CMSC414 - Project 2: Web Security

On Time Deadline: 6 March 2019, 11:59:59 PM

This project is worth 12% of the final grade.

This project will cover three broad classes of attacks that are common on the web today: Cross-Site Scripting (XSS), Cross-Site Request Forgeries (CSRF), and SQL Injection attacks.

Today's web is a complex system, consisting of multiple different kinds of protocols and technologies interacting with one another. To make the most out of this project, you will be working with several of them.

As a result, there are many things about this project that may be new to you; you will be digging into HTTP, writing some Javascript, SQL, and HTML code, and analysing some Python code, as well.

Start early.

Introducing *Nowshare*

In this project, you will be attacking an app named *nowshare*. Nowshare is a highly exclusive social media application written by members of the university's cybersecurity club for the purpose of this project. Nowshare is built on Python's Flask framework and uses a MySQL database. Flask allows developers to write web applications by defining routes and assigning those routes to functions written by the developer. The functions take the HTTP parameters as their arguments, and return the HTTP response. For example:

```
@app.route('/logout')
def logout():

# If the user is not logged in, then
# redirect user to the login
if not 'user' in session.keys():
return redirect(url_for('login'))

user = current_user()

# Clear the session token for the current user
if user != {}:
cursor = connection.cursor()
```

```
    cursor.execute("DELETE FROM sessions WHERE username = '" +
    user['username'] + "';")
    cursor.close()

    # Remove the session token from the user
    del session['user']

    return redirect(url_for('login'))
```
(from nowshare/app/app.py - line 313)

The above code assigns the endpoint `/logout` to the function `logout`. The function removes the user's session from the database and returns an instruction to redirect to the page `/login`.

Configuring Your Virtual Machine

For this project, you will use the same virtual machine that you used for P1. However, you will need to perform some additional configuration:

1. **Install docker-compose**: First, `docker-compose` needs to be installed, as that provides the environment in which the web server will run. To install this, open a shell and type:

   ```
   > sudo apt install docker-compose
   ```

2. **Install JRE and JDK**: We need these to in order to compile and run Java programs for our CSRF attack.

   ```
   > sudo apt install default-jdk
   ```

3. **Configuring /etc/hosts**: In order to access nowshare from http://now.share:5000 we will need to edit /etc/hosts. To do this, in your terminal type:

   ```
   > echo '127.0.0.1 now.share' >> temp

   > tail -n +2 /etc/hosts >> temp

   > sudo mv temp /etc/hosts
   ```

4. Next, you may find it useful to have the `HTTP Header Live` Firefox extension installed in your web browser. Visit the following webpage in your browser and install the extension:

https://addons.mozilla.org/en-US/firefox/addon/http-header-live/

5. Finally, download the project files from the class webpage at:

   https://www.cs.umd.edu/class/spring2019/cmsc414/projects.html

---

Nowshare Basics

Nowshare runs within *docker* containers.

Docker is a container management environment for running machine *images*, kind of like lightweight virtual machines. Docker images are defined in Dockerfiles and outline image properties such as 1) the container's operating system, 2) exposed ports, and 3) shareable files from the host OS to the container.

Nowshare uses docker-compose to create a network of docker containers on a single machine. We use `docker-compose.yml` to define two containers: **web** (which runs the nowshare web application) and **db** (which runs the MySQL database for nowshare).

To start the Nowshare application, visit the top-level P2 directory and use one of the following two commands:

```
> sudo ./start
```

or

```
> sudo docker-compose up -d
```

To stop the application, from the same directory use

```
> sudo ./stop
```

You can restart the service using

> sudo `./reset`

Which is just a shortcut for calling start and then stop.

Once you have started the Nowshare application, visit http://now.share:5000 in the browser to see the web service. Please note that it can take 30 seconds or longer for the application to fully start -- if you visit the site and get an error, refresh the site several times until it comes up. Also note that this site is only available locally in the VM where Docker is running -- now.share is an alias to localhost.

The Nowshare application is preconfigured with accounts for four users:

Username	**Password**

alice	al
bob	123
charlie	not
kathy	pass

This project consists of *ten* tasks, one of which is optional. The first three involve SQL injections, the next five tasks involve cross site scripting, and the last two involve cross site request forgery. The point values of each individual task for grading are to be announced.

