UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Fall 2020**

ASSIGNMENT 2

Updated Wednesday, November 4th, 2020 9:00 am

Milestone due date: **Monday, November 2nd, 2020 3:00 pm**
Assignment due date: **Friday, November 13th, 2020 3:00 pm**

Total marks: 92
Written Response Questions TA: Ezzeldin Tahoun
Programming Question TA: Matthew Rafuse

TA Office Hours: Wednesdays 11:00 - 12:00 EDT

Please use Piazza for questions and clarifications. We will be using Big Blue Button for TA office hours this term; we have separate online rooms for the written and programming parts. To attend office hours, access the corresponding URL for that assignment part and use the corresponding access code when prompted:
Written: https://bbb.crysp.org/b/you-9uu-vgr - Access code: 452674
Programming: https://bbb.crysp.org/b/you-ggq-k6d - Access code: 450936
When asked for your name please enter both your first and last name as they appear in LEARN.

What to Hand In

Using the “submit” facility on the student.cs machines (**not** the ugster machines or the UML virtual environment), hand in the following files for the appropriate deadlines:

1. **Milestone:**

   exploit1.tar Tarball containing your exploit for the first question of the programming assignment part. **Note that you will not be able to submit the tarball for the first exploit after this deadline.** To create the tarball, for example for the first question, which is developed in the directory `exploit1/`, run the command

   ```
   tar cvf exploit1.tar -C exploit1/ .
   ```

   (including the `.`). You can check that your tarball is formatted correctly by running “`tar -tf file.tar`”. For example (note that there are no folder names in the output):

   ```
   $ tar -tf exploit1.tar
   ./
   ./exploit.sh
   ./response.txt
   ```

2. **Rest of assignment:**

   a2.pdf: A PDF file containing your answers for all written questions. It must contain, at the top of the first page, your name, UW userid, and student number. **You will lose 3 marks if it doesn’t!**

   Be sure to “embed all fonts” into your PDF file. The file must be a valid PDF file, renaming a txt file to pdf will be corrupt and unreadable. Some students’ files were unreadable in the past; **if we can’t read it, we can’t mark it.**

   exploit{2,3,4,5,6,7}.tar: Tarballs containing your exploits for the rest of the programming portion of the assignment.

   To create the tarball from a directory that contains the exploit directories (`exploit1/`, `exploit2/`, ... `exploit7/`), run the command:

   ```
   for i in $(ls -d */); do tar cvf ${i%/}.tar -C {i} .; done
   ```

Written Response Questions (47 marks)

Note: Please ensure that written questions are answered using complete and grammatically correct sentences. You will be marked on the presentation and clarity of your answers as well as on the correctness of your answers.

Question 1. [Total: 10 marks]

Alice, Bob, Eve, Carol and Mallory all work for the same organization. The sensitivity levels within this organization form the following hierarchy:

Confidential (C) $\leq$ Secret (S) $\leq$ Top Secret (TS)

There are three types of documents stored in this system: Greenland (G), United States (US), Mexico (M). The clearance levels of each of the organization members are as follows:

- Alice: (Top Secret, $\{$G, M$\}$)
- Mallory: (Confidential, $\{$G, M$\}$)
- Carol: (Top Secret, $\{$US, M$\}$)
- Bob: (Confidential, $\{$G, US$\}$)
- Eve: (Secret, $\{$US, G$\}$)

1. (5 marks) Using the **Bell-La Padula Confidentiality Model**, state whether **Eve** has no access, read access, write access, or both to the files with the following integrity:
   (a) (Top Secret, $\{$US, G, M$\}$)
   (b) (Confidential, $\{$G$\}$)
   (c) (Top Secret, $\{$US,G$\}$)
   (d) (Secret, $\{$US, G$\}$)
   (e) (Confidential, $\{$US, M$\}$)
2. (5 marks) Using the **Dynamic Biba Integrity Model** with the low watermark property described in the slide handouts, describe how the integrity level of **Carol** and each file changes after each of the following operations (in order with changes persisting between accesses).
   (a) Carol reads from record1 having integrity: (Top Secret, $\{$US, G$\}$)

(b) Carol writes to record2 having integrity: (Confidential, {US})

(c) Carol reads from record3 having integrity: (Top Secret, {US,M})

(d) Carol writes to record4 having integrity: (Top Secret, {US, G, M})

(e) Carol reads from record5 having integrity: (Confidential, {US})

Question 2. [Total: 15 marks]

The organization is worried about the security of their password authentication mechanism. They require their users to use passwords that are ~~exactly at least~~ exactly 8 characters long and may only contain A-Z, a-z, 0-9 and the 8 special characters $\{!, \#, \$, \%, \&, (,), *\}$. The password database contains a pair $< uid, Hash(pwd) >$ for each user where the hash function used to encrypt the passwords is SHA-256.

