

Assignment 7

New Attempt

- Due Nov 6, 2024 by 11:59pm
- Points 100
- Submitting a file upload
- File Types pdf

Due: Tuesday, November 5, 2024 (11:59 PM). Please submit your assignment on Canvas as a PDF.

Web Exploitation

In this assignment, you will get familiar with SQL Injection attacks, Insecure Direct Object Reference, server-side template injection and Insecure deserialization.

SQL injection is one the most common and dangerous web hacking techniques targeting database systems in which malicious SQL statements are injected to exploit unfiltered entry fields. This type of attack is used to gain information from databases, create new entries or even delete a table and so on.

An **Insecure Direct Object Reference (IDOR)** vulnerability occurs when an application exposes internal implementation objects, such as files, database records, or URLs, to users. This allows users to directly access and manipulate these objects by simply modifying the input values.

SSTI (Server-Side Template Injection) occurs when user input is incorrectly rendered in a server-side template, allowing attackers to inject and execute malicious code. This can lead to unauthorized access, data leakage, and remote code execution.

Insecure deserialization happens when untrusted data is deserialized without proper validation, allowing attackers to manipulate serialized objects to execute arbitrary code, escalate privileges, or exploit application logic. This can lead to severe security breaches, including remote code execution.

Setup

You will be given two servers and a client written in Python and JavaScript.

Backend1:

- This is a node server that uses SQLite to store its data and is susceptible to SQL Injection and IDOR. It listens on port 8084.

Backend2:

- This is a Flask server which is vulnerable to SSTI and Insecure Deserialization. It listens on port 5000.

To run the backend and frontend in Kali perform the following steps:

- Download the backend (server) and frontend (client) code from [assignment7.zip](https://canvas.sfu.ca/courses/85026/files/24851272?wrap=1) (<https://canvas.sfu.ca/courses/85026/files/24851272?wrap=1>)_ ↓ (https://canvas.sfu.ca/courses/85026/files/24851272/download?download_frd=1) and unzip it.
- Run `md5sum assignment7.zip` and verify it's `eb00ee2681a63f98d58c0b7d93859500` to make sure you have the right code.

To bring up the backends and frontend, we'll use docker-compose as follows.

Using docker-compose up

- Install docker and docker-compose (If not already installed)
- Go to the assignment-7 directory with docker-compose.yml file and run ``docker-compose up`` (may need sudo if the user doesn't have permissions)
 - May need to run ``docker-compose build`` if you had built images from the older version.
 - This may need access to the internet as node may download and install packages at runtime. (Recommend running router VM)
- This will create the environment and bring up the backend, backend2 and frontend on ports 8084, 5000 and 3000 respectively.
- If you make changes to the code, you need to do a ``docker-compose build`` to rebuild the images and then run ``docker-compose up`` to run the new version.

After one of these steps, visit <http://localhost:3000> → (<http://localhost:3000/>) and you should be able to see the website.

sqlmap (15%)

In this set of tasks you are going to use a tool called `sqlmap`. sqlmap is an open source penetration testing tool that automates the process of detecting and exploiting SQL injection vulnerabilities.

In order to run sqlmap open a console in Kali and type:

```
sqlmap -u http://<vulnerable-website>/<vulnerable endpoint>
```

The server running at `http://localhost:8084` has a vulnerable endpoint at `/vulnerable`. You can visit the endpoint by opening a web browser in Kali and entering `http://localhost:8084/vulnerable?q=user`. You should see the username `user`. This endpoint takes a username as a query parameter and returns it if it exists in the database.

Task 1 (5%): Using `sqlmap`, list all the tables in the database by exploiting the vulnerable endpoint `/vulnerable`. What command did you use? What are the tables you found?

Task 2 (5%): Using `sqlmap`, list all the usernames and passwords you found in the tables. What command did you use?

Task 3 (5%): Now restart the server with docker compose and run the above command again. Do the credentials work? Why or Why not? What flag in the sqlmap command did you use to fetch the new credentials?

