IT security

(VIHIAC07)

Lab exercise

Client and server side web security lab

Author:
András Gazdag

March 6, 2025

Contents

1 Educational Objectives

In this exercise, you will learn about client and server-side web security issues and the dangers of various injection attacks. The goal is to gain hands-on experience with a real-world web application, understand the potential impact of injection attacks, become familiar with the JavaScript language, and understand the risks associated with SQL and JavaScript injection (such as Cross-Site Scripting (XSS)) attacks. During the exercise, you will perform attacks against the Juice-Shop web application to illustrate the effects of XSS attacks and the risks of injection attacks, which are common in database management. Your tasks will also include bypassing the login mechanism and password management and generally attacking the database communication.

2 Background Material

2.1 JavaScript

JavaScript (JS) (MDN: What is JavaScript?) is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions. While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as Node.js, Apache CouchDB and Adobe Acrobat. JavaScript is a prototype-based, multi-paradigm, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles.

The core client-side JavaScript language consists of some common programming features that allow you to do things like:

- store useful values inside variables,
- operations on pieces of text (known as "strings" in programming),
- running code in response to certain events occurring on a web page,
- and much more!

What is even more exciting however is the functionality built on top of the client-side JavaScript language. So-called Application Programming Interfaces (APIs) provide you with extra superpowers to use in your JavaScript code. APIs are ready-made sets of code building blocks that allow a developer to implement programs that would otherwise be hard or impossible to implement.

Browser APIs are built into your web browser, and are able to expose data from the surrounding computer environment, or do useful complex things. For example:

- The DOM (Document Object Model) API allows you to manipulate HTML and CSS, creating, removing and changing HTML, dynamically applying new styles to your page, etc. Every time you see a popup window appear on a page, or some new content displayed for example, that's the DOM in action.
- The Geolocation API retrieves geographical information. This is how Google Maps is able to find your location and plot it on a map.

- The Canvas and WebGL APIs allow you to create animated 2D and 3D graphics. People are doing some amazing things using these web technologies.
- Audio and Video APIs like HTMLMediaElement and WebRTC allow you to do really interesting things with multimedia, such as play audio and video right in a web page, or grab video from your web camera and display it on someone else's computer.

Third party APIs are not built into the browser by default, and you generally have to grab their code and information from somewhere on the Web. For example:

- The Twitter (or X) API allows you to do things like displaying your latest tweets on your website.
- The Google Maps API and OpenStreetMap API allows you to embed custom maps into your website, and other such functionality.

Browser Security rules guarantee that each browser tab is its own separate bucket for running code in (these buckets are called "execution environments" in technical terms). This means that in most cases the code in each tab is run completely separately, and the code in one tab cannot directly affect the code in another tab or on another website. This is a good security measure because if this were not the case, then pirates could start writing code to steal information from other websites, and other such bad things.

As a preparation to the lab you should learn the basic concepts of the JavaScript language from the Mozilla website.

2.2 Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script (JavaScript), to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page. Further details about the different XSS attacks are described in the next sections.

2.2.1 Stored XSS (AKA Persistent or Type I)

Stored XSS generally occurs when user input is stored on the target server, such as in a database, in a message forum, visitor log, comment field, etc. And then a victim is able to retrieve the stored data from the web application without that data being made safe to render in the browser. With the advent of HTML5, and other browser technologies, we can envision the attack payload being permanently stored in the victim's browser, such as an HTML5 database, and never being sent to the server at all.

2.2.2 Reflected XSS (AKA Non-Persistent or Type II)

Reflected XSS occurs when user input is immediately returned by a web application in an error message, search result, or any other response that includes some or all of the input provided by the user as part of the request, without that data being made safe to render in the browser, and without permanently storing the user provided data. In some cases, the user provided data may never even leave the browser (see DOM Based XSS next).

2.2.3 DOM Based XSS (AKA Type-0)

DOM Based XSS is a form of XSS where the entire tainted data flow from source to sink takes place in the browser, i.e., the source of the data is in the DOM, the sink is also in the DOM, and the data flow never leaves the browser. Modern web applications (e.g. SPAs: Single-page application vs. multiple-page application) render the final web page on the client side in most cases. Most frameworks have basic protection, but it is possible to make mistakes when using them. Applications that implement this functionality without a framework are typical targets for such an attack. For example, the source (where malicious data is read) could be the URL of the page (e.g., document.location.href), or it could be an element of the HTML, and the sink is a sensitive method call that causes the execution of the malicious data (e.g., document.write).

