CS 473: Network Security – HW 1

XSS, CSRF and SQL Injection Attacks

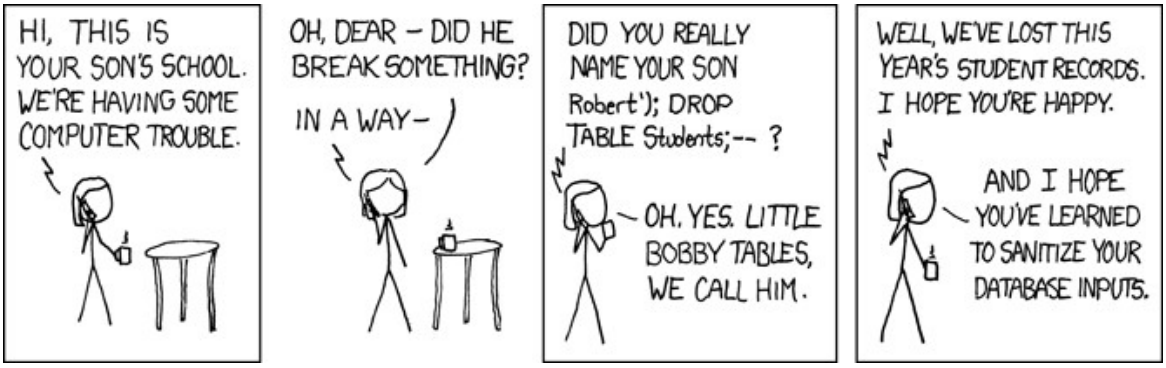
Wednesday 29th Jan 2020

All the following tasks must be completed by **Friday 7th February, 11:55 pm**. You will have to submit a **SINGLE document** on LMS, containing a screenshot of the webpage after the attack as a proof, the attack input string and a brief description about how you carried out the attack, for each task specified.

For *T. 3: Vulnerability Defense* task, additionally, you also have to submit the patched code and describe the defenses that you introduced, in the document.

T. 1: SQL Injection

SQL injection vulnerabilities allow attackers to inject arbitrary scripts into SQL queries. When a SQL query is executed it can either read or write data, so an attacker can use SQL injection to read your entire database as well as overwrite it, as described in the classic Bobby Tables XKCD comic. If you use SQL, the most important advice is to avoid building queries by string concatenation.



For the purpose of the following tasks you will have to construct simple (Select X From Y) queries, if needed, and no complex SQL queries are needed for SQL Injection. You can use https://www.w3schools.com/sql/ for refreshing your MySQL basics.

Using Group_concat() function in the select clause, - - comment out symbol and | | symbol for string concatenation will be helpful for your task. All SQL queries are concatenated as strings, so when carrying out the attack, keep an eye out for ending and starting string quotes. Semicolons are used to end SQL statements.

Feel free to observe and statically analyze the webserver code provided to you which contains the SQL queries, to think of the malicious SQL Injection input strings.

Start up the squigler web server using:

python webserver.py

Open your chrome browser, interact with the squigler website by typing http://localhost:8080/ in the URL bar. During your task attempts, if the server crashes, just go to home page or click back button to attempt again.

T. 1.1: Change a Users Password **(15)**

For this task you need to login in as **arel**, who is also an active member of squigler.com. The trick is you do not know their password, but you figure out the login page is vulnerable to SQL injection.

Your task is to, go to the login page(http://localhost:8080/login) and enter a malicious SQL string in the context of the program such that is allows you to update arel's password to your liking, and then you can login to arel's account with your own specified password.

The vulnerable SQL query in the program is:

```
"SELECT password from accounts where username='%s'" % user
```

[The above query is executed when a user enters their credentials and clicks login button. Using the above query the program extracts the user's password and then later checks if the input password matches the password in the database (line of code: 92)]

Tip: You will have to use the following SQL query to update arel's password to qwerty. it will be *part of your exploit string* into the **username** input box:

```
UPDATE accounts SET password='qwerty' WHERE username='arel';
```

How to craft the input is up to you. **You have to:**

1. Document about the malicious input string you used.
2. Paste a screenshot of you logged in as "arel".
3. Briefly explain why you observed the behaviour.

