CS 473: Network Security – Lab 1

XSS, CSRF and SQL Injection Attacks

Thursday 31st Jan 2019

Pre-Requisites

For this lab you need to have an Ubuntu Machine or Ubuntu VM, as previously instructed. Please make sure you have Sqlite3 and python installed as well. Lastly, use **Google Chrome**, for the purpose of this lab.

Basic concepts of HTML and JavaScript

This lab requires some basic understanding of HTML and JavaScript. If you already have this, great. If you do not, this section will give you the most basic concepts you need to know to complete the lab. It is not the aim of this course, or this lab to teach you either HTML or JavaScript, therefore http://w3schools.com will serve as a helpful resource and is allowed to be used for the lab. Happy Hacking.

1. **HTML**
   HTML is a markup language used to describe web pages. Broadly speaking, it consists of tags and text. Tags are indicated by angular braces, e.g. <div>. Tags can have a matching closing tag, indicated by a forward slash after the opening brace: </div>. Tags can also "close themselves", by ending with a forward slash before the closing brace: <img />. Tags can also have attributes, which have a name and a value. Different tags have different valid attribute names, each with an associated meaning for the value. Attributes are specified inside a tag following the syntax <tag attribute="value">. Single quotes are also permissible: <tag attribute='value'>. HTML is also fault tolerant. If the HTML given to a browser is not entirely correct, it will take a guess and evaluate it anyway. E.g. <p>foo<p>bar will be interpreted as <p>foo</p><p>bar</p>..

2. **JavaScript**
   The basic syntax of JavaScript required for this lab should be familiar to you. A function 'foo' is called with foo(). It can be passed arguments if desired: foo(bar), foo(bar, baz), etc. Strings are delimited by either single, or double quotes, i.e. 'foo' and "foo" define the same string. Strings can be concatenated with the + operator, e.g. "foo" + 'bar' is equal to "foobar". Separate statements can be placed on separate lines, and/or separated by a semicolon.

3. **JavaScript with HTML**
   Finally, a few details on invoking JavaScript from HTML:

   3.1: First, the **<script>** tag is used to in include JavaScript directly. Everything between the opening and closing script tags will be executed as JavaScript. For Example, <script>alert(0)</script> will call the alert box function.

   3.2: Second, many tags support attributes which can include JavaScript. These include onmouseover, onclick, and onerror. A tag which supports this is, for example, the **<img>** tag, which is used to load images (The image URL is specified through the src attribute). Finally, JavaScript can be executed by navigating to a URL beginning with **'javascript:'**. In this case, the browser will execute the remainder of the URL as JavaScript. For example, <img src="doesnotexist.jpg" onerror='javascript:alert('File does not exists')'> html element will generate an alert box for the user if the program cannot find "doesnotexist.jpg" file and throws an error.

Javascript with HTML example: https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_alert

A. *Cross site Scripting (XSS)*

For this task, you need to play through a google training game for cross site scripting (XSS) attacks, found at https://xss-game.appspot.com/. The game consists of a total of six levels, but for the purpose of this lab you need to pass only the first **four** levels. Each level has a task to exploit different vulnerabilities in toy websites. The game was designed for, and works in, Google Chrome.

For each level you have three hints in the program. If you use all your three hints, you cannot ask the TAs for any assistance for that particular level. Each level contains a **Toggle** option, which you can click to see the toy websites HTML and Javascript source code. Furthermore, you are not just restricted to use the Toggle option only, but it is highly suggested to use Chrome Developer Tools (Right click on a webpage and click on Inspect). The Console and the Elements tab will be useful for your hacking experience.

The site store cookies, so even if you close the website, you can restart from the level you left off from. Get your tasks checked for marks before leaving the lab. Skip to the next section, if you get stuck on a level, manage your time properly.

Level 1 **(2.5)**
Tip: This one is really simple. No input sanitization.

Level 2 **(7.5)**
Tip: When inserting HTML content in the DOM using innerHTML, <script> tags inside it will not load or run. But no problem, we had an another way to execute Javascript in HTML, right?