Task 1: SQL Injection with SELECT

In the first task, you will take advantage of the vulnerable login service to log in without a password. Normally, when a user tries to log in, they fill in the username and password fields, and an SQL query is performed to check the for the existence of the username and password combination in the database. The website uses the data provided in the “username” and “password” boxes directly to construct the SQL query. If the query returns a match, it means that the username password combination exists, and Nowshare logs the user in.

The login service is implemented in `nowshare/app/app.py`. The following SQL query is constructed to authenticate user logins:

```
"SELECT * FROM users WHERE username='" +
username + "' and password = '" + password + "';"
```

(from nowshare/app/app.py - line 379)

If at least one matching record is found, the user will be logged in; otherwise the login will be unsuccessful.

Your task is to log into the app as Alice without knowing the correct password.

Note: Look carefully at the function constructing the SQL query. It passes your inputs through some basic defenses to prevent SQL injection. You will need to find a way around these.

Submission: Submit a file called “sql-login.txt”. The first two lines should be the values you entered in the login and password fields. The two lines should look exactly as follows, with your inputs in place of the ...

username=”...”
password=”...”

Be sure to include the double quotes so we can detect if there is any whitespace at the beginning or end of your submitted input. Follow this with a blank line and then a short (3 sentences or less) plain-English explanation as to why the input you provided will successfully exploit the vulnerable web service and allow you to log into Alice’s account without knowing the password.

Task 2: SQL Injection with UPDATE

In this task, we want to use SQL injection to change information on another user’s profile without being able to login as them. In Nowshare, users can update their profiles by visiting `/settings`, where they can fill out a form to update the profile information. After the user sends the update request to the server, a SQL UPDATE statement will be constructed in app.py.

```
UPDATE users SET full_name='" + fn + "', description='" + desc + "'
        WHERE username = '" + user["username"] + "';”
```

(from nowshare/app/app.py - line 342)

Your task: While we are logged in as Alice (which we are able to do because we completed task 1), we want to be able to change Charlie’s full name from “Charlie Miller” to “Charlie Chaplin”. Log on as Alice, and click on “settings”.

Note: Look carefully at the function constructing the SQL query. It passes your inputs through some basic defenses to prevent SQL injection. You will need to find a way around these.

Submission: Submit a file called “sql-update.txt”. Enter the text that you would insert into the form, while logged in as Alice, in order to change Charlie’s name. The file should consist of 5 lines which should look exactly as follows, with your inputs in place of the …

fullname=”...”
description=”...”

As before, be sure to include the double quotes so we can detect if there is any whitespace at the beginning or end of your submitted input. Follow this with a blank line and then a short (3 sentences or less) plain-English explanation as to why the input you provided will successfully exploit the vulnerable web service and allow you to change Charlie’s name without needing to log in as Charlie.

Task 3: SQL Injection with INSERT

Now that we are logged in as Alice, we would really like for Kathy to follow us, however Kathy refuses. We would like to craft an SQL injection attack in order to force Kathy to follow Alice.

The request to follow or unfollow a person is made as a GET request through the `modify_relation` route in app.py. When the request is made, it inserts a row indicating a following relation between two people, or deletes the row if it already exists.

If the request is made and the target user is not yet being followed, the server constructs the following query:

```
"INSERT INTO following_relations (followed_uname, follower_uname)
                          VALUES ('" +
follow + "', '" + user['username'] + "');"
```

(from nowshare/app/app.py - line 465)

Your task: While we are *logged in as Alice*, construct a url such that after we visit it, Kathy is following Alice.

Hint: In order to pass certain special, non-alphanumeric characters into a url for a GET request, you must use percent encoding on those characters. See:

https://en.wikipedia.org/wiki/Percent-encoding

Submission: Submit a file called “sql-insert.txt”. Enter the url that you would access, while logged in as Alice, in order to be followed by Kathy. The file should consist of 1 line which should look exactly as follows, with your inputs in place of the …

url=”...”

As before, be sure to include the double quotes so we can detect if there is any whitespace at the beginning or end of your submitted input. Follow this with a blank line and then a short (3 sentences or less) plain-English explanation as to why the input you provided will successfully exploit the vulnerable web service and allow you to have Kathy follow you without needing to log in as Kathy.

Task 4: XSS Warmup (no submission)

The objective of tasks 4-8 is to embed a Javascript program in your Nowshare profile, so that when another user views your profile, the JavaScript program will be executed. We will build up to more complicated objectives in a number of steps.

