CS 5435 – Computer Security

Homework 2

Instructor: Tom Ristenpart, TAs: Paul Grubbs, Bijeeta Pal, Nirvan Tyagi *Fall 2019*

Instructions.

The deliverables in this homework are marked in blue boxes as exercises, and the end of the document includes a list of files that should be turned in. The task boxes are for your edification only, no deliverable is associated with tasks. You should submit your homework on CMS as a PDF (we encourage using the provided LaTeX template) along with any code files indicated by the lab. You may complete the lab on your own or in a team of two, which you will indicate on CMS and within the PDF.

Part 0: Getting Started

In this lab, you will be exploiting and fixing various web security vulnerabilities in the coin application from the first homework. We will be using the same course VM as before. If you need to download it again, it is available `here`. Login with account `student` and password `5435`.

We will be using a modified version of the coin application from the first homework that purposefully exposes some vulnerabilities. Retrieve the source code by cloning the repository from Git:

```
student@CS5435-VM:  git clone https://github.com/nirvantyagi/cs5435-lab2.git
Cloning into 'cs5435-lab2'...
```

Inside the directory, you will find two application directories, `app/` and `malicious_app/`. The `app/` directory will be familiar from the first homework, as that is the coin application. The new application provided, `malicious_app/`, will be used to mount attacks on `app/`. Some exercises will require having both applications running at the same time. To do this, you will need to run the following commands from the `cs5435-lab2/` directory in two separate terminals:

```
student@CS5435-VM:  python3 -m app
Bottle v0.12.17 server starting up (using WSGIRefServer())...
Listening on http://localhost:8080/
Hit Ctrl-C to quit.
```

```
student@CS5435-VM:  python3 -m malicious_app
Bottle v0.12.17 server starting up (using WSGIRefServer())...
Listening on http://localhost:8081/
Hit Ctrl-C to quit.
```

Note that even though `app/` and `malicious_app` are both running on `localhost`, they will be treated as different domains under the same-origin policy since they are running on different ports, `8080` and `8081`. This will be important for later exercises.

Some of the exercises will require you to write attacks using JavaScript, HTML, and CSS. There exist subtle differences in the way different browsers interpret this code, meaning some attacks may work differently on Chrome or Safari versus on Firefox or Edge (for example). Therefore, we will be grading your exploits on the latest version of Firefox and recommend that you develop and test your code on Firefox as well.

In debugging your attacks, you may find Firefox's site inspection tools helpful. When you are viewing a website on Firefox, you can right-click anywhere on the page and "View Page Source" to open a window with the page HTML. You can also right-click and "Inspect Element" to bring up a panel. For the purposes of this homework, there are three tabs in particular that you may find helpful: the "Inspector" tab displays the HTML and CSS and allows you to dynamically change their values to see how the site responds, the "Network" tab shows the HTTP requests made to the web server which can be clicked on to retrieve more information, and the "Console" tab allows you to test JavaScript.

Part 1: Cross-site request forgery (CSRF) attacks

In a CSRF attack, a user is tricked into making an unintended request to the vulnerable domain, either by clicking a link that directly makes the request or loading an HTML document from another domain that makes the request on behalf of the user. In this homework, you will take the second route, crafting an HTML document on `malicious_app/` that makes a forged request to pay coins from the victim's account on `app/` to the attacker's account. The browser will send the victim's cookies making it seem like a legitimate pay request was performed by the victim.

We will build up to the final attack HTML file through a series of intermediate tasks. These tasks are designed to help get the idea on how to build the final attack, but you may skip ahead to the final exercise if you prefer. If you are not able to complete the final exercise, submitting an intermediate task will be scored for partial credit.

An attacker account with username and password `attacker` and a victim account with username and password `victim` have been automatically preregistered on startup of `app/` for your convenience. Feel free to add more preregistered accounts to `app/scripts/registration.py` for your own testing purposes.

Task 1.1: Create a form on `malicious_app/views/csrf.html` to pay 10 coins to the `attacker` account.

- You can find the HTML form source for the original pay request on `app/` by logging in to `http://localhost:8080/login` and inspecting the profile page on your browser (e.g., on Firefox, right click → "View Page Source"). Alternatively, you can find the source at `app/views/profile.html`.
- Modify the form on `csrf.html` to pay 10 coins to `attacker` when the user presses the submit button, without requiring them to fill out any of the fields. You will need to modify the `action` attribute and `input` fields of the form. Hint: look at the `value` attribute of `input`.

Test by loading `http://localhost:8081/csrf` and verifying the coins are transferred when the form is submitted.

Next, we want the CSRF attack to proceed without any further action from the user other than visiting the webpage. Namely, we want to remove the requirement that the user presses the submit button for the form.