In the threat model, they assume an attacker with access to a machine that can compute 5 million hashes per second and his goal is to recover as many passwords as possible.

1. [2 marks] The attacker comes across the following hash dumps from the database. With some guess work, he is able to find out the passwords. Find the passwords that hash to each output. Note that some passwords can be up to 12 characters long. (**HINT**: Try to search for common passwords)

```
<1 :  17f80754644d33ac685b0842a402229adbb43fc9312f7bdf36ba24237a1f1ffb>
```
```
<2 :  e4ad93ca07acb8d908a3aa41e920ea4f4ef4f26e7f86cf8291c5db289780a5ae>
```
```
<3 :  15e2b0d3c33891ebb0f1ef609ec419420c20e320ce94c65fbc8c3312448eb225>
```
```
<4 :  5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8>
```

2. [2 marks] Assuming now that users choose their passwords at random, how many years would it take for the attacker to crack one specific hash by brute force, on average? How many machines would he need to crack the password in one hour?

   It seems that the attacker needs to step up his game. He decides to build a hash table i.e. for all possible passwords, he stores the $< pwd, hash(pwd) >$ in a file. This way, anytime he is given a hash, he can quickly look it up in the table and recover the associated password.

3. [2 marks] How many gigabytes of data does the attacker have to store?

   Seeing that he might not have enough disk space to store the hash table, the attacker goes on to make a rainbow table, a space efficient data structure for storing precomputed hash values.

   A rainbow table consists of k chains. Chains are constructed using both a hash function H and a reduction function R. The reduction function R deterministically transforms a hash to a valid password. Each chain in a rainbow table starts with a random valid password called the *root*. Next, the hash of the root password is computed and the reduction function is applied to it to get a new password. This process is repeated $l-1$ times resulting in a chain of length l. The last password in the chain is called the *endpoint*. The following is an example of a rainbow table with 3 chains of length 3 when the reduction function simply takes the first 8 charachters of the hex encoded hash as a new password. Note that the values are not the actual hashes but only examples.

(1)	rootroot $\xrightarrow{H}$ 12dergms..rve8 $\xrightarrow{R}$ 12dergms $\xrightarrow{H}$ A6g2n!gz..fg09 $\xrightarrow{R}$ A6g2n!gz
(2)	free2020 $\xrightarrow{H}$ zis85Bqc..v5yz $\xrightarrow{R}$ zis85Bqc $\xrightarrow{H}$ Rt7bo6ww..ica9 $\xrightarrow{R}$ Rt7bo6ww
(3)	realfake $\xrightarrow{H}$ mnUsag90..1bax $\xrightarrow{R}$ mnUsag90 $\xrightarrow{H}$ Cb9\$b0Om..pd3t $\xrightarrow{R}$ Cb9\$boOm

To store a rainbow table, it is enough to keep the root and the endpoint of each chain. To find any password, the attacker essentially traverses a chain in reverse. Given a specific $hash$, the attacker computes $R(hash)$ and searches among the endpoints for a match. If a match is found, the corresponding chain is reconstructed from the root to get to the password. If no match is found, the attacker computes $R(H(R(hash)))$ and again searches for a match and repeats this process (maximum $l - 1$ times) until a match is found.

4. [2 marks] For simplicity, assume that the attacker's rainbow table uses a reduction function such that each possible password is represented exactly once. If the attacker opts for chains of length $l = 60000$, how many gigabytes does the attacker have to store?

5. [2 marks] How long does it take for him to crack one specific password? (Assume that the reduction functions have no computation cost)

6. [3 marks] Compare these numbers to your results from (2) and (3). How does the rainbow table help? What is the inevitable cost that the attacker has to pay?

