Spokane Cyber Cup III

**Coaches Pamphlet.**

If you are not an assigned Spokane Cyber Cup coach **DO NOT VIEW THIS document.** This contains answers for the challenges. Viewing as a student is in direct violation of the rules of the contest.

**Coaching Advice**

* The goal of this event should be for the students to learn! Even though this is pitched as a contest, feel free to help students however you see fit.
* Please help the students in however you feel necessary. In general, try to not put hands to the keyboard to solve the problem for the student. Instead, explain, write on paper or a white board.
* Do not give students full answers to the solutions; give them pieces of the puzzle if they are stuck. The students having that ‘Ah ha’ moment is really important, as this drives a thirst for knowledge.
* If the students have found an issue in the challenge, or it appears to be unobtainable, please contact Max Dulin with these problems. Additionally, if a challenge appears to be too hard let Max know; we may add extra hints to the scoreboard to make it easier.
* Walk around and help all students; do not stick around a single team for the bulk of the contest.
* Ask students if they need help; some people are too nervous and/or embarrassed to ask themselves. Be inviting and helpful.
* Please talk to students about your career! The main reason we force this to be an onsite event is to make professionals in the field easy to talk to.
* Tell kids they are doing a good job when they solve a problem. It always feels good to do something right :)
* Have fun and help lots of people! Whoop whoop!

Organization:

1-X: Web

X-X: Binary

X-X: Linux/Coding

X-X: OSINT

X-X: Crypto

X-X: MISC

X-X: Malware Analysis

Al Qaeda – 1

# The terrible Al Qaeda has been out recruiting people again! See if you can gain access to the admin panel :) You will know the flag when you see it.

# Hint: No brute force is required for this challenge. Hint: What is SQL and how can it be manipulated to do bad things?

# Hints to Give

* Explain what SQL and SQL injection are.
* Debug a few points for flag 2: UNION operator, ordering of the fields and amount of fields in the UNION query.

# Solution

SQL is a database querying language. However, if unescaped user input is passed into it, the *meaning* of the query can be altered! By affecting the query itself, we can return more information than we should be able to. The blog post has a SQL *injection* vulnerability when viewing individual blog posts at http://url.com/blog.php?id=here .

The query being used is   
*SELECT id, title, author, content   
FROM blog   
WHERE id =* ***<<INPUT >>*** *ORDER BY id DESC;*  
  
In practice, this is returning all data where the ID for the blog post matches the ID in the database. A classic SQL injection would be to make the ID ‘1 OR 1=1’, which would return all of the blog posts. This happens because ‘1=1’ evaluates to true every time.

The first flag comes from simply performing this SQL injection, with the example above: SC3{Come\_out\_all\_ye\_people\_for\_injections}. This is because there is a flag within one of the blog posts that cannot be directly referenced.

Flag: SC3{Come\_out\_all\_ye\_people\_for\_injections

Al Qaeda – 2

Once you have found the initial vulnerability, use it to dump the **users** table from the database. The admin password hash is the FLAG for the challenge. The flag is the ADMIN user password.   
Hint: What is the UNION operator?

# Hints to Give

* This is hard to get just right. With the UNION operator, *ordering* of the fields and *amount* of fields in the UNION query both matter.

# Solution

The second flag comes from dumping the *users* table. This can be done by exploiting the SQL injection with the ***UNION*** operator. This operator is used to concatenate the output of two separate queries into one. After the ID field, we can use the UNION operator to add more data to our query. Using this, we can siphon out the entire users table when displaying the blog post. A query to get the users in a table would look like “*SELECT* ***\**** *FROM* *users*;” normally.

There are a few little tricks to this though! We need to add a *comment* to the end of the query in order to keep it valid with the rest of the query potentially causing syntax errors. Another thing to consider is that the query has 4 fields. Since this is the case, the UNION operator must have four fields as well; this can be easily achieved by padding fields with a 1. The final trick is that the *ordering* does matter on the UNION. Only the second, third and fourth fields are displayed on the PHP site; this means that we need to make sure the username and password fields are in those slots.

Using the UNION operator, we can add information from the *user* table. This has two important fields for our purposes: *username* and *password*. By setting this query properly, we can get all the users in the *user* table to the output of the blog posts. Putting this altogether, we get ‘*3 UNION select 1, username, password,1 FROM users; --’* as the ID to steal all of the username information. For the flag, simply put the ‘admin’ user password hash: 5a2207692843e047f337502c63c318dd.

Flag: 5a2207692843e047f337502c63c318dd

Al Qaeda – 3

Once you have dumped the admin password hash, crack the password and login. The flag will be waiting :)

HINT: It uses MD5 for passwords.

# Solution

A hashing function is a one-way function that is used to quickly verify information with knowing the actual value of it. Since these passwords are hashes, they are not very useful. By putting the hashes into an MD5 rainbow table, such as <https://crackstation.net/>, we can recover the original password: admin – almostthere and hashing – hashing. If you login with the credentials of the admin user, there is a flag waiting for you.

Flag: SC3{SQLi\_t0\_site\_compromise!}

Can you Sign my JWT?

Can you make or modify your own token? I wonder if you can brute force this? John the ripper is great! If you succeed, access the *‘/admin’* endpoint as the admin user to get the flag.

Hint: The *signature* is generated using an HMAC. If you know the key, you can digitally sign your own tokens.

# Hints to Give

* Explain what a *brute force* attack is.
* Tell them the password is simple. The ‘RockYou.txt’ wordlist would be enough.
* Demonstrate how to sign a JWT and what digital signatures are. Point to the jwt\_tool if people have problems.

