

# PA3 Discussion

...

# Part 1: SQL Injection

SQL injection is the placement of malicious code in SQL statements, via web page input.

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



# No defenses

You can use any attack.


Think about how will the input from the form be translated to an SQL command to the DB.

```
SELECT * FROM USERS WHERE USERNAME = 'victim' AND PASSWORD = '!'
```

## 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`

# Simple escaping

Try a different password from the No Defences one.

The server escapes single quotes (') in the inputs by replacing them with two single quotes.

# Some more resources to help

- <https://www.youtube.com/watch?v=ciNHn38EyRc>
- [https://www.youtube.com/watch?v=\\_jKylhItPmI](https://www.youtube.com/watch?v=_jKylhItPmI)
- [https://www.w3schools.com/sql/sql\\_injection.asp](https://www.w3schools.com/sql/sql_injection.asp)

## Part 2: Cross-site Scripting (XSS)





**Attacker**

1

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



**Victim**

2

Victim clicks on link and requests legitimate website



**Website**

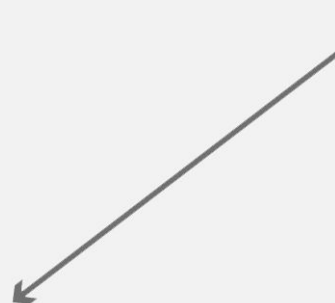
3

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

Malicious script sends victim's private data to attacker

4

```
101010111010101
10010101001010
11010101010101
101010101010
```



## Part 2: Cross-site Scripting (XSS)

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

- Steal the username and the most recent search the real user
- Send a GET request sending the username and last search :  
`http://localhost:31337/?stolen_user=username&last_search=last_search`

# XSS Sample

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

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

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

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

# Defenses

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

No defences : Any script can be run

Remove “script” : All occurrences of “script” is removed

Remove several tags : All the tags in the python script are removed

Remove some punctuation : The punctuation marks : ;\" are removed

# XSS submission

URL:

<https://bungle.sysnet.ucsd.edu/search?q=%3Cscript%3Ealert%28%27XSS%27%29%3C%2Fscript%3E>

Payload:

```
<script>
```

```
    alert('XSS')
```

```
</script>
```

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

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

What to expect:

- Log out Bungle so that you see “Not logged in.”
- Open the csrf\_0.html or csrf\_1.html
  - The page should be blank
- Go to Bungle again (or refresh), you should see “Logged in as attacker”

# Cross-site Request Forgery (CSRF)

How:

- Make a POST request to <https://bungle.sysnet.ucsd.edu/login>
- 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 (pay attention to withCredentials)
- JavaScript
- HTML <form> + JavaScript

What should the request contain:

- username, password, csrf\_token (for 3.1)
- Monitor the Network tab in Developer tools to see how Bungle send the request

# 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
- No XSS defense
  - You need to take advantage of this!
  - Think about <iframe>



# Random Token


- 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?

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 open the HTML files in browser, the page should be blank

# Writeup

You need to give some suggestions about how to defend against the attacks you did in this PA and also suggestions for this PA itself.