7. [2 marks] The organization's engineers decide to upgrade the username/password database. Instead of saving $< uid, Hash(pwd) >$, they now save the triple $< uid, salt, Hash(pwd||salt) >$ where $||$ means concatenation and $salt$ is just a random 64-bit number. Explain how this protects against a rainbow table attack?

Question 3. [Total: 12 marks]

After a recent breach of security, and loss of several confidential files, the organization has decided to set up its firewall again. You are tasked with this ordeal alongside the current network security expert Bruce Wayne. The organization owns the IP address range 16.18.20.0/25. The following are the network functionalities that are required for day-to-day operations:

- All employees of the organization should be able to browse the internet from within their network.
- Their public webpage which is hosted on an internal server (with the IP address 16.18.20.25 and served with HTTPS) must be accessible on the internet.
- No Employee should be able to use an internal mail server (with the IP address 16.18.20.85 and served with SMTP) from outside the network.
- The organization only trusts a special DNS server (located at the IP address 53.16.71.12) hosted by an allied organization to handle all of its DNS lookups. This DNS server is unique in that it serves requests on port 1773 and also expects the clients to send these requests from port 6000.
- The organization also maintains an IRC channel (on port 3223 of a server with IP address 16.18.20.10) which is meant to facilitate communications of their covert agent (with the IP address 8.18.10.218) with the rest of the organization.

1. [2 marks] Bruce says we should reinstate a blacklist with a list of all known malicious IP addresses along with the source IP address of the recent breach to protect against any future attacks. Is this a solid defense strategy? Elaborate why or why not? Based on that what do you advise be the default rule on the firewall?
2. [2 marks] While configuring the firewall, you get a request to give access to an internal FTP server to a range of IPs owned by the new branch of the organization in a different location. With further inspection, you find out that there have been outbound responses from the FTP server. What kind of attack is done on the organization? How does the request exacerbate it? What can you suggest to prevent such attacks?
3. [6 marks] Go ahead and configure the firewall by adding the required rules to meet the aforementioned requirements. Rules must include the following:
   - DROP or ALLOW
   - Source IP Address(es)
   - Destintation IP Address(es)

- Source Port(s)
- Destination Port(s)
- TCP or UDP or BOTH

Here is an example rule to allow access to HTTP pages from a server with IP address 5.5.5.5:

ALLOW 5.5.5.5 => 16.18.20.0/25 FROM PORT 80 to all BY TCP

(**HINTS:**

- CIDR Notation may be helpful for this portion of the assignment.
- Some requirements may need more than one rule.
- Ports can be specified as a singular value, range, as a set, or as ‘all’ as seen in the example above.

4. [2 marks] After setting up the firewall, Bruce suggests to migrate the DNS server to an internal server and also make it possible for employees to access the mail server from outside the organization. Using a DMZ (which includes your Web, FTP, Mail and DNS servers), you configure two firewalls. But instead of dropping illegitimate requests to the DMZ on the external firewall, you reject those requests and respond with ~~eiher a TCP ”RST” or a UDP ”Destination Unreachable”~~ an ICMP ”Destination Unreachable” packet. Analyze how can this affect the security of your network.

Question 4. **[Total: 10 marks]**

Consider the following scenario:

A computer is shared between Chris and Bob. Chris has an administrator account, Bob has a user account. For the purposes of this question:

(i) The hard disk the operating system is installed on is encrypted using an open-source implementation of AES.

(ii) Administrator accounts are given privileges to operate in read, write and execute modes on all files.

(iii) A discretionary access control protocol is being followed for **user files** in which a file's creator may specify access for other users.

(iv) Services are background processes, and are provided with all available privileges (i.e. they run at an administrator level regardless of function).

(v) File/memory access is checked for programs without administrator privileges.

(vi) Only programs validated and digitally signed by a trusted third party can be installed by users.

(vii) Unsigned programs may be installed by administrators.

Note these do not all necessarily constitute a vulnerability or violation. Consider the OS design policies in the course slides (Module 3, Part 5) whilst answering the questions below.

