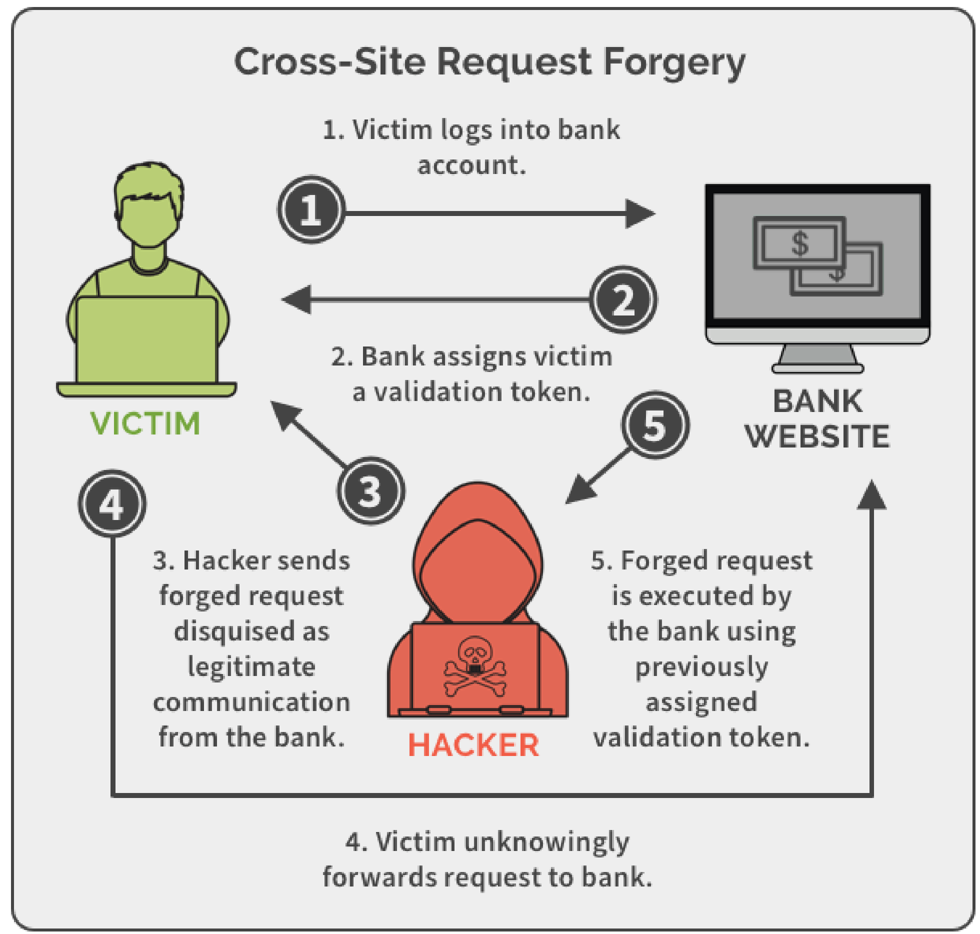
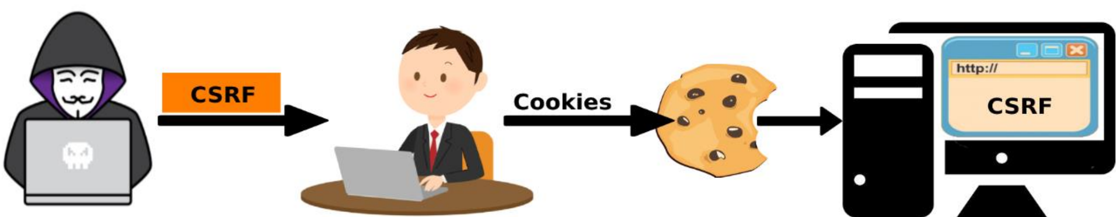
CROSS-SITE REQUEST FORGERY



What is CSRF?

A online security flaw known as Cross-Site Request Forgery (CSRF) enables a hacker to deceive a user into carrying out undesirable actions on a website where they have verified their identity. CSRF forges requests without the user's permission in order to take advantage of the confidence that a web application places in their browser. In order to carry out illegal operations on the online application, such as altering account information, transferring money, or carrying out other tasks that the authenticated user is allowed to carry out, the attacker coerces the victim into sending a maliciously designed request.

