Session 7

CSFR, Command Injection

Cross-site request forgery (CSRF)

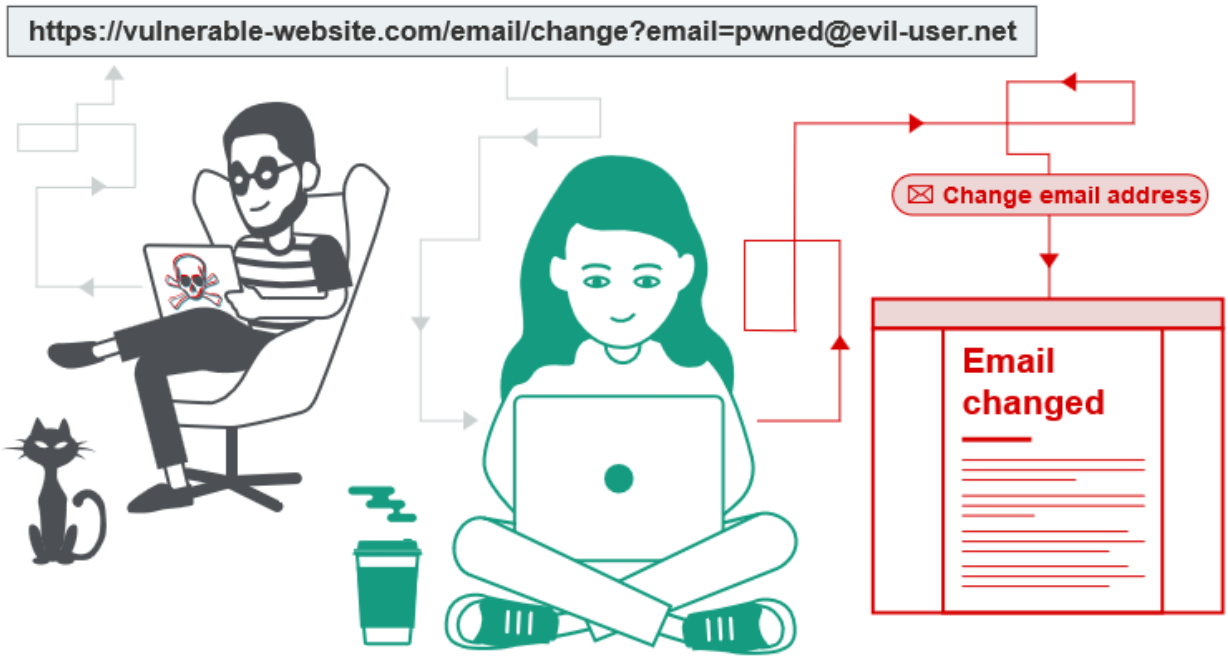
In this section, we'll explain what cross-site request forgery is, describe some examples of common CSRF vulnerabilities, and explain how to prevent CSRF attacks.

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.
-

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.

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">

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.

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.

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
```

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
```

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.

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.

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.

