Spokane Cyber Cup II

**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**

There are no specifics on what hints to give people for each challenge. However, there are some general guidelines :)

* 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 be key’s 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.
* There are several challenges that require manual verification with a coach. Ensure that the students have met the completion criteria for the challenge, then give them the flag.
* If the students have found an issue in the challenge, or it appears to be unobtainable, please contact either Brady or Max 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!

**Solutions**

This has the solutions for all of the challenges throughout the CTF. Please do not show this to the students; only use this as a guide to help yourself. Some of these do not have full explanations for them, as it would be too long. To get all information on a challenge, either go to the Github repository or talk to the creator of the challenge or Max.

**Privilege Escalation - 1**

The goal is to teach the students about file permissions in Linux. File permissions govern who can access what on a file system.

In Linux, there are three settings: owner privileges, group privileges and others. Within these different user types are different permissions: read, write and execute. These can be easily seen by using the *'ls -lA*' command on a file.

Another set of powerful tools exist: setuid, setgid and sticky bits. For this vulnerability, we will be abusing an issue with a poorly set ‘setuid’ bit on a binary. When the setuid bit is used, the behavior works as described above besides that modified so that when an executable is launched, it does not run with the privileges of the user who launched it, but with that of the *file owner* instead.

The vulnerability is in the **dd** and **date** binaries, which have the ‘setuid’ bit turned on. By giving both of these binaries the file ‘flag.txt’ file as the input, an error message will appear that prints the contents of the file. The following payloads will print the flag:

* Date -f flag.txt
* dd if=flag.txt

Flag: flg{file\_perms\_are+fun!}

**Basic Reverse - 3**

This is just a basic login form where the password is stored in plaintext in the binary.

In order to complete the challenge, just run `*strings reverse*`. This should output the flag and the password.

Flag: flg{N1ce70bHAc5er}

**Pokemon - 4**

In the original Pokémon series, the way in which EXP was calculated for a Pokémon at level 1 introduced an **integer underflow** in the EXP. This (should have been negative number) was now gigantic and huge! Instead of going to a very low level, the unsigned value rolled over to a super large value, moving the Pokémon to the max level (100). This game has recreated that flow :)

The vulnerability function is in the growth rates. All of these cannot go below 0 except the fast-medium growth type. The fast-medium growth type has an **integer underflow** which can wrap around the exp to be negative, creating an insanely large level (above 100). By reading through the stats.csv file and reversing the Pokémon storage format it is discovered that only the Pokémon **Shuckle** is in this growth rate group.

In addition to the particular growth rate group for this to be possible, a very small amount of EXP (experience) needs to be obtained in order to keep this in the negatives. In order to achieve this, both Pokémon should be a level 1 (the lowest possible level) and the Pokémon should have the smallest amount of HP (health points or bulk) as possible. There is only one Pokémon that has a small enough HP points; this is Pikachu.

In order to beat the challenge select **Shuckle** at **level 1** (as your Pokémon) and **Pikachu** as the opponent Pokémon at **level 1**. Once Shuckle wins the battle the EXP should be negative, resulting in a very large EXP gain to create an absurdly high level.

Flag: flg{integer\_underflows\_are\_fun!}

**SSH - 5**

Just use the credentials given in the box to SSH into the server. The flag is in the directory.

`*ssh* [*chal1@wargame2.spokane-ctf.com`*](mailto:chal1@wargame2.spokane-ctf.com%60)

Flag: flg{first\_flg!}

**Moving - 6**

There is a list of nested directories. In order to get the flag `*cd ./dir1/dir2`*. Then, just `*cat flag.txt`*. This will output the flag.

Flag: flg{you\_are\_a\_linux\_gururururu}

**UAF - 7**

The goal of the game is to get an impossible high score. The game randomly calculates four numbers and add them together (max of 100). In order to do get a high enough high score, the contestant needs to abuse a **use-after free** (UAF) vulnerability. This is when a heap pointer has been given back to the computer, but we still have an object pointed to this location that we control.

In this situation, the player object can be freed, resulting in the score variable to be overwritten by other aspects of the challenge. The player struct of 12 characters and an integer (16 bytes total) and the dynamically allocated array of 4 integers (4 bytes a piece, giving us 16 bytes) get allocated to the same range. *Because we still have pointer to the player object in the player score, this gets overwritten by an integer allocated into the array.*

Flow:

* Create player (create AAAA)
* Add one score (new)
* Reset the player (reset).
  + This frees the player, but still has a pointer to it.
* Add 20 scores (new) \* 20 in order to guarantee the setting of the value
* Check the score (won)
  + The score should have been corrupted to something like 103.