# Solution

The /admin endpoint checks for a JWT with a username of 'admin' AND validates the JWT signatures. Because the signature check is done, the only way to craft a fake token is to find the secret. Players will need to crack the secret key ('secret', should crack in about 5 seconds with rockyou.txt) to create their own JWT. What's the easiest way to do this crack!? You could write your own script or use something out of the box.

The tool <https://github.com/ticarpi/jwt_tool> works quite well to crack the secret and sign the secret. The command for using this tool looks like the following: *‘python3 jwt\_tool.py <token> -C -d rockyou.txt’* with the rockyou dataset at. <https://github.com/danielmiessler/SecLists/blob/master/Passwords/Leaked-Databases/rockyou-50.txt>.

Once we KNOW the secret, we need to create our own token. Which, again, there are a few ways to go about this. First, the website <http://jwtbuilder.jamiekurtz.com/> will build these for you as well, as long as you know the secret. <https://jwt.io> works as well, once the secret is known.

With the newly signed JWT in hand, set the value of the *session* cookie to be the newly created JWT. Now, when going to the 'admin' endpoint, you'll be greeted with the flag.

Flag: SC3{sudo\_gimme\_access}

JWT - Login

What does logging in give you access to?

# Solution

Simply login to the page. Then, go to the page ‘/private’. This is a gimme CTF challenge.

Flag: flag: SC3{Mr\_private\_page\_Im\_1n}

Korean Food

Prompt: Do you like Korean food!? Well, obviously you do, because it is delicious!

So, we have compiled a database full of delicious Korean foods for you! :)

We are hoping those damn hackers stop taking over our admin account though... We're still trying to figure out how they are doing it!

Hint: Open the \*browser developer tools\* to the \*network\* tab and view the requests.

Hint: Become the admin user! If you do this, you win :)

# Hints to Give

* Browser developer tools network tab.
* Point them to the *account creation* process.

# Solution

The goal of the challenge is to become an administrative user on the site. The request to create a user has three parameters: username, password and ***is\_admin***. By default, the *is\_admin* field is set to false. However, if you make the request with this set to ***true***, the user is NOW an admin user on the site. Once they log in, the flag will be outputted.

To make a request with the parameter set to ***true***, there are a few ways to go about it:

* Manually make a request in cURL with ***is\_admin*** to ***true*.**
* Intercept the request with a proxy (such as Burp Suite) to change the ***is\_admin*** parameter to be true.
* In the network tab of the browser dev tools, make a request to register a user. Then, right on the request to have the ***Edit and Resend*** option appear. Use this option to resend the request with ***is\_admin*** set to true.

Below is a programmatic example of how to send this via CURL in the CLI:

curl 'http://18.216.254.83:3001/register' \

-H 'Content-Type: application/json;charset=utf-8' \

--data-raw '{"username":"hackerman","password":"man","is\_admin":true}'

Flag: SC3{0h\_n0\_y0U\_bEcAm3\_AdMin!}

Lebean 1 – Simple XSS

Cross-site scriping (XSS) is an attack where JavaScript (or HTML) can be injected into the web page. Because of this, an attacker can do arbitrary action as the user, steal their session cookies and do plenty of bad things. The goal of these challenges is to bypass the filters put in place to get XSS. In general, the input from the user is being concatenated with a string in an insecure way.   
HINT: Look at the DOM with the reflected input.

HINT: What are HTML tags? What are script tags?

# Hints to Give

* Tell them to look at the reflection in the browser developer tools DOM.
* If the student has something that looks correct but is not outputting a flag, just give them the flag. The detection is very hacky and does not always work. Otherwise, let’s teach them to think like an attacker!

# Solution

The content is injected directly into the DOM. So, this is the easy ``script`` injection to add arbitrary JavaScript to the page. Simply putting ‘*<script>alert(1)</script>*’ should trigger the flag. There are many other ways to trigger XSS but it will only be found if the ‘<script>’ tag is used.

Flag: SC3{ezpz\_xss\_squeezy}

Lebean 2 – OnFun Event

Ok, I'm going to ban < and >. Get me to execute alert(1), please.   
Hint: Look at the page in the browser developer tools. Attributes!

# Solution

The user string is concatenated within an HTML attribute. By adding a double quote (") it is possible to escape the attribute to add our own attribute to get XSS. *" onfocus="alert(1)*`(or some other event) will trigger the XSS. The first double quote (“) escapes the attribute in the HTML. Once this is done, we can add arbitrary attributes in order to trigger the XSS. This solution is looking for an ‘onEvent’ handler.

Flag: SC3{xss\_on\_event\_fun\_fun}

Lebean 3 – String + Code = Problem

Ok, I'm banning even more. No more double quotes! Get me to execute alert(1), please.   
Hint: How are the strings being concatenated?

# Solution

The user string is concatenated within straight JavaScript code. By adding a single quote ('), we can escape the *string* and add our own code. *'); alert(1); //*  is a payload that will work. This works by first escaping the string with the single quote (‘) and ending the line with the semicolon. After this, any JavaScript is added (in this example, alert), with an ending ‘//’ to comment the rest of the code out. This solution is looking for ‘alert’ in the JavaScript.

Flag: SC3{xss\_sneaky\_clever\_fren}

Lebean 4 – Template is Power

No single quotes either, friend. Get me to execute ‘alert(1)’.   
Hint: What are template strings in JavaScript?

