then sent to client browsers.

It is the application team’s responsibility to identify which server-side operations are sensitive. The CSRF tokens must be a part of the HTML form—not stored in session cookies. The easiest way to add a non-predictable parameter is to use a secure hash function (e.g., SHA-2) to hash the user’s session ID. To ensure randomness, the tokens must be generated by a cryptographically secure random number generator.

Whenever a user invokes these critical operations, a request generated by the browser must include the associated CSRF token. This will be used by the application server to verify the legitimacy of the end-user request. The application server rejects the request if the CSRF token fails to match the test.

**Insecure Direct Object Reference (IDOR)**

An Insecure Direct Object Reference (IDOR) is a type of security vulnerability that occurs when an application provides direct access to objects based on user-supplied input. In simpler terms, it happens when an application exposes internal implementation objects to users and fails to properly check whether the user has the appropriate permissions to access those objects.

**To mitigate CSRF attacks, consider implementing the following techniques:**

**Anti-CSRF Tokens:** Include a unique and unpredictable token in each HTML form, and ensure that the token is validated on the server side for each submitted request. The token should be associated with the user's session and embedded within the form or included as a header.

**SameSite Cookies Attribute:** Set the SameSite attribute for cookies to "Strict" or "Lax" to control when cookies are sent with cross-origin requests. This helps prevent CSRF attacks by reducing the chances of a malicious website making unauthorized requests on behalf of the user.

**Referrer Policy**: Set an appropriate referrer policy for your web application to control how much information is included in the HTTP Referer header. Use the "strict-origin-when-cross-origin" policy to ensure that the Referer header is sent only when the request is made from the same origin**.**

**Custom Request Headers:** Include custom headers in your requests that are expected by the server and cannot be easily replicated by an attacker. Verify the presence of these custom headers on the server side to ensure that the request is legitimate.

**Examples of IDOR**

One common example of IDOR is accessing someone else’s private files or records just by changing a number in the URL. Say an app stores files with URLs like example.com/files/1, example.com/files/2, etc. If there’s no check to ensure you own file #2, you can access it just by entering that URL. Another IDOR vulnerability allows unauthorized access to sensitive data by guessing or brute-forcing the application's ID. For instance, an app may expose records at example.com/records/1, example.com/records/2, etc. An attacker could write a script to cycle through numbers until they find records belonging to other users. Some IDOR issues allow the manipulation of data through direct object references. For example, an app might have a “delete profile” feature at example.com/profiles/delete/1. By changing the 1 to another profile ID, an attacker could delete other users’ profiles. The bottom line is that if an application uses direct references to objects like files, records, or accounts, it needs to verify the user has permission to access or modify those objects. Failing to do so leaves the door open for IDOR, allowing unauthorized access and manipulation of sensitive data. Be on the lookout for these vulnerabilities in your software, and pressure developers to patch IDOR risks whenever they’re discovered. Our personal information is too valuable to leave exposed!

**How to Prevent IDOR Vulnerabilities**

To prevent IDOR vulnerabilities in your application, there are a few key steps you can take:

**Access Control**

Implement strong access control checks for all API endpoints and resources. Refrain from assuming that just because users can access one resource, they should have access to related resources. Explicitly check users' permissions for each request and verify their authorization for that specific resource.

**Randomize Identifiers**

Use randomly generated, opaque identifiers for resources instead of incremental IDs or names that can be easily guessed. This makes it much harder for an attacker to access resources by manipulating IDs.

**Validate User Input**

Carefully validate all user input for API requests to ensure the input matches the expected format. Don't trust that a user ID, file name, or other parameter is valid just because it was provided in the request. Malicious users may tamper with these values to try and access unauthorized data.  
**Audit Your Application**

Regularly audit your application for IDOR vulnerabilities, especially after any major changes. IDOR issues often arise from subtle oversights, so a fresh set of eyes can help identify any weak spots in your access control or validation logic. You can also use automated scanners to help detect potential IDOR flaws.

**Stay Up-to-Date**

Keep your application framework and all dependencies up-to-date with the latest versions. IDOR vulnerabilities are frequently disclosed and patched in third-party software libraries. Updating promptly helps ensure you have the latest fixes for any issues. Following security best practices like these can help reduce the risk of IDOR and keep your users' data safe. Be vigilant and proactively search for any weaknesses that could be exploited.

**To mitigate IDOR vulnerabilities, consider implementing the following techniques:**

**Access Controls and Authorization**: Implement proper access controls to ensure that users can only access the data and resources they are authorized to. Use role-based access control (RBAC) to define and enforce permissions based on user roles.

**Use Indirect Object References:** Instead of directly referencing sensitive objects in URLs or parameters, use indirect references like tokens, hashes, or identifiers that are more difficult for attackers to guess.

**Validate User Input**: Validate and sanitize all user inputs, including parameters in URLs, forms, and cookies. Check the validity of input against a whitelist of allowed values.

**Session Management:** Implement secure session management practices to prevent attackers from manipulating session data to gain unauthorized access. Regularly rotate session tokens and invalidate sessions after logout or inactivity.