Flag: flg{use\_after\_Fr33.Be\_g0n3\_satan}

**Grep - 8**

Grep is a wonderful tool for quickly searching a file or a group of files for text keywords. In order to solve this challenge, just grep for the prefix *flg{*. This should show a single value. The final command is: `*cat myfile | grep 'flg{'`*.

Flag: flg{grep\_is\_so\_cool!}

**Auth - 9**

This is a basic authorization prompt. However, the auth will always fail.

This is a basic buffer overflow vulnerability, where the username buffer does not have a bounds check, allowing for data to be overwritten. It should be noted that all exploit mitigations are turned off for this (DEP, ASLR, stack canaries, etc.)

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.

Please refer to the repository on Github for an in-depth explanation on this vulnerability and the exploitation.

Flag: flg{stack\_smashing\_for\_fun\_and\_profit}

**Wildcard Madness - 10**

The goal is to use non-alphabetic characters (besides 1) in order to beat this challenge. The key to solving this is wildcards.

* '\*' will call all existing. For example, ls \* will print all of the files in a directory.
* '?' will call all with a specific amount. For example, ls flag.??? will list all files that start with 'flag.' and end with 3 characters.

There are an infinite number of solutions to this problem. However, here are a few with explanations:

* This is the simplest solution that I could come up with: `*/???/??t ./????.???`.* This calls /bin/cat ./flag.txt. Doing this with a c (in Cat) may also work but will likely crash the shell.
* This solution uses echo and a few interesting directives. The initial `/???/e???` is calling /bin/echo. The second part `*$(<./????.???`)* is setting the file flag.txt as input into echo. The full solution is *`/???/e??? $(<./????.???)`.*
* This is also a really simple solution: *`/???/m??? ./????.???`.* This is very similar to number 1 but uses 'more' instead of cat.

Flag: flg{out\_in\_the\_wildwest}

**Library - 11**

The books\_by\_author?name=<something> API is vulnerability to SQL injection (SQLi). This is when a query can be maliciously altered by the user because characters, such as a single quote, are not escaped properly and executed as SQL by the engine.

This API must be exploited using a **UNION** clause or something else that concatenates data. The current flag that the hackers are searching for is in a separate table (secret). The final payload looks like this: %27%20UNION%20SELECT%20\*%20FROM%20secret;%20--%20. If you unURL encode this URL, it looks like: `*' UNION SELECT \* FROM secret; --* `. Please note the space after the comment!

How does this work:

* The query is just a string. Because of this, if values are just being added to the SQL query (such as parameters) then the QUERY ITSELF can be altered.
* SELECT b.title  
  FROM author as a, books as b  
  WHERE a.author\_id = b.author\_id AND a.name = '%s'
* Now, instead of the '%s' as a filler, let us insert a new string, such as ' OR 1=1 –
  + SELECT b.title  
    FROM author as a, books as b  
    WHERE a.author\_id = b.author\_id   
    AND a.name = '' OR 1=1 --
  + The query itself has been altered! With the single quote, we have escaped the string that was being used to encapsulate the input parameter. Now, we return everything from it.
  + Then, in order to get the value that we want, we use a UNION (which concatenates the results of two queries).
  + Below is what the query looks like with the final payload:   
    SELECT b.title  
    FROM author as a, books as b  
    WHERE a.author\_id = b.author\_id AND a.name = '%s'’ UNION select \* from secret; --

Flag: flg{SQLi\_1s\_fUn}

**Korean Food - 12**

The goal of this challenge is to become an administrative user (admin user) of the site.

In order to do this, we need to exploit a predictable session cookie vulnerability. Session cookies are used in order to keep track of state for a specific user. However, if these tokens are predictable, then other users can become other users.

The session cookies in the application (can be seen by going to devtools->application->cookies) are a single base64 encoded value. By base64 encoding the text ‘admin’ or YWRtaW4= you have now become the admin user. In order to get the flag, just do a simple Korean Food search and the flag will appear.

Flag: flg{randomness\_and\_crypt0\_is\_hard}

**Side Channel - 13**

In computer security, a side channel is information that exposes data that is not an issue with the algorithm itself but with the implementation. Common examples of this are timing information (this challenge), power consumption, electromagnetic information or sound.

The goal of this challenge is to figure out the pin of the application. Normally (knowing this is 8 characters long) it would take 10^8 tries. However, this can be put down to 10 \* 8 tries.

In order to solve this challenge, use the timing difference between a correct and incorrect character. For example, ./client 00000000 will return very quickly. However, ./client 10000000 will take significantly more time. Because of this time difference, we can figure out the entire passcode. This process could/should be completely automated in order to get the pin

Note: There are several known issues with this setup.

