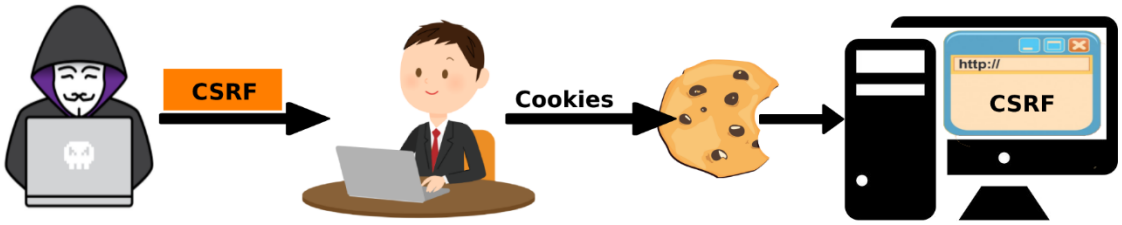
CSRF

Cross Site Request Forgery



What is CSRF (Cross Site Request Forgery)?

Cross-site request forgery (also known as CSRF) is a web vulnerability that allows attackers to trick users into performing unwanted actions. This allows an attacker to partially bypass the same-origin policy, which is intended to prevent different websites from interfering with each other.

It gets its long name from:

- "Cross-Site": originates on one site but performs an action on another
- "Request Forgery": it is not a genuine user request

CSRF attacks are particularly effective when the target site pre-authenticates the user's browser, meaning the user is already logged into the target site using the same browser that loads the attack page. When a request is sent to a website, the browser sends all the cookies stored for that website with the request. If these cookies contain something like the setting "logged_in=true", the request may be seen by the target server as coming from the currently logged in user. This exploit takes advantage of the fact that the server trusts the user's browser.

Attackers typically use CSRF attacks to make state-change requests. Although the request performs the action, it is still a request from the user's browser, so the attacker cannot see the results of the request.

How does the attack work?

There are numerous ways to trick end users into loading or submitting information to a web application. To carry out an attack, you must first understand how to generate a valid malicious request for execution by the victim.

Consider the following example.

Alice wants to send Bob her $100 using her Bank.com web application, which has a CSRF vulnerability. The attacker, Maria, tries to get Alice to send money to Maria on her behalf. The attack includes the following steps:

1. Building an exploit URL or script
2. Tricking Alice into executing the action with Social Engineering

GET scenario

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

```
GET http://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1
```

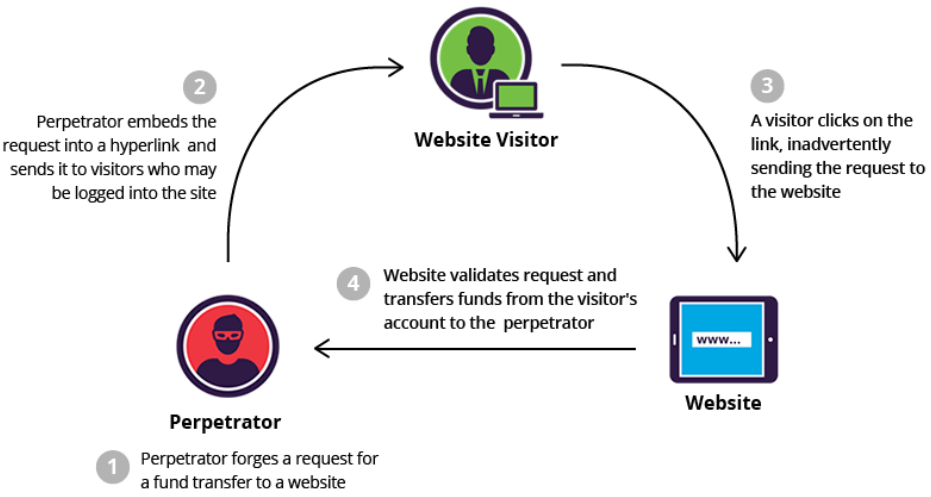
Maria(attacker) decides to exploit this vulnerability in the web application and uses Alice as a victim. Maria begins by creating the following exploit URL: This transfers her $100,000 from Alice's account to Maria's account. Maria takes her original command URL

and replaces her recipient's name with herself. This simultaneously significantly increases her remittance amount.

```
http://bank.com/transfer.do?acct=MARIA&amount=100000
```

The social engineering aspect of the attack tricks Alice into loading this URL when Alice is logged into the bank application. This is usually done with one of the following techniques:

- sending an unsolicited email with HTML content
- planting an exploit URL or script on pages that are likely to be visited by the victim while they are also doing online banking



The exploit URL can be disguised as an ordinary link, encouraging the victim to click it:

```
<a
href="http://bank.com/transfer.do?acct=MARIA&amount=100000"
>View my Pictures!</a>
```

Or as a 0x0 fake image:

```
<img
src="http://bank.com/transfer.do?acct=MARIA&amount=100000"
width="0" height="0" border="0">
```

If this image tag were included in the email, Alice wouldn't see anything. However, the browser will still submit the request to bank.com without any visual indication that the transfer has taken place.

POST scenario

The only difference between GET and POST attacks is how the attack is being executed by the victim. Let's assume the bank now uses POST and the vulnerable request looks like this:

```
POST http://bank.com/transfer.do HTTP/1.1

acct=BOB&amount=100
```

Such a request cannot be delivered using standard A or IMG tags, but can be delivered using a FORM tags:

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

<input type="hidden" name="acct" value="MARIA"/>

<input type="hidden" name="amount" value="100000"/>

<input type="submit" value="View my pictures"/>

</form>
```

This form requires the user to click a submit button. However, this can also be done automatically using JavaScript.

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

Testing for Cross-Site Request Forgery vulnerabilities

1. Manual Method

   To discover if a session is insecure, you need to examine your application's session. An application is vulnerable if session management occurs on the user's side and information is made available to the browser. "Client-side values" refers to HTTP authentication credentials and cookies.

   Resources that can be accessed via HTTP GET requests are definitely vulnerable, but POST requests are typically automated via JavaScript and are exposed to vulnerabilities. Therefore, using POST alone is not sufficient to address the occurrence of CSRF vulnerabilities.

2. Automated Tools

   I. OWASP ZAP

      ZAP detects anti-CSRF tokens solely by attribute names – that is considered to be anti CSRF tokens and is configured using the Anti CSRF in options. When ZAP detects these tokens it records the token value and which URL generated the token.

II. CSRF Tester

CSRF Tester is a project by OWASP, created by a group of developers for developers, to verify the integrity of HTTP requests in their web applications. CSRF Tester provides a PHP library and an Apache Module for cautious mitigation.

III. Pinata-csrf-tool

Intended to be used by advanced application security professionals. It generates the proof of concept CSRF HTML given an HTTP request to automatically discover whether it is a GET or a POST request with further validation for standard POST and Multipart/form POST. The tool creates HTML corresponding to the type of the request.

impact of a CSRF attack

1. damaged client relationships.
2. unauthorized fund transfers.
3. alter passwords.
4. even data theft — including stolen session cookies.

Prevention for CSRF Attack

1. Make sure your web application has CSRF protection
2. Use advanced validation techniques to reduce CSRF
3. Conduct regular web application security tests to identify CSRF

References

1) https://sucuri.net/guides/what-is-csrf/
2) https://owasp.org/www-community/attacks/csrf#:~:text=CSRF%20attacks%20target%20functionality%20that,the%20response%2C%20the%20victim%20does.
3) https://portswigger.net/web-security/csrf