1. Is the concept of least privilege followed here? What about complete mediation? Explain your answers. **[4 marks]**
2. A file creation service assigns read and write permissions to every file it creates by default, unless specific permissions are provided to the service when called. Which OS design policy is violated and how? **[2 marks]**
3. Describe two potential vulnerabilities of this system and explain a potential solution for each. **[4 marks]**

Programming Response Questions [42 marks]

Background

You are tasked with performing a security audit of a custom-developed *web application* for your organization. The web application is a content sharing portal, where any user can view content which includes articles, links, images, and comments. Registered users can log in and then post content. Note that users in the process of using the website can inadvertently leak information that may assist you in solving your tasks. You have been provided black-box access to this application, that is, you can only access the website in the same manner as a user.

The website uses PHP to generate HTML content dynamically on the server side. The server stores the application data, which includes usernames, passwords, comments, articles, links and images, in an SQLite3 database on the server. There are two systems, a relational database and HTML forms, that by default do not validate user inputs.

You will be writing exploits that exploit various vulnerabilities in these underlying systems. The Background Knowledge section contains references for all background knowledge that is necessary to develop your exploits.

Web Application Setup

A web server is running on each of the ugster machines, and a directory has been created using your ugster userid. Your copy of the web application can be found at:

```
http://ugster40X.student.cs.uwaterloo.ca/userid
```

where `ugster40X` and `userid` are the machine and credential assigned to you previously from the Infodist system.

- **Connecting to the ugsters:** As with the previous assignment, the ugster machines are only accessible to other machines on campus. Since all of you will be working off-campus this term, you will need to first connect to the ugsters through another machine (such as `linux.student` or the campus VPN). You can use SSH Tunnelling to forward website access to your local machine:

  ```
  ssh -L 8080:{UGSTER_URL}:80 linux.student.cs.uwaterloo.ca
  ```

  This allows you to access your instance of the website at `http://localhost:8080`.
- **Resetting the web app:** If you need to reset the webserver to its original state, such as to

debug question 3 or to remove all new posts, comments and votes, simply access this URL:

`http://ugster40X.student.cs.uwaterloo.ca/reset/userid`

- **Do not** connect to the web server with a userid different from the one that is assigned to you.
- **Do not** perform denial of service attacks on either the ugster machine or the web server.
- Please avoid uploading large files to the web server as it has limited disk space. Any student caught interfering with another student's application or performing DoS attacks in any manner, will receive an **automatic zero** on the assignment, and further penalties may be imposed.

Writing Your Exploits

We recommend using the command-line utility `curl`[1], as you can solve all questions with just tweaks in command-line options and input arguments, with no need to interface HTTP-request processing libraries. However, you may use any language that is installed on the ugster machines to write your code. The only restriction is that your scripts should run correctly on the ugster machines in order to exploit the server, *without installing additional dependencies*. You may use external tools to help you gather information during programming, etc. Note that when tested, your scripts will not have access to further download and run any code.

Assignment Submission Rules

1. Each question lists **required** files. You can submit more files if needed. For instance, if you use anything other than a command-line utility (e.g. Python), you will need to include your source-code files in that language.
2. We have specified names for each required file; these names must be used. Submitting files with different names will result in a penalty.
3. All questions require a script file `exploit.sh` and a text file `response.txt` to be submitted.
    (a) `exploit.sh` - Our marking scripts will execute this file to mark the corresponding question. If you are using `curl`, then your script file `exploit.sh` for each question should invoke `curl` directly, passing the arguments supplied to the script. If you are using an interpreted language, you should include your source code files in that language in your exploit tarball, and your script file should invoke the interpreter on those files, taking care to pass the URL argument.

    For all questions, your `exploit.sh` has to accept **one** parameter — a URL of the

[1]Note that the versions of `curl` on the linux.cs student machines and the ugsters are different; you should ensure that in your script, you run the version of `curl` that is on the ugster machine.

website under attack. While you are working on the assignment, the URL under attack will be:

`http://ugster40X.student.cs.uwaterloo.ca/userid`

However, during marking it will be something else (but it will be a valid URL). Also note that there will be **no** trailing slash in the end of the URL.

You must not hardcode the URL of your ugster machine anywhere in your exploit files (including in HTML/JavaScript). Hardcoding will result in a penalty, if your submission can be graded at all.

