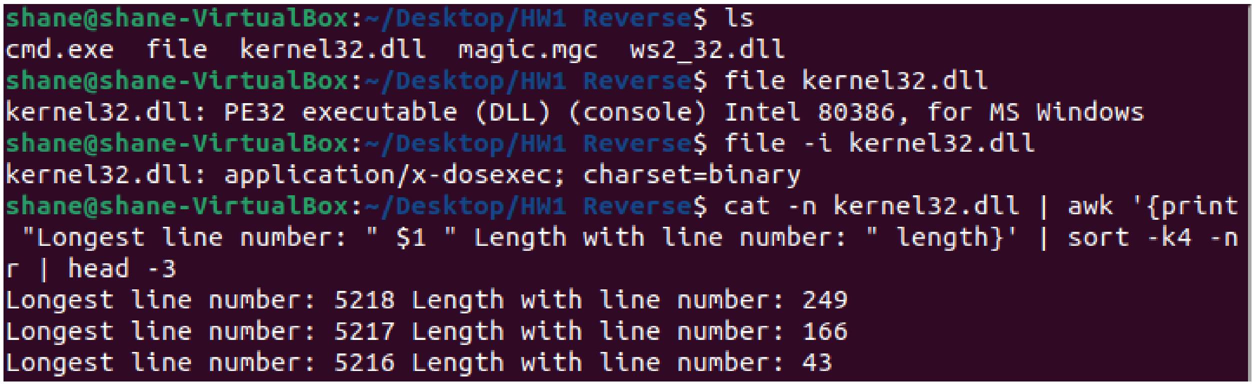
Shane Irons
Dr. Liu
CAP 5137 Software Reverse Engineering – FALL 2022
Due 9/14/2022

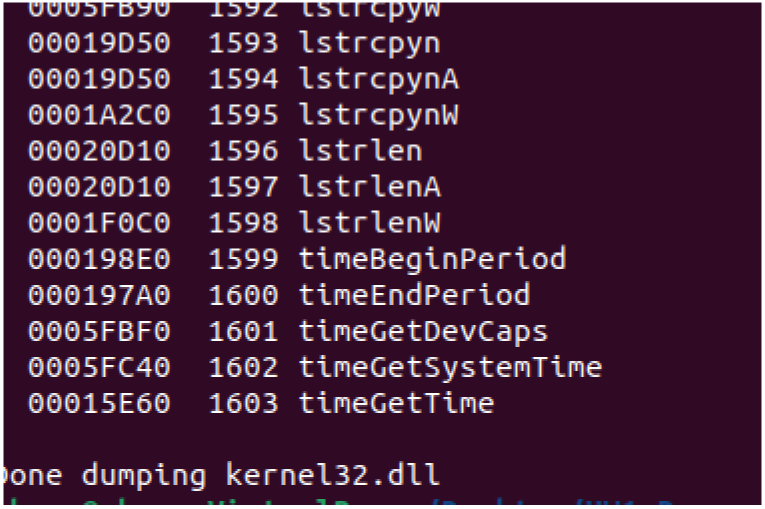
Homework 1

Question 1 (10 points) Using the “file”, “strings”, “hexdump” or any commands/tools to find out the file type,
its “magic numbers” as byte sequences (such as “\x4d\x5a”), the longest ascii string(s) (if there are ties, you need
to list all of them), the size of the .text section, and the first instruction in its .text for each of the following files:
1) kernel32.dll
2) cmd.exe
3) file
4) ws2_32.dll
5) a binary program you commonly use (such as /usr/bin/tcsh on linprog.cs.fsu.edu); for this one only, specify
the source and the size of the program in addition to your answers.
The first four files are available at
http://www.cs.fsu.edu/~liux/courses/reversing/assignments/files/ .

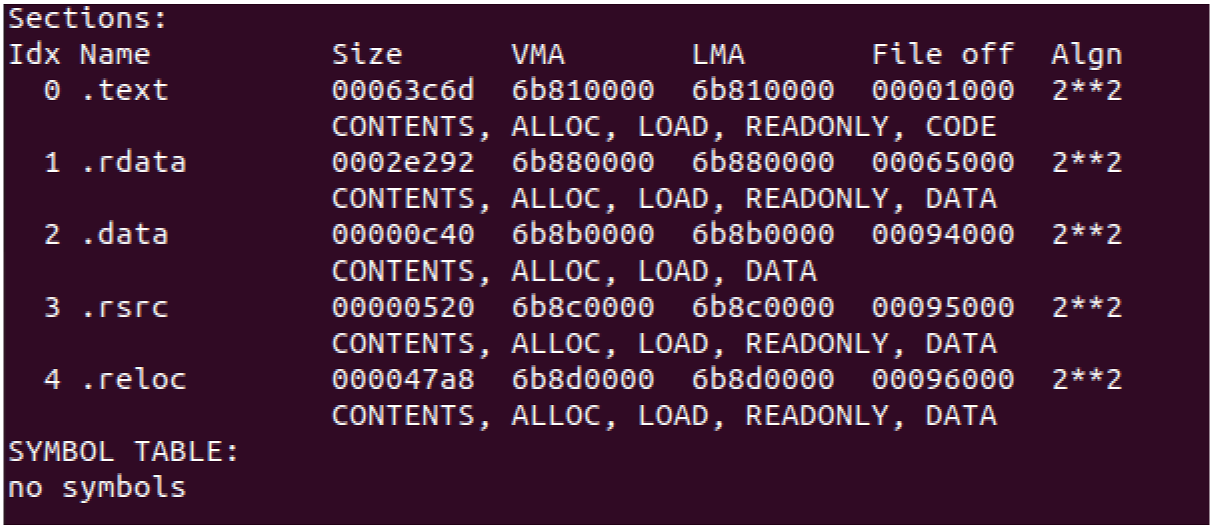
kernel32.dll:



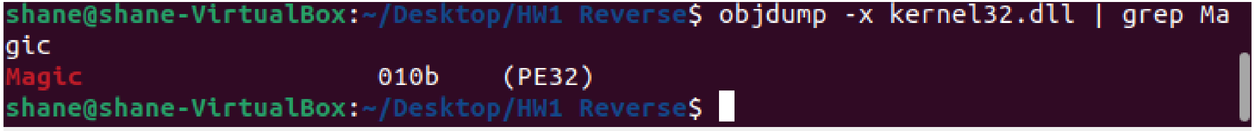
```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ ls
cmd.exe  file  kernel32.dll  magic.mgc  ws2_32.dll
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file kernel32.dll
kernel32.dll: PE32 executable (DLL) (console) Intel 80386, for MS Windows
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file -i kernel32.dll
kernel32.dll: application/x-dosexec; charset=binary
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ cat -n kernel32.dll | awk '{print
 "Longest line number: " $1 " Length with line number: " length}' | sort -k4 -n
r | head -3
Longest line number: 5218 Length with line number: 249
Longest line number: 5217 Length with line number: 166
Longest line number: 5216 Length with line number: 43
```



```
  0005FB90  1592 lstrcpyW
  00019D50  1593 lstrcpyn
  00019D50  1594 lstrcpynA
  0001A2C0  1595 lstrcpynW
  00020D10  1596 lstrlen
  00020D10  1597 lstrlenA
  0001F0C0  1598 lstrlenW
  000198E0  1599 timeBeginPeriod
  000197A0  1600 timeEndPeriod
  0005FBF0  1601 timeGetDevCaps
  0005FC40  1602 timeGetSystemTime
  00015E60  1603 timeGetTime

Done dumping kernel32.dll
```



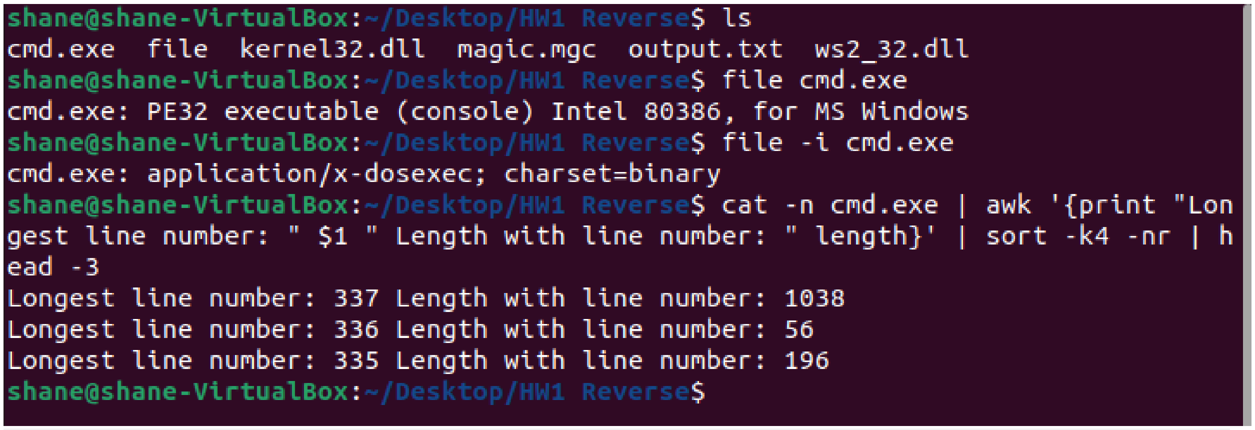
```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00063c6d  6b810000  6b810000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rdata        0002e292  6b880000  6b880000  00065000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .data         00000c40  6b8b0000  6b8b0000  00094000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .rsrc         00000520  6b8c0000  6b8c0000  00095000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .reloc        000047a8  6b8d0000  6b8d0000  00096000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
SYMBOL TABLE:
no symbols
```



```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ objdump -x kernel32.dll | grep Ma
gic
Magic                   010b    (PE32)
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$
```

