Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

Student Copy
AIEP 3 & 4 Session 4

CAPTURE THE FLAG

Capture the Flag (CTF) is a cybersecurity competition where participants find and exploit the vulnerabilities in a program or system in order to find a hidden secret string called a "flag." Introduced in the 1996 DEF CON (an annual conference attended by computer security enthusiasts and professionals), it gamifies the teaching of **ethical hacking skills**, such as reverse engineering, cryptography, and binary exploitation.

To get started, it is recommended to use the open-source **Linux** as your operating system, as it comes pre-installed with multiple useful tools. Popular Linux distributions ("distros") for CTF include **Ubuntu** and **Kali**; the former is more suited for general computing, while the latter is more tailored towards cybersecurity purposes. However, for introductory CTF, Ubuntu usually suffices.

> 💭 **Trying out Linux**
>
> A good place to try out Linux for CTF without installing it is via picoCTF's webshell.
> But, if you are considering using Linux for personal tasks, you may want to explore dual booting, using a virtual machine, or installing Windows Subsystem for Linux (WSL).

Basic Linux Commands

1. Downloading a file

```
wget link_to_file_to_be_downloaded
```

2. Reading a text file

```
cat file_name
```

3. Editing a text file (using GNU nano[1])

```
nano file_name
```

   To save the file, press Ctrl + S. To exit GNU nano, press Ctrl + X.

[1] Linux power users typically tend to hold strong opinions on the choice of text editor. Try searching for "vim vs emacs" on YouTube or Reddit — or take a look at this Wikipedia article on the "editor war."

4. Creating a new (empty) text file

```
touch file_name
```

5. Creating a new folder (making a directory)

```
mkdir folder_name
```

6. Going to a folder (changing directory)

```
cd folder_name
```

7. Going to special folders

`cd /` sends you to the root directory (i.e., the topmost directory in the Linux filesystem)

`cd ~` sends you to your home directory

`cd ../` sends you one directory level up

8. Running a Python script

```
python3 file_name
```

Some Python scripts include information on their usage. You can display this using either of these commands:

```
python3 file_name --help
python3 file_name -h
```

9. Running a Bash script[2]

```
bash file_name
```

10. Running an executable

```
./file_name
```

11. Clearing the screen

```
clear
```

12. Cancelling a running process

```
Ctrl + C
```

[2] Bash is a Unix shell and command language. The file extension of Bash scripts is `.sh`. However, in CTF, file extensions can sometimes be used for duping. Thus, knowing how to recognize a Bash script — or any script for that matter — by its contents (instead of simply relying on the file extension) is a valuable skill.

Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

> 💭 **It's the thought (process) that counts**
>
> The rest of this handout provides walkthroughs on solving illustrative CTF problems. Pay attention not only to the commands but, more importantly, to the thought process.

Forensics information (Link)

Files can always be changed in a secret way. Can you find the flag? cat.jpg

1. Start by downloading the given file:

   ```
   wget https://mercury.picoctf.net/static/a614a27d4cb251d04c7d2f3f3f76a965/cat.jpg
   ```

2. Running `file cat.jpg` will result in the following message:

   ```
   cat.jpg: JPEG image data <...>
   ```

   This suggests that `cat` is an image.

3. Opening the file with an image viewer will reveal the picture of a cute cat:

   This confirms that it is indeed an image.

4. Check the image properties by running `exiftool cat.jpg`
   Among the properties displayed, the license suspiciously resembles a ciphertext:

   ```
   cGljb0NURnt0aGVfbTN0YWRhdGFfMXNfbW9kaWZpZWR9
   ```

5. To decode this ciphertext, try out the different "recipes" in CyberChef.
   Using the `From Base64` recipe will reveal the flag 🏁.

General Skills Wave a flag (Link)

Can you invoke help flags for a tool or binary? This program has extraordinarily helpful information...

1. Start by downloading the given file:

   ```
   wget
   https://mercury.picoctf.net/static/cfea736820f329083dab9558c3932ada/warm
   ```

2. Displaying the contents of the file via `cat warm` will result in a bunch of garbage.

3. Running `file warm` will result in the following message:

   ```
   warm: ELF 64-bit LSB pie executable, x86-64 <...>
   ```

   This suggests that `warm` is an executable (or, at least, it is supposed to be an executable).

4. However, if we try to execute by running `./warm` will result in the following message:

   ```
   -bash: ./warm: Permission denied
   ```

5. The error message above suggests that we do not have execution privileges (that is, we are not allowed to execute the program).