* The timing differences can be different depending on the location of the student.
* A server can only have one connection at a time.

Flag: 12348219

**Arbitrary Redirect - 14**

Arbitrary redirects are really bad in general. They are bad for phishing campaigns because they convince users that the site is the correct one, and may not realize a similar looking site. Additionally, arbitrary redirects can make several vulns much worse, such as OAuth bugs.

There are two possible solutions for this:

* page=//google.com. The // tells the redirect to go to a direct domain
* page=http://<IP>. This redirects using an IP address to bypass the filter.

Flag: flg{these\_are\_badddd\_and\_obvious}

**Login Form: Flag 1 - 15**

This site is made only using static HTML and JavaScript on the frontend, with a small Django backend. This page only has a login page on it. The query, from the user, is being displayed on the page in order to make debugging easier.

The vulnerability exists in the login query, as SQL characters are not being escaped. The query looks like this: `*select \* FROM login WHERE username = ‘<username>’AND password = ‘<password>’`.* In order to cause an error, simply just add a single quote (‘). This alters the actual query itself!

In order to bypass the login mechanism, use the following username: `' OR 1=1 -- `. Broken down, the single quote escaped the string for the username. Then, we add an `OR 1=1` in order to get a statement that always returns true. Finally, the `-- ` (notice the space at the end) is a comment which escapes the rest of the query. This should return a flag to the user.

Flag: flg{sqli\_is\_super\_fun!}

**Login Form: Flag 2 - 16**

When creating this challenge, we accidentally created an HTML injection/XSS issue on this. So, for fun, I thought that we should keep this in here. XSS/HTML injection is having the ability to modify the content of the page directory with styling or actions.

The page does not sanitize the input going into the DOM. To exploit this, simply add an HTML page of any kind. For example, a username of <marquee>hello</ marquee> will alter the content of the page.

All the student needs to do is demonstrate that they can get a tag (bold, italics, etc.) to alter the look of the page or get an alert box to appear.

Flag: flg{crisscr0ss}

**Admin Panel - 17**

Source code analysis is quite common in the job of a pentester! Often, you may find a hardcoded password or other low hanging fruit. So, we have tried to simulate this aspect of pentesting directly!

The **ping.php** file has a command injection vulnerability in it. An attacker can inject content into the OS command being executed directly. For example, `ping google.com -c 1; whoami;` would display the current user running the web server.

In order to get the flag, the student needs to show that they understand the following things:

* What the vulnerability is (command injection)
* How they would exploit it. This just needs to be a general idea, they do not need to provide the full payload.

The more important of the two is the first one (what is the vulnerability). The second item does not matter as much; just try to push the students understanding.

Flag: flg{remote\_code\_execution:)}

**Diffie Hellman - 18**

The Diffie Hellman Key Exchange is an easy (and safe) way for two people to get the same key. The student needs to demonstrate that they understand the protocol.

Feel free to guide them throughout the workings of this, drawing diagrams and so on... Especially with the last one. The students need to get **every question correct** besides the optional one to get the points for this problem.

Here are the questions to ask, as well as the expected answers:

* What is Diffie Hellman used for?
  + Creating a shared key between two parities
* What numbers are out in the open for everyone to see? What are the requirements of these numbers?
  + P and g, or the prime number and the root of the prime.
  + Prime number(p) and primative root(g).
* What is kept secret? What is kept out in the open?
  + Alice and Bob each have a private value that they only keep to themselves
  + They each generate g ^ secret\_val % p. Then, they send this value away.
* What do Alice and Bob do with these values?
  + key = other\_secret\_to\_power ^ my\_secert mod p
  + This is freaking amazing! Now, they have the same value!
* *Optional*: How does this work?
  + Magic with how modulo and exponents work.
  + By rule, any value that is (x^y)^z = x^yz. So, g^ba mod p = (g^ab) mod p.

Flag: flg{diffie\_hellman\_key\_exchance}

**Huffman - 19**

Huffman encoding is a very common compression algorithm, that is used with zip files. In this challenge, the hackers are given the huffman table and the output string of the text, ran through the compression algorithm.

To solve it, do the following steps:

1. Understand the huffman compression algorithm.
2. Import the huffman table using a key-value pair (replacing the \s and \n with space and newline characters)
3. Import the cipher.txt into a string.
4. Write the decoding algorithm:
   1. Huffman has an interesting property about it: no individual complete table key (011010) is a substring of another key. So, once you see a value that is a key, then you *know* this is the correct entry in the table.
   2. Search character by character in the string. If the character is not a valid entry, then append the next character to the string and check again.
   3. Once a valid entry is found, remove the characters that made up the valid entry from the string. Place the 'valid entry' from the key-value pair (letter 'a') into the flag value.
   4. Repeat the process until the entire string of 0's and 1's is gone.