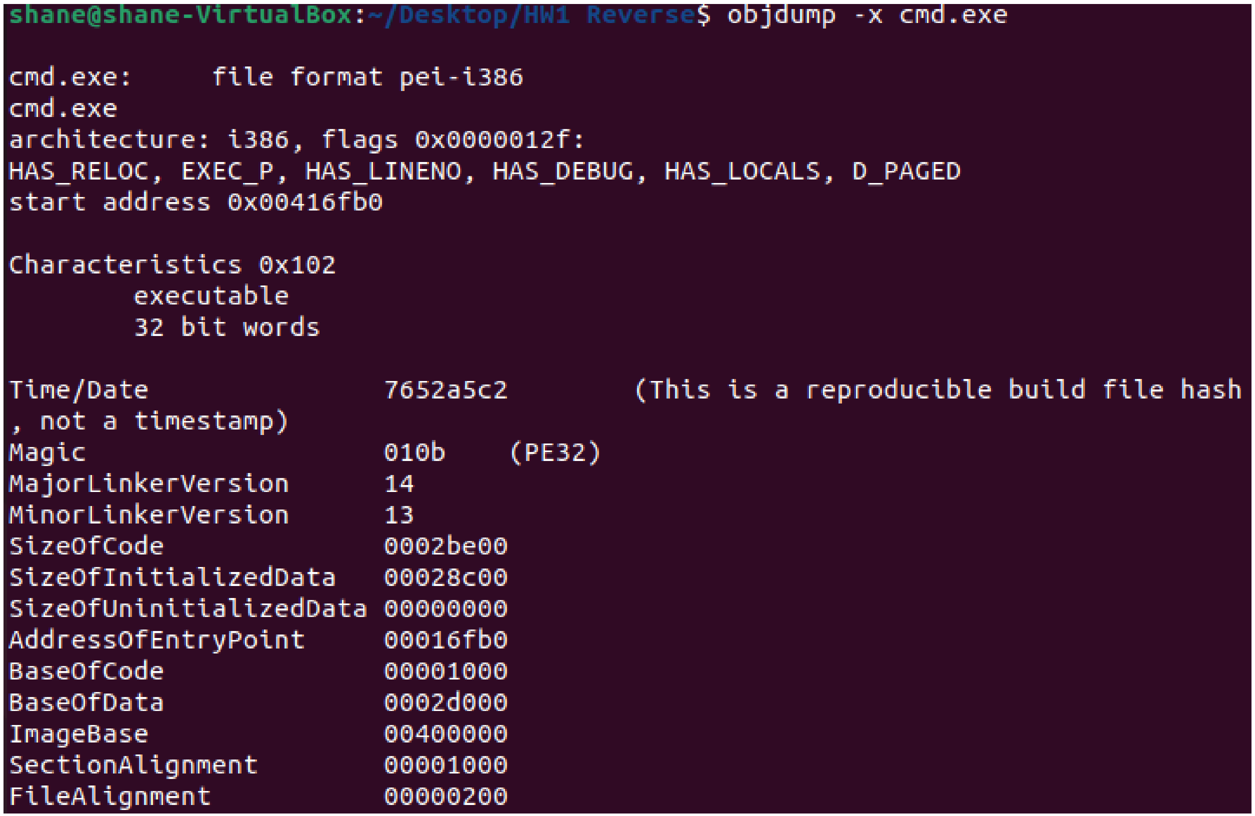
file and cat commands followed by a winedump can display some information about this dll file. We can see that it is a PE32 executable (aka DLL) and is in binary charset. The longest line number and the length is further showed below with a long command I discovered online that proves useful for this specific query. Unlike an ELF file format, DLL required some workaround to view a readable dump of the file, but once I discovered winedump I could see the dump in a more readable format. These were the only working commands I could find that displayed data relating to the things specified in the instructions.

cmd.exe:

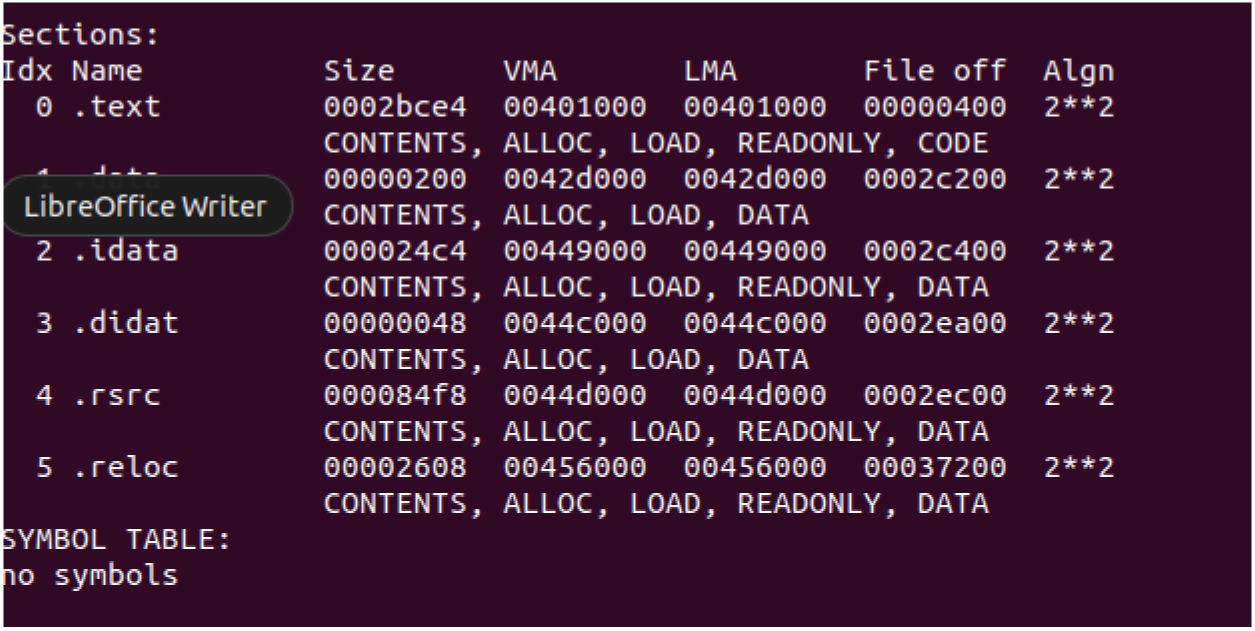


```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ ls
cmd.exe  file  kernel32.dll  magic.mgc  output.txt  ws2_32.dll
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file cmd.exe
cmd.exe: PE32 executable (console) Intel 80386, for MS Windows
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file -i cmd.exe
cmd.exe: application/x-dosexec; charset=binary
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ cat -n cmd.exe | awk '{print "Lon
gest line number: " $1 " Length with line number: " length}' | sort -k4 -nr | h
ead -3
Longest line number: 337 Length with line number: 1038
Longest line number: 336 Length with line number: 56
Longest line number: 335 Length with line number: 196
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$
```

for the cmd.exe file, it appears to be a PE32 executable file. The above commands show the file information, followed by the longest strings accompanied by line and line number. Cmd.exe is 236032 bytes in total. Below is the determined starting address and the magic 010b.