   To turn `warm` into an **executable** (that is, to change its mode to an executable), run the following command: `chmod +x warm`

6. Running `./warm` again will reveal the flag 🏁.

Binary Exploitation Stonks (Link)

I decided to try something noone else has before. I made a bot to automatically trade stonks for me using AI and machine learning. I wouldn't believe you if you told me it's unsecure! vuln.c `nc mercury.picoctf.net 59616`

1. Start by downloading the given file:

   ```
   wget https://mercury.picoctf.net/static/a4ce675e8f85190152d66014c9eebd7e/vuln.c
   ```

2. Displaying the contents of the file via cat `vuln.c` confirms that it is a C file.
   Programs written in C that accept string inputs are notoriously vulnerable to a slew of attacks, such as buffer overflow attacks and format string attacks.

3. Focus on the following lines of code in `vuln.c`:

   ```
   char *user_buf = malloc(300 + 1);
   printf("What is your API token?\n");
   scanf("%300s", user_buf);
   printf("Buying stonks with token:\n");
   printf(user_buf);
   ```

   In C, `printf()` and `scanf()` are the equivalents of Python's `print()` and `input()`. The function `printf()` can be used in two ways:

   a. The first way is by supplying two arguments: the format specifier and the value to be printed. Format specifiers in C include `%d` for printing integers, `%f` for printing floating-point numbers, `%s` for printing strings, and `%x` for printing hex values.

   b. The second way is by supplying only a single argument: the value to be printed.

   In the first line in bold, we can see that the value of `user_buf` is entered by the user (us). In the second line in bold, the value that we entered is printed. What will happen if we enter a format specifier as the value of `user_buf`?

4. To test out our curiosity, run `nc mercury.picoctf.net 59616`
   Enter `1` (for buying some stonks!).
   Enter `%x` (the format specifier for hex values) as the API token.

   Doing so will display a series of hex values similar to `96bc330` (the actual values that you will obtain may be different). These hex values are actually the contents of the stack "spilling out" because of our format string attack.

   To be more technical, function arguments are stored in a region of the computer memory known as the stack. Entering a format specifier like `%x` signals to the program that it is expecting a second argument. However, we did not supply any second argument. This tricks the program into thinking that the contents at the top of the stack are the second argument. In other words, we forced the program to print the contents of the stack — to which we should not be privy.

5. Entering a series of multiple `%x` will divulge more of the contents of the stack. Entering around a hundred `%x` will display something similar to the following series of hex values to be displayed (the actual values may be different):

   ```
   8ab3490804b00080489c3f7ecdd80ffffffff18ab1160f7edb110f7ecddc708ab218028a
   b34708ab34906f6369707b465443306c5f49345f74356d5f6c6c306d5f795f79336e3834
   313634356562ffe9007df7f08af8f7edb440417f710010f7d6ace9f7edc0c0f7ecd5c0f7
   ecd000ffe9cfa8f7d5b68df7ecd5c08048ecaffe9cfb40f7eeff09804b000f7ecd000f7e
   <...>
   ```

6. Enter the displayed hex values on CyberChef and use the `From Hex` recipe. Doing so will result in a string of garbage characters. However, in this garbage, something similar to this substring — which resembles a flag — can be found:

   `ocip{FTC0l_I4_t5m_ll0m_y_y3n841645ebÿéNUL}`

7. Grouping the characters in blocks of 4 and reversing the characters in each block will reveal the flag 🏁. Note that doing this will result in the non-English alphabet characters appearing after `}`, indicating that they should be excised from the flag.

Binary Exploitation tunn3l v1s10n (Link)

We found this file. Recover the flag.

1. Start by downloading the given file:

   ```
   wget https://mercury.picoctf.net/static/06a5e4ab22ba52cd66a038d51a6cc07b/tunn3l_v1s10n
   ```

2. Running `file tunn3l_v1s10n` will result in the following message:

   ```
   tunn3l_v1s10n: data
   ```

   This is not the most descriptive message. In particular, this message suggests that the `file` command failed to figure out anything about `tunn3l_v1s10n`. A possible reason is that the file is corrupted.