Level 3 **(10)**
Tip: Read the source code. Understand the flow. The execution is similar to level 2 :)

Level 4 **(10)**
Tip: This one is a bit tricky. Enter an apostrophe in timer user input box and press the create timer button. Open the javascript console window (right click, click on inspect and then click on console tab). You will see an error there. Inspect the line of code where the error is occuring. Now using different inputs try to see what dynamically changes at that line of code and how can you make your malicious code to execute. (Halved your work).

B. *Cross Site Request Forgery (CSRF)*

Setup and Installation
Have python 2.7 or 3.0+ and sqlite3 installed.

1. Login to your LMS and download “Lab1.tar.gz”.

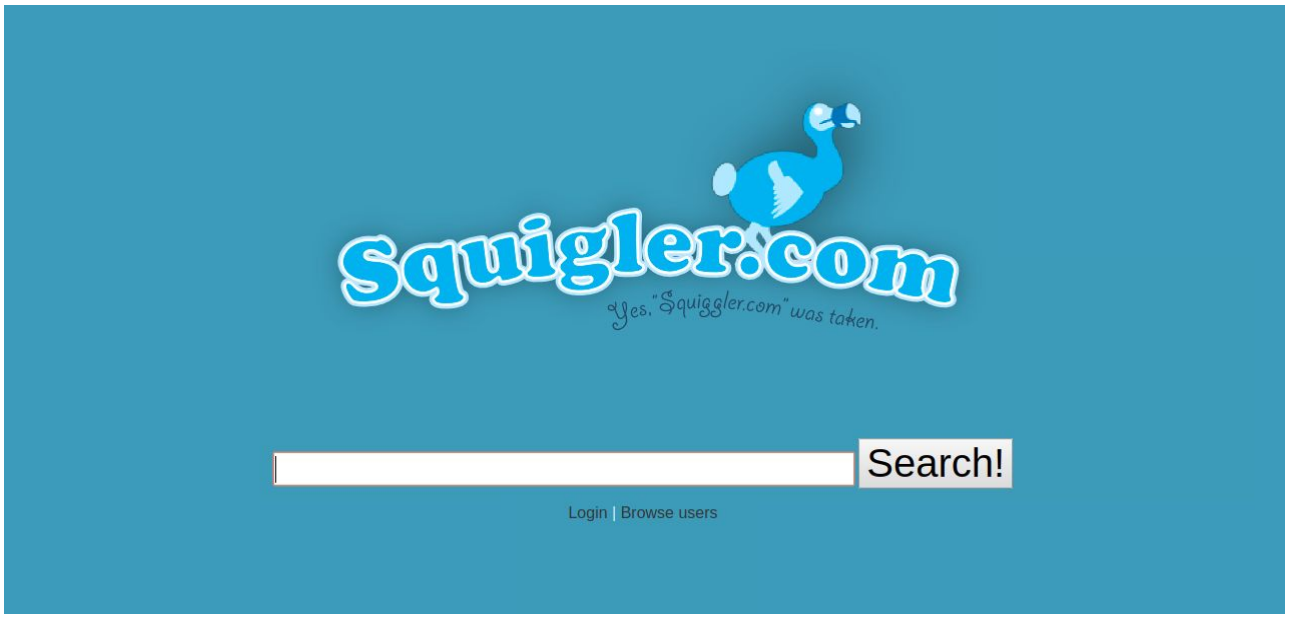
2. Extract the folder.

3. Open the terminal and run the web server:

python webserver.py

4. Open your chrome browser, interact with the squigler website by typing http://localhost:8080/ in the URL bar.

5. Squigler.com website will be open.



Login Credentials

Username	Password
dilbert	funny
bob	notcool

Using any of the above credentials you can login to your squigler account.

Note: All the following attacks needs to be performed through the squig post user input box (login and click on 'My Squigs') page, unless specified.

CSRF Attack
Squigler.com is a status posting web application, where each user can post their status (Squigs) and also delete them. All squigs are private so you can click on 'Browse Users' to see other users squigs.