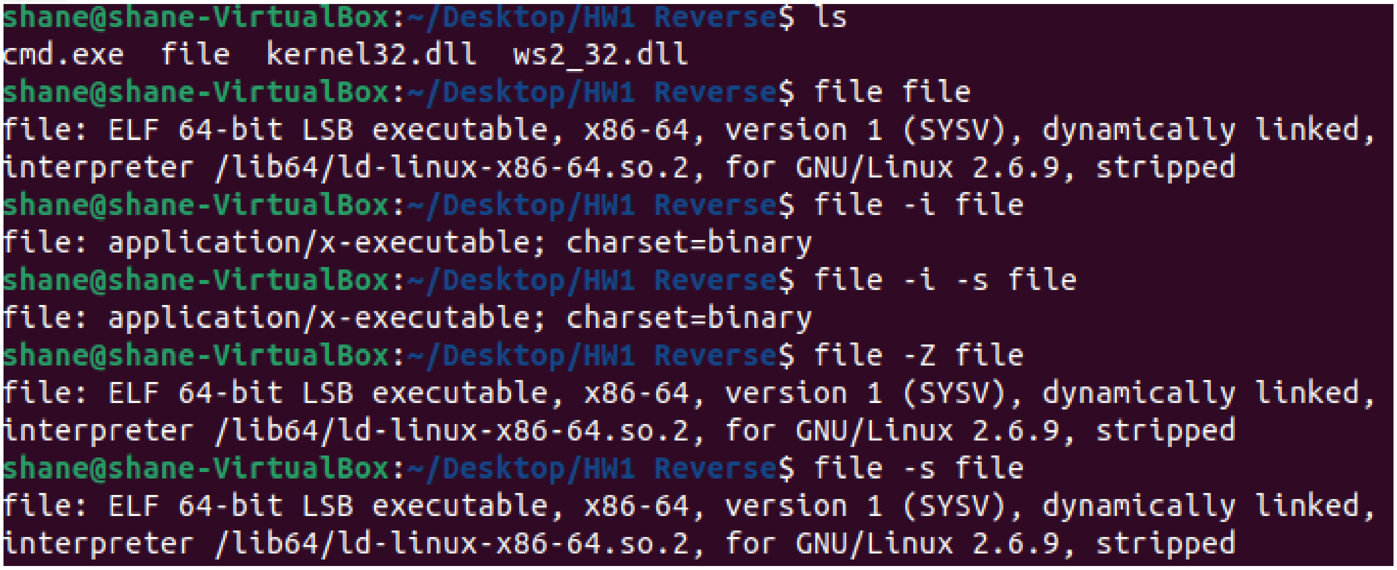
```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ objdump -x cmd.exe

cmd.exe:     file format pei-i386
cmd.exe
architecture: i386, flags 0x0000012f:
HAS_RELOC, EXEC_P, HAS_LINENO, HAS_DEBUG, HAS_LOCALS, D_PAGED
start address 0x00416fb0

Characteristics 0x102
        executable
        32 bit words

Time/Date               7652a5c2        (This is a reproducible build file hash
, not a timestamp)
Magic                   010b    (PE32)
MajorLinkerVersion      14
MinorLinkerVersion      13
SizeOfCode              0002be00
SizeOfInitializedData   00028c00
SizeOfUninitializedData 00000000
AddressOfEntryPoint     00016fb0
BaseOfCode              00001000
BaseOfData              0002d000
ImageBase               00400000
SectionAlignment        00001000
FileAlignment           00000200
```



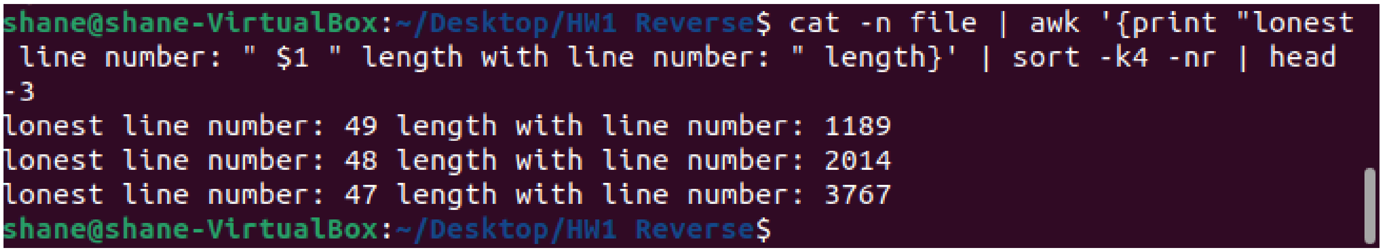
```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0002bce4  00401000  00401000  00000400  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000200  0042d000  0042d000  0002c200  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .idata        000024c4  00449000  00449000  0002c400  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .didat        00000048  0044c000  0044c000  0002ea00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .rsrc         000084f8  0044d000  0044d000  0002ec00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .reloc        00002608  00456000  00456000  00037200  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
SYMBOL TABLE:
no symbols
```

Furthermore, at the bottom of the ‘objdump -x’ command, the .text section is found which includes the size and address. The start address is 0x6b820010.

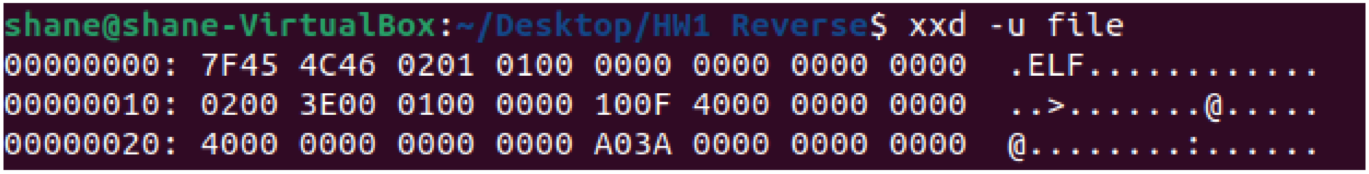
File:



