

School of Computing and Information Systems

COMP90074: Web Security, Semester 2 2023

Assignment 1 (worth 12.5%)

Due date: 11:59pm, Sunday August 27

1 Introduction

This assignment is one of two programming assignments in this subject (together worth 25% of your mark). This assignment is worth 12.5% of your mark. It is done individually.

You will find, analyse, exploit, and fix a range of XSS and SQL injection vulnerabilities in a simple web application. Your submission will include a short report (as a plain text file) plus client exploit code that you will write for this assignment.

Your code can be written in any language; however it should be packaged in a docker container with a script, or instructions on how, to run it. We provide a sample docker container for an exploit client written in Python, as a starting point.

2 The Web Application

The web application implements a simple authenticated, broadcast messaging system. Users log-in with username/passwords and can then post messages. All messages are displayed to all users; only logged-in users can see messages. Anyone can create a new account, allowing them to log-in and use the system. When creating an account, users are asked to supply a phone number (e.g. to be used for account recovery if they lose their password), which is stored along with the supplied username and password hash.

User account details (usernames, phone numbers, password hashes) are stored in an SQL database table `users`. A separate table `messages` is used to store the messages that have been posted.

Initial conditions. When the web application starts, there are no entries in the `messages` table and the `users` table contains a single `admin` account.

The `admin` account. The `admin` account is assumed to be trusted. It is allowed to post messages that contain rich HTML content. Messages posted by all non-`admin` users are filtered to prevent untrusted content leading to potential XSS vulnerabilities, etc. When the server application starts, a new password is generated for the `admin` account. This simulates that in reality passwords for `admin` accounts should be infeasible to guess, even if you have access to the source code for a web application.

Password storage. The Argon2id algorithm (mentioned in lectures) is used to hash passwords before they are stored in the SQL database. In addition, a random *pepper* value is prepended to each password before it is hashed. This pepper value is randomly generated each time the web application starts. It is not

stored in the database. In practice this means that if you create a user account with a particular password, and then stop and restart the server, clearing the database, and then re-create the user account with the same password as before, the hash stored in the database will be different to the hash originally stored.

3 Your Tasks

3.1 Find, Analyse, and Fix the Vulnerabilities (9 marks)

The application contains three known XSS/SQL injection vulnerabilities: at least one XSS vulnerability and at least one SQL injection vulnerability.

Your first task is to identify each vulnerability and to analyse it. What kind of vulnerability is it? What is its cause? How can it be exploited? If it is exploited, what bad things might an attacker be able to do?

Then, you should devise, implement and test fixes for the vulnerabilities.

You should write a short report as a plain text file `vulnerabilities.txt`. For each vulnerability you should document:

- The type of vulnerability (e.g., client reflected XSS / blind SQL injection, etc.)
- Where is it in the original source code (relevant files and line numbers) and its root cause (e.g., the vulnerability arises on line *X* due to [missing input validation on / constructing the SQL query via raw string manipulation of] the *Y* data, etc.)
- How the vulnerability could be exploited (e.g., this vulnerability could be exploited by an attacker [sending an email containing a malicious link to a user for instance `http://blah...` / submitting the log-in form using carefully crafted inputs for instance `username=blah...` and `password=meh...`] etc.)
- What the impact of the vulnerability could be (e.g., if exploited this vulnerability could allow an attacker to [extract sensitive information like *AAA* from the *BBB* table in the database / inject malicious code into the page that does *CCC*] etc.)
- How you fixed the vulnerability (e.g., I fixed this vulnerability by modifying line *X* to change `foo` to `baz`, and ..., etc.)

There are three marks allocated for each of the three known vulnerabilities: 2 marks for correctly describing the vulnerability, how it can be exploited, and its impact; and 1 mark for correctly fixing the vulnerability.

A bonus mark will be awarded to any student who identifies an unknown XSS/SQL injection vulnerability in the web application.

3.2 Exploit the Web Application (3 marks)

Understanding the impact of a vulnerability in isolation tells only part of the story. In reality, vulnerabilities are often most dangerous when they are leveraged together by an attacker to achieve something that couldn't have been achieved by exploiting each vulnerability on its own.

The vulnerabilities that this application contains can be leveraged together by an attacker to allow a non-`admin` user (i.e., an arbitrary person on the web who doesn't know the `admin` password) to cause malicious code to be run by *any* logged-in user of the application.

Your task is to work out how to combine the vulnerabilities you have discovered to achieve this effect.

To do this, you need to implement a client exploit program that sends and receives HTTP requests/responses to the server. Your client begins not knowing the `admin` password. Its job is to cause code, created by your client, to be executed by any logged-in user who visits the web application.

To do so, it may or may not need to exploit all of the vulnerabilities you identified.

Your client will be a *proof-of-concept*: that is, a program that demonstrates the potential for harm without actually causing any harm. Specifically, your client's job is to inject code so that any logged-in user of the web application sees a popup alert saying "This site has been hacked!".

Your client can be written in any programming language you choose. However, it *must* be packed in a docker container that comprises all dependencies required to run it (i.e., your client should be "dockerised"). We provide a sample docker container for a client written in Python that uses the `requests` library to send and receive HTTP requests to the server.

Handling Session Cookies Note that when a user logs in to the web application, a session cookie is set and is returned in the response. However, the response after logging in (that contains this cookie) is a 302 redirect response that redirects the client to load the login page.

