

# **EECS 4481 - Project Phase 2**

By: Derui Liu, Harsh Patel, Jihal Patel, and Lukas Rose

# Table of Contents

<b>1. Penetration Testing</b>	<b>1</b>
<b>1.1 Password Cracking</b>	<b>1</b>
1.1.1 Environment Setup	1
1.1.2 Testing for Password Cracking	3
<b>1.2 Cross-Site Request Forgery (CSRF)</b>	<b>5</b>
1.2.1 Environment Setup	5
1.2.2 Testing for CSRFS	7
<b>1.3 SQL Injection Attack</b>	<b>10</b>
1.3.1 Tautology Queries	10
1.3.2 PiggyBacked Queries	11
1.3.3 Illegal/Logically Incorrect Queries	11
<b>1.4 Cross-Site Scripting Attack (XSS)</b>	<b>12</b>
1.4.1 Persistent/Stored XSS	12
1.4.2 Reflected XSS	15
1.4.3 DOM-based XSS	15
<b>2. Static Code Analysis Test</b>	<b>16</b>
2.1 Cleartext submission of password	16
2.2 Password field with autocomplete enabled	17
2.3 Unencrypted communications	18
<b>3. Source Code</b>	<b>19</b>

# 1. Penetration Testing

## 1.1 Password Cracking

### 1.1.1 Environment Setup

In order to test the application if it is vulnerable to password cracking attacks, we leveraged the Kali command line program Hydra to perform brute force cracking on the admin login page of HelpDesk. The program is designed to go through a file of passwords, sourced by the user, and try every password until it finds one that is in the database and logs in the user.

The Hydra program has various options and functions we need to learn and understand in order to use it properly for our analysis.

```
(kali@kali)-[~]
$ hydra -h
Hydra v9.1 (c) 2020 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this is non-binding, these *** ignore laws and ethics anyway).

Syntax: hydra [[[-l LOGIN|-L FILE] [-p PASS|-P FILE]] | [-C FILE]] [-e nsr] [-o FILE] [-t TASKS] [-M FILE] [-T TASKS]
]] [-w TIME] [-f] [-s PORT] [-x MIN:MAX:CHARSET] [-c TIME] [-ISOuvVd46] [-m MODULE_OPT] [service://server
[:PORT][:OPT]]

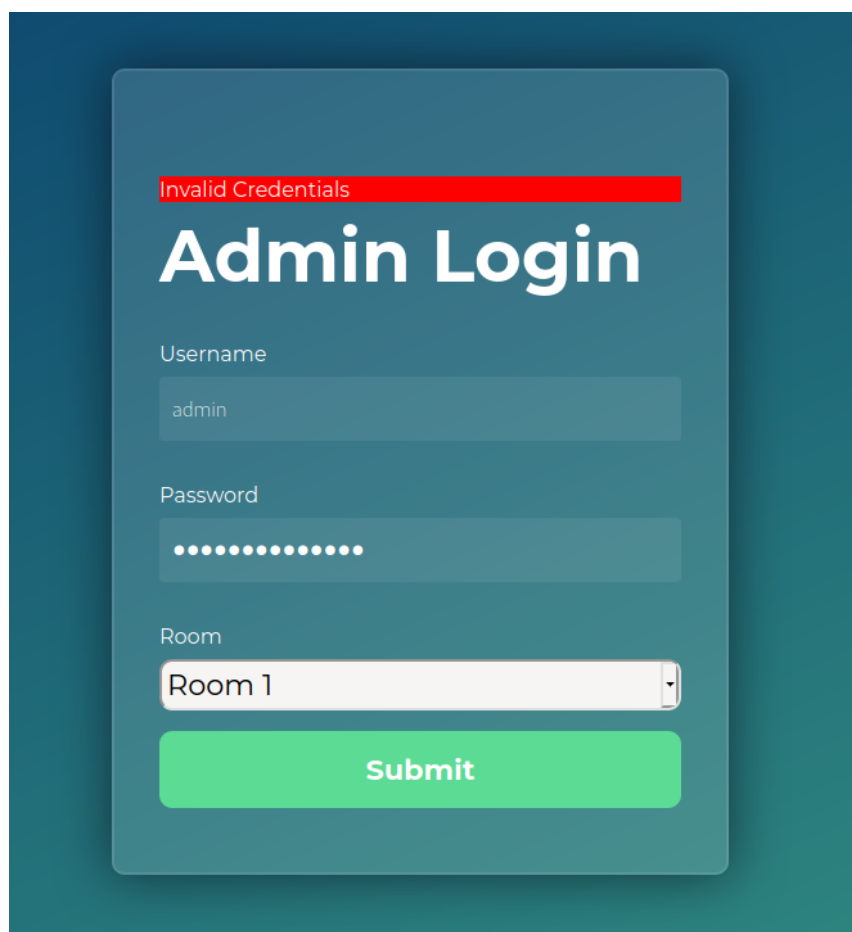
Options:
-R      restore a previous aborted/crashed session
-I      ignore an existing restore file (don't wait 10 seconds)
-S      perform an SSL connect
-s PORT if the service is on a different default port, define it here
-l LOGIN or -L FILE login with LOGIN name, or load several logins from FILE
-p PASS or -P FILE try password PASS, or load several passwords from FILE
-x MIN:MAX:CHARSET password brute-force generation, type "-x -h" to get help
-y      disable use of symbols in brute-force, see above
-r      rainy mode for password generation (-x)
-e nsr  try "n" null password, "s" login as pass and/or "r" reversed login
-u      loop around users, not passwords (effective! implied with -x)
-C FILE colon separated "login:pass" format, instead of -L/-P options
-M FILE list of servers to attack, one entry per line, ':' to specify port
-o FILE write found login/password pairs to FILE instead of stdout
-b FORMAT specify the format for the -o FILE: text(default), json, jsonv1
-f / -F exit when a login/pass pair is found (-M: -f per host, -F global)
-t TASKS run TASKS number of connects in parallel per target (default: 16)
-T TASKS run TASKS connects in parallel overall (for -M, default: 64)
-w / -W TIME wait time for a response (32) / between connects per thread (0)
-c TIME wait time per login attempt over all threads (enforces -t 1)
-4 / -6 use IPv4 (default) / IPv6 addresses (put always in [] also in -M)
-v / -V / -d verbose mode / show login+pass for each attempt / debug mode
-O      use old SSL v2 and v3
-K      do not redo failed attempts (good for -M mass scanning)
-q      do not print messages about connection errors
-U      service module usage details
-m OPT  options specific for a module, see -U output for information
-h      more command line options (COMPLETE HELP)
server  the target: DNS, IP or 192.168.0.0/24 (this OR the -M option)
service the service to crack (see below for supported protocols)
OPT     some service modules support additional input (-U for module help)

Supported services: adam6500 asterisk cisco cisco-enable cvs firebird ftp[s] http[s]-{head|get|post} http[s]-{get|p
ost}-form http-proxy http-proxy-urlenum icq imap[s] irc ldap2[s] ldap3[-{cram|digest|md5}][s] memcached mongodb mssq
l mysql nntp oracle-listener oracle-sid pcanywhere pcnfs pop3[s] postgres radmin2 rdp redis rexec rlogin rpcap rsh
rtsp s7-300 sip smb smtp[s] smtp-enum snmp socks5 ssh sshkey svn teamspeak telnet[s] vmauthd vnc xmpp

Hydra is a tool to guess/crack valid login/password pairs.
Licensed under AGPL v3.0. The newest version is always available at;
https://github.com/vanhauser-thc/thc-hydra
Please don't use in military or secret service organizations, or for illegal
purposes. (This is a wish and non-binding - most such people do not care about
```