2.3 XSS Prevention Rules

The following rules are intended to prevent all XSS in your application (Cross Site Scripting Prevention Cheat Sheet). While these rules do not allow absolute freedom in putting untrusted data into an HTML document, they should cover the vast majority of common use cases. You do not have to allow all the rules in your project. Many projects may find that allowing only Rule #1 and Rule #2 are sufficient for their needs.

1. HTML Escape Before Inserting Untrusted Data into HTML Element Content.
2. Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes.
3. Never Insert Untrusted Data Except in Allowed Locations.
4. JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values.
5. HTML escape JSON values in an HTML context and read the data with JSON.parse.
6. CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values.
7. URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values.
8. Sanitize HTML Markup with a Library Designed for the Job.
9. Avoid JavaScript URL's.

2.4 Code insertion

Injection flaws allow attackers to relay malicious code through an application to another system and the system executes that along with the original application code. These attacks include calls to the operating system via system calls, the use of external programs via shell commands, as well as calls to backend databases via SQL (i.e., SQL injection). Whole

scripts written in Perl, Python, and other languages can be injected into poorly designed applications and executed. Any time an application uses an interpreter of any type, there is a danger of introducing an injection vulnerability.

2.5 Database engines

A database engine (or storage engine) is the underlying software component that a database management system (DBMS) uses to create, read, update and delete (CRUD) data from a database. Most database management systems include their own application programming interface (API) that allows the user to interact with their underlying engine without going through the user interface of the DBMS.

2.5.1 Relational databases

A relational database is a type of database that stores and provides access to data points that are related to one another. Relational databases are based on the relational model, an intuitive, straightforward way of representing data in tables. In a relational database, each row in the table is a record with a unique ID called the key. The columns of the table hold attributes of the data, and each record usually has a value for each attribute, making it easy to establish the relationships among data points.

The relational model provides a standard way of representing and querying data that could be used by any application. From the beginning, developers recognized that the chief strength of the relational database model was in its use of tables, which were an intuitive, efficient, and flexible way to store and access structured information.

Over time, another strength of the relational model emerged as developers began to use structured query language (SQL) to write and query data in a database. For many years, SQL has been widely used as the language for database queries. Based on relational algebra, SQL provides an internally consistent mathematical language that makes it easier to improve the performance of all database queries.

2.5.2 NoSQL databases

NoSQL databases provide looser consistency restrictions than traditional SQL databases. By requiring fewer relational constraints and consistency checks, NoSQL databases often offer performance and scaling benefits. Yet these databases are still potentially vulnerable to injection attacks, even if they aren't using the traditional SQL syntax. Because these NoSQL injection attacks may execute within a procedural language, rather than in the declarative SQL language, the potential impacts are greater than traditional SQL injection.

NoSQL database calls are written in the application's programming language, a custom API call, or formatted according to a common convention (such as XML, JSON, LINQ, etc). Malicious input targeting those specifications may not trigger the primary application sanitization checks. For example, filtering out common HTML special characters such as `<>&;` will not prevent attacks against a JSON API, where special characters include `\{}:`.

There are now over 150 NoSQL databases available for use within an application, providing APIs in a variety of languages and relationship models. Each offers different features

and restrictions. Because there is not a common language between them, example injection code will not apply across all NoSQL databases. For this reason, anyone testing for NoSQL injection attacks will need to familiarize themselves with the syntax, data model, and underlying programming language in order to craft specific tests. NoSQL injection attacks may execute in different areas of an application than traditional SQL injection. Where SQL injection would execute within the database engine, NoSQL variants may execute within the application layer or the database layer, depending on the NoSQL API used and data model.

Typically NoSQL injection attacks will execute where the attack string is parsed, evaluated, or concatenated into a NoSQL API call.

2.6 SQL language

SQL (pronounced "ess-que-el") stands for Structured Query Language. SQL is used to communicate with a database. It is the standard language for relational database management systems. SQL statements are used to perform tasks such as update data on a database, or retrieve data from a database. Some common relational database management systems that use SQL are: Oracle, Microsoft SQL Server, MySQL, SQLite etc. Although most database systems use SQL, most of them also have their own additional proprietary extensions that are usually only used on their system. However, the standard SQL commands such as `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE` and `DROP` can be used to accomplish almost everything that one needs to do with a database. You must learn the basics of the language from the w3schools website: https://www.w3schools.com/sql/default.asp.