# Solution

The user string is put directly into a [template string](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals). The template string has properties that allow it to run JavaScript and do other crazy things. By using the `$`, we can run JS in the context of the template string. A simply solution to this is ***‘${alert(1)}’***

Flag: SC3{xss\_templeet\_string}

Lebean 5 – iFrames are Secure

Prompt: That's it, friend. I'll HTML-encode everything! Get me to execute alert(1), please.   
Hint: What are JavaScript URIs?

# Solution

The user string is added directly to an iFrame source attribute. By using a JavaScript URI, the code within the link will be executed. An example of this is ***javascript:alert(1)***

Flag: SC3{iframe\_shenanigans\_xss}

Lottery

Can you win the lottery? We use a secure random number generator in order to guarantee the randomness of our numbers!

# Hints to Give

* Point them to the code for *seeding* the random number generator.
* Tell them to use Python to solve this. Ensure that Python3 is being used, as Python2 will give the wrong solution.
* Ensure that the clock on the computer is correct. If they’re doing most things right but the time is off, just give them the flag.

# Solution

The program is generating random numbers then expecting the users to guess this number. The vulnerability comes from *how* the random numbers are being generated.

Once per minute (at the 0 second mark of a minute), the random number generator is being *seeded* with the starting point to use. This *seed* comes from the current time! Since the current time is predictable, we know the seed being used. Since the seed determines the random number being outputted, we can *predict* the lottery number!

This takes a little bit of Python3 code to do. First, we need to *seed* the random number generator with the proper time. Since the random number is calculated at the beginning of every minute, we need to round *down* to the nearest Unix timestamp for a given minute.

Finally, we run the same function that generates the random number to get the lottery number! The code for this is labeled below:

seed(int(time.time()) - int(time.time() % 60))

print(randint(0, ﻿100000000000)) # Contains the random lottery number!

Flag: ﻿SC3{crYpt0\_1n\_Th3\_Mr0ng}

Nailterest

Do you know that anything and everything you post on the internet stays? This is a social media platform that is up front about what happens with its data. Welcome to Nailterest!

Hint: What is server-side request forgery (SSRF)?

Hint: How is text processed on the backend?  
Hint: There is a server running on the LAN on port 1337. This is where the flag is at with flag.txt.

# Hints to Give

* Point them to how .txt and .md files are processed on the backend. This is the vulnerable code.

# Solution

Images and text files are processed in different ways when they are displayed on the website.

* Images: Processed with a GET request directly from the browser.
* Text (.txt and md): A request is made on the backend. Then, the data is sent back to the frontend, after being processed.

The ***text*** file is processed with a request from the backend. This means that the request has access to the LAN (local area network). Some services are only available on the LAN, but not the WAN. Being able to make requests on the LAN allows us to access services on the LAN from the WAN! This attack is known as server-side request forgery (SSRF).

There is a server running on <http://localhost:1337> on the LAN with an endpoint at /flag.txt. By making a request to this local server, we can get the flag; this can be done with 127.0.0.1 or localhost. By adding a link to [***http://localhost:1337/flag.txt***](http://localhost:1337/flag.txt), it will output the flag on the profile page when attempting to render a text file.

Flag: SC3{sErv3r\_31De\_ReqUest\_FForgery\_1s\_A\_BBBIIGGG\_Dea!}

POST\_DELETE

Prompt: Make a POST and a DELETE request to the server to route1 and route2 respectively.

Hint: HTTP Methods: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

Hint: CURL: <https://curl.haxx.se/docs/manpage.html>

# Hints to Give

* Explain what HTTP and the different verbs are.

# Solution

The purpose of this challenge is to force students to either use CURL or postman to make a request to a server that is not a GET request in the browser. In the land of HTTP, these different verbs represent the type of action to perform. For instance, a GET request is simply getting the data. So, there are two requests that students must make: POST and DELETE.

For simplicity, I will be using CURL to make the requests. In CURL, you can specify the HTTP verb of the request with the ‘-X’ flag. The two commands are shown below:

curl -X POST '<<domain:port>>/route1

curl -X DELETE '<<domain:port>>/route2'

By making the two requests above, the two parts of the flag will be outputted.

Flag: SC3{HTTP\_re3pon3e\_c0de3!}

What’s Inside the JWT?

What's inside the JWT?

HINT: Look up the format of the JSON Web Token (JWT).   
HINT: Learn about base64 encoding

# Solution

The JWT itself is a base64 encode collection of JSON objects. To be precise, there are three fields: header, payload and signature. These fields are separated by periods. JWTs itself should not contain any secret information when used in a real environment.

In this situation, they are stored as a cookie on the web page. Open up the 'developer tools'->'application' and cookies to see the JWT. To get the flag, base64 decode the *payload* of the JWT (data between the first and second period). This can be done with Python, cyberchef, jq and other things. The best (and nicest) way to see the data is at <https://jwt.io/>. This decodes the token and shows the fields in a very nice way.

Flag: SC3{Reading\_the\_json\_web\_token\_for\_the\_win}

Auth

Try to bypass this!

Hint: Use GDB to debug the exploit.

Hint: Use a buffer overflow to hijack the flow of execution to jump to the function ‘*do\_valid\_stuff*’.

# Hints to Give

* Show them *how* to use GDB
* Explain how the *call stack* and RET command works

# Solution

This challenge has a different flow than the previous 2 but suffers from a very similar vulnerability as before. ‘fscanf’ is called with a buffer of size 32. However, there is no sanity check for the data being read in, resulting in a buffer overflow.