Task 1.2: Modify `csrf.html` so that it submits the form automatically when it is loaded. This will involve adding some JavaScript under a `<script>` tag. Hint: `document.forms` gives you a way to access forms in the current document, and once accessed can be submitted using the `submit()` method.

You will have noticed that submitting the form causes the page to redirect to the profile page of the user in which they can see that something is wrong and they have lost coins. Next, we would like the CSRF attack to proceed without allowing the victim to notice much difference in their normal browsing. In particular, we will modify the attack to redirect to our course webpage, `http://github.com/tomrist/cs5435-fall2019`, after completing the payment. Importantly, we will require that at no point does the location bar of the browser show the address of the vulnerable application and that the malicious form is not visible to the user.

Task 1.3: Modify `csrf.html` so that it redirects to `http://github.com/tomrist/cs5435-fall2019` after the payment is completed while showing only a blank page in the meantime.

- You may want to make use of CSS `<style>` tags which you can use to select specific styles for HTML elements. Hint: the `visibility` property may be useful for hiding elements.
- To prevent the form from redirecting to the profile page, you may consider using the `<iframe>` tag along with the `target` attribute of the form.
- Depending on how you write your code, you need to be careful of race conditions. You should not redirect to the course webpage until it is confirmed that the payment is complete. One way to do this is to use `addEventListener()` to listen for the `load` event on an `iframe`.

Exercise 1.4: Modify `malicious_app/views/csrf.html` to pay 10 coins to `attacker` when visited by a logged-in user. The attack should do the following with no input from the user:

- Redirect to the course webpage, `http://github.com/tomrist/cs5435-fall2019`.
- Never visit the vulnerable webpage, `http://localhost:8080`.
- Hide all trace of the attack during the redirect, i.e., show only a blank page.

Note: The result of Tasks 1.1-1.3 satisfy the above requirements.

Next, we will fix the coin application to be protected against these types of CSRF attacks through the use of a session anti-CSRF token. The application already tracks a unique session token per user to track logged in users and allow logged in users to access site contents. You will extend the usage of the session token to prevent CSRF attacks on the payment form.

Exercise 1.5: Modify any of `app/api/profile.py`, `app/views/profile.html`, and `app/api/pay.py` to use the user session token for anti-CSRF protection when submitting the payment form.

Short Answer Questions 1.6:

1. The countermeasure you implemented in Exercise 1.5 re-uses the session token as a CSRF token. This prevents the session token from having the "HTTPOnly" attribute.
   - Why does re-using the session token in this way prevent it from being marked HTTPOnly?
   - What are the security implications of not having HTTPOnly set on a session token?
2. In class we discussed a *login CSRF* attack, where an attacker logs a user into a service (using the attacker's credentials) without their knowledge. Does the countermeasure you implemented in Exercise 1.5 prevent login CSRF? Why or why not? If not, how would you build a token-based login CSRF countermeasure?

Part 2: Cross-site scripting (XSS) attacks

XSS attacks are some of the most prevalent across the internet and typically come in two forms: (1) reflected XSS attacks, and (2) stored XSS attacks. In a reflected XSS attack, a malicious user crafts a malicious URL that when followed by a victim tricks the vulnerable application into serving a malicious script to the victim. In a stored XSS attack, a malicious user is able to store a malicious script on the vulnerable application that will be served persistently to any victims that visit the vulnerable application.

We will start by crafting a reflected XSS attack that sends the session cookie of the victim to the attacker. Again, we will work up to the final attack through a series of intermediate tasks that can be submitted for partial credit.

Task 2.1: Craft a URL along the profile path, `http://localhost:8080/profile/<your attack here>`, such that when visited by a logged in user, it prints out their session cookie using `alert()`.

- Calling `alert()` requires you to inject some JavaScript to be run in a `<script>` tag. Where is the input to the profile path HTTP request reflected to the victim's browser? Look at `app/api/profile.py` and `app/views/profile.html`.
- How can you craft your input so that a script you include is run? Hint: look at the surrounding structure of where your input is displayed in `app/views/profile.html`. Note that the input you want to send across in the URL string will be interpreted via URL encoding. You may find this URL encoding reference and this conversion tool helpful.
- In building your script to inject, you may find `document.cookie` helpful in accessing the user cookies.

The browser may cache the results of loading your HTTP request. Explicitly clicking the reload button should force the browser to bypass its cache.

Next, you will modify your attack to communicate the victim's cookie to `malicious_app`.

Task 2.2: Modify your attack URL to send the victim's session cookie to the attacker through the `http://localhost:8081/xss_out` GET path in `malicious_app/app.py`. You can verify the cookie was received by adding a print statement to `app.py`. Hint: You might consider using the `<img>` tag to help make the GET request.

Like before, we will want to hide any trace of the attack from the victim.

