A learning path for Injection Vulnerabilities

Injection ranked as 3rd in the 2021 OWASP top 10 list.
Over 500,000 applications were tested for some form of injection with the average incidence rate of 3.37% with a maximum of 19% and 274k occurrences. [1]

Advanced Level

This attack type is considered a major problem in web security. In fact, injection attacks, such as SQLi and XSS, are very dangerous and also widespread, especially in legacy applications.

As untrusted data flows through an application, it can be split into parts, combined with safe data, transformed or decoded in multiple different ways. In fact, a single piece of data could go through multiple steps before going into an interpreter and this makes identifying injection problems rather difficult.
Injection can be really complex, the details of data flow, parsers and escaping are overwhelming even for security specialists.
Increasing awareness on this topic is more and more important because as applications get more and more interconnected the likelihood of a buried attack being decoded and executed by a downstream interpreter increases rapidly.
These attacks are a huge percentage of the serious application security risks.

One of the causes is poor attention on security controls that are being put in place to prevent injection attacks. Vague recommendations for input validations and various encoding are not enough to fully prevent these flaws.
A better way to prevent these attacks is to use a strong set of controls integrated into the application frameworks and trying to keep data separate from commands and queries.

Developers have to carefully code the application following some best practices to defend from this kind of attacks.
There are several different strategies to do this. Traditionally, input validation has been the preferred way to deal with untrusted data, but it is not the most efficient way to protect because usually validation is made when the destination of the data is unknown. This means that you do not know which characters are significant to the interpreter and input validation must be not too strict to avoid particular, but valid and not harmful, input to be rejected from the application.
Input validation is very important and should always be performed but cannot be the only countermeasure to injection attacks. It must be combined with other good practices, such as escaping.

Escaping is used to ensure that characters are treated as data, instead of characters that are relevant to the interpreter's parser and therefore untrusted data cannot be used for injection attacks.
There are a lot of different ways to perform escaping; some just require a special escape character and others a more complex syntax.
There is no harm in escaping data properly. It only ensures that the interpreter knows that the data is not intended to be executed, preventing attacks from working.
Another protection against injection attacks is to ensure that the web application runs with only the privileges it absolutely needs to perform its function.

In conclusion, some of the possible best practices and strategies to defend against injection attacks can be the following:

- Validating user input by creating a whitelist for valid statements and configuring input for user data by context
- Using prepared statements with parametrized queries that help distinguish between code and user input and do not mistake statements for commands
- Limit special characters to disallow string concatenation
- Escaping all user-supplied input
- Enforcing the least privilege and strict access by allowing only the necessary privileges for an account. [2]

The most important advice to remember is: never trust user input. The more you restrict, control and monitor any form of user input, the more you can avoid your application being hacked.

Cross site scripting attacks are a type of injection, in which malicious script are injected into otherwise trusted websites.
XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different user.

The methods of injection can have a lot of different variations; in some cases, the attacker may not even need to directly interact with the web functionality itself to exploit it. Any data received by the web application that can be controlled by an attacker could become an injection vector. If the attackers can control the value of the input field, they can easily construct a malicious value that causes their own script to execute.
XSS attacks can also be performed without using the <script> tag, even other tags can do the same thing. For example, using unload, onmouseover or onerror attributes or using strings with encoded characters can allow the attack to be performed.
Or you can even encode the script in base64 and place it in a <meta> tag. [3]
So, there are a lot of different ways to perform this attack and this is why it is so popular and so difficult to defend against.

The consequences of an XSS attack are the same, regardless of whether it is stored or reflected. The difference is on how the payload arrives at the server. The most severe XSS attacks involve disclosure of the user's session cookie, allowing an attacker to hijack the user's session and to take over the account. Other damaging attacks include the disclosure of end user's files, installation of malicious programs, redirecting the user to some other page or site, or modifying presentation of content. [3]

XSS flaws can be difficult to identify and remove from a web application. The best way to find flaws is to perform a security review of the code and search for all places where input from an HTTP request could possibly make its way into HTML output.

This is a very difficult task: a lot of HTML tags could be used to transmit malicious code. There are available different tools to scan and check a website for these flaws. They can be very helpful, but they can only scratch the surface. If one part of the website is vulnerable, many other problems are likely to be there.