For example, if you are using `python`, your source-code file named `exploit.py` can be invoked like this within your script: `python3 exploit.py $1`. Refer to the Background Knowledge section for help on languages and scripting.

(b) `response.txt` - Your exploit should output the HTTP responses when *you* tested your code into this file. In case your script does not work on the ugsters in our marking setup, we may use your submitted file, along with the main script and the regenerated file, to determine what went wrong and give partial marks. It should contain the HTTP response headers for all HTTP requests that are sent in the exploit. Refer to the Background Knowledge section for background on all of these terms.

You should include the following *response* headers (the *values* of the headers are for a request sent for Question 1a):

Protocol, HTTP response status code → HTTP/1.1 302 Found
"Set-Cookie" header → Set-Cookie: CS458=j0kndm5bc5g94aq5pjadufmlcj; path=/
"Location" header → index.php
"Content-Length" header → Content-Length: 18
"Content-Type" header → Content-Type: text/html; charset=UTF-8

4. For each part of each question, submit your files as a tar file with the files at the top level (do **not** submit your files inside a folder). Submitting files inside a folder will result in a penalty. To create the tarballs correctly, refer to the instructions at the start of the assignment under What to Hand In.

Question dependencies: Question 1 is required to solve all questions. (All questions require some combination of logging-in as a user or impersonating one and then either creating a post or a comment or an upvote.) None of the other questions require that you have solved a previous question; so, if you struggle with one of these questions, proceed to the next one. However, a given question may require previous parts of the same question to be solved first.

Questions

1. [Total: 12 marks] **Get, Set, Go!**
   This question is designed to help you write setup code that is essential for solving all questions. To solve this question, you should experiment with your site and examine the form fields sent in GET or POST requests for logging-in, creating a post, commenting on a post and voting. **You should go through the Background Knowledge section before attempting to solve this question.**

   For this question, please use the following credentials:

   ```
   username:  alice
   password:  password123
   ```

   Write a *single* script named `exploit.sh`, that takes in the URL of a copy of the web application as an argument and performs the following tasks in the given order:

   (a) (2 mark) logs in as the user `alice` with the above password.

   (b) (4 mark) creates an *article* post as that user.

   (c) (4 marks) creates a comment on the post made in part 1b.

   (d) (2 mark) upvotes the post made in part 1b.

   Note that you will only get half marks for each of 1c and 1d if you comment or upvote on any post other than the one made in 1b. You should test your script with the input URL being the URL for your copy of the web application, that is, `http://ugster40X.student.cs.uwaterloo.ca/$userid`. Note that no ending forward slash will be supplied in the input URL. We will be testing your script from a different, but valid URL, without the ending forward slash. Your script should concatenate the HTTP response headers corresponding to the HTTP requests for each of the above parts into a single file named `response.txt`.
   What to submit: `exploit1.tar` file which contains the following files (**not** inside a folder): `exploit.sh` script and the `response.txt` text file.

User Impersonation

Your organization has heard of the dangers of allowing arbitrary internet visitors to sign-up to their website, and thus they have now removed the signing-up feature for this website. However, an arbitrary visitor to this website could still *impersonate* an existing user. The next three questions examine different ways that a visitor could do this.

2. [Total: 4 marks] **Easily Guessable Password**
   The site registration process did not enforce any password hardness measures. You are concerned that some user may have a password that is very easy to guess or discover. If

this is the case, any other user could log in and post as them! Check out the website and identify the user with the weak password. Note the following:

- The username-password pair used in question 1, is not the correct answer (`alice` and `password123`).
- The password is easy to guess *by human* users who are exploring the website, so simply using rainbow tables, which are generic, or brute-forcing is not the correct approach here.

Once you've guessed the password, write a *single* script that does the following:

(a) (2 marks) Logs-in to the website as that user. Saves the response from the log-in request into a file named `response.txt`. This text file should contain the username and the password.

(b) (2 marks) Creates a post on behalf of that user.

What to submit: `exploit2.tar` file which contains the following files (**not** inside a folder): `exploit.sh` script and the `response.txt` text file.