First, as a warmup, post a malicious message that will display an alert message when the profile is viewed. For this task, log in as Kathy. Go to the “settings” page and add the following text to the description field:

```
<script>alert(“my first xss”);</script>
```

Note that if you copy-paste this text (or any other text in the following tasks), you may end up with “smart quotes” (curly quote marks) rather than the straight up and down quotes that you need. You may need to adjust the quote marks manually.

Visit Kathy’s Nowshare feed (“Nowshare” tab) to trigger the alert. If you look carefully, you may notice that another user has already implemented some XSS in their profile.

Now try embedding the following script and see what happens:

```
<script>alert(document.cookie);</script>
```

Task 5: Stealing the Session Cookie

In this task, you will add to your XSS so that when someone views your profile, the script you have embedded will send you their session cookie automatically.

To achieve this, the malicious JavaScript code can send send a HTTP request to the attacker, with the cookie appended to the request. We can do this by, for example, having the malicious JavaScript try to load an image with src set to the URL of the attacker’s choice. When the browser tries to load this image, it sends an HTTP GET request to the attacker’s website, with the cookie information attached.

The attacker is running a TCP echo server that simply prints out each request it receives. The TCP server program is included in the starter files for Project 2. To run the echo server, simply cd into the `echoserver/` directory and run make. This will compile the program and create an executable file named `echoserver`. The command

```
> ./echoserver 5555
```

will run the echo server on port 5555. It will hang until it is killed via Ctrl+C and will print anything that is sent to localhost:5555.

Your task: Embed the cookie-stealing script in Kathy's profile so that anyone who reads Kathy's profile will send their cookie to the attacker.

Hint: The javascript command `document.write()` may come in handy here.

Submission: Submit a file called `task5.txt` containing the contents of the message you placed in Kathy's profile.

Grading: We will run the echo server on localhost on port 5555. We will edit the description field, acting as the user Kathy, using the message contents as specified by your task5.txt. Then, when Bob opens this message, Bob's cookie should be printed by the echo server.

Task 6: Impersonating the Victim

After stealing the victim's cookies, the attacker can do whatever the victim can do on the Nowshare web server, including posting a new message in the victim's name, deleting the victim's posts, etc. In this task, we will launch a session hijacking attack and write a program to post "Kathy is a l33t h4xor!" on Bob's profile description.

To do this, we must first find out what (legitimately) updating a profile looks like on Nowshare. Specifically, we must figure out what is sent to the server when a user updates her settings. The HTTP Header Live extension can help us with this; it can display the contents of any HTTP request message sent from the browser. From the contents, we can identify all the parameters in the request. You installed HTTP Header Live as part of the VM configuration steps above. (You can also obtain similar information from the built-in developer tools that come with Firefox.)

Once we understand what the HTTP request for updating settings looks like, we can write a program to send out the same HTTP request. The Nowshare server cannot distinguish whether the request is sent out by the user's browser or by the attacker's Java program. As long as we set all the parameters correctly, the server will accept and process the message-posting HTTP request. To simplify your task, we provide you with a sample Java program (HTTPSimpleForge_starter.java, in the starter files) that does the following:

1. Opens a connection to the web server.
2. Sets (some of) the necessary HTTP header information.
3. Sends the request to the web server.
4. Gets the response from the web server.

Your task is to update the HTTPSimpleForge java code to post “Kathy is a l33t h4x0r!” in the “description” field of Bob’s profile settings.

Submission: Please submit an updated file called HTTPSimpleForge.java.

Grading: Your java file will be compiled into bytecode and executed. Your java program should read from an input file called task6input.txt. The first line of the input file will contain Bob’s cookie (formatted as key=value, no quotation marks this time). When your Java program is executed, it should update the profile description for Bob to display the message.

Task 7: Writing an XSS Worm

In the previous task, we learned how to steal the cookies from the victim and then forge HTTP requests using the stolen cookies. In this task, we will modify our attack to forge the HTTP request directly from the victim’s browser, without requiring intervention from the attacker.

In this task, we will embed JavaScript in Kathy’s profile description. When another user (say, Bob) visit’s Kathy’s page, Bob’s profile description will automatically be updated to say “Kathy was here!”