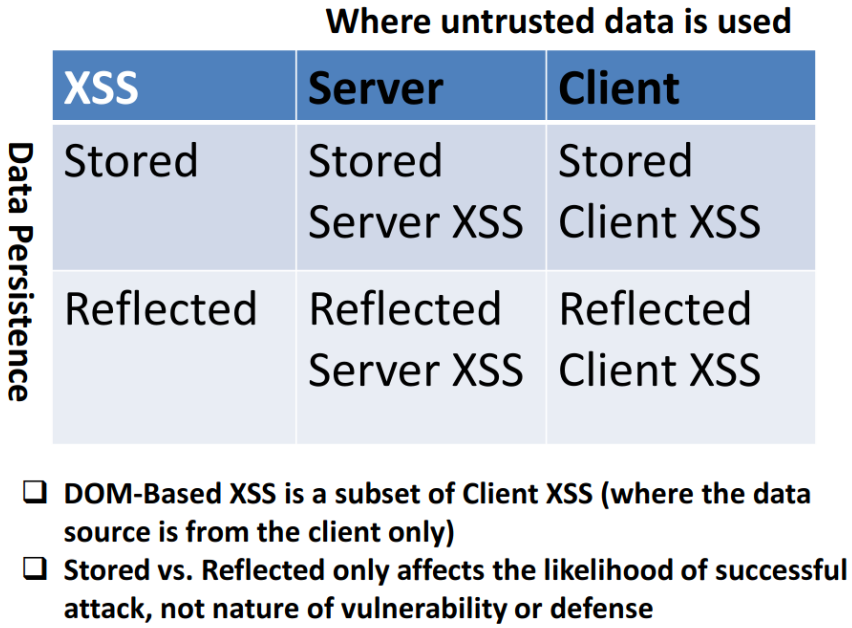

Fig. 3.3 XSS attack matrix [4]

For years, most people thought of Stored, Reflected, DOM as three different types of XSS, but in reality, they overlap. You can have both Stored and Reflected DOM Based XSS. You can also have Stored and Reflected Non-DOM Based XSS too, but that is confusing, so to help clarify things, starting about mid 2012, the research community proposed and started using two new terms to help organize the types of XSS that can occur: Server XSS and Client XSS.

Server XSS occurs when untrusted user supplied data is included in an HTTP response generated by the server. The source of this data could be from the request, or from a stored location. As such, you can have both Reflected Server XSS and Stored Server XSS.

In this case, the entire vulnerability is in server-side code and the browser is simply rendering the response and executing any valid script embedded in it.

Client XSS occurs when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call. A JavaScript call is considered unsafe if it can be used to

introduce valid JavaScript into the DOM. This source of this data could be from the DOM, or it could have been sent by the server. The ultimate source of the data could have been from a request, or from a stored location on the client or the server. As such, you can have both Reflected Client XSS and Stored Client XSS.
With these new definitions, the definition of DOM Based XSS does not change. DOM Based XSS is simply a subset of Client XSS, where the source of the data is somewhere in the DOM, rather than from the Server.
Given that both Server XSS and Client XSS can be Stored or Reflected, this new terminology results in a simple, clean, 2 x 2 matrix with Client & Server XSS on one axis, and Stored and Reflected XSS on the other axis as shown in Fig.1.11. [5]

There are some best practices to prevent or limit the impact of XSS vulnerabilities.
The application must validate all the input data and ensure that all variable output in a page is encoded before it is returned to the user. When you encode variable output, you substitute HTML markup with alternative representations that are not run by the browser. Each variable in a web application needs to be protected. Ensuring that all variables go through validation and are then escaped or sanitized is known as perfect injection resistance. Any variable that does not go through this process is a potential weakness.
Server XSS is caused by including untrusted data in HTML responses. Input validation or data sanitization should be performed to help prevent server XSS.
Client XSS is caused when untrusted data is used to update the DOM with an unsafe JavaScript call. So, the strongest defense is using safe JavaScript APIs.
If you know that a JS method is unsafe, the primary recommendation is to find an alternative safe method to use. Otherwise, context sensitive output encoding can be done in the browser, before passing that data to the unsafe JavaScript method.

It is also crucial to turn off HTTP TRACE support on all web servers. An attacker can steal cookie data via JavaScript even when document.cookie is disabled or not supported by the client. This attack is mounted when a user posts a malicious script to a forum so when another user clicks the link, an asynchronous HTTP Trace call is triggered which collects the user's cookie information from the server, and then sends it over to another malicious server that collects the cookie information. So, the attacker

can create a session hijack attack. This is easily mitigated by removing support for HTTP TRACE on all web servers. [3]
Fewer XSS bugs appear in applications built with modern frameworks. These frameworks bring developers towards good security practices and help mitigate XSS by using templating, auto-escaping, and other good practices. It is important for developers to understand how the used framework prevents XSS and where it has gaps. Frameworks are not perfect and security gaps still exist in popular frameworks like React and Angular. Output Encoding and HTML Sanitization help address those gaps.

SQL injection is a code injection technique used to attack data-driven applications.
A SQL injection attack consists of insertion or “injection” of a malicious SQL query via the input data from the client to the application. [6]

SQL injections can be typically distinguished into three categories: In-band SQLi (also called Classic), Inferential SQLi (also called Blind) and Out-of-band SQLi. These categories can be distinguished based on the methods used to access backend data and the damage potential. [7]

In-band SQLi:
The attacker uses the same channel of communication to launch their attacks and to gather their results. In-band SQLi’s simplicity and efficiency make it one of the most common types of SQLi attack. There are two sub-variations of this method:
- Error based SQLi: the attacker performs actions that cause the database to produce error messages. The attacker can potentially use the data provided by these error messages to gather information about the structure of the database.
- Union based SQLi: this technique takes advantage of the UNION SQL operator, which fuses multiple select statements generated by the database to get a single HTTP response. This response may contain data that can be leveraged by the attacker.