3. We start our attempt to fix the corrupted file by opening it in a **hex editor** like https://hexed.it/

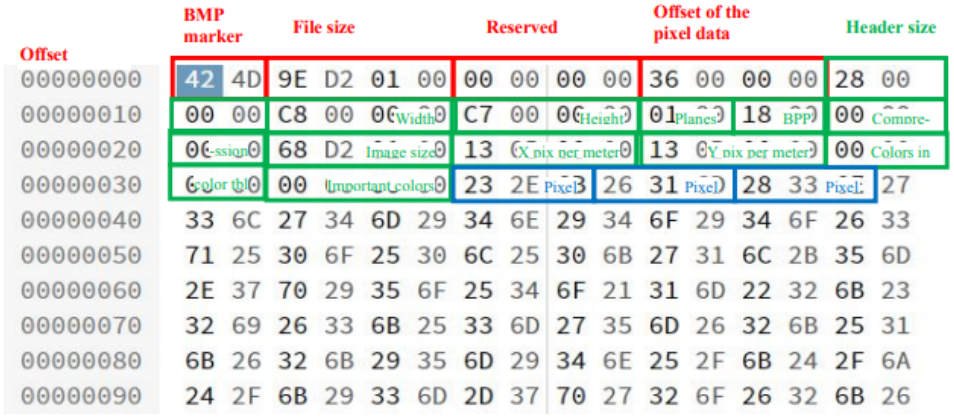
   Note that every file is just a sequence of bits. However, reading raw 1s and 0s can be terribly confusing; hence, we usually group bits into blocks of 4; four bits comprise a **hex digit.** Editing a file using a hex editor is equivalent to editing its bits.

4. Opening the file in a hex editor will display the following hex values (we add a vertical bar for readability, but it is not relevant):

   ```
   42  4D  8E  26  2C  00  00  00  |  00  00  BA  D0  00  00  BA  D0
   <...>
   ```

   On the right side of the editor, you will see a "translation" that starts with **BM**. This `BM` (or **`42 4D`** in hex) is the marker or identifier of a **bitmap (BMP) file** — that is, every bitmap file should have this at the start.

   Refer to the picture on the next page for the meaning of each hex digit in a bitmap file. A summary of the BMP file specification can be found here.



Taken from https://i.stack.imgur.com/Oh1wS.png

5. Compare the first row of hex values of our (corrupted) file with the first row of hex values in the BMP specification (the image above). For easy reference, the first row of hex values of our corrupted file is

   ```
   42  4D  8E  26  2C  00  00  00  |  00  00  BA  D0  00  00  BA  D0
   ```

   The file size is, of course, different from image to image. However, the BMP marker, the reserved hex digits, the offset of the pixel data, and the header size should be the same across all BMP images.

   Therefore, to fix the corrupted file, we have to fix the hex digits in red to match the hex digits in the specification. Doing so will result in the first row of the file being

   ```
   42  4D  8E  26  2C  00  00  00  |  00  00  36  00  00  00  28  00
   ```

6. Save the edited hex file, and append the `.bmp` file extension (since we now know what type of file it is). Opening it with an image viewer will reveal this photo:

7. Unfortunately, the flag is not there (and the photo knows it)! However, looking at the photo, it appears to have been unnaturally "cropped"; in particular, it seems to be "missing" some of its height.

   Refer to the hex values in the BMP specification, and identify which hex values specify the image height.

8. Open our uncorrputed-but-cropped photo in a hex editor. We focus on the second row, and color the hex values specifying the image height in red:

   ```
   00  00  6E  04  00  00  32  01  |  00  00  01  00  18  00  00  00
   ```

9. We experiment with changing these hex values. Some values will corrupt the image file such that it cannot be opened with an image viewer. However, some values will "extend" its height and reveal more of the image. For example, changing the second row of hex values to

   ```
   00  00  6E  04  00  00  32  03  |  00  00  01  00  18  00  00  00
   ```

   will result in this photo:

Changing the second row to

```
00  00  6E  04  00  00  52  03  |  00  00  01  00  18  00  00  00
```

will result in a better photo that completely exposes the flag 🏁 in plain view:

Some Afterthoughts:

1. Running `exiftool tunn3l_v1s10n` even before fixing the corrupted file will actually reveal that it is a bitmap (BMP) file.
2. Opening the corrupted `tunn3l_v1s10n` with ImageMagick (an image viewer that is widely used for CTF) will give us a hint that the header (the hex digits at the start of the file) is corrupted.
3. Try coming up with more sophisticated strategies for changing the hex values corresponding to the image height.

-- return 0; --