The very initial information that Hydra needed in order to perform the password cracking was a known username that it can try to brute force crack the password with. In our case we used the username 'admin' for all testing purposes. Next we needed to supply the program a file of all passwords to let Hydra try until it found valid passwords. We were able to locate a very well known text file 'rockyou.txt' that includes a large amount of possible passwords. The backstory of how this file has become popular for password cracking is that back in 2009, a company named RockYou was hacked. This wouldn't have been too much of a problem if they hadn't stored all of their passwords unencrypted, in plain text for an attacker to see. They downloaded a list of all the passwords and made it publically available.

Various other information was indeed necessary to Hydra for a password crack. These included the IP address or Hostname which we had as 'localhost'. The HTTP method which was 'http-get-form'. The directory path to the admin login page, ours being '/adminform.html'. Our request body for username/password, which is 'username=admin&password=password'. And at last also needed was a way to identify Failed attempts, which we had a message saying "Invalid Credentials".



The image shows a web form titled "Admin Login" on a dark teal background. At the top, a red banner displays the error message "Invalid Credentials". Below the title, there are three input fields: "Username" with the value "admin", "Password" with masked characters (dots), and "Room" with a dropdown menu showing "Room 1". A green "Submit" button is at the bottom.

Invalid Credentials

# Admin Login

Username

Password

Room

Submit

All in all, the complete command window prompt for the Hydra platform to make a password cracking attack on our Admin Login page was:

```
sudo hydra -l admin -P /home/kali/Documents/rockyou.txt
localhost http-get-form
'/adminform.html:username=^USER^&password=^PASS^&room=Room+1:Invalid Credentials'
```

## 1.2.1 Testing for Password Cracking

In order to start testing a general password cracking attempt. The attack resulted in the following information and output in the Kali command window:

```
(kali@kali)-[~]
$ sudo hydra -l admin -P /home/kali/Documents/rockyou.txt localhost http-get-form '/adminform.html:username="USER"&password="PASS"&room=Room+1:Invalid Credentials'
Hydra v9.1 (c) 2020 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this is non-binding, these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2022-03-16 21:10:53
[DATA] max 1 task per 1 server, overall 1 task, 1 login try (l:1/p:1), -1 try per task
[DATA] attacking http-get-form://localhost:80/adminform.html:username="USER"&password="PASS"&room=Room+1:Invalid Credentials
[80][http-get-form] host: localhost login: admin password: 123456
```

Zoomed in:

```
(kali@kali)-[~]
$ sudo hydra -l admin -P /home/kali/Documents/rockyou.txt localhost http-get-form '/adminform.html:username="USER"&password="PASS"&room=Room+1:Invalid Credentials'
Hydra v9.1 (c) 2020 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this is non-binding, these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2022-03-16 21:10:53
[DATA] max 1 task per 1 server, overall 1 task, 1 login try (l:1/p:1), -1 try per task
[DATA] attacking http-get-form://localhost:80/adminform.html:username="USER"&password="PASS"&room=Room+1:Invalid Credentials
[80][http-get-form] host: localhost login: admin password: 123456
```

Here we can see that because the user 'admin' had a very easy password, the password cracking platform was easily able to find it.

To further understand, we decided to also attempt a brute force password cracking attack on another user account which we knew had a more secure password than '123456'. The results of the attack are as follows:

```
(kali@kali)-[~]
$ sudo hydra -l test1 -P /home/kali/Documents/rockyou.txt localhost http-get-form '/adminform.html:username="USER"&password="PASS"&room=Room+1:Invalid Credentials'
Hydra v9.1 (c) 2020 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this is non-binding, these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2022-03-16 21:21:13
[DATA] max 1 task per 1 server, overall 1 task, 1 login try (l:1/p:1), -1 try per task
[DATA] attacking http-get-form://localhost:80/adminform.html:username="USER"&password="PASS"&room=Room+1:Invalid Credentials
[80][http-get-form] host: localhost login: test1 password: test1
```

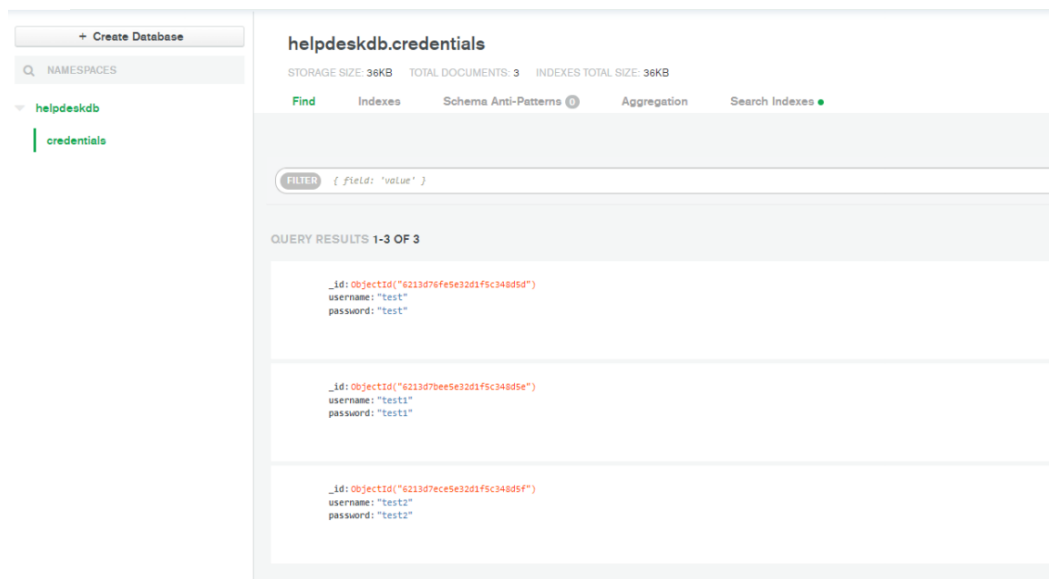
Zoomed in:

```
(kali@kali)-[~]
$ sudo hydra -l test1 -P /home/kali/Documents/rockyou.txt localhost http-get-form '/adminform.html:username="USER"&password="PASS"&room=Room+1:Invalid Credentials'
Hydra v9.1 (c) 2020 by van Hauser/THC & David Maciejak - Please do not use in military or secret service organizations, or for illegal purposes (this is non-binding, these *** ignore laws and ethics anyway).

Hydra (https://github.com/vanhauser-thc/thc-hydra) starting at 2022-03-16 21:21:13
[DATA] max 1 task per 1 server, overall 1 task, 1 login try (l:1/p:1), -1 try per task
[DATA] attacking http-get-form://localhost:80/adminform.html:username="USER"&password="PASS"&room=Room+1:Invalid Credentials
[80][http-get-form] host: localhost login: test1 password: test1
```