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
```

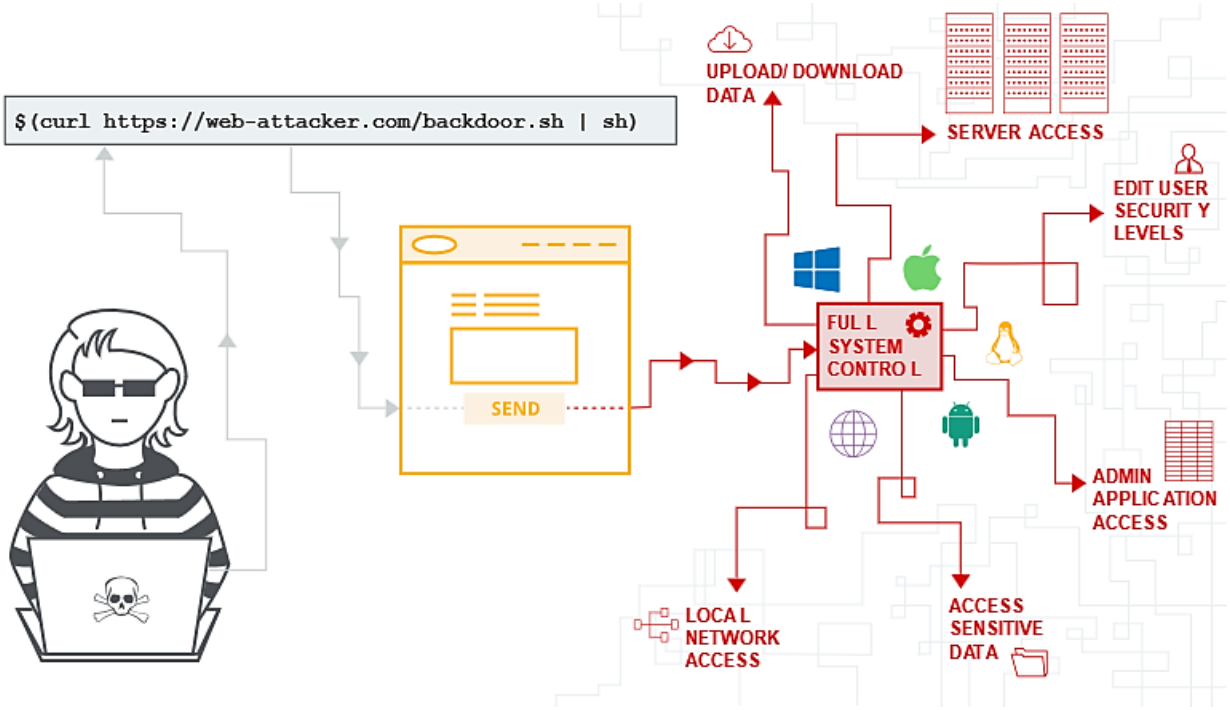
In this situation, the attacker can again perform a CSRF attack if the web site contains any cookie setting functionality. Here, the attacker doesn't need to obtain a valid token of their own. They simply invent a token (perhaps in the required format, if that is being checked), leverage the cookie-setting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

OS command injection

In this section, we'll explain what OS command injection is, describe how vulnerabilities can be detected and exploited, spell out some useful commands and techniques for different operating systems, and summarize how to prevent OS command injection.

What is OS command injection?

OS command injection (also known as shell injection) is a web security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server that is running an application, and typically fully compromise the application and all its data. Very often, an attacker can leverage an OS command injection vulnerability to compromise other parts of the hosting infrastructure, exploiting trust relationships to pivot the attack to other systems within the organization.



Executing arbitrary commands

Consider a shopping application that lets the user view whether an item is in stock in a particular store. This information is accessed via a URL like:

https://insecure-website.com/stockStatus?productID=381&storeID=29

To provide the stock information, the application must query various legacy systems. For historical reasons, the functionality is implemented by calling out to a shell command with the product and store IDs as arguments:

stockreport.pl 381 29

This command outputs the stock status for the specified item, which is returned to the user.

Since the application implements no defenses against OS command injection, an attacker can submit the following input to execute an arbitrary command:

& echo aiwefwlguh &

If this input is submitted in the productID parameter, then the command executed by the application is:

stockreport.pl & echo aiwefwlguh & 29

The echo command simply causes the supplied string to be echoed in the output, and is a useful way to test for some types of OS command injection. The & character is a shell command separator, and so what gets executed is actually three separate commands one after another. As a result, the output returned to the user is:

Error - productID was not provided
aiwefwlguh
29: command not found

The three lines of output demonstrate that:

- The original stockreport.pl command was executed without its expected arguments, and so returned an error message.
- The injected echo command was executed, and the supplied string was echoed in the output.
- The original argument 29 was executed as a command, which caused an error.

Placing the additional command separator & after the injected command is generally useful because it separates the injected command from whatever follows the injection point. This reduces the likelihood that what follows will prevent the injected command from executing.

Useful commands

When you have identified an OS command injection vulnerability, it is generally useful to execute some initial commands to obtain information about the system that you have compromised. Below is a summary of some commands that are useful on Linux and Windows platforms:

Purpose of command	Linux	Windows
Name of current user	whoami	whoami
Operating system	uname -a	ver
Network configuration	ifconfig	ipconfig /all
Network connections	netstat -an	netstat -an
Running processes	ps -ef	tasklist

Blind OS command injection vulnerabilities

Many instances of OS command injection are blind vulnerabilities. This means that the application does not return the output from the command within its HTTP response. Blind vulnerabilities can still be exploited, but different techniques are required.

Consider a web site that lets users submit feedback about the site. The user enters their email address and feedback message. The server-side application then generates an email to a site administrator containing the feedback. To do this, it calls out to the mail program with the submitted details. For example:

```
mail -s "This site is great" -aFrom:peter@normal-user.net feedback@vulnerable-website.com
```

The output from the mail command (if any) is not returned in the application's responses, and so using the echo payload would not be effective. In this situation, you can use a variety of other techniques to detect and exploit a vulnerability.

Detecting blind OS command injection using time delays

You can use an injected command that will trigger a time delay, allowing you to confirm that the command was executed based on the time that the application takes to respond. The ping command is an effective way to do this, as it lets you specify the number of ICMP packets to send, and therefore the time taken for the command to run:

```
& ping -c 10 127.0.0.1 &
```

This command will cause the application to ping its loopback network adapter for 10 seconds.

Exploiting blind OS command injection by redirecting output

You can redirect the output from the injected command into a file within the web root that you can then retrieve using your browser. For example, if the application serves static resources from the filesystem location /var/www/static, then you can submit the following input:

```
& whoami > /var/www/static/whoami.txt &
```

The > character sends the output from the whoami command to the specified file. You can then use your browser to fetch https://vulnerable-website.com/whoami.txt to retrieve the file, and view the output from the injected command.