Security: Pen-testing 1

Made by Alex, Carmen, Daniel, & Jacob

Setup of server

We have made a script called control.sh. Here it has three functions. Start, stop and killdb. Start calls app.py to ensure that the app runs on the server. Stop kills the server to ensure that there is no longer anything running. Killdb removes the current database such that anything created on the database not longer exists until the app is run again and does its initial inserts.

Fixed vulnerabilities

Using `secrets`-library instead of rand

We chose to replace the rand-method with the `secrets`' modules method for getting a random number. This is because the rand method is actually predictable if the seed is known or guessed. `secrets` is also a module that is used to generate cryptographically strong random numbers suitable for managing data.

Fixing SQL injection

Most of the SQL queries used in the application were prone to SQL injection, e.g:

```
statement = "SELECT * FROM notes WHERE assocUser = %s;" %session['userid']
c.execute(statement)
```

We have fixed this vulnerability by making the queries parameterized:

```
statement = "SELECT * FROM notes WHERE assocUser = ?;"
c.execute(statement, (session['userid'],))
```

To do this when initializing the database, we can not use `executescript`, as it does not support parameterized queries. Therefore, we use execute instead.

Hash passwords

We have also fixed the vulnerability of storing passwords in plain text. We now hash the passwords before storing them in the database. We use `werkzeug.security`-module for this.

Do not check if or inform about passwords already in use when registering

We have fixed the vulnerability of informing the user if a password is already in use when registering. This was a security risk, as it could be used to guess passwords. Different users should be able to use the same password, so we have removed this check completely. The user will now be informed if the username is already in use, but not if the password is already in use, and the user will be able to register with the same password as another user.

Cross-site scripting attacks (XSS)

We had a look at XSS to see if it was necessary for us to sanitize the user input. After doing some research, we figured that Flask uses Jinja2 as the template engine, and because of the way a note is displayed, it would not matter if a user had submitted html or JavaScript code. Jinja2 automatically escapes variables to prevent XSS attacks.

Secure Session Cookies

We have set the session cookie to be secure, so it will only be sent over HTTPS. This is done by setting the `SESSION_COOKIE_SECURE` to `True` in the config. We also set the `SESSION_COOKIE_HTTPONLY` to `True` to prevent the cookie from being accessed by JavaScript.

Installed packages:

We use crontabs this has been installed by running apt install cron. We have this version: cron/focal,now 3.0pl1-136ubuntu1 amd64

We use werkzeug version 0.16.1 We use Flask version 1.1.1 We use Jinja2 version 2.10.1 We use python version 3.8.10 We use sqlite version 2.6.0

Introduced vulnerabilities

1: User Access - SQL Injection

We have set up the first part of the assignment where the attackers are intended to gain user access by using SQL Injection. We have done this by making the notes clickable, and upon clicking, it redirects you to the note API that provides the relevant metadata about that note. The URL is SQL injectable and can be exploited by changing the URL from something like `http://127.0.0.1:5001/note?noteid=6850822111` to `http://127.0.0.1:5001/note?noteid=1' OR '1'='1`.

2: Root Access - Command Injection

We have set up a CRON job that runs on the server using the student user's privileges. The student user has root privileges, so anything run by that user is executed with root permissions. Additionally, we have created another user on the server who only has access to edit the script executed by the CRON job.

Intended path for attackers

The intended way for our backdoors to work is for the attackers to figure out that the notes are clickable and that they are able to SQL-inject the way a note is displayed (through the note id). Doing this displays all the notes in the database, where some of the notes indicate that a user - `mrdata` - exists and that the password is also stored in one or more notes in some way. This is the way for an attacker to gain access to the server. `mrdata` is a user on the server with very limited permissions, but is allowed to edit our `backup.sh`-script that is also executed as a CRON-job by our `student`-user that has root privileges. Editing this script to somehow gain more access will therefore work.

Step-by-step guide

1. Create a new user on the webapplication and make *some* note.
2. Click the note to view its metadata.
3. Change the URL to something like `http://127.0.0.1:5001/note?noteid=1' OR '1'='1` which will display all notes in the database.
4. Locate the note telling you where to look for the password (password: `svoemmendejulemand`).
5. Locate the note telling you what the username to the server is (username: `mrdata`).
6. Access the server using the mentioned username and password. **This is user access gained**.
7. Now you are able to change the `backup.sh`-script that is ran as a CRON-job by the `student`-user. **This is root access gained**.