While on this attempt the Hydra platform did take a significantly longer time, it eventually did crack the password.

The outcomes of the password cracking attempts and further analysis points gave our group valuable information on where our current security stands regarding login authentication. It is very evident that our current course of actions are not enough to keep our application safe from attacks such as brute force password cracking. When taking a glance at the admin database it is evident that easy to crack passwords are allowed and as well stored unencrypted in the database.



Thus, our team has put in plans to update various details within our admin login page to mitigate the vulnerabilities that currently exist. These plans are all focused on the improvement of login authentication and against password cracking attacks. We find it is necessary that we implement a rule that all admin accounts need to follow certain guidelines for their passwords such as:

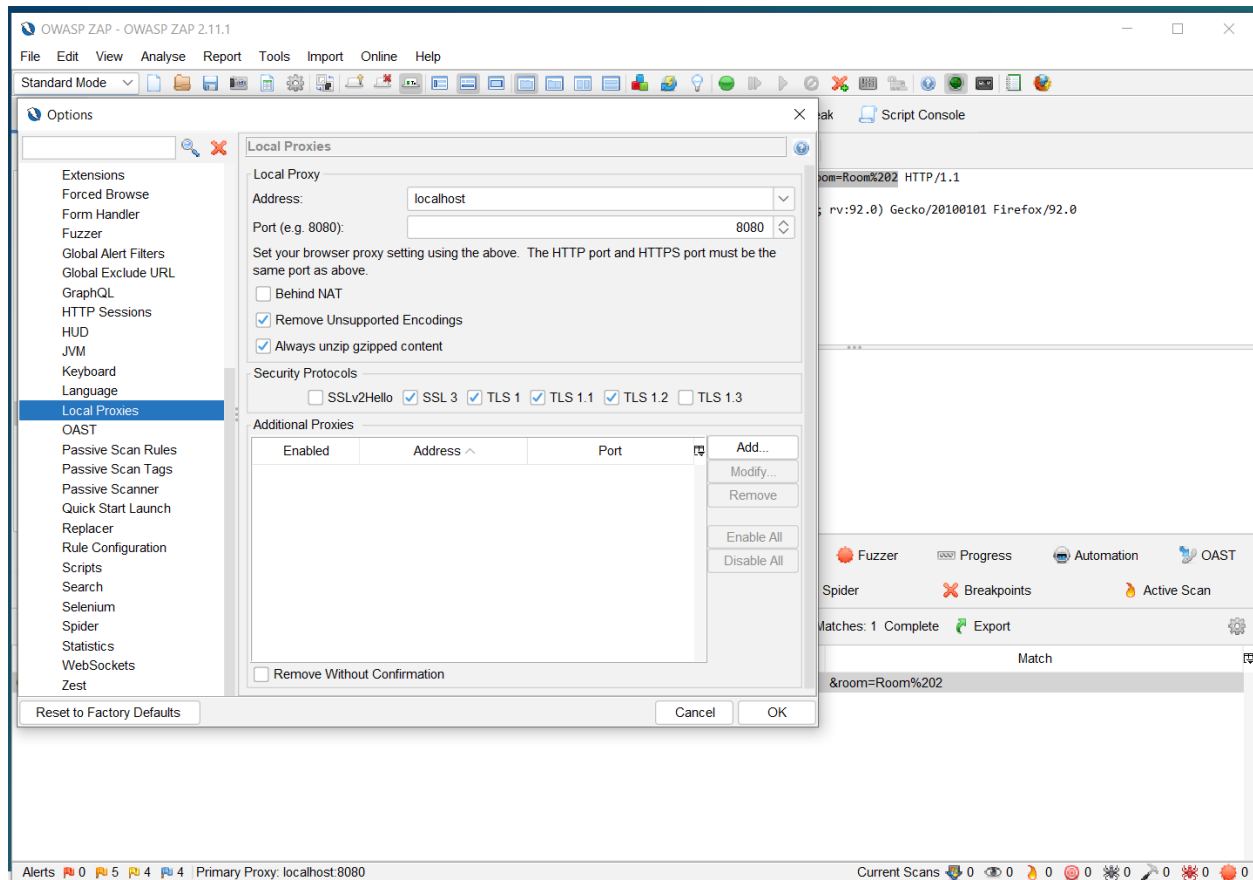
- At least 8 characters
- A mixture of both uppercase and lowercase letters
- A mixture of letters and numbers
- Inclusion of at least one special character, e.g., ! @ # ? ]

We strongly believe this will strengthen our current biggest weak point. On top of this we also plan to store passwords encrypted into the database, as our current state of storing in plaintext is very vulnerable to attacks on the database itself. At last, the use of Hydra in analyzing password cracking attacks has given us valuable data to build on and improve the HelpDesk application.