To do this, we must embed JavaScript that can issue an HTTP POST request. Our POST request will contain essentially the same content (headers, data) as the POST request we forged in Task 6. Viewing the HTTP POST headers and body content may again be useful.

To do this, we will use an XMLHttpRequest object. We provide a skeleton for doing this below. You may need to add or edit several things in this skeleton code in various ways. In addition, here is some documentation describing the XMLHttpRequest API:
https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

You may also need to debug your JavaScript code. You may find the built-in Firefox developer tools -- namely the Console -- helpful for this, as it can show you what JavaScript runs on a page and point you to any errors that exist.

```
<script> var xhr=null; xhr = new XMLHttpRequest(); xhr.open("POST",
"http://now.share:5000/", true); Ajax.setRequestHeader("Host",
"now.share:5000"); Ajax.setRequestHeader("Connection", "keep-alive");
var content = "something"; Ajax.send(content); </script>
```

Your task: Create a script, modeled on the example above, such that if Kathy injects this script into her profile description, and then Bob views Kathy's profile, Bob's profile description will be updated to read "Kathy was here!"

Hint: Be careful when dealing with an infected profile. After all, the profile is where your malicious script is stored, but the profile is also what your malicious script overwrites. If you are not careful, you may end up removing the XSS worm from the profile.

Submission: Submit a file called task7.txt which contains the script that Kathy injects into her profile in order to modify the victim's profile.

Grading: We will inject the worm as user Kathy. When Bob logs in and views Kathy's profile, "Kathy was here!" will be displayed on Bob's profile.

Task 8: Self-Propagating Worm (Extra Credit)

To become a real worm, the malicious JavaScript program we built in Task 7 should be able to propagate itself. Namely, whenever someone views an infected profile, their profile will be updated not only to say "Kathy was here!", but also to include the malicious script itself. That way, whenever anyone views that profile, the worm will continue to spread. (This is analogous to the Samy worm we discussed in lecture.)

To achieve self-propagation, the malicious script must copy *itself* to the victim's profile. The following guidelines will help you to make this work:

1. **ID Approach**: If the entire JavaScript program (i.e., the worm) is embedded in the infected profile, the worm code can use DOM APIs to retrieve a copy of itself from the web page. An example of using DOM APIs is given below. This code gets a copy of itself, and displays it in an alert window:

   ```
   <script id="worm"> var strCode = document.getElementById("worm");
   alert(strCode.innerHTML); </script>
   ```

2. **URL Encoding**: All messages transmitted using HTTP over the Internet use URL Encoding, which converts all special characters (such as space) to special codes. In writing your self-propagating worm, you may need to encode some or all of your

malicious script. The escape function can be used to URL encode a string. An example of using the escape function is given below:

```
<script> var strSample = “Hello, world!”; var urlEncSample =
escape(strSample); alert(urlEncSample); </script>
```

3. **Concatenation:** Under the URL encoding scheme, the “+” symbol is used to denote space. In JavaScript programs, “+” is used for both arithmetic operations and string concatenation operations. To avoid this ambiguity, you may use the concat function for string concatenation. For example:

   ```
   <script> var onestring=“abc”; onestring =
   onestring.concat(“def”); </script>
   ```

   Avoid using addition if you can. If you find that you need addition for something, try subtracting a negative number instead.

Hint: When you do `getElementById.innerHTML`, the output will not contain the outer tags <scriptid = worm> and </script>. You will have to figure out how to include these as the worm propagates. You may need to look up and use escape codes for < > / and whitespace to help you. Here is a HTTP encoding reference: https://www.w3schools.com/TAGs/ref_urlencode.asp

Submission: Submit a file called task8.txt containing the script Kathy should inject into her profile description in order to start the self-propagating worm.

Grading: We will inject the worm as Kathy. When Bob view’s Kathy’s profile, “Kathy was here!” will be displayed on Bob’s profile. Then, when Charlie view’s Bob’s profile, his profile is in turn modified.

Task 9: Cross-site Request Forgery - GET

In previous tasks we developed attacks that would alternate now.share’s expected behaviour by injecting code into the website. For the next two tasks, we will create attacks that will work even if the attacker doesn’t have access to the website. All the attacker needs is a user of now.share to go to the attacker website while also being signed into now.share on another tab!