T. 1.2: Password Dump through Post **(20)**

For this task you have to carry out SQL Injection through the Squig post input box. Craft a user input such that when you press Squig it button and the page refreshes, all usernames and their corresponding passwords are dumped out as a post on the web application. Such as:

The SQL queries that are vulnerable for this task and can be exploited are:

(1) "INSERT INTO squigs VALUES ('%s', '%s', datetime('now', 'localtime'));" % (user, squig)

[When posting on squig, the above SQL query stores/inserts that post for a particular user, with the datetime of the post made, into the database. (line of code: 124)]

(2) "SELECT body, time from squigs where username='%s' order by time desc limit 10" % (user)

[After posting, when the page refreshes the above, SQL query is made to extract out all the 10 newest posts (ordered by time) and display them on the page. (line of code: 115)]

Tip: Try to enter a SQL string, such that it becomes part of (1). Instead of the your post, something else from the database gets stored in 'squigs' Table. When (2) is called to display the posts, all passwords and corresponding usernames are displayed as the most recent squig. You have to exploit (1), (2) will work naturally.

You have to:

1. Document about the malicious input string you used.
2. Paste a screenshot of the password and username SQL dump as a post.
3. Briefly explain why you observed the behaviour.

T. 2: Extended CSRF **(15)**

In the lab you generated exploit hyperlinks which resulted in account log out and malicious posts, when a victim clicked on the socially engineered link. Now, you have to carry out a similar attack but instead whenever a victim clicks on the link, they see an alert box which contains their cookie (session_id), everytime their squig post's page is refreshed. You can get the value of cookie using, **document.cookie**. Think along the lines of what you did in the lab and Reflected XSS attacks. Can CSRF and reflected XSS be merged together in someway?

You need the following to show up for any user that clicks on your malicious hyperlink, everytime their page refreshes: (your cookie value can be different)

You have to:

1. Document a screenshot of the above attack, as illustrated
2. Write your exploit input string
3. Briefly explain why you observed such a behaviour.

T. 3: Vulnerability Defenses **(20)**

Now it's time to think about the defenses. Consider that you have been hired as a freelancer to fix the squigler web application server side code for security bugs. For this task you have to patch the webserver.py code file that is provided to you. The web application is vulnerable to Cross site scripting (both reflected and DOM based), CSRF attacks and SQL Injection, as you have already seen.

Your task is to make sure when you retry all of the above vulnerabilities explained in this handout, none of those pass your security filter and the website **functions normally**.

(For XSS you only have to take care of injecting javascript using <script> tags)

Search up for secure safety practices against these three attacks. **You have to:**

1. Document briefly about each patch you introduce for each of the three vulnerabilities and how the vulnerability will no longer exist due to it. You can document about multiple patches, if needed.
2. Submit a copy of the modified safe webserver.py code.

If a broken program is submitted, no marks will be given for this task.

T. 4: XSS - Cross Site Scripting **(10 + 10)**

Complete the whole https://xss-game.appspot.com/ XSS game. There are a total of six levels and hopefully, you completed all 4 levels in the lab :D. Complete the remaining two. **After completion of each level you have to document the following:**

1. A screenshot of the 'Level completion' page.
2. A brief description of your thought process and how you carried out the attack.

T. 5: Briefly answer these questions **(20)**

In this task, you have to answer the following questions:

Q-1:
Provide an example of a simple test to determine if XSS vulnerability is present in a web application.

Q-2: Is your example above for reflected XSS or stored XSS? And which type of XSS attack can affect wider audience? Why?

Q-3: One of your colleagues has developed a mechanism to prevent XSS attack by removing any instances of word "script" in any input. The prevention is as:
1) User enters a string to input field and presses "enter"
2) Your code at server side, takes the string and runs the following code before doing anything else:

```
1  def removeEvil(text):
2      indicesToRemove = []
3      for i in range(0, len(text)):
4          if text[i:i+6] == "script":
5              for j in range(i, i+6):
6                  indicesToRemove += [j]
7      nonEvilText = ""
8      for i in range(0, len(text)):
9          if i in indicesToRemove: continue
10         nonEvilText += text[i]
11     return nonEvilText
```

Is there anything fishy you can still do? Note that you are bound to only use “script” tags for injecting any scripts so figure out if this is still possible.

Q-4: Tell whether a white-list approach or a black-list approach would be a better solution for prevention against CSRF attacks. If your answer is white-list approach, clearly explain the disadvantages of black-list approach.

Q-5 Write a short note explaining SQL Prepared Statements.

Submission Guidelines

You have to submit, a **single** document containing screenshots and brief descriptions, as specified at the end of each task and the modified version of webserver.py in part (T. 3). The document should have proper headings and each task’s answer must be labelled.

Place both the files in a zipped folder, labelled as your_roll_numer.zip and upload it on LMS.