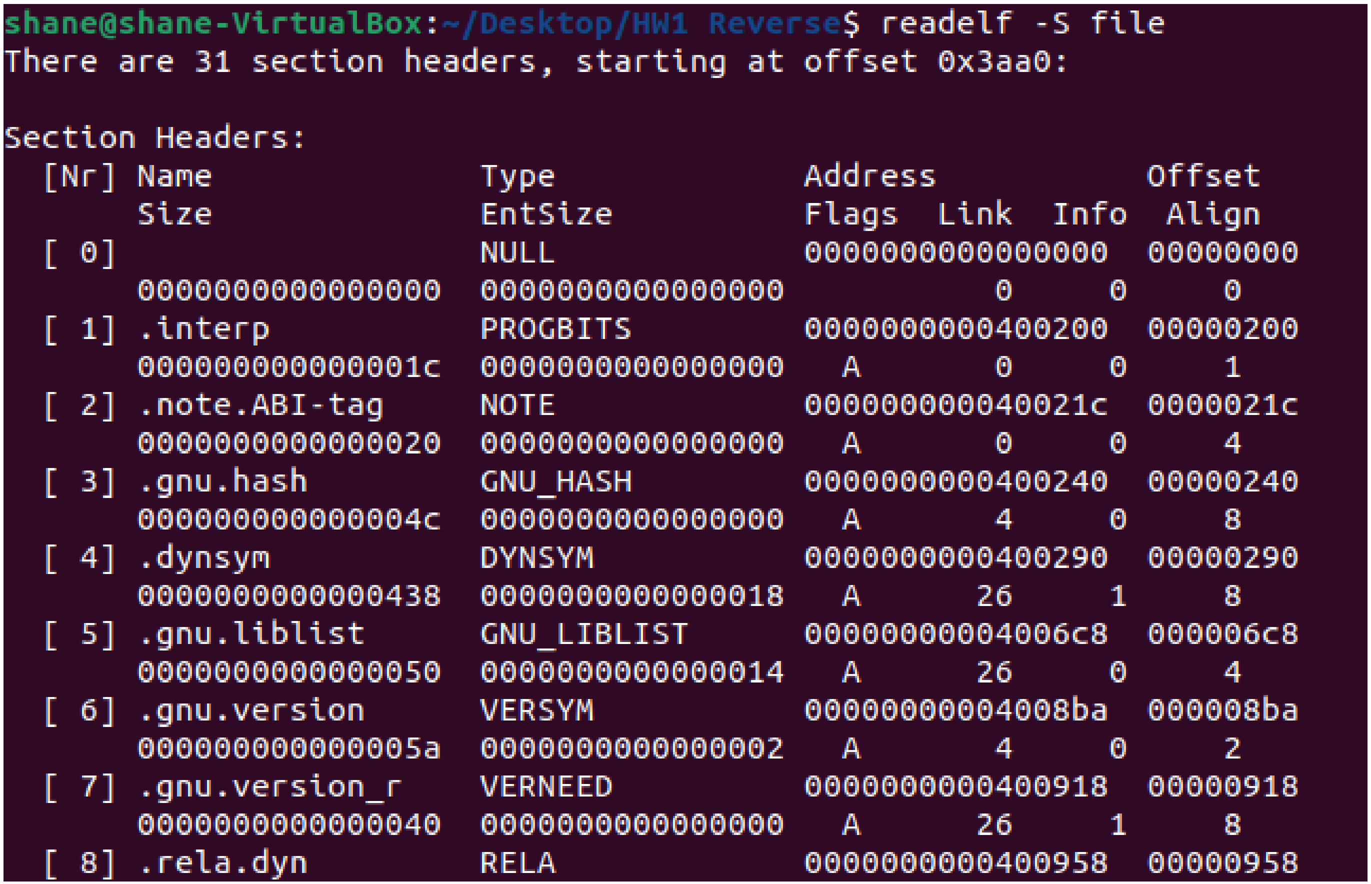
```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ ls
cmd.exe  file  kernel32.dll  ws2_32.dll
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file file
file: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.9, stripped
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file -i file
file: application/x-executable; charset=binary
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file -i -s file
file: application/x-executable; charset=binary
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file -Z file
file: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.9, stripped
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file -s file
file: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.9, stripped
```



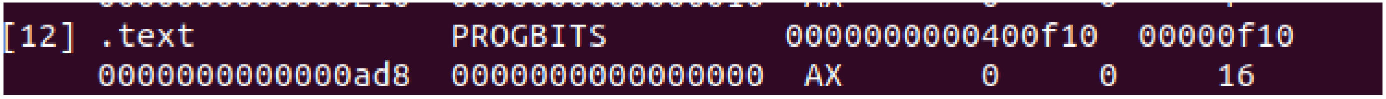
```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ cat -n file | awk '{print "lonest
 line number: " $1 " length with line number: " length}' | sort -k4 -nr | head
-3
lonest line number: 49 length with line number: 1189
lonest line number: 48 length with line number: 2014
lonest line number: 47 length with line number: 3767
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$
```



```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ xxd -u file
00000000: 7F45 4C46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0200 3E00 0100 0000 100F 4000 0000 0000  ..>.......@.....
00000020: 4000 0000 0000 0000 A03A 0000 0000 0000  @........:......
```