3. [Total: 6 marks] **Confirmation Code**

   One of the users on the website has been inactive for some time and now must confirm their account using a confirmation code in order to be able to use the website. This confirmation code should be pseudorandom, however, the website developers may have used deterministic inputs to compute the confirmation code for any user. Note that a confirmation code can only be used once, so while debugging this question, you should reset your website after each attempt.

   Identify an inactive user by examining the website. Write a script that:

   (a) (2 marks) Includes a simple function that computes the confirmation code for any user.

   (b) (2 marks) Logs in as the inactive user, using the confirmation code generated by running your function in part 3a for that user. Save the response from the code confirmation request into a `response.txt` file.

   (c) (2 marks) Creates a post on behalf of the user you just impersonated.

   What to Submit: `exploit3.tar` file which contains the following files (**not** inside a folder): `exploit.sh` script and the `response.txt` text file.

4. [Total: 6 marks] **SQL Injection**

   User-generated data is directly passed into database queries in several contexts. For example, to insert or modify user-generated content in a database, or to query a database that contains user credentials for authentication. A user can craft their input to include malicious database queries, such as to alter queries or insert/modify records. Refer to the Background Knowledge section for necessary background and examples on SQL and SQL injection attacks.

   Write a script that:

(a) (4 marks) Uses an SQL injection attack on the log-in form to log in as the user "mrafuse". Saves the response from the log-in request into the file `response.txt`. Note that you will only receive half of the total marks for this question if you log-in as another user. You must log in as this user to receive full marks.

(b) (2 marks) Creates a post on behalf of the "mrafuse" user.

What to submit: `exploit4.tar` file which contains the following files (**not** inside a folder): `exploit.sh` script and the `response.txt` text file.

5. [Total: 14 marks] **Advanced SQL and Path Traversal**

A web application often hosts many files that are not webpages (that is, in PHP or JS) and are intended to remain confidential. Website developers often forget to configure the web server to restrict access to these files during the processing of HTTP requests. As a consequence, any visitor to the website might access confidential files and directories; this is known as a path traversal attack.

In this question, you will conduct a simple path-traversal attack to learn sensitive information about the content and structure of the database. Based on this information, you will execute an SQL injection attack to stealthily insert a new user into the database.

(a) (2 marks) Examine the content of the web portal for possible paths for the SQLite database. Write an exploit script that obtains a dump of this database and saves it as `data.db` alongside the exploit file.

(b) (12 marks) By examining the structure of the database file from part 5a, construct SQL queries that insert a new user into the database of the website, such that the new user is indistinguishable from a real user registered on the website. (You may need to insert the user into multiple tables in order to satisfy the above condition.) Include these queries as human-readable strings within your file. Again, you can refer to the Background Knowledge section for necessary background and examples on SQL and SQL injection attacks.

Next, identify an appropriate target for your second SQL injection attack. The log-in page is vulnerable to the SQL injection attack in question 4. The confirmation codes need to be preserved over time, and thus are probably stored in the database. Therefore, the confirmation page would likely also be using SQL queries to obtain the correct confirmation code as well as to reset the activation status of a user upon successful confirmation.

i. (8 marks) Conduct an SQL injection attack on one of the log-in or confirmation pages to insert a new user into the database, using the queries that you constructed above. Save the username and the password of the new user into the file `credentials.txt`. Note that you will need to go through either the previous part (5a) or question 3 to obtain information to include in your SQL injection.

ii. (4 marks) Log-in as the new user that you created, using the known username and password, and create a post as that user.

What to submit: `exploit5.tar` file which contains the following files (**not** inside a folder):

- `exploit.sh` — shell script that downloads the database, injects a user into it, logs-in as that user and creates a post by that user.
- `data.db` — database file downloaded from the server.
- `credentials.txt` — file containing the username and password of the new user. Include username on the first line and password on the second line, followed by the new line character and nothing else.
- `response.txt` — file with concatenation of all HTTP response headers from the server.

For the rest of the assignment, please use the following credentials:

```
username:  alice
password:  password123
```

6. [Total: 16 marks] **Cross-Site Scripting Attack**

Cross-site scripting attacks involve injecting malicious web script code (known as the "payload") (including HTML, Javascript, etc) into a webpage via a form. As a result, the document object model (DOM) of the HTML webpage changes whenever the code is executed. Depending on whether the changed DOM persists across reloads of the webpage, XSS attacks are classified as persistent or non-persistent attacks.