To trigger a crash using this vulnerability input 44 characters. If you input 48 characters, then control of the *instruction* *pointer* has been gained! Since there is no Nx, ASLR or stack canaries on the binary, the crash is from an invalid instruction pointer and not a result of the protections.

Generally, we need to overwrite the return address of the function to jump to the function *do\_valid\_stuff*. By putting in an excessive amount of input, this will happen quite easily. The difficult part is setting the address to *do\_valid\_stuff*. The key is to remember to flip ever byte of data because the architecture is little endian. An explanation of the endianness flip can be seen in the ‘Dead’ challenge.

To find the address of ‘*do\_valid\_stuff’* open up GDB. Then, type in ‘*print do\_valid\_stuff*’. This will output the *address* in memory that the function that we want to call is at: ﻿0x80485c3. Since we know the amount of bytes prior to the instruction pointer (44), the next 4 bytes should be the address of ‘*do\_valid\_stuff’.* A full payload is shown here: ‘*python -c 'print "A" \* 44 + "\xC3\x85\x04\x08"' | ./auth’.*

Flag: ﻿SC3{Hijack\_the\_flowzzz}

Dead

I don’t want to fix my buffer! Try to overwrite my buffer with the value **0xdeadbeef.**

# Hints to Give

* Review Endianness. The program is 0x86, making it Little Endian.
* Make sure they are writing exactly 8 bytes to start with.
* Use `python -c ‘print “A” \* 0x10 + “ABCD”’ | ./dead` to write the bytes you want.
* Use ‘hexdump’ to see raw bytes being printed and GDB to debug the exploit.

# Solution

This is a basic buffer overflow challenge; this is when a memory writes outside the bounds of its allocated size. Instead of simply corrupting the value, we need to write specific data to this integer.

The stack overflow is the same as before. The buffer is only 8 bytes in size, but we can write more than this. The next 8 bytes are an *integer* that we need to make to **0xdeadbeef**. As a result, the filler, prior to the corruption, is **8** bytes in size.

Once we have our corruption, we need to write the value **0xdeadbeef** to *cool\_int*. The first problem is how do I send bytes? Using Python, as shown in the hint above, we can write the raw bytes of a character. For instance, 0x41 (ASCII code for A) can be written with *‘/x41’*. This representation can be used for all the raw bytes that we want to send. Now, our payload looks like *“\xde\xad\xbe\xef”*.

The final problem we need to solve is the *endianness*. *Endianness* refers to the ordering of memory. For instance, 0x4142 could have the most significant byte as 0x41 or 0x42, drastically changing the value of the number. In x86, strings are represented in little endian. As a result, we need to *shift* the ordering of the bytes in our payload to be reversed since there is a difference in the ordering when writing an integer vs. writing a string. For the payload, we now get *“\xef\xbe\xad\xde”*.

The final payload is `python -c 'print "A" \* 0x10 + "\xef\xbe\xad\xde"' | ./dead`.

Flag: SC3{Endianness\_is\_such\_a\_pain!}

Firsty

Can you overflow my buffer?

Hint: What happens when you give more bytes than the size of the buffer?

# Solution

This is a basic buffer overflow challenge; this is when a memory writes outside the bounds of its allocated size. Since the buffer is **8** bytes in size, anything larger than 8 would go outside the specified buffer. Provide a string with more than 8 characters to solve this challenge. For instance, the text *“AAAAAAAAAAAA”* , being 12 in size, would overflow the *cool\_int* value.

Flag: SC3{FIRST\_0verf1ow!}

Floating Bank

“Hey man... I found a vulnerability in how a bank handles money! Want to break the system with me? We can be like robinhood and give it back to the poor."

"Or, we can buy ferrari's and drive off in to the sunset".

The goal of this exercise is to break the bank and steal money from it. Enjoy :)

Hint: Floating point numbers are real hard! <https://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)FloatingPoint.html>

# Hint to Give

* What happens when you subtract from a large number? Does the total amount change?

# Solution

Floats are how fractional numbers in stored in C. An example of a float is 12.33333. Floating point numbers can only represent a finite set of values. As a result, ***rounding*** occurs in unexpected ways, especially with the higher numbers. Rounding can be a HORRIBLE issue when dealing with things that require extreme precision, such as money. For instance, numbers between 9999998976 and 10000000000 will round to the closest number. So, 999999897**7** will round to 9999998976 and 9999999999 will round to 10000000000.

The maximum amount of money that the program can store is 1e10 or 10 billion. By abusing the *withdraw* and *deposit* functionality, we can force *rounding* to occur on the *withdraw* (subtraction), to give us money without subtracting from the total amount! For instance, if we deposit $10000000000 into the account then subtract $1, the *total* amount does not change! To do this as quick as possible deposit the maximum amount (10000000000) then withdraw $500 three times. At this point, $1500 has been stolen from the bank, which is enough to output the flag.

Flag: SC3{fl0A11ing\_p0InTs\_are\_too\_hArdddddd}

Hello World

I've written my own 'echo' Hello world function! Can you find anything wrong with it? Hint: What is this error message about? `warning: format not a string literal and no format arguments [-Wformat-security]` when I compile?

# Hints to Give

* What are formats strings? For instance, what does *‘%x’* do?
* If they are using the ‘%x’ format string specifier, tell them to use a *hex to ASCII converter* if the data is confusing to them.
* If they are using a hex to ascii converter but are confused on the flag, tell them to change the *endianness* of the content.