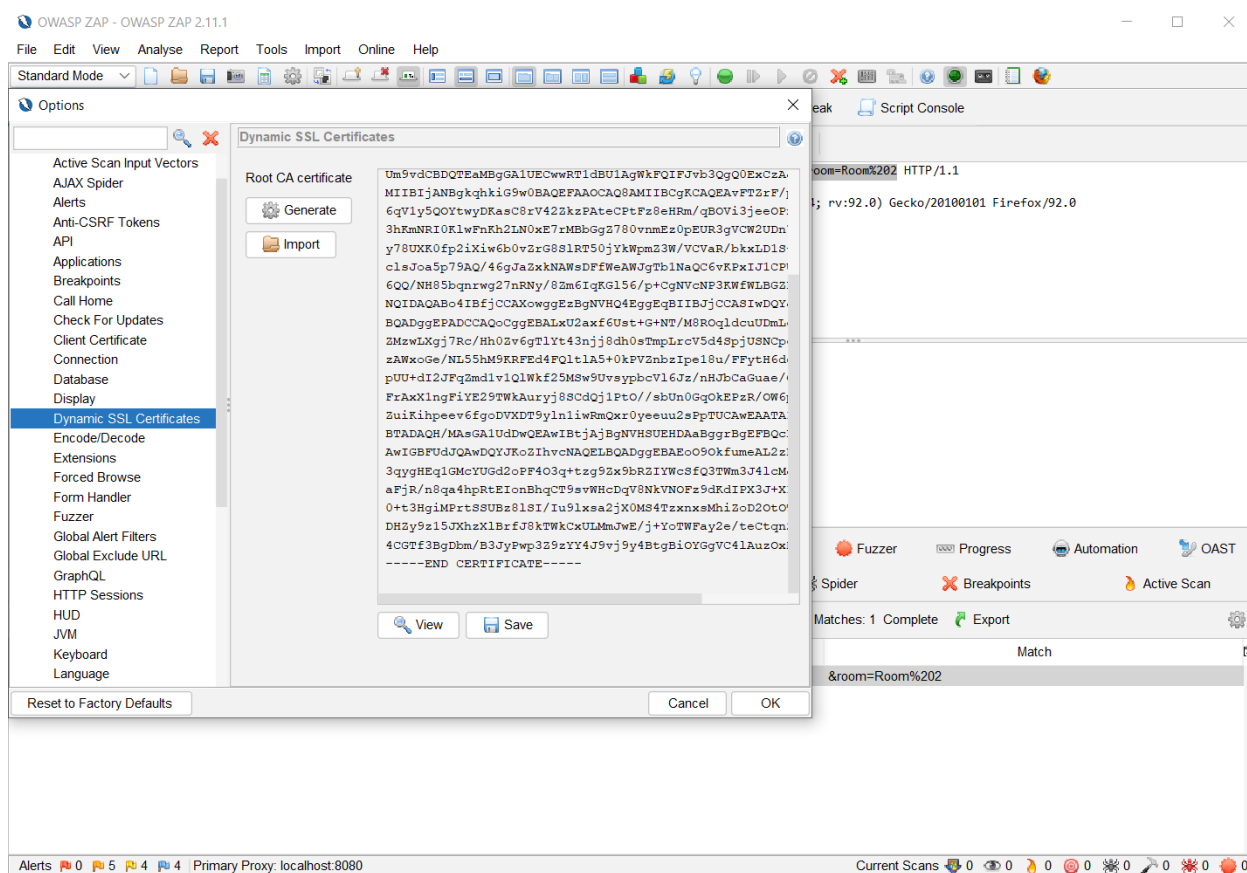
## 1.2 Cross-Site Request Forgery (CSRF)

### 1.2.1 Environment Setup

In order to test against cross-site request forgery attacks, we initialized and set up the web application on our localhost and installed OSWAP Zap. To begin testing, we first obtained the local proxy address and port for Zap as shown below.



The second course of action was to get Firefox to trust the reroutes through the proxy which was done by importing the Dynamic SSL Certificates of Zap which was found under options as shown below.



Lastly, we installed a Foxy Proxy add-on in Firefox to reroute our traffic through Zap with the following configurations:

The screenshot shows the 'Edit Proxy Zap' configuration window in Firefox. The window has a title bar with a fox icon and the text 'Edit Proxy Zap'. Below the title bar, there are several input fields and a dropdown menu. The 'Title or Description (optional)' field contains 'Zap'. The 'Color' field shows a green color swatch with the hex code '#66cc66'. The 'Proxy Type' dropdown menu is set to 'HTTP'. The 'Proxy IP address or DNS name' field contains 'localhost'. The 'Port' field contains '8080'. The 'Username (optional)' field contains 'username'. The 'Password (optional)' field is empty. At the bottom right, there are four buttons: 'Cancel', 'Save & Add Another', 'Save & Edit Patterns', and 'Save'.

After refreshing our website running on localhost:3000, we can see the information in Zap as shown below.



In order to start testing a general spider attack was run. The attack resulted in the following information and output in VS Code:

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
{ category: 'Admin Login' }
Admin Login Page
Credentials Entered --> { username: 'ZAP', password: 'ZAP', room: 'Room 2' }
Invalid Credentials: Authentication Failed
joinForm is called!!
{ category: 'Admin Login' }
Admin Login Page
Credentials Entered --> { username: 'ZAP', password: 'ZAP', room: 'Room 2' }
Invalid Credentials: Authentication Failed

```

Under the parameters caterogy in Zap, a list of parameters and their types were returned as follows:

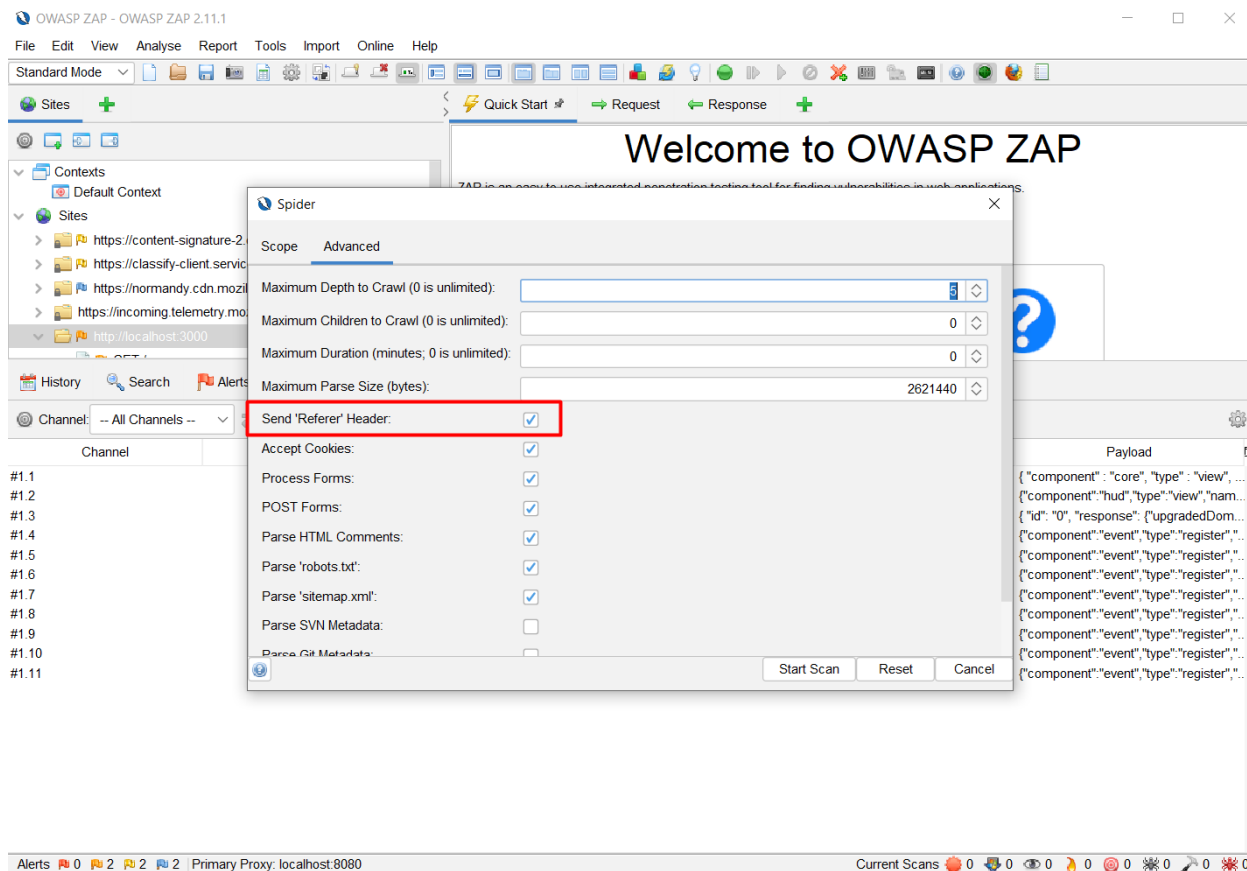
Site: localhost:3000 Export						
Type	Name	Used	# Values	% Chan...	Flags	Values
FORM	category	1	1	0		Admin Login
FORM	password	1	1	0		ZAP
FORM	room	1	1	0		Room 2
FORM	username	1	1	0		ZAP
Header	Accept-Ranges	7	1	0		bytes
Header	Cache-Control	7	1	0		public, max-age=0
Header	Connection	14	1	0		keep-alive
Header	Content-Length	14	11	78		143, 2090, 37, 246, 149, 811, 1262, 150, 161, 173, 43
Header	Content-Security-Policy	5	1	0		default-src 'none'
Header	Content-Type	14	4	28		text/html; charset=utf-8, text/html; charset=UTF-8, text/plain; charset=utf-8, text/css; charset=UTF-8
Header	Date	14	2	14		Wed, 16 Mar 2022 17:31:24 GMT, Wed, 16 Mar 2022 17:30:56 GMT
Header	ETag	7	4	57		W/"6-17f9385f59d", W/"82a-17f9385f59c", W/"4ee-17f9385f59b", W/"32b-17f9385f59e"
Header	Keep-Alive	14	1	0		timeout=5
Header	Last-Modified	7	1	0		Wed, 16 Mar 2022 16:18:24 GMT
Header	Location	3	3	100		/unauthenticated.html, /adminform.html, /css/
Header	Vary	2	1	0		Accept
Header	X-Content-Type-Options	5	1	0		nosniff
Header	X-Powered-By	14	1	0		Express

From the list of parameter types, we can see that there are some URL type requests meaning that a request by changing parameters within an URL would be possible.

Site: localhost:3000 Export						
Type	Name	Used	# Values	% Chan...	Flags	Values
FORM	category	3	2	66		Admin Login, Client Login
FORM	password	3	2	66		test, ZAP
FORM	room	3	2	66		Room 2, Room 3
FORM	username	3	2	66		test, ZAP
URL	EIO	6	1	0		4
URL	room	1	1	0		Room 3
URL	sid	5	1	0		V_BjTYHbn22gr-UAAAD
URL	t	5	5	100		N-JqbZC, N-JqbbZ, N-JqbbW, N-JqbdT, N-JqbdS
URL	transport	6	2	33		websocket, polling
URL	username	2	2	100		test, ZapTestUsername
Header	Accept-Ranges	17	1	0		bytes
Header	Cache-Control	19	1	0		public, max-age=0
Header	Connection	35	2	5		keep-alive, Upgrade
Header	Content-Length	32	22	68		2090, 37, 1262, 2230, 150, 161, 173, 174, 143, 2, 1115, 246, 3314, 149, 3256, 811, 60, 62, 1847, 97, 43, 76
Header	Content-Security-Policy	5	1	0		default-src 'none'
Header	Content-Type	34	8	23		text/html; charset=utf-8, text/html; charset=UTF-8, text/html; charset=utf-8, text/plain; charset=utf-8, text/css; charset=UTF-8, application/javascript
Header	Date	34	7	20		Wed, 16 Mar 2022 17:31:24 GMT, Wed, 16 Mar 2022 17:41:45 GMT, Wed, 16 Mar 2022 17:30:56 GMT, Wed, 16 Mar 2022 17:40:0
Header	ETag	19	10	52		W/"6-17f9385f59d", W/"737-17f9385f59b", W/"82a-17f9385f59c", W/"8b6-17f9385f59d", W/"45b-17f9385f59c", "4.4.1", W/"cf2-17f
Header	Keep-Alive	34	1	0		timeout=5
Header	Last-Modified	17	1	0		Wed, 16 Mar 2022 16:18:24 GMT
Header	Location	6	5	83		/unauthenticated.html, /clientform.html, /adminform.html, /chat.html?username=test&room=Room%203, /css/
Header	Sec-WebSocket-Accept	1	1	0		taOqmhzpBAXJJh75QWl8z3A=
Header	Upgrade	1	1	0		websocket
Header	Vary	5	1	0		Accept

There are also FORM based requests that take form-based inputs such as category, password, room, and username. These fields may be manipulated using a CSRF attack.

While these forms can be manipulated using a CSRF attack, our app employs the referrer attribute in requests. This means that our application will block the request cross-domain by checking the Referer header. However, this type of protection is weak because once the Referer header is removed the request will be successfully accepted. Removing the Referer header is pretty trivial as it can be done through the options of a spider attack through Zap as shown in the image below.



At this stage of development our application relies on HTTP and does not employ cookies. A user loses access if they leave the webpage. While this gives a certain level of protection against CSRF, it is very inconvenient and trivial to bypass. We plan on changing this to add sessions, hidden CSRF tokens and HTTPS connection.

Due to lack of implementation of cookies and sessions we can not thoroughly test for CSRF, but given the information collected we can conclude that the application tentatively has CSRF vulnerability.

## 1.3 SQL Injection Attack

In order to make our application more secure we decided to use NoSQL with MongoDB instead of traditional SQL. This was done to provide a layer of protection against SQL injections.

This methodology however opened us to NoSQL injections. While traditionally NoSQL itself doesn't have many ways to mitigate injections and is just as susceptible to injection attacks as SQL. An attacker will most often assume an application is using SQL as it is the most widely used database management system.

We further secured our database by using MongoDB to host our NoSQL database. This way we can use MongoDB's security capabilities such as input validation, to further secure our application.

### 1.3.1 Tautology Queries

We performed some NoSQL injections to bypass authentication. We attempted to bypass basic authentication using not equal (\$ne) or greater (\$gt) tautology injections.

Some tautologies that we used are listed below. We have also provided screenshots of the results.

`{ $ne: null } { $ne: 1 } { $gt: '' } { $gt: '' } [ $ne ] = 1 { $ne: 1 } { "$ne": "" } where '1=1' etc`

