Vulnerabilities

XSS = Cross-site scripting

javascript injection into a website
caused by improper sanitization of user input on the page

Types

Stored (persistent)

when the script is permanently stored for example in the database and included on the page

Reflected (non-persistent)

when the script is reflected from the web server and included the page's source code
(for example, a parameter from the URL is reflected on the website)

DOM-based

happens on the client-side when the payload modifies the DOM in some way
(for example, using eval(), innerHTML, setting javascript to <a href=>)

Blind

a variant of stored XSS, when the payload is executed on a different page
(for example XSS in the contact form that is executed in the admin dashboard)

Prevention

Sanitize user input

- ➔ the most important prevention for all kinds of attacks
- ➔ be aware of in what context the input is (tag contents, tag attribute, script tag) and escape it accordingly
- ➔ for example in PHP use `htmlspecialchars` (escape angle brackets, quotes, etc.)

Content Security Policy

- ➔ header sent in page's HTTP response, whitelisting allowed URLs
- ➔ in CSP v2 we can use the nonce parameter with a random value that is generated every on every request, and is appended to every script that we want to run

Trusted Types

- ➔ is a CSP directive to prevent DOM XSS attacks
- ➔ by default disables potentially dangerous functions, for example: `eval, innerHTML, insertAdjacentHTML, document.write, setting <iframe srcdoc>`

Correct Content-Type for JSON requests

- ➔ always set an appropriate Content-Type header for the request (`application/json; charset=utf-8`)

- ➔ using `Content-Type: text/html` for a JSON HTTP response will execute the HTML when the request is viewed directly

CSRF = Cross-site request forgery

an attack that changes data on behalf of an authenticated user
works because the browser automatically appends all cookies to the request

Examples

Logout using img tag (low severity)

```
<img src="//example.com/logout">
```

Change user data

```
<form action="https://example.com/api/changeProfile" method="POST">
      <input type="text" name="name" value="New name">
      <input type="text" name="email" value="new@example.com">
      <input type="text" name="address" value="123 Road">
</form>
<script>
      document.forms[0].submit();
</script>
```

Prevention

CSRF tokens

- ➔ generated on the server and sent to the user
- ➔ appended in the request body (for example, input type="hidden") or in the headers
- ➔ the server then verifies the token

SameSite cookies

- ➔ mitigate the risks of CSRF by marking cookies to be sent only from the same site
- ➔ supported in most browsers
- ➔ doesn’t replace CSRF tokens, only as an additional layer of defense

Verify the Origin

- ➔ check if the Origin header in the request matches the correct origin of the website

Verify the Content-Type

- ➔ using an HTML form you can only submit GET requests and POST requests
- ➔ form enctype cannot be set to `application/json`, so if you’re sending a JSON request, verify if the Content-Type is correct
- ➔ also only serves as an additional layer of defense

SQL injection

an attack where untrusted user input is inserted into an SQL query
can enable the attacker to read, modify or delete data from the database

Example

User submits a login form containing a `username` and `password` field. It is then submitted via an HTTP request and the server builds an SQL query by concatenating the query with these fields.

```
"SELECT * FROM users WHERE username='" + username + "' AND hash = '" + hash
+ "'"
```

If the user submits this username: `admin' AND 1=1--`, the query becomes:

```
"SELECT * FROM users WHERE username='admin' AND 1=1--' AND hash = '123'"
```
which is equivalent to:
```
"SELECT * FROM users WHERE username='admin'"
```

Since the rest of the query is commented out using two dashes, the user will be able to log in as ‘admin’.

Types

Blind SQLi

when the server doesn’t directly show the output or error of the query
can be exploited for example by using a SLEEP function

Second-order SQLi

when a user-submitted value containing an SQL injection payload is first correctly stored in a database, but then when building another query it is incorrectly deemed safe and unsafely used in an SQL query

Prevention

Prepared statements

Using prepared statements instead of directly concatenating the query is the best approach to prevent SQLi.

Least privilege

Set the least possible privilege for the database user to prevent eventual damages by an SQLi.

LFI = Local File Inclusion

vulnerability allowing the attacker to include a file from the server because of incorrectly sanitized/validated input

Example

```
<?php
include($_GET['url']);
```

https://example.com/?url=/etc/passwd

Prevention

Not using user-input directly for resolving a file

Instead of directly including the filename passed from the user, have a list of the files and load the file by an assigned unique token/identifier.

XXE = XML External Entity

attack on an application that parses XML, allowing, for example, to include files from the filesystem or perform SSRF

Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>
```

Prevention

Correctly configure the XML parser

Disable DTD (Document Type Definition) in the XML parser and set other recommended configurations for the specific parser.

Security Misconfiguration

occurs when the server of a web application is not correctly configured, leading to various security flaws

Examples

Default credentials

Default username and password like `admin:admin` may allow attackers to gain access to the system.

Directory listing is enabled

An attacker will be able to see the list of all files and directories in the current directory.

Outdated software

Using outdated software with known security issues may allow attackers to use those vulnerabilities.

Debugging enabled

Showing verbose error messages / stack traces / debugging information may help attackers to misuse this information.

Prevention

Change the credentials

Update the default credentials with a safe password.

Disable directory listing

Disable listing files in directories/buckets by default.

Update the software

Regularly install the latest updates and patches.

Disable verbose error messages

Disable debugging or error messages that might reveal unwanted information.

IDOR = Insecure Direct Object Reference

combined with an access control vulnerability allows the attacker to access an object by directly providing its identifier, for example using incremental IDs

Examples

Incremental IDs

Using incremental IDs and not checking if the user has permission to access them.

```
GET /api/users/12345
```

```
POST /api/users/12345
Content-Type: application/json
…

{
	"email": "evil@example.com"
}
```

Prevention

Enforce access control

Implement user authorization to check if the user has permission to access/modify the resource.