Report for the findings of the link

https://0a670012039d2c28808817c0002c0045.web-security-academy.net/

Sql Injection

What is SQL injection (SQLi)?

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

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

What is the impact of a successful SQL injection attack?

A successful SQL injection attack can result in unauthorized access to sensitive data, such as:

Passwords.

Credit card details.

Personal user information.

SQL injection attacks have been used in many high-profile data breaches over the years. These have caused reputational damage and regulatory fines. In some cases, an attacker can obtain a persistent backdoor into an organization's systems, leading to a long-term compromise that can go unnoticed for an extended period.

How to detect SQL injection vulnerabilities

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

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

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

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

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

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

SQL injection examples

There are lots of SQL injection vulnerabilities, attacks, and techniques, that occur in different situations. Some common SQL injection examples include:

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

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

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

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

Blind SQL injection vulnerabilities

Many instances of SQL injection are blind vulnerabilities. This means that the application does not return the results of the SQL query or the details of any database errors within its responses. Blind vulnerabilities can still be exploited to access unauthorized data, but the techniques involved are generally more complicated and difficult to perform.

The following techniques can be used to exploit blind SQL injection vulnerabilities, depending on the nature of the vulnerability and the database involved:

You can change the logic of the query to trigger a detectable difference in the application's response depending on the truth of a single condition. This might involve injecting a new condition into some Boolean logic, or conditionally triggering an error such as a divide-by-zero.

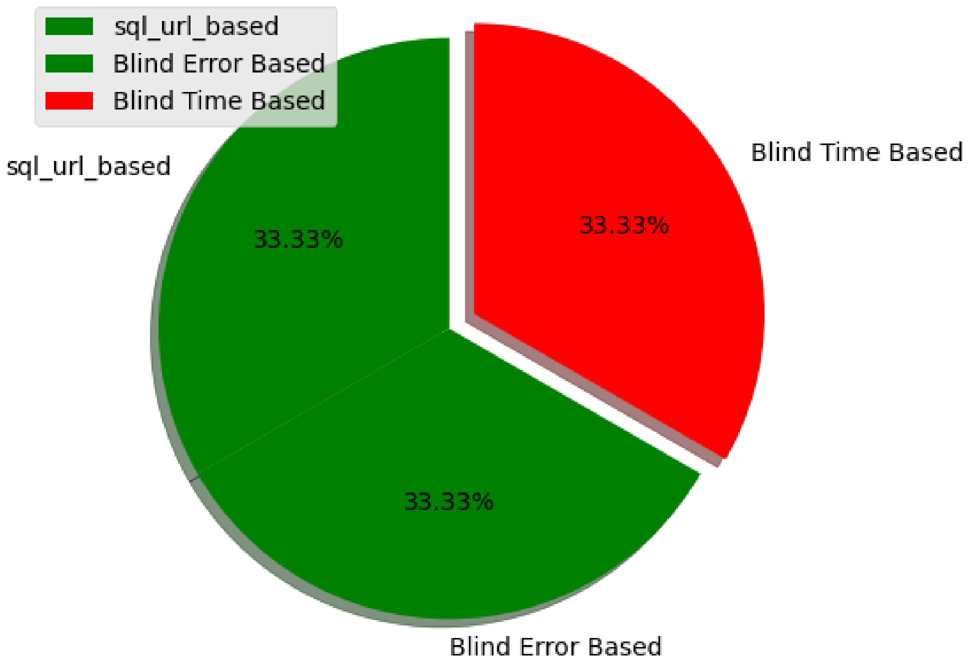
You can conditionally trigger a time delay in the processing of the query. This enables you to infer the truth of the condition based on the time that the application takes to respond.

You can trigger an out-of-band network interaction, using OAST techniques. This technique is extremely powerful and works in situations where the other techniques do not. Often, you can directly exfiltrate data via the out-of-band channel. For example, you can place the data into a DNS lookup for a domain that you control.