```
Invalid Credentials: Authentication Failed
Admin Login Page
Credentials Entered → { username: '{"$ne": ""}', password: '{"$ne": ""}', room: 'Room 1' }
Invalid Credentials: Authentication Failed
Admin Login Page
Credentials Entered → { username: 'test', password: '{"$ne": ""}', room: 'Room 1' }
Invalid Credentials: Authentication Failed
```

```
{ category: 'Admin Login' }
Admin Login Page
Credentials Entered → { username: "{$gt: ''}", password: "{$gt: ''}", room: 'Room 1' }
Invalid Credentials: Authentication Failed
Admin Login Page
Credentials Entered → { username: 'test', password: "{$gt: ''}", room: 'Room 1' }
Invalid Credentials: Authentication Failed
```

```

Admin Login Page
Credentials Entered → { username: 'test', password: '$ne: null', room: 'Room 1' }
Invalid Credentials: Authentication Failed
Admin Login Page
Credentials Entered → { username: '$ne: null', password: '$ne: null', room: 'Room 1' }
Invalid Credentials: Authentication Failed
Admin Login Page
Credentials Entered → { username: '{$ne: null}', password: '{$ne: null}', room: 'Room 1' }
Invalid Credentials: Authentication Failed
Admin Login Page
Credentials Entered → { username: 'test', password: '{$ne: null}', room: 'Room 1' }
Invalid Credentials: Authentication Failed
Admin Login Page
Credentials Entered → { username: 'test', password: '{"$ne": ""}', room: 'Room 1' }
Invalid Credentials: Authentication Failed
Admin Login Page
Credentials Entered → { username: '{"$ne": ""}', password: '{"$ne": ""}', room: 'Room 1' }
Invalid Credentials: Authentication Failed

```

```

Admin Login Page
Credentials Entered → { username: '{"$ne": ""}', password: '{"$ne": ""}', room: 'Room 1' }
Invalid Credentials: Authentication Failed
Admin Login Page
Credentials Entered → { username: 'test', password: '1=1', room: 'Room 1' }
Invalid Credentials: Authentication Failed
Admin Login Page
Credentials Entered → { username: 'test', password: '{"$ne": ""}', room: 'Room 1' }
Invalid Credentials: Authentication Failed

```