Some libraries (including the Python `requests` library) will automatically follow redirects. But in doing so, the final response they return may not contain the session cookie that was set after logging in.

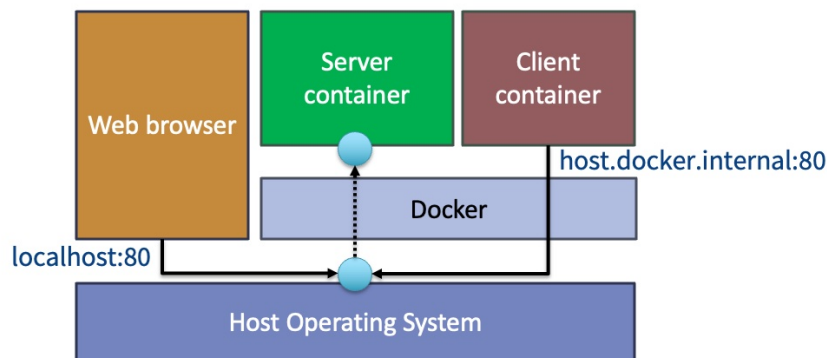
Therefore, in such cases, if your exploit client is logging-in to the application, it may need to manually disable redirect following when sending the login request, so that it can extract the session cookie from the response. That session cookie will need to be included in subsequent requests to the application that the client makes, after logging-in.

Client communication from its own docker container. This means your client will be running inside its own docker container. To communicate with the server, this means your client needs to send requests to the host `host.docker.internal`.

The server is also running inside its own docker container. It listens on port 80. The server can be accessed from outside the server docker container because when the server is started, docker instructs the host operating system to forward requests to port `localhost:80` to port 80 inside the docker container. This is why you can connect to the server by visiting `localhost` in your web browser.

The exploit client, inside its own docker container, can talk to the host operating system by connecting to `host.docker.internal`. If it connects to `host.docker.internal` on port 80, then that connection is forwarded to port 80 inside the server docker container, thereby allowing the client to communicate with the server even though they are both inside docker containers.¹

This situation is depicted in the diagram below.



Running your client. You should provide a `run_client.sh` script that executes the necessary docker commands to start your client. A sample `run_client.sh` script is provided for the sample dockerised client in Python.

Alternatively, you can document the necessary commands in a `README` file.

¹Of course there are other ways the client could communicate with the server using docker networking features; however, the above scenario is perhaps simplest for our purposes.

We will test your client by first starting the original, vulnerable server. Then in a separate terminal we will run your client. After running your client, we will connect to the server in a web browser and create a new user account and log-in. After logging in, we expect to see an alert popup message appear in the browser saying “This site has been hacked!”.

Clients that successfully carry out the entire exploit will receive 3 marks. Those that demonstrate an exploit of at least one identified vulnerability will receive at most 1 mark. Those that demonstrate exploitation of at least two vulnerabilities but without achieving the entire exploit as described above will receive at most 2 marks.

3.3 Clarity (0.5 marks)

The remaining 0.5 marks will be awarded for solutions whose report and exploit client are both clearly written and well-documented.

4 Submission

You should submit a single ZIP file that contains at least the following:

- `vulnerabilities.txt` : Your vulnerability report.
- `Dockerfile` : The docker file for your client.
- `run_client.sh` or `README.md` : A script or instructions for running your dockerised client.
- ...remaining code for your client ...

You should *not* submit your fixed version of the server. Instead your fixes should be clearly and unambiguously described in your `vulnerabilities.txt` file.

5 Academic Integrity

All work that you submit must be your own or if derived from elsewhere it must be appropriately acknowledged (see below).

You can discuss the assignment with others to help clarify your understanding of what is being asked. You *cannot* share your answers or your thinking about how to answer a particular part of the assignment with others.

You can make use of certain online resources to help you write code for this assignment, *so long as you acknowledge the use of such sources and clearly document which code came from them*. These include passive searches using Google, or asking questions to ChatGPT, Github Copilot and similar tools.

All code derived from such sources needs to be appropriately labelled, including the source from which it was derived and how (see below). Otherwise you risk an academic misconduct allegation.

For example, if you derive part of your code from ChatGPT, Copilot, etc., then you need to add comments identifying which pieces of code were derived and what was the prompt you used to query ChatGPT, Copilot, etc.

Similarly, if you derive code from code you find on the web, you need to add comments identifying which code was derived and the source from which it was derived (i.e., a URL that points to the original code / stackoverflow post, etc.)

This also applies to code that you derive by looking at the subject materials (lecture demo code, workshop/tutorial code, lecture slides, tutorial handouts etc.). If you derive your code from such sources, you must label which code was derived this way and the original source.

To repeat: **all derived code that you submit needs to be appropriately labelled, including the source from which it was derived and how it was derived. Otherwise you risk an academic misconduct allegation.**

If you are unsure, email the subject coordinator to ask.

6 Late Penalty and Extensions

Late submissions will attract a 10% penalty (1.25 marks) for each day they are late.

If you require an extension (e.g., for medical reasons, or other reasons supported by the university's extension policy, see <https://ask.unimelb.edu.au/faq/5667/applying-for-an-extension/>) of less than 10 days then you should email the subject coordinator *as soon as you become aware of the circumstances that have impacted your ability to complete the assignment*. Don't wait until the last minute to ask for an extension.