The website is vulnerable to CSRF attack and you being a malicious squigler want to exploit it and cause trouble. You know that when you post a squig, the web application stores your squig in the sql database and refreshes the page, which causes all stored posts in the database to be shown on the screen and **become part of the html.**

The terminal window that you used to start the webserver, will have logs of all **url redirects and GET requests** that are made from your interaction with squigler website. Use that for your advantage as well, using it as a simplified local sniffer (packet capturer).

For the purpose of the following tasks, you will have to **create hyperlinks using anchor tags**. Use the following resource if you don't know how to: https://www.w3schools.com/html/html_links.asp

The goal here is to find a way to perform an account changing action on behalf of a logged in Squigler user without their knowledge.

Task 1: Log em out **(10)**

Post a socially engineered squig, such as:

"OMG check out these cute kittens: www.cute-kittens.com"

So that whenever a user clicks on the specified link, **he is logged out of his account instead**. Use dilbert's account to carry out the attack, and bob's account as a victim for the attack illustration, by logging in as bob, going to dilbert's profile using the "Browse Users" tab and opening the link.

Task 2: Did I post that? **(15)**

Use the same socially engineered post as before craft your malicious squiq input in such a way that **whenever a user clicks on the link, they automatically post a squig stating "I got pwned"**. Use the dilbert's account to carry out the attack, and bob's account for the attack illustration by going to dilbert's profile using the "Browse Users" tab and opening the link. Bob should have a squig posted on his timeline.

Note: After completing both of the above tasks, give the TA a live demo of the attacks to earn points. No points will be given for any task if you leave before getting it checked by the TA.

You have to ATLEAST get your first two levels of XSS and both tasks of CSRF checked in lab by the TA. Other remaining tasks of the lab can be submitted by 11:00 pm TODAY on LMS, by submitting a single document containing a screenshot of task completion as a proof and a brief description about your approach.

Take Home Tasks

All the following tasks must be completed by **Wednesday 6th February, 11:59 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)]