How does a CSRF Attack work?



1. User Authentication on Targeted Website: The user creates a session after logging onto a website (like a banking site). As long as the user is logged in, the browser automatically includes a cookie from the server with the session information with each request to the website.
   For example, when we log into ‘bank.com’, the browser stores a session cookie as-
   Set-Cookie: sessionid=abcd1234; Domain=bank.com

2. Attacker Prepares a Malicious Request: Knowing that the user has authenticated on a particular website, the attacker creates a fraudulent request that causes the website to perform an action (such sending money to the attacker's account).
   - The malicious request is embedded by the attacker into a forged link or a hidden form. An example of a request for a money transfer could be like follows:
     <form action="https://bank.com/transfer" method="POST">
     <input type="hidden" name="amount" value="1000">
     <input type="hidden" name="account" value="attacker-account">
     <input type="submit" value="Submit">
     </form>
   - Alternate link the attacker can use is:
     <img src="https://bank.com/transfer?amount=1000&account=attacker-account">
     This will trigger the request when the image will load.

3. Tricking the User: By deceiving the user, the attacker sends the forged request unintentionally. This can be accomplished by inserting the malicious code into a third-party website, phishing link, or email. The request is immediately sent to the server as soon as the user clicks the link or loads the webpage that contains the hidden form or picture.
4. Execution of Malicious Request: The user's browser adds the session cookie when requesting the desired website because they have already successfully authenticated. When a request like this one arrives to the server with a valid session ID and comes from the user's browser with proper credentials, the server considers it to be authentic.
5. Server Processes the Request: Assuming that the request was made by the user, the server processes it and carries out the requested action (such as moving money from the victim's account to the attacker's).

EXAMPLE SCENARIO:

Imagine a situation when a person is both visiting other websites and logged into their bank.com online banking account. The following form is present on the malicious webpage that the attacker is hosting:

```
<form action="https://bank.com/transfer" method="POST">

  <input type="hidden" name="amount" value="5000">

  <input type="hidden" name="account" value="attacker123">

</form>
```

The form is configured to submit a request for a money transfer to the bank. While remaining logged in to the banking website, the victim unintentionally accesses the malicious website. The malicious page loads quickly and uses the victim's session to submit the form to the bank, prompting the bank to transfer $5,000 from the user's account to the attacker's account.
Since the request originates from an authenticated session, the banking website treats it as though the user has voluntarily made it, thus there's no need to be suspicious.

A REAL WORLD EXAMPLE:

A CSRF attack happened in the real world in Gmail (2007). A CSRF exploit was created by attackers to alter Gmail accounts' forwarding configurations. An image tag was embedded in a rogue website to carry out the attack:

‘ <img src="https://mail.google.com/mail/?view=up&act=cf&forward=attacker@gmail.com"> ’

The request was issued using the authenticating cookies of the logged-in Gmail user, and Gmail inadvertently altered the user's forwarding address to the attacker's email.

How to Test For CSRF Vulnerability?

1. Review Application Behaviour: Verify whether sensitive actions—like changing a password or transferring money—need to be authenticated and whether they're started by sending HTTP POST requests.
2. Verify Use of Anti-CSRF Tokens: Verify if a distinct anti-CSRF token that is updated for every session or request is included in every sensitive request. Check to determine if the server accepts the request by removing or changing the token.
3. Check Referer Header: Examine whether the application verifies the request's source by looking at the ‘Referer’ header in incoming requests.
4. Create Malicious HTML/JS Code: By hiding the susceptible action in a script or hidden form, you can simulate a cross-site request forgery (CSRF) attack and see if the server handles it when the user logs in.

   For example, create a form like-

   ```
   <form action="https://victim-website.com/transfer" method="POST">
     <input type="hidden" name="amount" value="1000">
     <input type="hidden" name="account" value="attacker-account">
     <input type="submit" value="Submit request">
   </form>
   ```

   The server is probably vulnerable if it handles this request without verifying the origin.

Example:

Assume for the moment that we are testing an application that allows users to add things to their carts. Normally, a POST request such as this would be needed:

POST /add-to-cart HTTP/1.1

Host: shop.com

Cookie: sessionid=abcd1234

If forming a GET request allows us to carry out the same action:

‘ <img src="https://shop.com/add-to-cart?item=123"> ’

and the item is successfully added to the cart, the application probably has CSRF vulnerability.