# Solution

This is a *format string* vulnerability. The C function family of functions *printf* accepts variable length arguments. For instance, *‘%s’* is the specifier to display the variable as a string. ‘*printf(“%s”, my\_string”);*’ would print the variable *my\_string* as a string.

As a result, if the WRONG number of parameters are specified, then the user can control the format string specifiers! In this case, ‘*printf(arg\_buf);*’ is the code used for this program. Since there is one variable, but no format string provided, the program will interpret the *arg\_buf* as the format string specifier. Since this is quite powerful to control, we can do things such as write arbitrary data in memory or read out arbitrary data.

The flag for this challenge lives on the *stack*. To get the flag, we can use the format string specifier *‘%x’* to leak out the contents of the stack, including the flag. Using a few hundred ‘%x’ in the input of the program will output a bunch of hex values. Using a hex to ASCII converter, such as <https://www.rapidtables.com/convert/number/hex-to-ascii.html> , will get something that sort of looks like a flag: ‘{3CSsihtees\_i\_smsopmisss}elb’.

However, the flag looks garbled. This is because of memory on x86 is arranged in little endian format where the strings get reversed. When we are printing this via the `%x` specifier, we are printing out hex integers. Because the formatting is unexpected, it's printed out wrong. This means that the data is *reversed* from what is stored in memory in groups of 4. For instance, '{3CS' is the first part of the flag. If we reverse this, we get ‘SC3{‘, which is the prefix for most flags in the CTF. If this is done for all the bytes in the flag, we get the final flag.

Flag: SC3{this\_seems\_impossssible}

Just Keeping Rolling

Alright… here’s my final shot! I’m using a unique XOR for each byte of the secret. I bet you cannot figure it out now.

Hint: Hexdump and Ghidra are extremely helpful for this.

# Solution

The program XORs each part of the flag string with the corresponding section of the key. This can be found out by reverse engineering with a tool such as Ghidra, IDA, hexdump and many others. This key is long but has a value starting with ﻿*12245678912345678912345678912345*. The output of the XOR has a value of *‘﻿bq\x01\x4f\x7e\x05\x52\x48\x6b\x01\x7e\x7f\x7d\x5b\x64\x78\x54\x55\x78\x7c\x4e*’ with this being mostly hex escaped characters.

To solve for this, we need to XOR the ASCII value of each of the numbers (1 is 0x31 in ASCII) with the hex value of the output. For instance, the ASCII value of ‘b’ is 0x62 and the ASCII value of ‘1’ is 0x31. If we XOR these together, we get 0x53. The ASCII character for 0x53 is ‘S’, which is the beginning of our flag. Do this for every character in the *output* string until the final flag is recovered.

Flag: ﻿SC3{K3epR0LLInROllIN}

Patch

You have been given a file with protections on it... Can you ALTER the binary in order to get it to execute the other part of the if statement? In order to execute the binary, send it to <IP> with the client.

Hint: This is known as *patching*.

Hint: <https://www.youtube.com/watch?v=LyNyf3UM9Yc>

# Hints to Give

* Go through the disassembled code with them.
* Show them how to use vim, xxd and hexdump.
* Talk through ways to modify the code to do different things.

# Solution

The goal of the challenge is to ***patch*** the binary in order for a different path to be taken. Patching refers to making subtle changes in the binary itself. Once a small change (no one than 10 bytes) has been made, the file can be sent to the server and executed.

The code itself is a simply if statement to print or not print the flag.

int i = 1;

if(i == 0){

// Flag

puts("Congratzzzzz");

system("cat flag");

}else{

puts("So sad :(");

}

At a high level, there are several changes that can be made to this. However, the disassembly makes more sense to look at. This can be found via a disassembler (such as Ghidra) or *hexdump*.

400566: 55 push %rbp

400567: 48 89 e5 mov %rsp,%rbp

40056a: 48 83 ec 10 sub $0x10,%rsp

40056e: c7 45 fc 01 00 00 00 movl $0x1,-0x4(%rbp) <-- Put 0x1 into variable

400575: 83 7d fc 00 cmpl $0x0,-0x4(%rbp) <-- Compare the variable with 0x0

400579: 75 16 jne 400591 <main+0x2b> <-- Jump to Failure case if NOT equal

40057b: bf 34 06 40 00 mov $0x400634,%edi

400580: e8 ab fe ff ff callq 400430 <puts@plt>

400585: bf 41 06 40 00 mov $0x400641,%edi

40058a: e8 b1 fe ff ff callq 400440 <system@plt> <-- Print the flag

40058f: eb 0a jmp 40059b <main+0x35>

400591: bf 4a 06 40 00 mov $0x40064a,%edi

400596: e8 95 fe ff ff callq 400430 <puts@plt>

40059b: b8 00 00 00 00 mov $0x0,%eax

4005a0: c9 leaveq

4005a1: c3 retq

From looking at the disassembly above, there are several ways to go about this. But, a few will be mentioned.

* Set ‘i’ to be 0. In the machine code, this is done by altering the ‘mov’ instruction at 40056e.
* Change the JMP instruction at 400579to be something else. For instance, changing the JNE to a JEQ would only jump if the comparison was true. Since this is NOT the case, the flag would be printed.
* Change the comparison at 400575 to be done with 0x1 instead of 0x0.

All three are valid options. For example, we will go with the third option. The hex bytes (machine code) that represent this instruction are ’*83 7D FC 00’*. The ‘00’ represents the ‘0’ in the CMP instruction. If we set the ‘00’ to be ‘01’, then the comparison would pass since 0x1 == 0x1. This is a good change to do!