In parts 6a and 6b, you will be writing Javascript functions that modify the webpage (that is, affect the *integrity* of the webpage) and log the user's presence on that webpage (break *confidentiality* of the client-side cookie) respectively. These functions will then form the "payload", by being included in a post, such that all users clicking on the post will be victims of two XSS attacks. Refer to the resources for Questions 6 and 7 in the Background Knowledge section for necessary background, examples and sample code for XSS attacks.

(a) (4 marks) *Non-persistent XSS:* Write a Javascript function in a Javascript file named `xss.js`, that first checks if it is running on a page with the URL `view.php?id=<any post id>`. Otherwise, it should return immediately without doing anything. If the function is running on a `view.php` page, then it should modify the webpage HTML (not the database) so that it appears as if the post was written by the user "bob". That is, the function should replace the webpage's text "posted by __" with "posted by bob".

(b) (8 marks) *Persistent XSS:* You will write two Javascript functions, that when chained together, will allow you to leak the user's cookies in a comment on any page with the URL `view.php?id=<any post id>`. (Note that again, these functions should do nothing on pages that do not have this URL and all functions should be included in the `xss.js` file.)

i. (2 marks) Write a JavaScript function `getID` that returns the integer ID of the current post when run on any page with the above URL.

ii. (6 marks) Write a JavaScript function `leakCookies` that accepts a post ID `postID`, and leaves a *comment* on the `view.php?id=postID` page that contains *all* cookies for this website.

Your functions, when composed like this: `leakCookies(getID)` should leave the user's cookies as a comment on a page with the URL `view.php?id=<any post id>`.

(c) [4 marks] Write a shell script that logs-in as the user `alice` and creates a new post that contains and runs the JavaScript functions from parts 6a and 6b. When a user views this post on a page with the URL `view.php?id=<new post id>`, it should appear as if it was written by "bob" and the user's browser should automatically leave a comment with all of their cookies for this site.

What to submit: `exploit6.tar` file with the following files (**not** inside a folder):

- `xss.js` — File with the Javascript functions as described in parts 6a and 6b. Please keep the code in this file readable.
- `exploit.sh` — Shell script that creates the new post as described in 6c, using the code from `xss.js`.
- `response.txt` — File with concatenation of all HTTP response headers from the server.

7. [Total: 14 marks] **Cross-Site Request Forgery Attack**
Consider the following attack: the user `alice` has logged-in to our web portal in one tab on her browser and in another tab, she views a promotion email that contains a link to the portal. This link can upvote content or create many posts on behalf of Alice, without her knowledge. If Alice clicks on a link for the portal while being logged in to it, then the CS458 cookies which were used for authenticating her to the client would also be sent to the portal. This is a cross-site request forgery (CSRF) attack. Here, the attacker exploits the fact that state information maintained by the user's client (browser) for a given domain is sent with subsequent requests to that domain, *even* if the user did not intend to send the request.

As we do not currently host other means of delivering the link, e.g. through emails, we will host these links on the web portal itself. (That is, we use the web portal as both the attacker-controlled site, which hosts the malicious links, and the target website, which is affected by the user's unintended behaviour.) Specifically, in the following questions, we host links that *execute code* on the web portal, by exploiting stored XSS vulnerabilities, which we explored in question 6b. URLs on the website are created using the html tag `<a>`, as follows: `<a href="[URL]">[TITLE]</a>`. For all parts of this question, you can assume that your

`<a>` tags will be clicked/executed from the main `/index.php` page of the website by a logged-in user.

Refer to the resources for Questions 6 and 7 in the Background Knowledge section for necessary background, examples and sample code for CSRF attacks.

(a) (4 marks) Write a script that creates a post, such that whenever a user clicks on the `<a>` tag, the post with id "2" is upvoted by 1000 points on behalf of that user. **You should not use any JavaScript for this question.** Essentially, you will be replacing the **[URL]** with HTML code that performs upvoting. Include the HTML code for the `<a>` tag in a separate file named `upvote_url.html`.