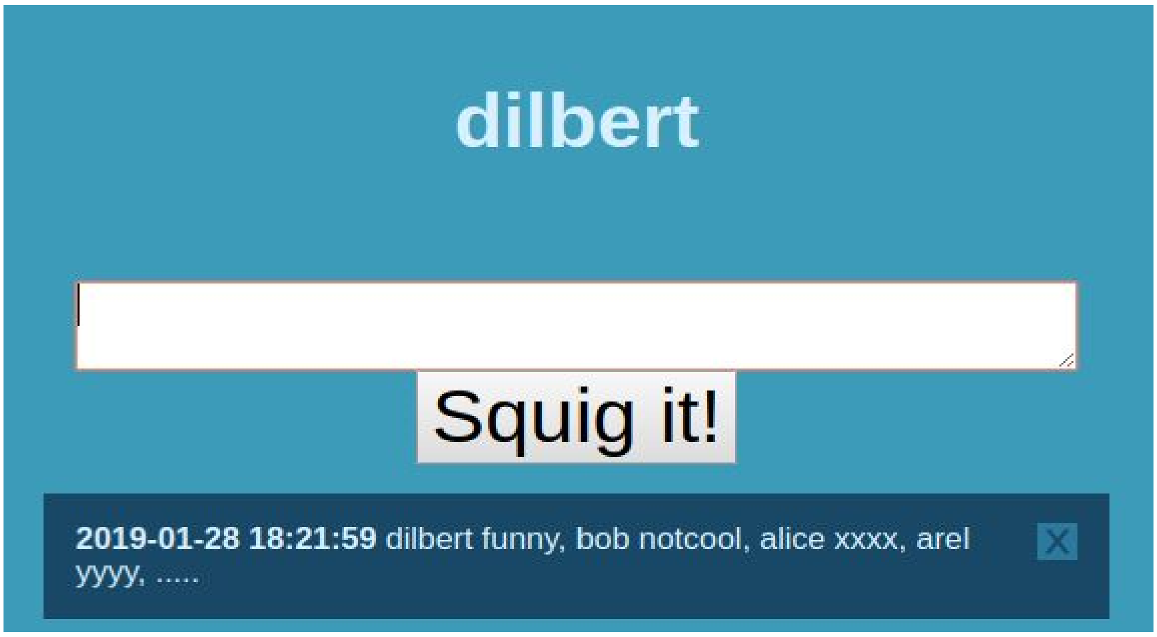
Tip: You will have to use the following SQL query to update arel's password to qwerty. it will be *part of your exploit string* into the **username** input box:

```
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:



2019-01-28 18:21:59 dilbert funny, bob notcool, alice xxxx, arel yyyy, ...

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.

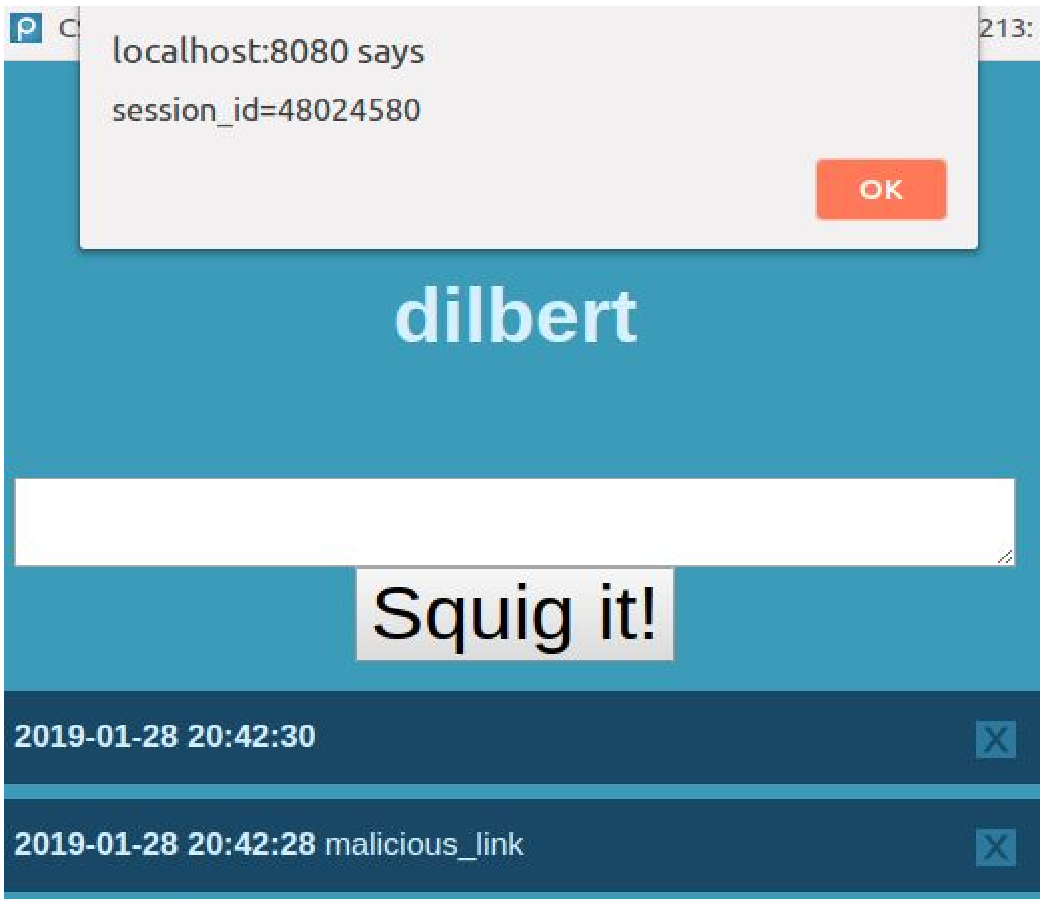
You have to:

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)



localhost:8080 says

session_id=48024580

2019-01-28 20:42:30

2019-01-28 20:42:28 malicious_link

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: (BONUS) XSS - Cross Site Scripting **(5 + 10)**

This is a bonus task, 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.

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.

Resources (Appendix)

For refreshing your Cross-Site Scripting, Cross-Site Request Forgery and SQL Injection concepts:

Cross-Site Scripting (XSS)
https://www.hacksplaining.com/exercises/xss-stored

Cross-Site Request Forgery (CSRF)
https://www.hacksplaining.com/exercises/csrf

SQL Injection
https://www.hacksplaining.com/exercises/sql-injection