To physically alter the binary, there are many ways to go about this including ghidra, IDA and vim. Since most Linux machines have Vim, we will use this text editor.

1. Open vim into ‘xxd’ mode with the binary; this will show the hex values of the binary instead of ASCII text. This can be done via the command ‘:%!xxd’ while in Vim.
2. Once in ‘xxd’ mode, search for the bytes around the CMP instruction bytes. ‘/83’ should find the CMP instruction. Then, go through all of the bytes until ‘7D FC’ is found.
3. After ’83 7D FC’, is the 00 byte, which is the value being loaded, can be seen. Alter this to be ‘01’.
4. Write the data back to the binary after we have reversed the xxd mode. This can be done by ‘:%!xxd -r’ to reverse the xxd operation and ‘:wq’ to save the binary.

At this point, send the file to the server, using the client, and the flag should be outputted!

Flag: SC3{dr1nk\_a11\_the\_B00z\_hack\_a11\_tHe\_things}

String For the Moment

Remember those times of having source code? Well, instead of that, you can read assembly or machine code! Good luck finding the flag in here!

# Solution

The flag is in the binary itself. Although you could disassemble this, simply using the *strings* utility will work. The command *‘strings 1 | grep SC3’* will output the flag.

﻿Flag: SC3{Strings4EV4}

XORs Fun Properties

Okay… the flag is encrypted with XOR. Can you get it?   
Hint: Hexdump and Ghidra are extremely helpful for this.

# Solution

﻿The program just a XOR of the key with 6, so that the characters in the answer are still printable (and also it flips the {} brackets, which I hope is a little hint. To find this, some disassembly needs to be done with something lik ghidra or hexdump:

* The value being XORed is ﻿"UE5}DoRK2RNKgaoE'{“. This can be found via the *strings* command as well.
* The disassembly will show that each value in the string is being XORed with 0x6.

By doing this operation in the opposite direction, we can get back our original flag. For instance, ASCII ‘U’ has a numerical value of 85. 85 ^ 6 is 83, which is numerical ASCII value of S. S is the first part of the flag. Doing this for each of the characters in the string will output the flag.

Flag: ﻿SC3{BiTM4THMagiC!}

Auth Handler

Trying to login? Just use the *uid* (user id) of the user! Sorry for the bad error handling... error handling is hard.

Hint: To win the challenge, become a user at level 10.

# Solution

﻿The goal is to become a user with level 10. If this is attempted directly, then it will not allow it. Additionally, you cannot simply login with the built-in administrative user.

When attempting to login, there is *logic* for handling IDs that are set to the admin user or negative. However, this is within a *try-catch* block. If an error occurs in the processing, then the *except* block of code is hit without exiting properly. In this case, if a non-number, such as ‘A’, is attempted to be converted into an integer, then the *except* block will be hit. After printing an error message, a user ID is returned with the default value. Since the *default* value is **0**(same as the admin user), this returns the ID of the ***admin*** user.

Once you have logged in as the admin user, attempt to use the ‘Super Admin’ option (number 5) and the flag will be outputted.

Flag: ﻿SC3{errors\_are\_hArrrrrrrrd}

Deeper

Login using SSH to HOST. This can be done with ‘*ssh first\_ssh@<ip> -p 2226’*. Go deeper into the directory than you have ever gone before!

# Solution

Simply SSH into the machine. Then move a few directories via the *‘cd’* command. The flag can be printed via the command *‘cat flag’.*

Flag: ﻿SC3{A\_l1TT13r\_b1t\_HA7d3R\_N0w!}

Even

Odd characters are overrated :) Who needs the '/', a,c,e,i,o,u and 'y' for these problems!? The flag is at ~/flag.txt.

Hint: Where are all the places that binary commands are stored at?  
Hint: The flag begins with ‘SC3{‘

# Hints to Give

* What are wildcard characters? ‘\*’
* If they are lost in the output, tell them to use *‘grep’* with ‘SC3’ to sort through all of the output of the program.
* Read the manual for different programs on the box!

# Solution

The application restricts single wildcard (‘?’), directory slash (‘/’), lots of vowels (a,c,e,i…) and dashes (‘-‘). What is interesting though, is that the wildcard character (‘\*’) can still be used. Since this substitute in possible characters afterwards, this is the key to the challenge.

Besides the wildcard character, we need to have a command to pass parameters to. Since we cannot use many characters, we are stuck with short commands with our allowed characters. On the box, *‘hd’*, *‘pr’* and *‘ptx’* can be used to display the flag.

For instance, ‘*hd \*’* will output all the files in the current directory into a hexdump. Within the hexdump, the string representation is used as well. From this, the flag can be seen in plaintext.

Flag: SC3{even\_Is\_not\_odd!gotta\_love\_the\_\*}

First SSH

Login using SSH to HOST.

# Solution

Simply SSH into the machine. flag can be printed via the command *‘cat flag’.*

Flag: SC3{FIRST\_SSH!}

Linux Hardening

Servers, servers, everywhere... but are they secure?

HINT: What is a REALLY bad password for the *root* user on FTP?  
HINT: What services are running? FTP on port 2121 and HTTP on port 8090.

# Solution

FTP server uses UNIX accounts for authentication. Root credentials can be found in Dockerfile. The root credentials are very weak and can easily be guessed. The username-password combo is *root:root*. This should be guessable! The command to run to access FTP is *‘ftp -p HOST 2121’*.