2.7 Attacks

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM) or LDAP injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

2.7.1 SQL Injection

SQL injection is a particularly widespread and dangerous form of injection. To exploit a SQL injection flaw, the attacker must find a parameter that the web application passes through to a database. By carefully embedding malicious SQL commands into the content of the parameter, the attacker can trick the web application into forwarding a malicious query to the database. These attacks are not difficult to attempt and more tools are emerging that scan for these flaws.

Injection vulnerabilities can be very easy to discover and exploit, but they can also be extremely obscure. The consequences of a successful injection attack can also run the entire range of severity, from trivial to complete system compromise or destruction. In any case,

the use of external calls is quite widespread, so the likelihood of an application having an injection flaw should be considered high.

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL command contains both structure and hostile data in dynamic queries, commands, or stored procedures.

A typical **vulnerable** implementation of a SQL command is the following:

```
String query = "SELECT * FROM users WHERE name = '" + userName + "' AND
    password = '" + password + "'";
```

This implementation works correctly as long as the user only enters real user names and passwords. But an attacker could use the password field as a potential input for SQL injection. In this scenario, if the attacker enters an input like this:

- username: `whoever`
- password: `asd' OR 'a'='a`

This could modify the applications original behavior. This input creates the following SQL query:

```
SELECT * FROM users WHERE name = 'whoever' AND password = 'asd' OR 'a'='a'
```

The additional `OR 'a'='a'` part completely modifies the original query. Instead of returning just one single user it now returns every user form the database. If the application logic just takes the first row from the response then the first user (which is usually the administrator) is considered to be the logged in user.

Another typical tool of an attacker is the usage of the comment character in the SQL language. This character may differ between different SQL database engines. The usage of the comment character could effectively disable the remaining parts of a query. An example for an attack like this is the following:

- username: `name' OR 'a'='a' #`
- password: `whatever`

These inputs will result in the following SQL query:

```
SELECT * FROM users WHERE name = 'name' OR 'a'='a' #
```

The database engine will no longer evaluate the password condition.

Note: The previous code examples are not completely syntactically correct. Their purpose is only to demonstrate the concept of a SQL injection.

2.7.2 Injection prevention

Preventing an injection requires keeping the data separate from the commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs). Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter. *Note:* SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use `LIMIT` and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

2.7.3 Denial-of-service

In a denial-of-service (DoS) attack, an attacker attempts to prevent legitimate users from accessing information or services. By targeting your computer and its network connection, or the computers and network of the sites you are trying to use, an attacker may be able to prevent you from accessing email, websites, online accounts (banking, etc.), or other services that rely on the affected computer.

3 Setting Up the Environment

3.1 Using a Virtual Machine

Using the VirtualBox application, import the released OVA file and start the VM. No further preparation is required using the provided VM.

The steps to start Juice Shop are:

1. Launch the Terminal application.
2. Enter the Juice Shop library: `cd ~/juice-shop/`.
3. Launch the Juice Shop application: `npm start`.
4. Depending on the load of the machines, the start-up can take a few minutes.

The Juice Shop runs in interactive mode, so it will not be able to issue any further commands in the terminal you have been using to start the application. If you close the terminal window, the application will also stop. At the end of the measurement, you will be able to shut down the application normally by using the `CTRL + C` key combination.

Once the app is launched, it can be accessed at `http://localhost:3000`.

Figure 1. Expected result

3.2 Setting up your home environment

The steps to start Juice Shop in your home environment are:

1. Install Node.js in a way that is compatible with your operating system using the following link. For supported versions of Node.js, see the following page
2. Download the Juice Shop GitHub project. For example, clone it from your terminal using `git clone https://github.com/juice-shop/juice-shop.git`
3. Install the application as described in the following link.
4. Follow the steps as described in the **Using a Virtual Machine** section.

Note: if you want to use the application in a different way, for example using Docker, the link shared in step 3 provide various installation instructions to setup the application in different ways.

4 Tasks

This lab exercise consists of 6 exercises: 3 XSS - 3 SQL injection related exercises. Each task is in ascending order of difficulty within a "block", so it is recommended that you solve them in the order described. The Juice Shop app will be used in the lab.

Commonly used XSS attacks have been collected on several sites. Such a list can also help to solve the exercises, and it is recommended to gather ideas for attempts. You can find a list like this in the following page: XSS Filter Evasion Cheat Sheet.