```

Admin Login Page
Credentials Entered → {
  username: "true, $where: '1 = 1'",
  password: 'test',
  room: 'Room 1'
}
Invalid Credentials: Authentication Failed

```

### 1.3.2 PiggyBacked Queries

Just like tautology, queries with malicious code were converted to string by MongoDB and were unsuccessful.

### 1.3.3 Illegal/Logically Incorrect Queries

Mongodb also does not return an error for a conversion error, so we can not apply Illegal/Logically Incorrect Queries.

## 1.4 Cross-Site Scripting Attack (XSS)

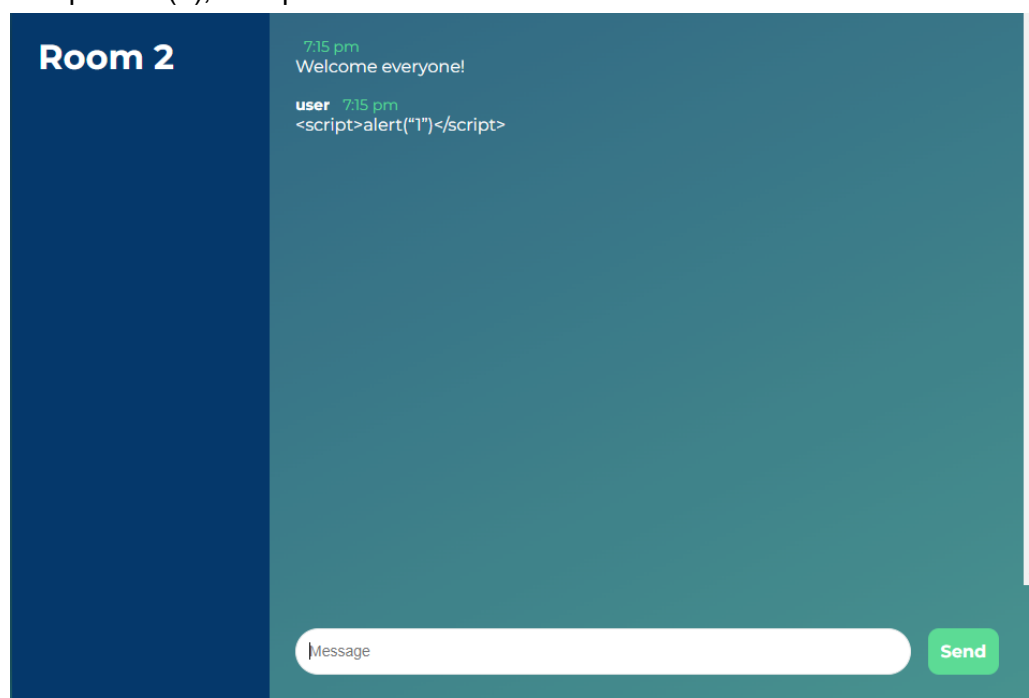
Cross site scripting attacks are common in applications that take information from an untrusted source, and includes that information in responses to users. Our application is a live-chat application which takes inputs from many users, and sends it to other users in the chatroom. This makes the application exceptionally open to XSS attacks.

There are three possible XSS attack avenues to consider: Persistent XSS, Reflected XSS and DOM-based XSS.

### 1.4.1 Persistent/Stored XSS

We performed a persistent XSS attack by injecting the following script block into the chat room, sending the script as a chat message:

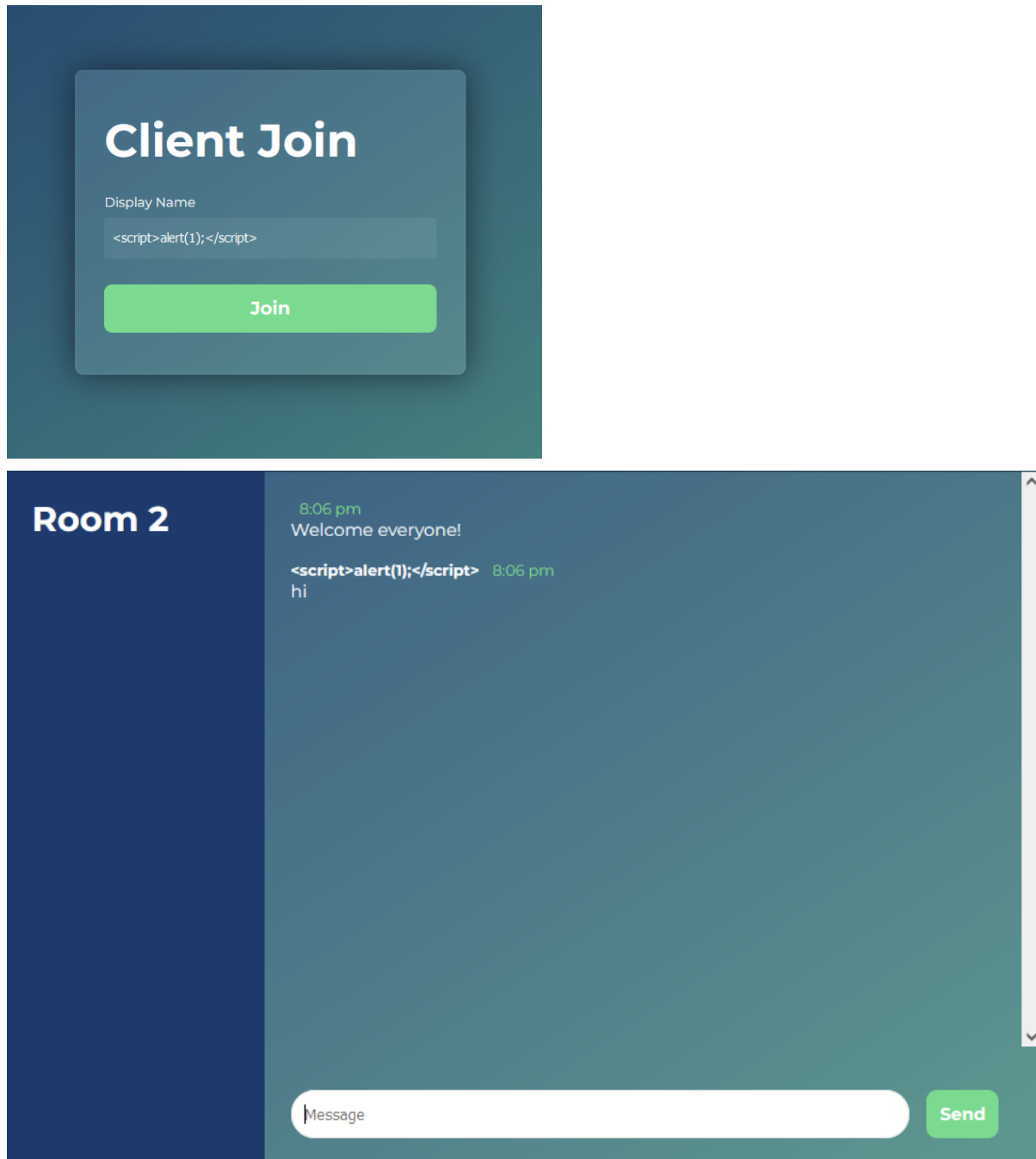
```
<script>alert(1);</script>
```



This does not work. The scripts we use to render the chat messages correctly render the injected script block as pure text, and it does not execute the script.

Another XSS attack can be performed by injecting the following script block as a user's display name:

```
<script>alert(1);</script>
```



This also does not work. The application safely renders the malicious code block into benign text, and it displays without executing.

Another, more persistent XSS attack could see an admin user with their display name in the database replaced with a script block. While the application currently does not support the creation of users, it may be possible for a disgruntled employee or sysadmin to create a malicious set of credentials.

To test this, we created a new set of credentials, with the following username:  
badmin<script>alert(1);</script>

## Insert to Collection

VIEW

1	_id : ObjectId("623282e340e8665627ba0a3e")	ObjectId
2	username : "badmin<script>alert(1);</script>///"	String
3	password : "password///"	String

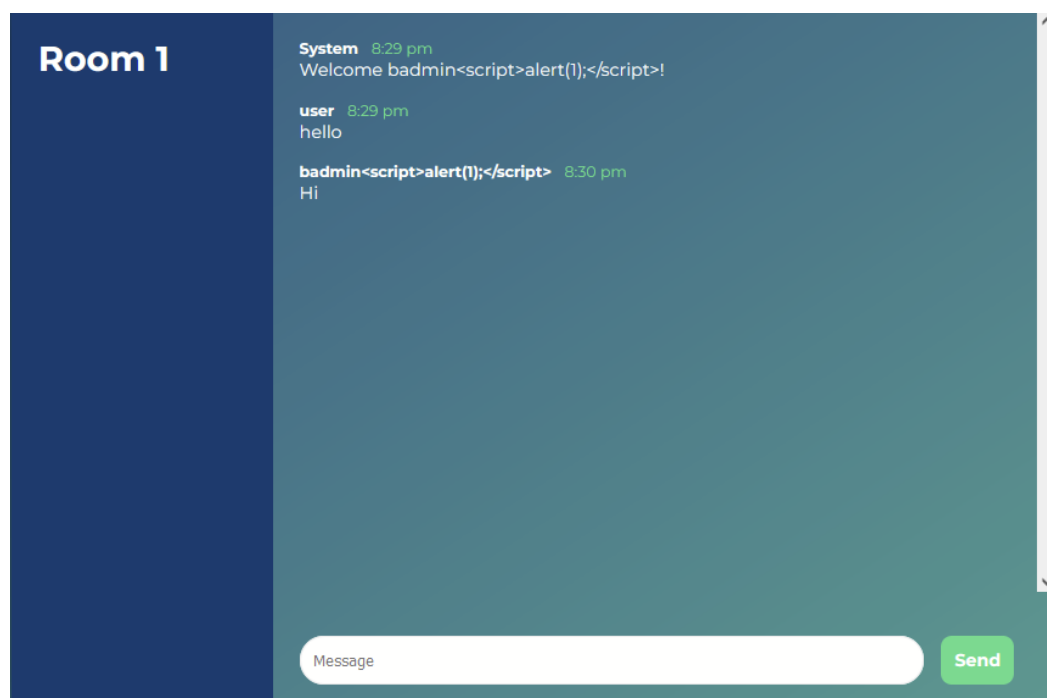
# Admin Login

Username

Password

Room



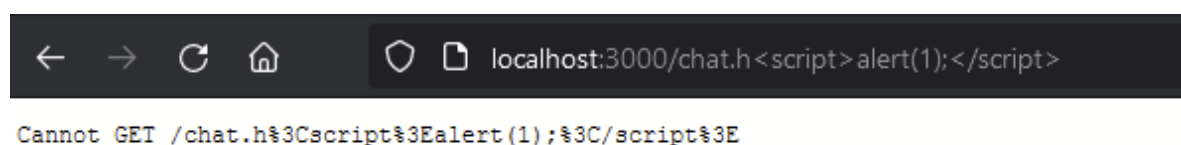


This also does not work. Just as it did with the anonymous user, the application safely renders the malicious code block into benign text, and it displays without executing.

### 1.4.2 Reflected XSS

We performed a reflected XSS attack by using this modified URL:

`http://localhost:3000/chat.h<script>alert(1);</script>`

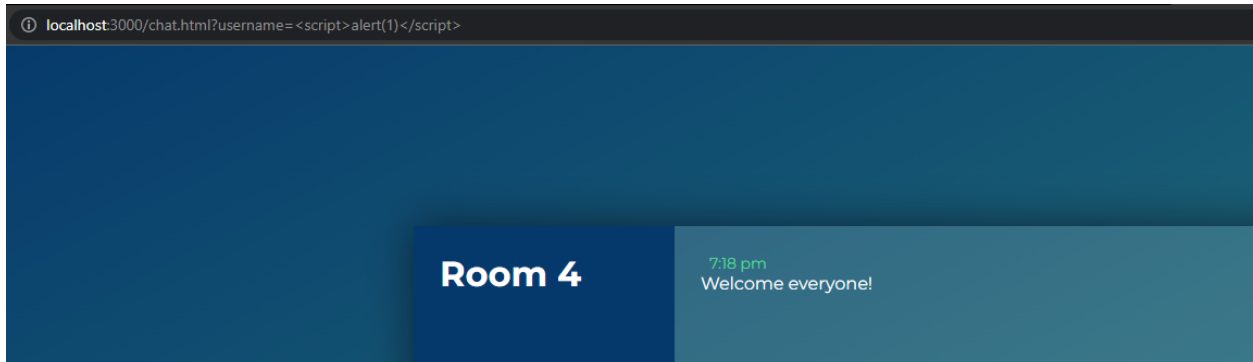


This does not work. The node.js server correctly encodes the special characters in the string, such that the information reflected in the error message is no longer a script block, and does not execute.

However, the application should not be including the encoded script in the error message at all. This is still an issue that needs to be addressed.

### 1.4.3 DOM-based XSS

We performed a DOM-based XSS by using this modified URL:  
`http://localhost:3000/chat.html?username=<script>alert(1);</script>`



This does not work. The document object model is unaffected by this url, as the Node.js server interprets this as an incoming connection by a user with the name: “<script>alert(1);</script>” joining the chat. Just like with the persistent XSS, the application safely renders the script into text, and protects the user from an XSS attack.

## 2. Static Code Analysis Test

In order to perform static code analysis we used Burp Suite Pro and let it do a crawl and audit. Over the next hour it managed to find 3 vulnerabilities.

### 2.1 Cleartext submission of password

	Severity:	High
	Confidence:	Certain
	Host:	http://localhost:3000
	Path:	/adminform.html

#### Issue detail

The page contains a form with the following action URL, which is submitted over clear-text HTTP:

- <http://localhost:3000/adminLogin>


The form contains the following password field:

- password

The Password is being sent to the database for authentication in clear/plain text. This is a big vulnerability because an attacker may eavesdrop on the client's network traffic and obtain the client's credentials. This vulnerability is very severe because the client may have reused this password for other websites, this vulnerability may give the attacker access to not only the client's credentials for HelpDesk Application but also to the credentials of other websites.

This vulnerability should be handled by first encrypting/hashing the password on the client's device before being sent to the database for authentication. This however is not enough, hackers usually have a stored database of common passwords and their respective hashes. Hence the application should use transport-level encryption (SSL or TLS) to protect all sensitive communications passing between the client and the server. The application should also use HTTPS to further protect traffic between the client and server.

## 2.2 Password field with autocomplete enabled

	Severity:	Low
	Confidence:	Certain
	Host:	http://localhost:3000
	Path:	/adminform.html

### Issue detail

The page contains a form with the following action URL:

- `http://localhost:3000/adminLogin`

The form contains the following password field with `autocomplete enabled`:

- `password`

Most browsers have a functionality to remember user credentials that are entered into HTML forms. If the auto fill function is enabled, then credentials entered by the user are stored on their local computer and retrieved by the browser on future visits to the same application.

The stored credentials can be captured by an attacker who gains control over the user's computer. Furthermore, this vulnerability can also be exploited by an attacker who finds a cross-site scripting vulnerability to retrieve a user's browser-stored credentials.

To prevent browsers from storing credentials entered into HTML forms, the attribute `autocomplete="off"` should be included within the form tag (to protect all form fields) or in input tags (to protect specific individual fields).

## 2.3 Unencrypted communications

The application HelpDeskApp allows users to connect to it over unencrypted connections (HTTP). An attacker eavesdropping on clients' network traffic could record and monitor their interactions with the application and obtain all the information the client provides.

The application should use transport-level encryption (SSL or TLS) to protect all sensitive communications passing between the client and the server. The application should also use HTTPS to further protect traffic between the client and server.

### 3. Source Code

All of our source code is provided in the GitHub repository. The GitHub repository for the help-desk chat software can be found at the following link:

<https://github.com/Harsh-B-Patel/HelpDeskApp/tree/main>. Please make sure that you utilize the **main branch** in the repository.