Inside of the FTP section, there is a folder called *notes.* The folder in FTP directory contains clues to HTTP server credentials from previous years. These will help find the password for the web server:

* note.txt is signed *jeff*. This is the HTTP server username.
* pwds.txt contains a list of previously used passwords. The same base password is used each year but only changing the year on the password. One can infer that the current password may follow the pattern and thus be `i\_like\_c0ff33\_2022` for this year.

Logging in at the HOST:8090 should produce the flag.

Flag: SC3{Y0u\_founD\_m3\_m@t3!}

Processes

Login via SSH to the third challenge. To find the flag, check out the running processes :)

Hints to Give

* If they are using the proper command, ensure that terminal screen is large enough for the full command with the full to show.

# Solution

At the beginning of the Dockerfile, a script is ran that puts a `secret` into a CLI parameter. In order to see this, we can use the ``ps`` command in order to see all of the running processes on the box. To see the *full* command, use the ‘-v’ or ‘-f’ flag. Additionally, use ‘-a’ to see all running processes. Putting this altogether, run ‘*ps -av’* to see all running process information. It should be noted that the output will be truncated if the terminal is not large enough.

Flag: ﻿SC3{psssssssssss333st}

Signals

Prompt: Moxie Marlinspike has called you up to learn about SIGNALS! See if you can continue the execution of the program in order to earn the flag! :)

Hint: <https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/19/lec.html>, <http://man7.org/linux/man-pages/man7/signal.7.html>, <https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html>

Hint: ***kill*** is how to send signals; it does not necessarily terminate the process.

# Solution

The purpose of this challenge is to get students to learn about signals and syscalls. The program works as following:

* + First, we call fork on the process. This starts a new process off of the parent process:
    - The parent process returns the pid of the child process.
    - The child process return the pid of 0. This is how we know which process is which in the code.
  + The child process will send the signal (SIGSTOP) to itself. This essentially **pauses** the process.
  + The parent process will send (with the ***kill*** function) a signal call and a process id, which are both specified by the user.

To get the flag, the child process needs to continue execution. This can be done by sending the SIG\_CONTINUE signal to the child process has a value of 18. Since we have made the process continue via our signal, the process will continue on and print the flag.

Flag: SC3{sIgNA1s\_ma[3\_7HE\_W0rLd\_G0\_R0uND}

Car

Who's arm is pictured in this photo? Flag format is SC3{firstname\_lastname}

Hint: This vehicle is probably older than you are.

# Solution

The picture of the car is a *Chrysler* car from the 1990s, which can be seen from the logo on the car. By looking at *Chrysler* cars from the 1990s, it is easy to determine that this is a *Voyager.* There is also a license plate number on the car as well. By looking at <http://www.worldlicenseplates.com/> will yield this being a plate from Germany.

Putting this altogether into a Google Search, will yield articles from a person named David Tracy.

Flag: SC3{david\_tracy}

Location 1

­­For each of these challenges, the goal is to find exactly where the photo was taken. The flag would be the location all lowercase and spaces replaced with underscore. This photo is taken at a famous landmark. What’s the name of that famous landmark? Flag format is SC3{location}

# Solution

Simply uploading the image to *google images* for a search will output Copley Square as the location.

Flag: SC3{copley\_square}

Location 2

Prompt: This one is a little tougher. This photo was taken from a hotel window. What is the name of the hotel that this photo was taken? Flag format is SC3{location}

# Solution

View the building in front; it says Westin. Use Google maps to find all Westin hotels and see all the Westins in America. One should look familiar in terms of landscape. Now, you know it’s in Boston Massachusetts. The tough part is finding which exact hotel you are staying at. The tall building in the back is the Prudential Building. Finally, you see a hotel that is across the street of the Westin and in the distant is the Prudential Building. This is the *Fairmount Copley Plaza*.

Flag: SC3{fairmont\_copley\_plaza}  
  
Location 3

Prompt: You have been working hard and your friends are seeing this. They offer to pay for your dinner. You then ask where is the restaraunt, but they simply reply back to you with an image. What is the exact name of the restaraunt? Flag format: SC3{Name of restaturant}

# Solution

Seach up store to the right, hoooop. You see there are a bunch in Japan. Find the one that looks similar to the one in the photo. You find the correct one and see the building to the left of it. In google maps you can see the name of the restaurant as ‘おおき食堂’.

Flag: SC3{おおき食堂}

Astros

Can you pick the coaches signs? He's giving signs to his catcher from the dugout. Good luck, cheater ;)

# Hints to Give

* Pattern matching! Check for an *indicator* that a pitch is coming.
* Is there any action that is only shown ***once****?*

# Solution

The ***nose*** is the indicator. After the nose is selected, the ***second*** sign after this is the pitch being thrown. The pitches are as follows:

* **Cap**: Fastball
* **Left Ear**: Change Up
* **Chin**: Curveball
* **Right Ear**: Knuckleball

For example, the sequence “*Cap, Left Ear, Nose, Right Ear, Cap, Chin, Cap, Cap, Left Ear, Chin”* will be used. The nose is the *third* sign. This means that the actual pitch call will be the *fifth* sign. Since the *fifth* sign is the ***cap***, then a fastball will be thrown. Do this correctly 10 times to get the flag.

Flag: ﻿SC3{d0nt\_ch3at\_k1dS}

Cesar Cipher

Let's do some crypto! No, not Bitcoins, cryptography! Can you decrypt the flag?