Sql injection detected:

Target URL: https://0a670012039d2c28808817c0002c0045.web-security-academy.net/

Blind Time based sql injection possible https://0a670012039d2c28808817c0002c0045.web-security-academy.net/ -> Using payloads -> ["1' AND 1337=(SELECT 1337 FROM PG_SLEEP(10)) AND '1337'='1337"]



Mitigation:

How to prevent SQL injection

You can prevent most instances of SQL injection using parameterized queries instead of string concatenation within the query. These parameterized queries are also know as "prepared statements".

The following code is vulnerable to SQL injection because the user input is concatenated directly into the query:

```
String query = "SELECT * FROM products WHERE category = '"+ input + "'";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(query);
```

You can rewrite this code in a way that prevents the user input from interfering with the query structure:

```
PreparedStatement statement = connection.prepareStatement("SELECT * FROM products WHERE category = ?");
statement.setString(1, input);
ResultSet resultSet = statement.executeQuery();
```

You can use parameterized queries for any situation where untrusted input appears as data within the query, including the WHERE clause and values in an INSERT or UPDATE statement. They can't be used to handle untrusted input in other parts of the query, such as table or column names, or the ORDER BY clause. Application functionality that places untrusted data into these parts of the query needs to take a different approach, such as:

Whitelisting permitted input values.

Using different logic to deliver the required behavior.

For a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant. It must never contain any variable data from any origin. Do not be tempted to decide case-by-case whether an item of data is trusted, and continue using string concatenation within the query for cases that are considered safe. It's easy to make mistakes about the possible origin of data, or for changes in other code to taint trusted data.

Security Headers

What are Security headers?

Security headers are HTTP headers that web servers use to enhance the security of web applications by providing additional layers of protection against various types of attacks. These headers are sent as part of the HTTP response from the server to the client's browser and instruct the browser on how to handle certain aspects of the web page. Here's a brief explanation of each of the headers you mentioned:

1. Strict-Transport-Security (HSTS):

Purpose: Enforces the use of HTTPS by instructing the browser to always load the website over a secure, encrypted connection.

Example: Strict-Transport-Security: max-age=31536000; includeSubDomains

2. Content-Security-Policy (CSP):

Purpose: Defines a policy to control which resources are allowed to be loaded on a web page, mitigating risks such as cross-site scripting (XSS) attacks.

Example: Content-Security-Policy: default-src 'self'; script-src 'self' https://trusted-scripts.com

3 .X-Frame-Options:

Purpose: Prevents a web page from being embedded within an iframe, providing protection against clickjacking attacks.

Example: X-Frame-Options: DENY

4. X-Content-Type-Options:

Purpose: Prevents browsers from interpreting files as a different MIME type than declared by the server, reducing the risk of certain types of attacks, such as MIME-sniffing.

Example: X-Content-Type-Options: nosniff

5 .Referrer-Policy:

Purpose: Controls how much information is included in the HTTP referer header when navigating from one page to another. This helps to protect user privacy.

Example: Referrer-Policy: no-referrer

6 .Permissions-Policy:

Purpose: Allows a website to control which browser features can be used and by whom. It helps in limiting the capabilities of web pages, enhancing security and privacy.

Example: Permissions-Policy: geolocation=(self "https://example.com")

Security Misconfiguration detected:

Target URL: https://0a670012039d2c28808817c0002c0045.web-security-academy.net/

Score: E

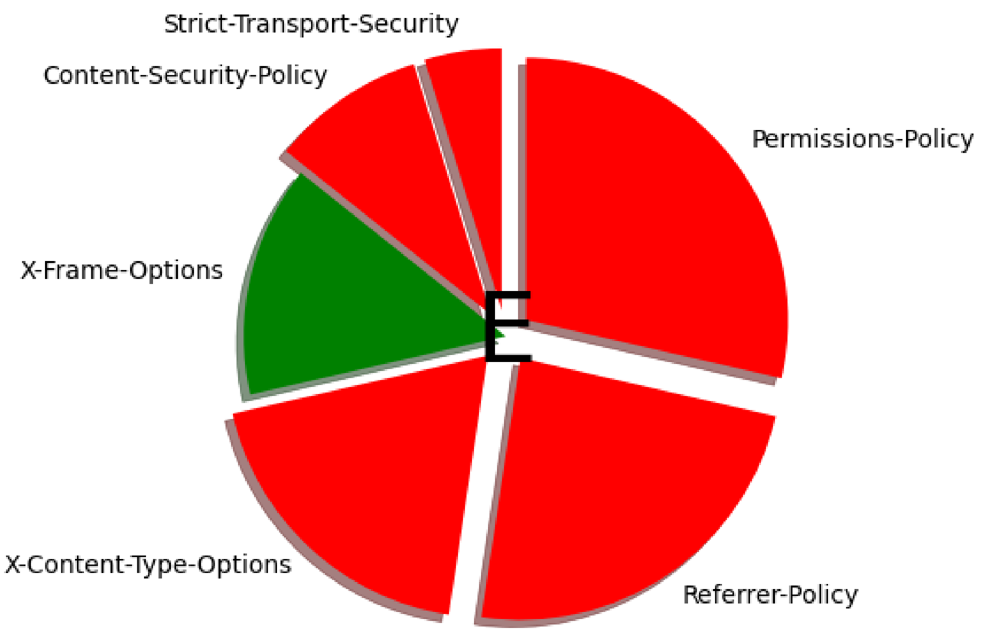
RISK: Strict-Transport-Security -> High risk of man-in-the-middle attacks

RISK: Content-Security-Policy -> High risk of xss and data injection attacks

RISK: X-Content-Type-Options -> High risk of MIME-based attacks

RISK: Referrer-Policy -> User privacy is at risk

RISK: Permissions-Policy -> Security policy not set for features like camera,microphone and more



Mitigation:

Mitigations for common web security vulnerabilities often involve a combination of secure coding practices, server configurations, and the use of security headers. Here are some general mitigations for the security headers you mentioned:

1. Strict-Transport-Security (HSTS):

Mitigation: Ensure that your website is accessible only over HTTPS. Configure HSTS headers with an appropriate max-age directive. Also, consider including subdomains if applicable (includeSubDomains).

Considerations: Be careful when implementing HSTS, as mistakes can lead to users being unable to access your site if it does not support HTTPS.

2. Content-Security-Policy (CSP):

Mitigation: Develop a strong CSP policy that restricts the sources of content, scripts, and other resources. Regularly review and update the policy as your website evolves.

Considerations: Test your CSP thoroughly to ensure it doesn't interfere with the normal operation of your web pages. Use the CSP report feature to receive reports on policy violations.

3. X-Frame-Options:

Mitigation: Set X-Frame-Options to DENY to prevent your web pages from being embedded in iframes. Alternatively, you can use SAMEORIGIN if your pages need to be embedded only on the same origin.

Considerations: Evaluate the impact on legitimate uses of iframes on your site. Adjust the setting accordingly.

4. X-Content-Type-Options:

Mitigation: Set X-Content-Type-Options to nosniff to prevent browsers from MIME-sniffing and interpreting files as a different MIME type than declared.

Considerations: Ensure that your server is correctly serving the appropriate MIME types for all content.

5. Referrer-Policy:

Mitigation: Set Referrer-Policy to no-referrer to limit the information included in the HTTP referer header.

Considerations: Be aware that strict referrer policies may break some legitimate functionality, so test thoroughly.

6. Permissions-Policy:

Mitigation: Define a restrictive policy using Permissions-Policy to limit access to browser features. Regularly review and update the policy based on your application's requirements.

Considerations: Test your web pages to ensure that necessary features are not inadvertently restricted. Use the reporting features to monitor policy violations.

Thank you