The Github repo has a Python script that works to decode this.

Flag: do this with code not by hand. Hack the planet!

**Cesar - 20**

Encryption is the process of encoding a message or information in such a way that only authorized parties can access it and those who are not authorized cannot.

The Cesar Cipher is a quite famous and well-known cipher, as it is essentially the beginning of Cryptography! The cipher works by shifting a letter ‘x’ times (the key is x) to encrypt it and the other way to decrypt it. For example, MOM with a key of 5 would translate to 12 14 14. With the shift, this becomes 17 19 17 or rtr.

If the value is greater than 25 (0-25 are valid because there are 26 letters in the alphabet) then we mod (take the remainder) of the value. For instance, Z, with a key of 3 would go from 25 to 28. 28 % 26 would give use a key of 2. The same works but in the negative direction.

The challenge has a key of 14 (shifted 14 times). In order to solve the challenge, shift each character 14 times to the left (with wraps).

Flag: flg{cesar\_was\_a\_genius}

**Hamming Weight 1 - 21**

The Hamming Weight is simply the amount of 1’s in each string.

weight(A) = weight("1101 1110") = 6

weight(B) = weight("1111 0000") = 4

weight(C) = weight("0101 1101") = 5

The flag is just these answers concatenated together.

Flag: 645

**Hamming Weight 2 - 22**

To find the max distance between strings, find the # of places where they are different.

d(A,B) = xor("1101 1110", "1111 0000") = 4

d(A,C) = xor("1101 1110", "0101 1101") = 3

d(B,C) = xor("1111 0000", "0101 1101") = 5

flag: B,C

**Bowling - 23**

In order to do this, we must consider 2 things: The number of pins still up and the locations of each pin. Because order of the pins (pin 1 and pin 2 vs pin 2 and pin 1) does not matter, this is a combination problem and not a permutation problem.

For a single set of pins, we need to consider C(n,i) or n! / ((i!)(n - i)!) where n is the total amount of pins and i pins left. For the problem (15 pins), this would be 15! / ((i!)(15 - i)!) iterating over each amount of pins still standing. For example (6 pins) would look like C(6,0) + C(6,1) + C(6,2) + C(6,3) + C(6,4) + C(6,5). The same applies for 15 pins, just C(15,0) + C(15,1)…+ C(15,14). This ends up being 32767 possible spare combinations. Note: We do not count C(6,6) because this would mean that all of the pins are down.

Flag: 32767

**DNA - 24**

Take all combinations of the letters up to strings of length 10. Run this through an MD5 hash. Eventually, you will find the combination that matches. The Github repo has Python code that does this programmatically.

Flag: AATGTACACT

**Morse - 25**

The file given is a morse code audio recording. The audio file gives off the following scheme:

..-. .-. --- -- | - .... . | -. --- .-. - ....

F R O M T H E N O R T H

It is likely that students will get tripped up at the TH in THE. This because there is a single -, then 4 dots, which can also be interrupted as a 6. If students are stuck on this, then tell them to listen very carefully for the pauses.

Flag: FROM THE NORTH

**Passive - 26**

The file is a pcap file, which records network traffic. The easiest way to view this file is with Wireshark.

The easy way to find the correct traffic is to sort by protocol. The type of traffic being sent to a website is going to HTTP traffic.

By clicking at the 6 HTTP requests, it is found that the /login.cgi (request number 34). There are several form items in this request, where the form item is 'login\_authorization'. This form value is YWRtaW46MTIzNDU=, or admin: 12345.

Flag: YWRtaW46MTIzNDU=

**XOR - 27**

XOR has a property that, given two of the values, the other value can be discovered. In this case, the key is known and the ciphertext is known. So, we should be able to find the plaintext for the value by reversing the process.

The easiest way to do this is to swap the key with the ciphertext, then give the key as the input text in the C binary already given to the user. This will output the ciphertext. Or, this can be done by iterating the ASCII value of each index in both the ciphertext and key string. Then, put these ASCII values back into strings to get the flag.

Flag: MerryHAd71TT13123WB

**Comment - 28**

Within the page, there is a comment that has the flag inside of it. In order to do this, open up the dev tools to view the source code of the application. The flag will be within the DOM of the challenge box.

