Lab 08 - Web Security

Objectives

- Web vulnerabilities, both server-side and client-side
- Server-side SQL injection
- Cross-Site Scripting, Cross-Site Request Forgery

Background

SQL Injection

SQL Injection is a server-side code injection vulnerability resulting from improper (unsanitized) input directly concatenated into SQL queries. Typical server queries are built as strings:

```
sql = "SELECT * FROM table WHERE item = '" + user_input_variable + "' <other expressions>";
database.query(sql);
```

Note that the user may choose to escape the SQL quotes and alter the SQL statement, e.g.:

```
user_input_variable = "' OR 1=1 -- "; // example input given by the user
sql = "SELECT * FROM table WHERE item = '' OR 1=1 -- ' <other expressions>";
```

An SQL injection exploit ultimately depends on the target SQL expression (which is usually unknown to the attacker) and query result behavior (whether the query contents arem displayed on screen or the user is blind, errors reported etc.).

Make sure to check those cheatsheets out:
https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/SQL%20Injection
[https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/SQL%20Injection]
and:
https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/SQL%20Injection/MySQL%20Injection.md
[https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/SQL%20Injection/MySQL%20Injection.md]

Other server-side vulnerabilities

The SQL injection is a popular server-side code injection vulnerability, but there are many mistakes that a website developer / system administrator can make (*expect to find some of them in your homework :P*):

- code injection (LDAP, eval, shell execution etc.);
- broken authentication or access control (authorization);
- sensitive data exposure (e.g., backups / keys forgotten inside web root);
- path traversal;
- server misconfiguration;
- *and many more [https://owasp.org/www-community/vulnerabilities/]*

There are even free or commercial web vulnerability scanners [https://owasp.org/www-community/Vulnerability_Scanning_Tools] for testing a server's sercurity!

Client-side vulnerabilities

Browsers are now among the most targeted pieces of software on the Internet. This is mainly because of the large threat vector resulting from the complexity of the web ecosystem, requiring features such as fancy HTML+CSS rendering, animation and even sandboxed, untrusted JavaScript code execution.

Even when the browsers do a good job at protecting against attacks, sometimes trusted websites themselved may contain bugs that directly affect the security of their users.

A major threat, **Cross Site Scripting (XSS)** is a JavaScript code injection vulnerability where an attacker that found a way to post public HTML scripts into an unprotected website (e.g., by using comments forms or forum responses). Those scripts, if served to other visitors, will execute with the credentials of their respective users, making it possible for the attacker to scam, exfiltrate personal data or even push malware into the victim's PC.

Another typical client-side vulnerability that the web developers need to protect their websites against is **Cross-Site Request Forgery (CSRF) [https://research.securitum.com/what-is-the-csrf-cross-site-request-forgery-vulnerability/]**. In this attack, the victim is tricked into opening an attacker-controlled web page which then issues custom requests (either using concealed elements that do external requests - `img`, `form`, or by using JavaScript / AJAX) to another (target) website. The browser will happily make the requests using the target domain's cookies and credentials. If the target website has URLs that execute certain actions (e.g., `POST https://my-blog/post.php` [`https://my-blog/post.php`]) without verifying the source of the request, any malicious page can execute them. Note

that the attacker cannot see the results of those requests (unless authorized by CORS headers by the target). In practice, any URL endpoint executing sensitive actions needs to be protected using either referer validation or CSRF tokens.

Setup

You will be using a OpenStack VM [https://cloud-controller.grid.pub.ro/] for your tasks.

Remember that the instances have private IPs, `10.9.x.y`, inaccessible from anywhere but the campus network. Since we need to use a local browser to access a web server running inside the VM, we will employ a SSH tunnelling + proxy trick to accomplish this.

First, copy your private key from **fep** to your workstation (since authentication with the VM will be realized from your PC)
(*either do this, or assign a key existing on your machine to your OpenStack account and VM instance)*:

```
scp <first.lastname>@fep.grid.pub.ro:.ssh/id_rsa ~/.ssh/fep_rsa
```

Make sure to replace `id_rsa` with your actual key name (if you didn't use the default one).