SQL injection (35%)

In the following set of tasks, you are going to perform SQL injection without using `sqlmap`. Please do not use `sqlmap` for any of the tasks below.

Task 4 (5%): On the home page of the provided website click Login User and try to gain access to the webpage using SQL injection (**The User panel should show the data for this to be a valid submission**). Report what you did.

Hint: `user` is a sample username for website users, and `admin` is the username of the website admin.

Task 5 (5%): After gaining access, logout and go to User Login. Try to change the password of the `user` using SQL injection. Report how you did it.

Task 6 (10%): After exploiting SQL injection in the User Login go to the User Panel (after you login with the proper user's credentials) with the newly set password and now, you can see the `user`'s data. You can update all of the data fields except the `user`'s salary. Try to exploit SQL injection from the User's Panel to double the `user`'s salary. Report what you did.

NOTE: Any SQL Injection from your part that treats salary as a TEXT and not as a NUMERIC is not considered a valid answer. The problem is that salary is defined as NUMERIC but sqlite treats TEXT as a NUMERIC without complaints. So for example if you do

```
' , salary='20000
```

will not be considered a valid answer.

Task 7 (5%): Try to delete the users table using SQL injection from the login page. What actions did you take? How can you confirm that the table was deleted?

Task 8 (10%): Try to fix the bug in the server for the vulnerable endpoint `/vulnerable`. The bug makes the endpoint vulnerable to SQL injection. The bug exists in the backend directory in the file `index.js` towards the end of the file and the corresponding code is:

```
app.get("/vulnerable", async (req, res, next) => {
  const db = await dbPromise;
  let ret;
  try {
    ret = await db.get(
      `SELECT username FROM users WHERE username='${req.query.q}'`
    );
  } catch(err) {
    ret = "error";
  }
  res.send(ret);
});
```

Submit the correct code. You do not need to submit the whole `index.js`, just the corrected code as above.

Hint: You may find useful the documentation of SQLite package [here](https://www.npmjs.com/package/sqlite) (<https://www.npmjs.com/package/sqlite>)

Insecure Direct Object Reference (15%)

Task 9 (10%): Identify and exploit the IDOR vulnerability to get user data. Provide screenshots.

Task 10 (5%): Fix the IDOR vulnerability and share the updated function highlighting what was changed.

Server Side Template Injection (20%)

<http://localhost:5000/ssti>  [\(http://localhost:5000/ssti\)](http://localhost:5000/ssti) is vulnerable to server side template injection.

Task 11: (10%): Read the contents of /etc/passwd using SSTI. Provide screenshots with the payload.

Task 12 (10%): Fix the arbitrary code execution in the /ssti endpoint and share the patched code.

Insecure Deserialization (15%)





<http://localhost:5000/deserialize>  [\(http://localhost:5000/deserialize\)](http://localhost:5000/deserialize) is vulnerable to insecure deserialization.

Task 13 (10%): Establish a reverse shell with the local system on port 8989 by exploiting the vulnerability. How did you generate the payload? Provide screenshots.

Task 14: (5%): Fix the vulnerability and share the patched code.

Additional Resources

There are various types of attacks on web applications that can be explored in the following resources:

- Portswigger tutorial and labs: <https://portswigger.net/web-security/all-topics>  [\(https://portswigger.net/web-security/all-topics\)](https://portswigger.net/web-security/all-topics)
- OWASP Juice Shop <https://owasp.org/www-project-juice-shop/>  [\(https://owasp.org/www-project-juice-shop/\)](https://owasp.org/www-project-juice-shop/)
- Damn Vulnerable Web Application <https://github.com/digininja/DVWA>  [\(https://github.com/digininja/DVWA\)](https://github.com/digininja/DVWA)
- Hacktricks: <https://book.hacktricks.xyz/>  [\(https://book.hacktricks.xyz/\)](https://book.hacktricks.xyz/)