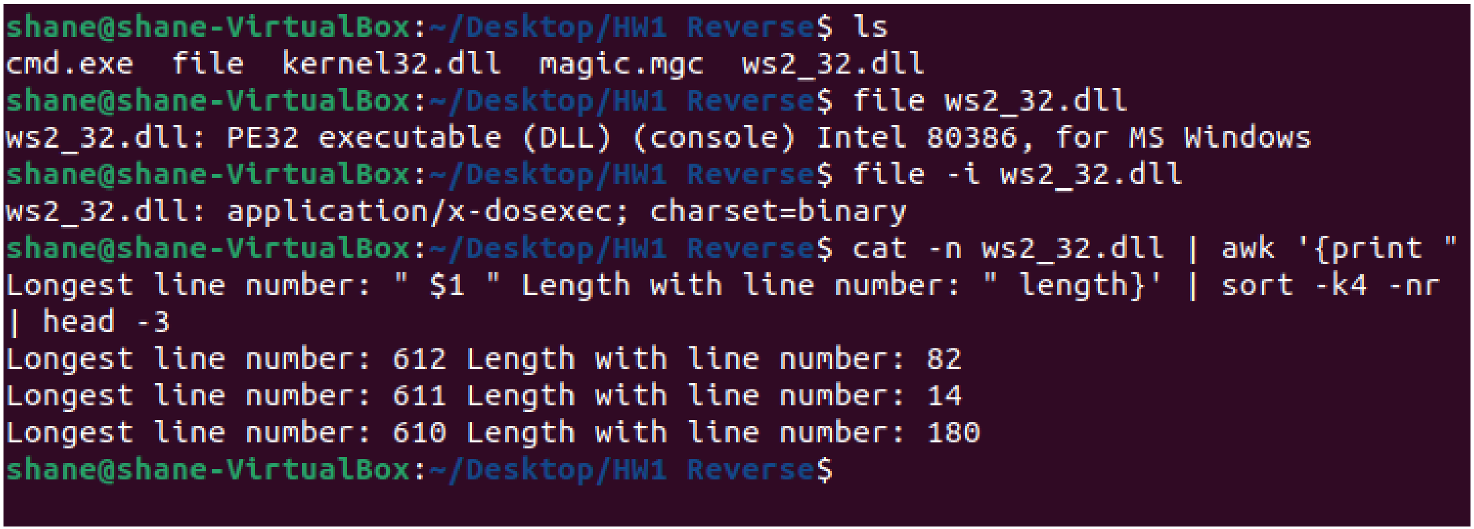
```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ readelf -S file
There are 31 section headers, starting at offset 0x3aa0:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400200  00000200
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             000000000040021c  0000021c
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .gnu.hash         GNU_HASH         0000000000400240  00000240
       000000000000004c  0000000000000000   A       4     0     8
  [ 4] .dynsym           DYNSYM           0000000000400290  00000290
       0000000000000438  0000000000000018   A      26     1     8
  [ 5] .gnu.liblist      GNU_LIBLIST      00000000004006c8  000006c8
       0000000000000050  0000000000000014   A      26     0     4
  [ 6] .gnu.version      VERSYM           00000000004008ba  000008ba
       000000000000005a  0000000000000002   A       4     0     2
  [ 7] .gnu.version_r    VERNEED          0000000000400918  00000918
       0000000000000040  0000000000000000   A      26     1     8
  [ 8] .rela.dyn         RELA             0000000000400958  00000958
```



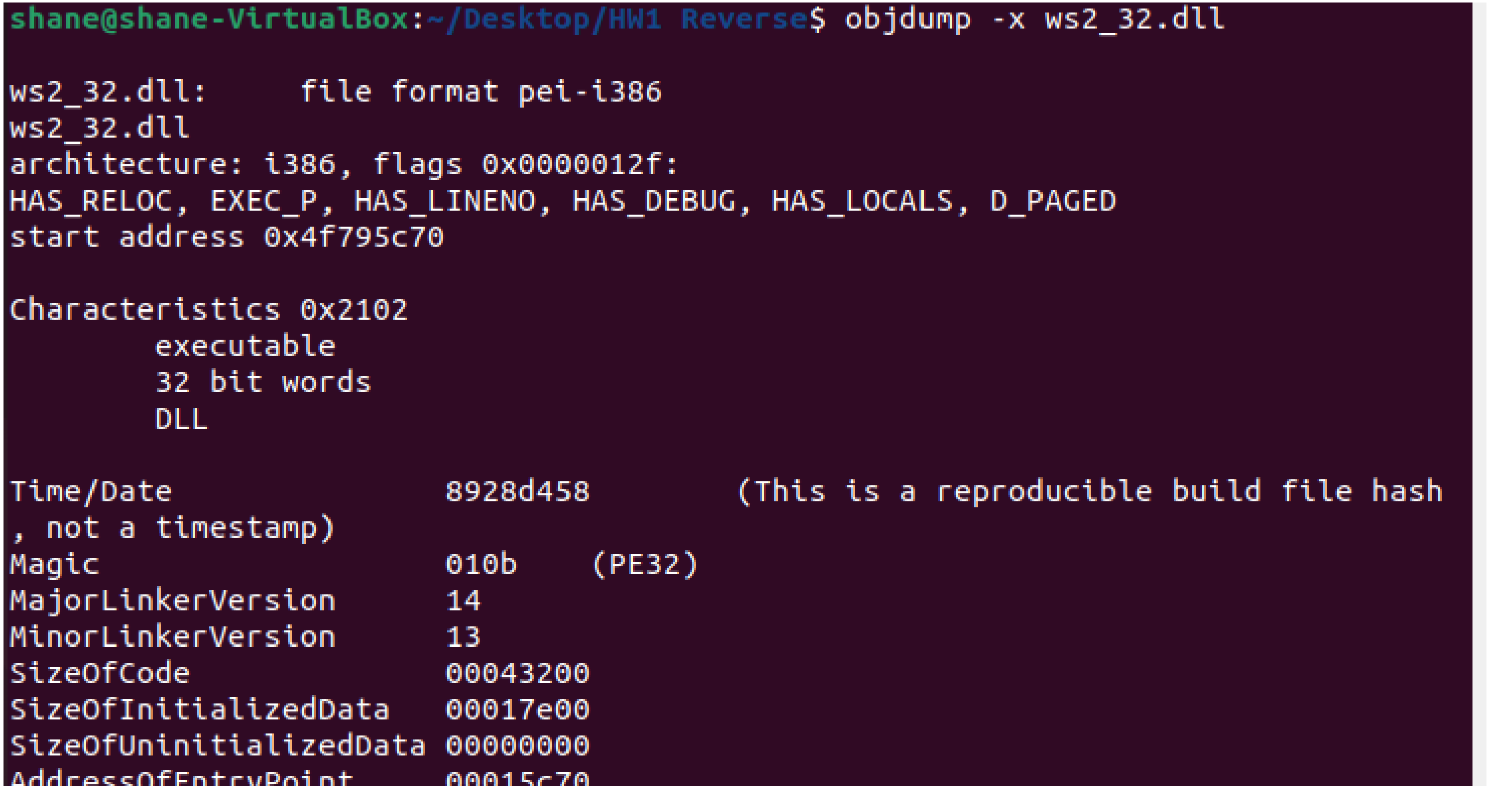
```
[12] .text             PROGBITS         0000000000400f10  00000f10
     0000000000000ad8  0000000000000000  AX       0     0     16
```