Task 2.3: Modify your attack URL to hide any red error messages. The landing page of the attack URL should look as if no error had occurred. You may find the `visibility` attribute helpful again.

Exercise 2.4: Craft a URL in `xss_url.txt` such that when it is visited by a logged-in user, the user's session cookie is sent to the attacker via the `http://localhost:8081/xss_out` path. Your attack URL must:

- Begin with `http://localhost:8080/profile/`.
- Redirect to a page that looks the same as the user's profile with no errors.

Note: The result of Tasks 2.1-3 satisfy the above requirements.

Next, you will build a powerful type of stored XSS attack known as a worm. A worm, in the context of web security, is a script that is injected into pages by an attacker (XSS), and that also automatically spreads to

more pages once they are executed in a victim's browser. The Samy worm spread to over one million users on MySpace in under 20 hours. You will create a similar worm to inject into the `attacker` "About me" section for their profile, such that when a victim visits the attacker's profile, the victim pays one coin to the attacker and the worm code is replicated into (infects) the "About me" of the victim. Thus, any future visits to the victim's profile will cause one coin to be sent to the attacker (from the new victim) and the worm to spread again.

Exercise 2.5: Craft a text input in `profile_worm.txt`, such that when the text within is copy-pasted into the "About me" section of `attacker` the following occurs:

- When an infected user's profile is visited, 1 coin is paid from the viewing user's account to `attacker`.
- When a user visits an infected profile, the worm is spread to the viewing user's profile.
- The browser location should remain at the infected user's profile throughout the payment and worm replication.

One way to go about starting this exercise is in the following order: (1) Craft text input for "About me" to pay 1 coin from viewer to `attacker`, (2) Hide any trace of the attack, (3) Set up the worm to spread to the viewing user. Some further useful pointers:

- Since you are now on the same domain, you do not need to use the `iframe` trick used to submit a form from Part 1, but can make a POST request directly using `XMLHttpRequest`.
- You may need to get access to content within certain elements on the document for which `getElementById()` may be helpful.

XSS attacks can be prevented by properly "sanitizing" the user-provided inputs before serving them as part of an HTML document.

Short Answer Questions 2.6: Identify 1 other reflected XSS vulnerability and 1 other stored XSS vulnerability in `app/`. Briefly describe how you could exploit them.

Task 2.7: For the two XSS vulnerabilities you exploited and any further XSS vulnerabilities you identified, sanitize the appropriate locations in the HTML templates using Jinja's built-in tool. Confirm that your attacks no longer work.

Short Answer Questions 2.8:

1. What is the difference between reflected and stored XSS?

2. Cross-origin resource sharing (CORS) is a way to relax the same origin policy and allow scripts from other origins to access some site resources via XMLHTTPRequest. To implement CORS, the web server uses the header "Access-Control-Allow-Origin" to tell the browser which origins can access its resources. Describe what attacks can arise if a web server set this header's value to "*" (meaning any origin can access it).

Part 3: SQL injection (SQLi) attacks

A SQL injection attack consists of an insertion or modification of a SQL query used to learn some information about the contents of the SQL database. In this part, you will craft a SQL attack to learn the password of any user in the system. Note that this version does not have the hash and random salt password modifications that you made in the first homework.

You will attack the SQL query made in `app/api/pay.py` when submitting the payment form. You can test your attack on the preregistered user `admin` to try to learn their password.

Exercise 3.1: Implement the SQL injection attack in `sqli_attack` of `sqli.py`. It takes in the username of the user to attack and should return the password of that user as a string. Some ideas to get you started on crafting your attack:

- What happens when the SQL query searching for the recipient in `pay.py` does not find a match?
- Can you modify the SQL query so that it both checks if the recipient username is present and some other condition?
- Can you use the "LIKE" keyword to build a second condition to learn some information about the recipient's password? Hint: look at the wildcard characters available in the "LIKE" query.
- With many queries, can you eventually reconstruct the entire password? Hint: you may find it helpful that the payment form allows you to pay 0 coins.

Short Answer Questions 3.2:

1. Imagine a SQL injection countermeasure that used client-side Javascript to preprocess user input and remove strings that looked like SQL injection attacks before sending the input to the server. Is this an effective SQL injection countermeasure? Why or why not?

2. Compare and contrast server-side sanitization (for example, removing SQL comment characters) and prepared SQL statements as SQL injection countermeasures. Is one clearly better than the other?

Final deliverables. You will submit the following on CMS:

- PDF of solutions to Exercises 1.6, 2.6, 2.8, 3.2
- Python files: `app/api/profile.py`, `app/api/pay.py`, `sqli.py`
- HTML files: `app/views/profile.html`, `malicious_app/views/csrf.html`
- Text files: `xss_url.txt`, `profile_worm.txt`