Flag: flg{comment\_comet}

**OS Image – 29**

On the image file, run the following command: *volatility -f Capturetheflag.raw imageinfo*. The important part of the command is the *imageinfo*. This will get the information needed. Within the results of this is the ‘suggested profiles’. This is just the likely OS being used. The first one is the correct answer.

Flag: Win7SP1x64

**Parent Process ID (PPID) - 30**

Use the following command: *volatility -f Capturetheflag.raw --profile=Win7SP1x64 pslist.* This command prints the processes currently running. The key to this is the ‘*pslist’* . The only process that has a parent ID process that is **not** currently running is *explorer.exe*. This has the process number 1860.

Flag: 1860

**Open Process on Port- 31**

Run the following command: *volatility -f Capturetheflag.raw --profile=Win7SP1x64 netscan*. This will show all running processes that have ports associated with them. The key to this is the *netscan* command.

After running this command, there are quite a few items being outputted. By looking at the output of the *Local Address* field, port 554 can be seen in use. The only process associated with port 554 is *wmpnetwk.exe*.

Flag: wmpnetwk.exe

**Port 554 - 32**

Port 554 is being used. Simply just google what this port is used for, and the solution will be there. The answer has to a 4 letter acronym.

Flag: Real Time Streaming Protocol (rtsp)

**MD5 Hash of Process - 33**

First, the process needs to be dumped. This can be done by running the following command: *volatility -f Capturetheflag.raw --profile=Win7SP1x64 procdump -p 2464 --dump-dir=/some/dir.* This will dump the process memory into the file, which can then be used.

Finally, take the md5 hash of the file. This can be done in multiple ways, depending on the OS being used. An OS agnostic way to do this is to upload the file to a site that automatically hashes the file for you (Virus Total will actually do this for you).

Flag: d55e41e669cb6527c7346232b96eae58

**Virus Total – 34**

Virus Total is an amazing site for easily and quickly checking to see if a file is dangerous or not. Simply just upload the file to Virus total and the only Anti-virus to see the issue with the file is CrowdStrike Falcon.

Flag: CrowdStrike Falcon

**MZip - 36**

MZip is a zip file format. The vulnerability is a directory traversal bug. The unzipper trusts that the files will go to known and safe locations, without verifying if it is in the correct location or not. In order to get the flag, create a file/folder within the /flag directory.

In order to understand the vulnerability, we need to understand the file structure:

- zip\_file\_name:

- .content: Holds all of the content for each file and name for all of the directories

- .hierarchy.key: Holds the hierarchy data for the location of each file

- .huffman.key: Holds the huffman table used for compression

The .hierarchy.key file, holds information about where the following information should be extracted to. In order to exploit this, start changing the paths within the .hierarchy.key file.

For example: change `zipped/test\_folder/file.md` to `../../../../../../../../../../../../../../../../../../../../flag/file.md`. This will create a file at /flag/file.md. Once a file is created here, the student will get the flag sent back to them.

It should be noted that using an absolute file path will not work because the unzip function concatenates the file location with the current location. So, the directory traversal (../) is the only way that this will work.

Flag: flg{winrar\_oh\_winrar}

**Total Attempts - 37**

The goal of the challenge is to find the total amount of login attempts in the sshd log file. A single line is going to be a single attempt. So, just find out the amount of lines in the file to get the number of attempts. The code for this would look like `*cat auth.log | wc -l`*.

Flag: 424

**Failed Attempts - 38**

The goal is to view the amount of **failed** login attempts. The easiest way to do this is to find a keyword that exists within a failed attempt (fail). Now, use grep to search through the file for all cases where *fail* is found. Finally, output the amount of total lines that were outputted. The code looks like `*cat auth.log | grep ‘fail’ -i | wc -l`*.

Flag: 418

**Correct Attempts - 39**

The goal is to view the amount of **valid** login attempts. The easiest way to do this is to find a keyword that exists within a valid attempt (accept). Now, use grep to search through the file for all cases where *accept* is found. Finally, output the amount of total lines that were outputted. The code looks like `*cat auth.log | grep ‘accept -i | wc -l`*.

**Oddball Attacker - 40**

There are three categories of logins: public key accepted (5), password failed (418) and password accepted(1). There is only a single *password accepted* login attempt. So, this clearly shows something weird has happened.

Now, the IP address for this is on the same line as the only valid password login.

Flag: 37.204.31.33

**Location Location Location - 41**

Where are all of the attacks coming from? Based upon the IP’s, this can be figured out by using a site such as ‘WhatIsMyIp.com`. Simply enter all of the IP’s into this website to figure out the location of it. There are three locations where calls are coming from: China (399), Russia (20), USA (5). Because China has the largest number of attempts, the answer is China 399 (with the space in between China and 399).