This file was the easiest to find data on in the terminal. Due to its file type, ELF, it seems to have multiple tools that easily work to project file information. The start address for ‘file’ is 0400f10. The magic number is seen from the screenshot containing the command ‘xxd -u file’. This magic number is 7F454C4602010100. This is the magic number for an ELF file, as seen later on since /bin/ls has the same starting number.

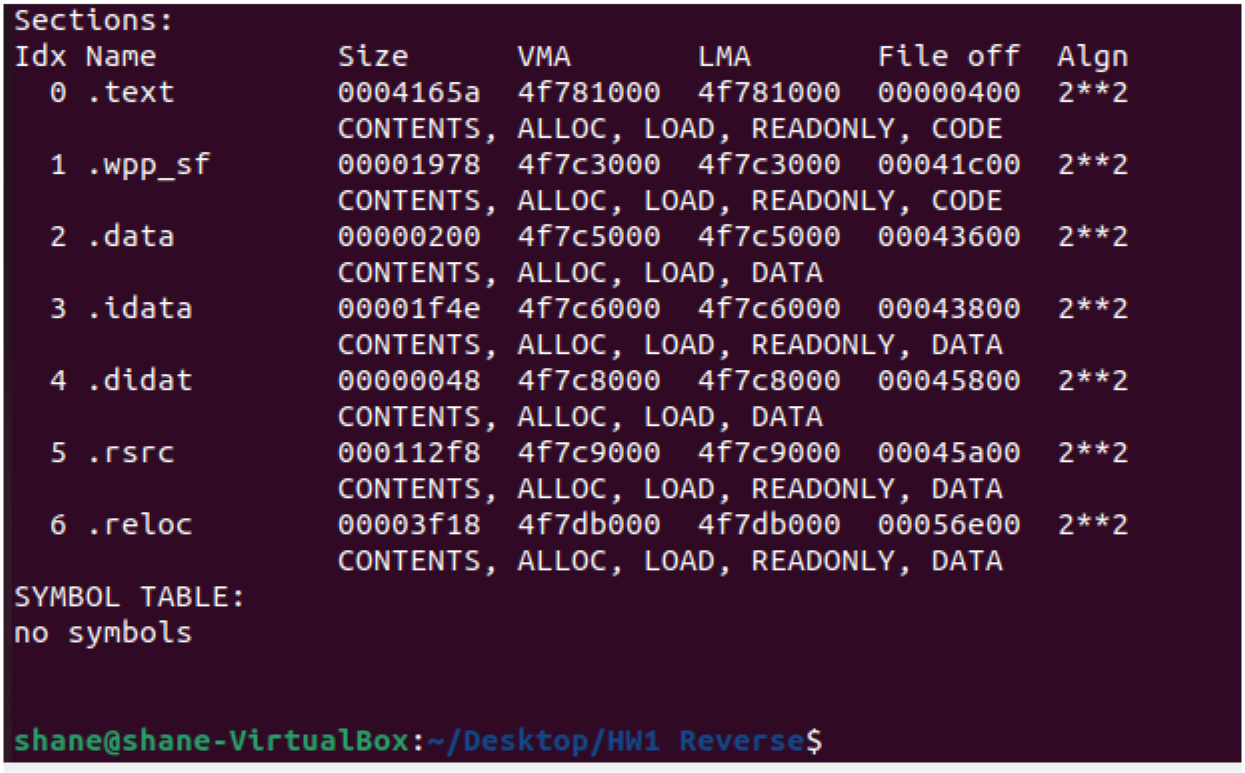
Ws2_32.dll:



```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ ls
cmd.exe  file  kernel32.dll  magic.mgc  ws2_32.dll
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file ws2_32.dll
ws2_32.dll: PE32 executable (DLL) (console) Intel 80386, for MS Windows
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file -i ws2_32.dll
ws2_32.dll: application/x-dosexec; charset=binary
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ cat -n ws2_32.dll | awk '{print "
Longest line number: " $1 " Length with line number: " length}' | sort -k4 -nr
| head -3
Longest line number: 612 Length with line number: 82
Longest line number: 611 Length with line number: 14
Longest line number: 610 Length with line number: 180
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$
```



```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ objdump -x ws2_32.dll

ws2_32.dll:     file format pei-i386
ws2_32.dll
architecture: i386, flags 0x0000012f:
HAS_RELOC, EXEC_P, HAS_LINENO, HAS_DEBUG, HAS_LOCALS, D_PAGED
start address 0x4f795c70

Characteristics 0x2102
        executable
        32 bit words
        DLL

Time/Date               8928d458        (This is a reproducible build file hash
, not a timestamp)
Magic                   010b    (PE32)
MajorLinkerVersion      14
MinorLinkerVersion      13
SizeOfCode              00043200
SizeOfInitializedData   00017e00
SizeOfUninitializedData 00000000
AddressOfEntryPoint     00015c70
```

