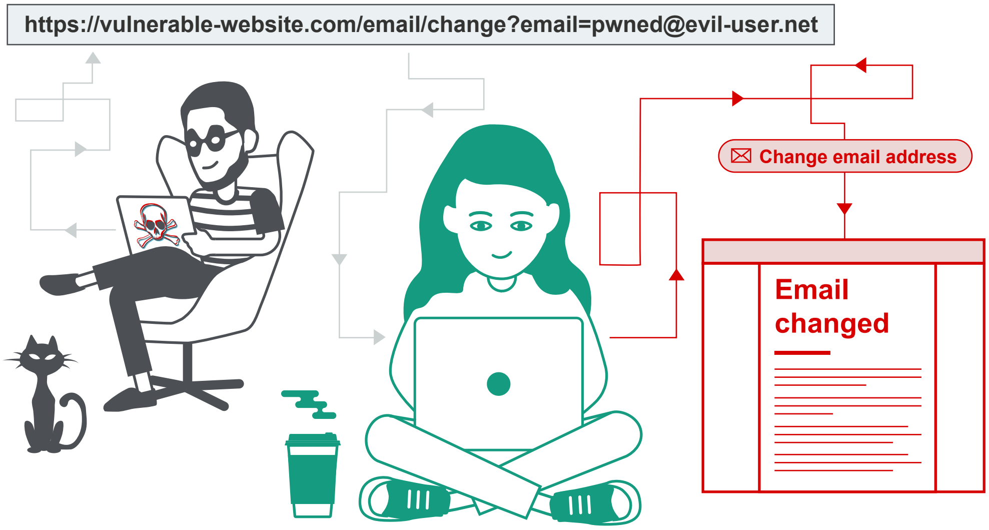
Cross-site request forgery (CSRF)

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.


https://vulnerable-website.com/email/change?email=pwned@evil-user.net
Change email address
Email changed

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
```

In this topic
CSRF »
CSRF tokens »
XSS vs CSRF »
SameSite cookies »

All topics
SQL injection »
XSS »
CSRF »
Clickjacking »
DOM-based »
CORS »
XXE »
SSRF »
Request smuggling »
Command injection »
Directory traversal »
Access control »
Web cache poisoning »
WebSockets »

```
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 »

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:

```
<img src="https://vulnerable-website.com/email/change?email=pwned@evil-user.net">
```

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

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 »

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 »

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 »

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 »

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 »

Validation of Referer can be circumvented

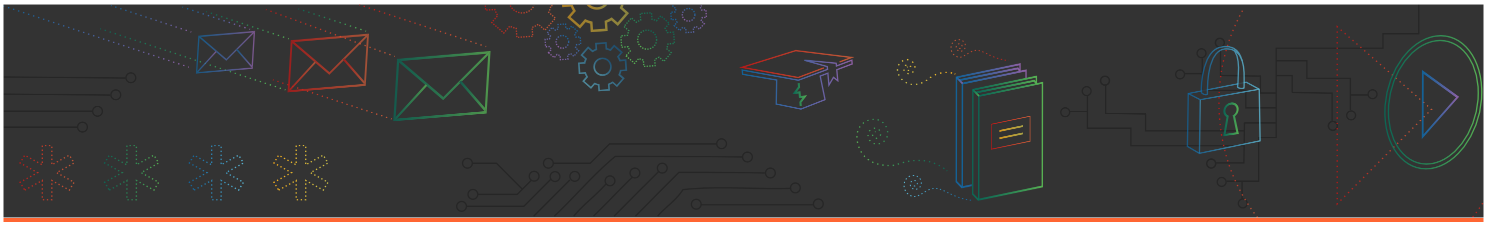
Some applications validate the Referer header in a naive way that can be bypassed. For example, if the application simply validates that the Referer contains its own domain name, then the attacker can place the required value elsewhere in the URL:

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

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
```

LAB CSRF with broken Referer validation »



Burp Suite

Web vulnerability scanner
Burp Suite Editions
Release Notes

Vulnerabilities

Cross-site scripting (XSS)
SQL injection
Cross-site request forgery
XML external entity injection
Directory traversal
Server-side request forgery

Customers

Organizations
Testers
Developers

Company

About
PortSwigger News
Careers
Contact
Legal
Privacy Notice

Insights

Web Security Academy
Blog
Research
The Daily Swig

PORTSWIGGER
WEB SECURITY

Follow us