(b) (4 marks) Similarly to 7a, create a post, such that whenever a user clicks on the link, a *new* post is created on behalf of that user. **You may use JavaScript for this question.** As before, include the HTML (and any Javascript) code for the link in a separate file named `post_url.html`; for this question, also include any Javascript code that's in this HTML file, in a human-readable format (with proper formatting) in a file named `post_human.js`.

Note that clicks on `<a>` links result in GET requests by the browser, which might not work for creating a post. There are some tricks how to override this behaviour of the `<a>` tag, for example, as in this post on stackoverflow.

(c) (6 marks) Create a new post, which contains the `<a>` link with the malicious URL similar to the one in part 7b. **You may use JavaScript for this question.** Construct it such that whenever a user clicks on the link, a new post that has the *same link* is created on behalf of the user who clicked on the link. So, the link in the new post should create another new post, which has a link which will create another new post, which has a link... and so on, recursively.

As before, include the HTML (and any Javascript) code for the link in a separate file named `recursive_post_url.html` and also include any Javascript code that's in this HTML file, in a human-readable format in a file named `recursive_post_human.js`.

Go through the resources for Questions 6 and 7 in the Background Knowledge section for help.

What to submit: `exploit7.tar` file with the following files (**not** inside a folder):

- `upvote_url.html` — `<a>` tag with [URL] substituted for 7a.
- `post_url.html` — `<a>` tag with [URL] substituted for 7b.
- `post_human.js` — A human-readable version of the JavaScript code contained in the [URL] section of the `<a>` tag in 7b.
- `recursive_post_url.html` — `<a>` tag with [URL] substituted for 7b.
- `recursive_post_human.js` — A human-readable version of the JavaScript code contained in the [URL] section of the `<a>` tag in 7c.

- `exploit.sh` — Shell script that creates three new posts, ones for each of the three parts, using the HTML files.
- `response.txt` — File with concatenation of all HTTP response headers from the server.

Background Knowledge

- Question 1: You must familiarize yourself with HTTP request methods (GET and POST for our application) and sending data through the `<form>` element. Refer to the HTTP status response codes while debugging this question. Go through the first section on MDN Cookies to understand why cookies are necessary and how they work. You should figure out how to access the cookies set by the website, through your language of choice (e.g. for curl), in order to conduct stateful transactions (questions 1b, 1c and 1d) once you have logged-in as a user. In general, understanding HTTP 1.1 messages will be helpful for this assignment (e.g. for generating the correct `response.txt` file).
- Question 3 and onwards: For some requests, you will need to understand URL encodings in order to successfully deliver data to the server over the URL.
- Question 4 and 5b: If you are unfamiliar with relational databases or SQL syntax, you can refer to the W3Schools site or the Tutorialspoint. You will need to understand how the following SQL clauses work: SELECT, FROM, WHERE and INSERT. Once you are familiar with this syntax, refer to the SQL injection examples on the OWASP SQL site.
- Question 6 and 7: For step-by-step examples with sample code of non-persistent and persistent attacks, go through the short "Exploit examples" section on the OWASP XSS site. Similarly, for simple step-by-step examples of CSRF attacks for the GET and POST HTTP request methods, read the "Examples" section on the OWASP CSRF site. Note that these examples and sample code are intended to provide you with some background in order to develop your own code. To develop Javascript code for these questions, you should have a basic understanding of the following syntax:
  - In order to send HTTP requests through Javascript, you may use the XMLHTTPRequests API or the more recent Fetch API.

    You can specify URLs in the XMLHttpRequest API using either the absolute URL or the relative URL (difference between the two). Since the URL that you are using to test your exploits and the URL that we will use for marking will be different, you may not assume that a request to
    `http://ugsterXX.student.cs.uwaterloo.ca/userid/index.php`
    will always do what you expect. Therefore, you should use relative URLs, so that you do not have issues where you do not know the rest of URL.
  - The Document Object Model (DOM) is used by the browser to process HTML doc-

uments. You will need to interact with the DOM for some parts of these questions. The following DOM features may be useful: document.querySelectorAll():, document.cookie, document.location and outerHTML.

- You may find Javascript regular expressions and string manipulation methods helpful for processing URLs.

You can easily test your JS code in chrome's or for firefox's developer tools console.