In this task, Charlie wants Alice to follow him, but Alice refuses to do so. Charlie decides to use a CSRF attack to achieve his goal. He sends Alice a URL (via direct messaging or on his wall); curious, Alice clicks on the URL. Pretending that you are Charlie, construct a web page so that as soon as Alice visits the webpage, she follows Charlie.

In this task, as in Task 3, we will work with the `modify_relation` route in app.py.

In this task, you are not allowed to write any JavaScript code to launch the CSRF attack. Instead, your job is to make the attack happen as soon as Alice visits the webpage. The attack must be triggered by Alice visiting the webpage and not by clicking anything on the webpage. (Hint: you can use the `<img>` tag, which automatically triggers an HTTP GET request).

Because the web browser automatically attaches the session cookie to the request, the trusted site cannot distinguish the malicious request from a genuine one, therefore compromising the victim user's session integrity.

Observe the request structure for following via HTTP Header Live or the built in Firefox developer tools. Once you understand the structure of the GET request, you can forge a new request for the web service. When the victim user visits your malicious web page, a request for following a user should be injected into the victim's active session.

Your task: Create a webpage (in html) such that when Alice visits the page while being signed into nowshare, she will begin following Charlie.

Submission: Submit a web page in the file "csrf-get.html".

Task 10: Cross-site Request Forgery - POST

Now Charlie wants Bob to change his description field to "badf00d 4 badd00d5". Bob thinks this is juvenile and refuses. However, Charlie is committed and will not take no for an answer. Charlie, once more, decides to use a CSRF attack. To do this, Charlie must create an attacker webpage that is able to issue HTTP POST requests.

The objective of this task is to modify the victim's profile. In particular, the attacker needs to forge a request to modify the profile information of the victim user of nowshare. Allowing users to modify their profiles is a feature of nowshare. If users want to modify their profiles, they go to the settings tab after signing in, fill out a form, and then submit the form—sending a POST request—to the server-side script app.py, which processes the request and does the profile modification. In this task, you are required to use the POST request. Namely, attackers (you) need to forge an HTTP POST request from the victim's browser, when the victim is visiting their malicious site. Attackers need to know the structure of such a request. You can observe the structure of the request, i.e the parameters of the request, by making some modifications to the profile and monitoring the request using HTTP Header Live.

Now, using the information you gathered from observing the request, you can construct a web page that injects the message “badf00d 4 badd00d5” into the victims description field. To help you write a JavaScript program that sends a HTTP POST request, we provide the sample code named csrf-post-sample.html. You can use this sample code to construct your malicious web site for the CSRF attacks.

Your task: Create a file named csrf-post.html. When the victim user Bob is logged into now.share:5000 in one browser tab, and visits the attacker website csrf-post.html in another tab, Bob's description will be changed to say “badf00d 4 badd00d5”. To test it, you will need to simply open your csrf-post.html file on the same browser Bob is signed into. For this task, when opened, we expect the csrf-post.html to not redirect to/render anything related to now.share:5000.

Submission: You are required to submit a file named csrf-post.html. You may submit multiple files for this part however, when all files are put in the same directory and csrf-post.html is opened in the browser, the attack must work.

List of all files to submit:

- `sql-login.txt`
- `sql-update.txt`
- `sql-insert.txt`
- `task5.txt`
- `HTTPSimpleForge.java`
- `task7.txt`
- `task8.txt` (optional)
- `csrf-get.html`
- `csrf-post.html`

Include all of these files in a file of ZIP format (your zip file may contain other files in addition to these). Upload this zip file to the CS submit server as project2.

Notes and Tips

- Sometimes the VM can be quite slow. You may find it less frustrating to install docker directly on your own machine (rather than in the VM) and work with the project there during your testing. We note that this approach is unsupported, so we can't help you solve any problems you have setting it up, and you should absolutely ensure that your solutions work in the VM before submitting. Grading will all take place in the VM, and only solutions that work in the VM will receive credit. Alternatively, you might want to allocate more RAM for your VM from machine>settings>system when the VM is shut down.

- If you attempt an attack that doesn't work (especially for the SQL injection tasks), you may put Nowshare into an incorrect state where it does not work properly. If this happens, use the `./reset` utility to start over. Don't be afraid to reset early and often.

- We have included a script in your starter files titled `sanity_test.py`. When you execute it, it will check to make sure all the required files for your submission are present and that they contain all the required values. This file will not test whether or not your plain english explanation is present where required, and it will not verify the correctness of your solution.