# Solution

The data is encrypted using the classic Cesar Cipher. This algorithm takes the numeric value of an ASCII character and shifts it several characters (in a circle) for the encryption process. In this case, the shift is 11 characters.

Flag: SC3{the game is zork}

Othello

So, you think you're good at board games, yeah? If you're so good, let's see you try to beat the A.I for the retro board game Othello! I can tell you, the game is NOT over until it is over. Don't celebrate too early ;)

Hint: There is no exploit in the game! The goal is to beat the A.I. (that's it).  
Hint: You will not beat the A.I. with random moves. Read up on strategy guides for the game.

# Hints to Give

* Keep the corners as long as possible. If this is done, you have a good chance of winning.
* Read up the rules of Othello!

# Solution

Simply beat the A.I! This can be done by understanding basic strategy about the game. In Othello, giving up corners is really bad, since these coins cannot be flipped. This is the main strategy implemented by the A.I.

The A.I. has two major flaws:

* It will not play in spots around the corners essentially no matter what.
* It does not value the edges well enough.

In order to exploit these flaws (to win) do the following:

* DO NOT play in spots directly around the corners (i.e. 2A, 1B and 2B). This will force the A.I. to make bad decisions on placement. This is the key.
* Take control of the sides early! This makes it difficult for the A.I to control the full board.

Flag: SC3{#World\_Champion\_AI\_:3}

SOC 1 – What IP?

The Security Operations Center (SOC) has indicated some suspicious behavior on an endpoint and have sent you the Windows Security and PowerShell event logs to identify the Indicators of Compromise and any additional information. The two files are CTF\_Security.evtx & CTF\_Windows PowerShell.evtx.

What is the IP address that the malicious Powershell script reaches out to? Find this from viewing the Windows Event Logs.

HINT: Use an event log viewing tool for EVTX files. 'Windows Event Viewer' will do this by default.

HINT: Look for outgoing connections that use powershell

# Hints to Give

* Go to the log entry with the base64 powershell script. Point out this data is base64 encoded and looks interesting.
* If the data looks invalid to them, tell them to remove the invalid data. Cyberchef is really awesome for this!
* Another solution is simply looking for output connections

# Solution

Open the file ‘*CTF\_Windows PowerShell.evtx’* file in any Event Log analysis tool. Even the default tool that will open the file in Windows will work fine for this. Within the logs is a complete powershell script that is base64 encoded, which is directly below the *HostApplication* variable. Base64 decode this data using a tool like Cyberchef.

Once the data is decoded (and the junk between each character is removed), there is an IP address that begins with ‘*http:*’. This IP is the correct answer for this.

Flag: 172.16.28.5

SOC 1 – Which User?

What User account was able to log into the system from the system identified in the PowerShell script?

HINT: What's the event code for LOGIN on Windows?

# Solution

Open the ‘*CTF\_Security.evtx’* file. Since we are checking for *valid logins* it would make sense to find out the event type ID for this event. From a google search, we find out that the event ID for a logon is *4624*.

In our Event Log analysis tool, we can use *filter* this file by the Event ID *4624* or sort by event IDs. If you look through all of the events for the *‘Account Name*’, then you will eventually find the user that was logged in. The name of the user is ‘JAdams’.

Flag: JAdams

SOC 2 – What’s My Hash?

Flag: JAdams

The SOC has also forward potential malware sample to you for evaluation. They have sent you a 7Zip folder with the file name CTF\_Spokane2022.7z with the password “infected”. The SOC manager is requesting the following information concerning the payload.

What is the SHA-256 Value of the powershell script?

HINT: Just hash the whole file

# Solution

Take the SHA-256 hash of the file. This can be done via bash, an online website or several other ways. The flag is the SHA-256 hash output.

Flag: 6881531ab756d62bdb0c3279040a5cbe92f9adfeccb201cca85b7d3cff7158d3

SOC 2 – What’s My IP?

What IP address does the script establish a connection with?

HINT: Base64 \*decode\* the data in the powershell script.

HINT: Decode the final binary that gets executed in the powershell script via <https://gchq.github.io/CyberChef/>.

# Solution

The Powershell script is heavily obfuscated. In the top of the file, it is base64 decoding a string then executing these characters. To find the original powershell code being ran, un-obfuscate this by base64 decoding the content.

The newly decoded data is attempting to execute a binary via another base64 encoded blob. To find the original IP, we need to deobfuscate the base64 encoded blob and XOR it with 33. The easiest way to do this is to use <https://gchq.github.io/CyberChef/> with a base64 decode and XOR with 33. At this point, the data looks like garbage; this is because the decoded data is a *binary executable.* If we look for strings within the binary, a single IP address pops out: 47.242.164.33.

Flag: 47.242.164.33

SOC 2 – Virus Totalz?

What has Microsoft identified this file as? Go to <https://virusTotal.com> to find out!

# Solution

Upload the whole file or the hash of the file to Virus Total. On the *‘detection*’ page, there are opinions about the file; one of them is Microsoft. The fourth column of this table shows the *type* of malware that the file is. Microsoft classifies this as ‘TrojanDropper:PowerShell/Cobacis.B’.

Flag: TrojanDropper:PowerShell/Cobacis.B

SOC 2 – Tooling

What tool produced this script?

# Solution

Upload the whole file or the hash of the file to Virus Total. Under the *‘Community’* tab, a user named VMRay has notes on the malware. One of the notes is ‘*Threat Name*’ as *‘Cobalt Strike*’, which is the tool that generated the payload.

Flag: Cobalt Strike