Inferential SQLi:
The attacker sends data payloads to the server and observes the response and behaviour of the server to learn more about its structure. This method is also called

blind SQLi because the data is not transferred from the website database to the attacker, thus the attacker cannot see information about the attack in-band.
Blind SQL injections rely on the response and behavioral patterns of the server so they are typically slower to execute but may be just as harmful. Blind SQL injections can be classified as follows:
- Boolean: the attacker sends a SQL query to the database prompting the application to return a result. The result will vary depending on whether the query is true or false. Based on the result, the information within the HTTP response will modify or stay unchanged. The attacker can then work out if the message generated a true or false result.
- Time based: attacker sends a SQL query to the database, which makes the database wait for a certain amount of time before it can react. The attacker can see from the time the database takes to respond whether a query is true or false. Based on the result, an HTTP response will be generated instantly or after a waiting period. The attacker can thus work out if the message they used returned true or false, without relying on data from the database.

Out of band SQLi:
This type of attack is not very common, mostly because it depends on features being enabled on the database server being used by the web application. Out-of-band SQL Injection occurs when an attacker is unable to use the same channel to launch the attack and gather results, or when a server is too slow or unstable for these actions to be performed. This form of attack is used as an alternative to the in-band and inferential SQLi techniques and count on the capacity of the server to create DNS or HTTP requests to transfer data to an attacker.

SQL injection attacks can be easily prevented by simple measures. The first step is input validation or sanitization, which is the practice of writing code that can identify illegitimate user inputs. [7]
While input validation should always be considered best practice, it is rarely a foolproof solution. The reality is that, in most cases, it is simply not feasible to map out all legal and illegal inputs, at least not without causing a lot of false positives, which interfere with user experience and an application functionality.

For this reason, a Web Application Firewall (WAF) is commonly employed to filter out SQLi, as well as other online threats. A WAF typically relies on a large, and constantly updated, list of meticulously crafted signatures that allow it to surgically identify malicious SQL queries. Usually, such a list holds signatures to address specific attack vectors and it is regularly patched to introduce blocking rules for newly discovered vulnerabilities.
WAF products cannot prevent SQL injection vulnerabilities from entering into a codebase, but they can make discovery and exploitation much more challenging to an attacker.
For example, a web application firewall that encounters a suspicious, but not outright, malicious input may cross-verify it with IP data before deciding to block the request. It only blocks the input if the IP itself has a bad reputational history.

Developers can also use ORM frameworks (Object relational mappers) to create database queries in a safe and developer friendly way. Since database queries are no longer constructed as strings, there is no danger of an injection vulnerability.

Another popular, though error-prone, way to prevent injections is to attempt to escape all characters that have a special meaning in SQL. The manual for an SQL DBMS explains which characters have a special meaning, which allows creating a comprehensive blacklist of characters that need translation. Routinely passing escaped strings to SQL is error prone because it is easy to forget to escape a given string. This technique should only be used as a last resort, when none of the above are feasible. [8]

Limiting the permissions on the database used by the web application to only what is needed may help reduce the effectiveness of any SQL injection attacks that exploit any bugs in the web application.

Most instances of SQL injection can be prevented by using parameterized queries instead of string concatenation within the query. [9]
With most development platforms, parameterized statements that work with parameters can be used instead of embedding user input in the statement. A placeholder can only store a value of the given type and not an arbitrary SQL fragment.

Hence the SQL injection would simply be treated as a strange and invalid parameter value.

They cannot 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 those parts of the query will need to take a different approach, such as white-listing permitted input values, or using different logic to deliver the required behaviour.

Bibliography

[1] «A03:2021 - Injection,» [Online]. Available: https://owasp.org/Top10/A03_2021-Injection. [Consulted 2023].

[2] «Injection attack types,» [Online]. Available: https://crashtest-security.com/different-injection-attack-types/. [Consulted 2023].

[3] «Cross Site Scripting,» [Online]. Available: https://owasp.org/www-community/attacks/xss/. [Consulted 2023].

[4] [Online]. Available: https://owasp.org/www-community/assets/images/Server-XSS_vs_Client-XSS_Chart.png.

[5] «Types of XSS,» [Online]. Available: https://owasp.org/www-community/Types_of_Cross-Site_Scripting. [Consulted 2023].

[6] «SQL Injection,» [Online]. Available: https://owasp.org/www-community/attacks/SQL_Injection. [Consulted 2023].

[7] «What is SQL injection, SQLi Examples & prevention,» [Online]. Available: https://www.imperva.com/learn/application-security/sql-injection-sqli/. [Consulted 2023].

[8] «SQL injection prevention - OWASP Cheat Sheet Series,» [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html. [Consulted 2023].

[9] «What is SQL injection,» [Online]. Available: https://portswigger.net/web-security/sql-injection. [Consulted 2023].