If a logged-in user is required to perform a task, it is recommended to use the administrator user. To do this, enter **username:** `admin@juice-sh.op` and **password:** `admin123`

When solving frontend-related tasks, the goal is always to demonstrate how to run Javascript code, which is most easily done by calling the `alert()` function available in JavaScript. The goal is to throw up a window/frame (`<iframe>` tag) containing the `xss` text.

Attacks are often complicated by the need to ensure that quotation marks are used correctly when inserting strings. The original source code also uses the `'` and `"` characters in order. In such cases, it may also be useful to try the ‘ (backtick) character, which can also be used as a string start and end character.

When solving SQL injection related tasks, it is worth trying out the vulnerability attacks in a simpler version first to see if you are really trying to solve the tasks in the right place. A good example of this is to test a SQL injection attack first with minimal input such as `'`

or `';` characters. By analyzing the responses to such inputs, you can determine whether the vulnerability is indeed exploitable, and so it is then worthwhile to compose more complex input to solve the exercies.

4.1 XSS attacks

4.1.1 Perform a reflected XSS attack!

Look for an input field where its content appears in the response when its form is submitted.

Hint: the vulnerability is in the order tracking feature. (This can be accessed by clicking on the small truck icon on the Order History page.)

Try probing for XSS vulnerabilities by submitting text wrapped in an HTML tag which is easy to spot on screen (see previously detailed expected result)!

4.1.2 Perform a DOM XSS attack!

This challenge is almost indistinguishable from *Perform a reflected XSS attack* if you do not look "under the hood" to find out what the application actually does with the user input.

Repeat the attack against the website's search function and then look in the source code to find where the vulnerable code snippet is located. The code to process the search can be accessed via this link: https://github.com/juice-shop/juice-shop/blob/master/frontend/src/app/search-result/search-result.component.html.

4.1.3 Perform a persisted XSS attack bypassing a client side security mechanism!

All new user data is stored in the database. Proper verification of all data during the registration process is a critical task for security.

In the Juice Shop application, this protection has been implemented on the client side, which can be bypassed by an attacker. The client-side protection can be easily bypassed by manually modifying the HTML content of the website.

Perform a persistent XSS attack to demonstrate this vulnerability! The result of a successful attack can be seen after opening the `.../#/administration` URL!

This challenge is founded on a very common security flaw of web applications, where the developers ignored the following golden rule of input validation:

Be aware that any JavaScript input validation performed on the client can be bypassed by an attacker that disables JavaScript or uses a Web Proxy. Ensure that any input validation performed on the client is also performed on the server.

Hint: the vulnerability is in the in the registration form. You can bypass the "client-side security" of the registration by manipulating the DOM tree of the form.

4.2 SQL injection

4.2.1 Log in with the administrator's user account!

Find a vulnerable input in the web application and gain administrator access to it. Although the task can be performed by trying to guess the password, the goal is to perform an SQL injection.

Hint: The server side code for the login can be found here: https://github.com/bkimminich/juice-shop/blob/master/routes/login.js

4.2.2 Let the server sleep for some time!

This challenge is about giving the server the chance to catch a breath by putting it to sleep for a while, making it essentially a stripped-down denial-of-service attack challenge.

The Juice Shop web application uses a MongoDB derivate as its NoSQL database to store information about the products such as its reviews. Your task is to find the relevant requests to the backend and perform an injection attack against it. For this attack you should use the sleep function of MongoDB, the documentation for which can be found here: https://docs.mongodb.com/manual/reference/method/sleep/

Hint: The server side code for the product reviews can be found here: https://github.com/bkimminich/juice-shop/blob/master/routes/showProductReviews.js

Comment: flooding the application with requests will not solve this challenge. That would probably just kill your server instance.

4.2.3 Reset Jim's password via the Forgot Password mechanism.

Use the built in feature of the web application to change the password of the user Jim (jim@juice-sh.op). The password reset function has built in protection to stop unauthorized modifications. Use social engineering to circumvent the protection!

Jim picked one of the worst security questions and chose to answer it truthfully. As Jim is a celebrity, the answer to his question is quite easy to find in publicly available information on the internet.

Hint:

- Look for the 'Forgot password' feature in the web app!
- After entering Jim's email address, please note Jim's security question!
- Find Juice Shop products that Jim has written a review for! Search on the second page. The information you find may give you an idea about Jim.
- Use a method similar to the SQL injection method from the first exercise in this block to log in as Jim and then look up the shipping addresses he provided.
- Find out who Jim is online, based on the information you get from reviews and addresses.
- Use the available information about Jim to answer the security question you found earlier!