Connect to the VM using (**ssh on Linux / WSL**):

```
ssh -i <fep key path> -L "8080:localhost:8080" -o 'ProxyCommand=ssh <first.lastname>@fep.grid.pub.ro -W %h:%p' student@10.9.X.Y
# after entering the password for fep and trusting the server's public key,
# you should see the 'student@host:~$' prompt!
```

If you use **OpenSSH for Windows**, you must use the full path for `ProxyCommand`:

```
ssh -i <fep key path> -L "8080:localhost:8080" -o 'ProxyCommand=C:\Windows\System32\OpenSSH\ssh.exe <first.lastname>@fep.grid.pub.ro -W %h:%p' student@10.9.X.Y
```

This uses *fep.grid.pub.ro* as a proxy (`-W` is stdin/stdout forwarding), successfully connecting to the SSH server running on the OpenStack VM (inaccessible otherwise) and requesting it to forward (`-L <local addr:port>:<remote addr:port>`) its HTTP server port (`localhost:8080`) back to your phisical machine (same port, for convenience)!

Note: replace the `<fep key path>`, `<first.lastname>` and `X.Y` placeholders with the actual path to your SSH key / username / VM IP address!
You can use the same trick to forward any port from a OpenStack VM to your local host!

For this lab, **you are required to take a screenshot proving each task's completeness** (usually, of the exploited webpage). For the SQL injection tasks, it would be nice if you saved a list of successful inputs in a .txt file. For the XSS / CSRF injection tasks, please include the HTML / JavaScript code you used into the archive.

Tasks

1 [20p]. SQL Injection

- Start the web server by using the following sequence:

```
# First, start the MySQL instance in background
docker run -d --rm --name mysql ropubisc/lab08-mysql
# Wait until the MySQL server fully starts:
docker logs mysql -f
# Ctrl+C and continue when mysql is 'ready for connections'


# Finally, start the sample web server
docker run -it --link mysql:mysql -p 8080:8080 ropubisc/lab08-web-server
```

- Connect to the application using http://localhost:8080/ [http://localhost:8080/] (assuming you forwarded the port correctly)
- Now: You don't know any user / password for this guestbook. Try to log in using SQL Injection!
- Hint: Try the apostrophe in one of the login fields, check if it shows an error!
- If you ever want to exit the MySQL server:

```
docker kill mysql
```

2 [20p]. Advanced SQL Injection

- Start the web server from the first task again.
- What if I told you there is a hidden **flag** inside the database? Find it!
- Hint: where do you have query feedback inside the application? try to do an `UNION` hack!
- Note: for UNION to work, you must SELECT exactly the same number of columns as in the original query!
  Check out the cheatsheets from the Background section + here's a UNION example:

```
SELECT col1, col2... FROM tablename WHERE name='' UNION (SELECT 1, 2, ...)  -- bruteforce the number of columns! '
```

(don't type this into the input field, extract the relevant parts!)

- Hint: first, you need to discover the names of tables in the database (to avoid extra steps, the schema name is `guestbook`); afterwards, find the column containing the flag, then extract it!

3 [20p]. Cross-Site Scripting

- Once logged in, try to do a Cross-Site Scripting attack using the message.
- Hint: The message accepts any HTML code!

4 [20p]. Cross-Site Request Forgery

- On your local station, create a simple HTML page that posts a hidden message to http://localhost:8080/guestbook/post [http://localhost:8080/guestbook/post]
- The user will click a button and the hidden message will be posted ;)
- Run the page using your local web browser and try it!
- (Make sure you are logged in inside the guestbook before you test it!)
- Hint: Check the HTML of the guestbook for a `<form>` example!
- Hint: Use an input with `type="hidden"`
- For bonus, you can try to do a cross-site AJAX posting the message automatically (without requiring user interaction!).

5 [20p]. Server Reconnaissance

- Can you steal the source code of the server-side code using HTTP only?
- Once you found it, try to find the database credentials!
- Hint: try to guess the path to a common file [https://docs.npmjs.com/files/package.json] that all NodeJS projects have! It may reference the main script's name!
- Also try it by using a tool: nikto [https://cirt.net/nikto2], `apt install nikto`

Feedback

Please take a minute to fill in the feedback form [https://forms.gle/5Lu1mFa63zptk2ox9] for this lab.

isc/labs/08.txt · Last modified: 2020/04/14 10:38 by florin.stancu