Impact of CSRF

- Financial Loss: Unauthorized fund transfers resulting from CSRF can seriously harm one's finances.
- Account Compromise: Account takeover can occur when hackers alter email addresses, passwords, or account settings.
- Data Integrity Breach: Malicious alterations to user data, such as preferences or profile information.
- Service Disruption: Administrator rights can be abused by attackers to stop or interfere with services.

Mitigation Measures

1. Anti-CSRF Tokens: Introduce distinct anti-CSRF tokens for every form submission or session. Requests should not be processed until the server has validated the token.
   Example- Including a hidden input field in forms:
   <input type="hidden" name="csrf_token" value="generated_token">

2. SameSite Cookie Attribute: Utilize the ‘SameSite’ property in cookies to prevent cross-origin requests from being sent with them.
   Set-Cookie: sessionid=abcd1234; SameSite=Strict

3. Double Submit Cookie: Adopt a method wherein a cookie and a form parameter are used to store an anti-CSRF token. For the request to be considered valid, both values must match.

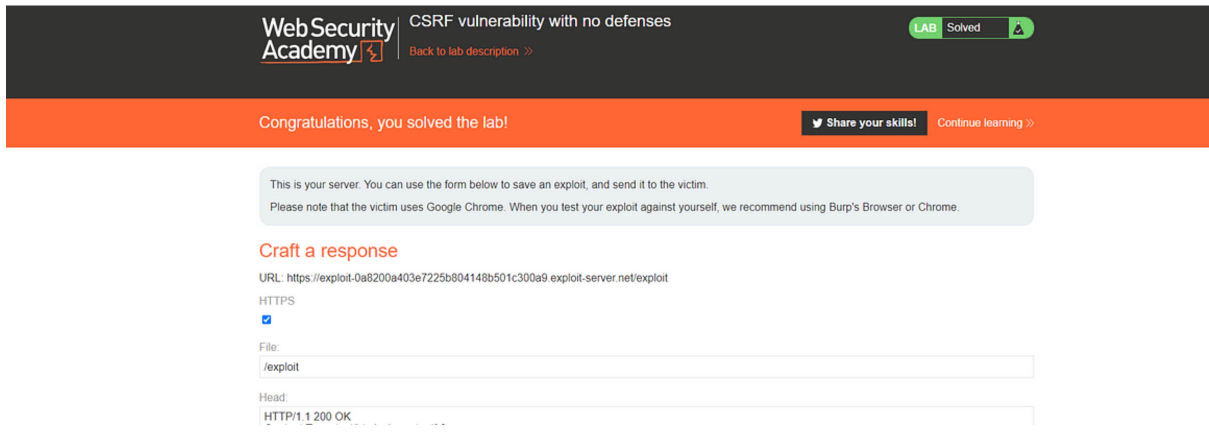
4. Referer and Origin Header Validation: Verify that requests are originating from a reliable domain by looking at the "Referer" or "Origin header."

5. User Interaction Verification: Before processing sensitive actions like password changes or fund transfers, require re-authentication or CAPTCHA.

6. Framework Security Features: Use the built-in security capabilities that contemporary web frameworks (like Spring Security and Django's CSRF middleware) offer.

A major online application vulnerability called Cross-Site Request Forgery (CSRF) may allow a user to take unapproved actions. Developers should use security headers, use anti-CSRF tokens, and impose stringent validation for sensitive actions in order to stop these attacks. To guarantee application safety, a thorough web security testing approach should include regular CSRF testing.

PRACTICAL TASKS DONE



Web Security Academy
CSRF where token validation depends on request method
Back to lab description
LAB Solved
Congratulations, you solved the lab!
Share your skills!
Continue learning
Craft a response
URL: https://exploit-0a3d00b80335a2afc226a786019e0081.exploit-server.net/exploit

REFERENCES

1. https://owasp.org/www-community/attacks/csrf
2. https://portswigger.net/web-security/csrf
3. https://brightsec.com/blog/cross-site-request-forgery-csrf/
4. https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/06-Session_Management_Testing/05-Testing_for_Cross_Site_Request_Forgery
5. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html
6. https://brightsec.com/blog/csrf-mitigation/