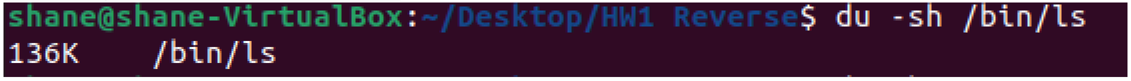


```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0004165a  4f781000  4f781000  00000400  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .wpp_sf       00001978  4f7c3000  4f7c3000  00041c00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .data         00000200  4f7c5000  4f7c5000  00043600  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .idata        00001f4e  4f7c6000  4f7c6000  00043800  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .didat        00000048  4f7c8000  4f7c8000  00045800  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .rsrc         000112f8  4f7c9000  4f7c9000  00045a00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .reloc        00003f18  4f7db000  4f7db000  00056e00  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
SYMBOL TABLE:
no symbols


shane@shane-VirtualBox:~/Desktop/HW1 Reverse$
```

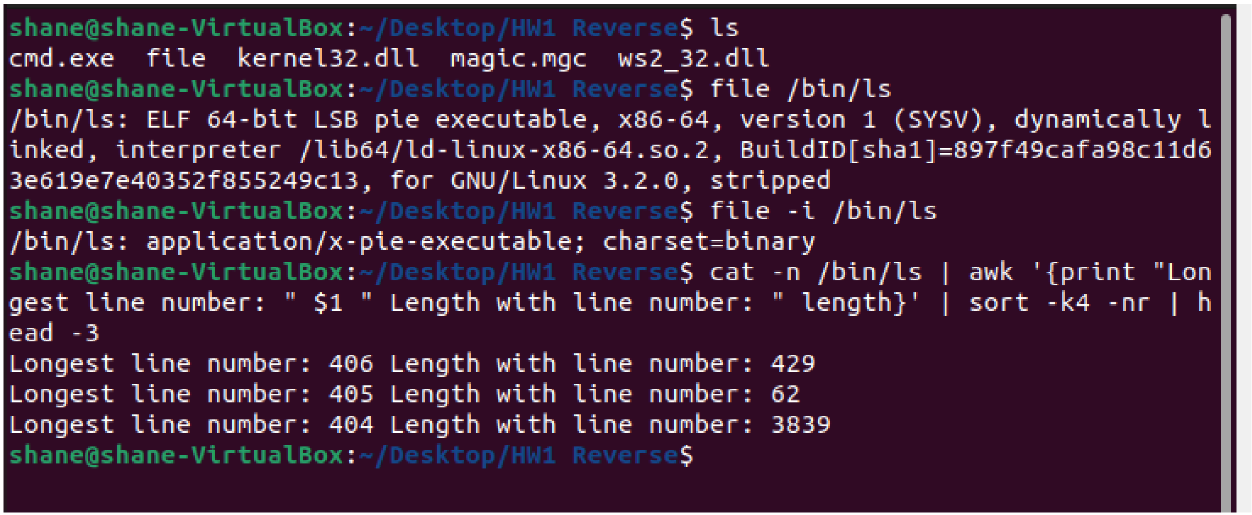
Here, the information regarding file ws2_32.dll is displayed. This dll file was a bit simpler to work with once I discovered the objdump command. The '-x' attribute in the objdump command gives me most of the information I was searching for, as displayed in the screenshots.

/bin/ls:

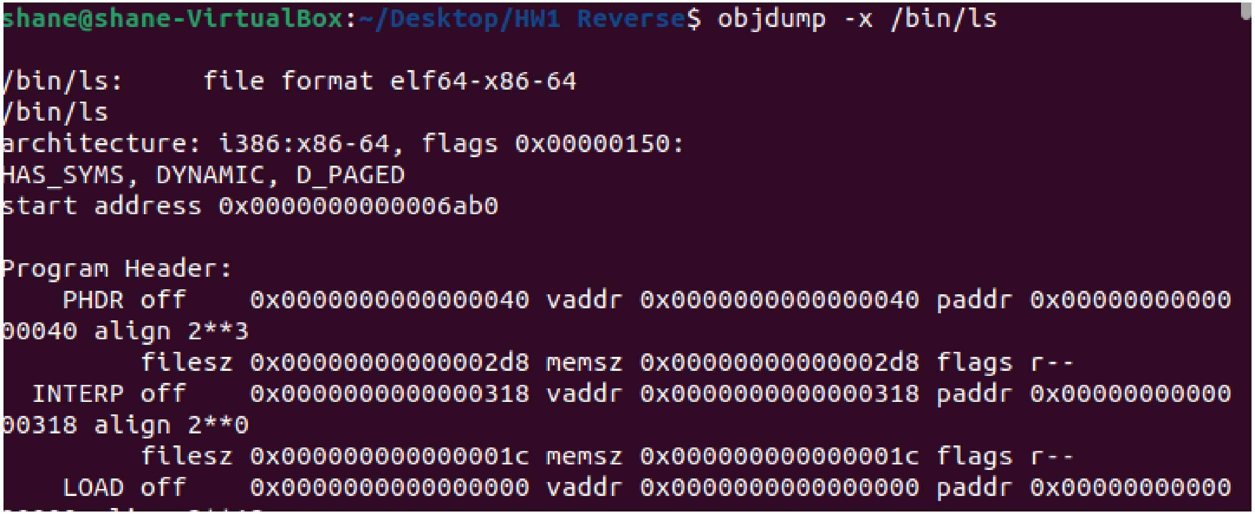
/bin/ls is the binary file that is used for the 'ls' command in linux terminals. It is in the /bin directory. It has a size of 136k.



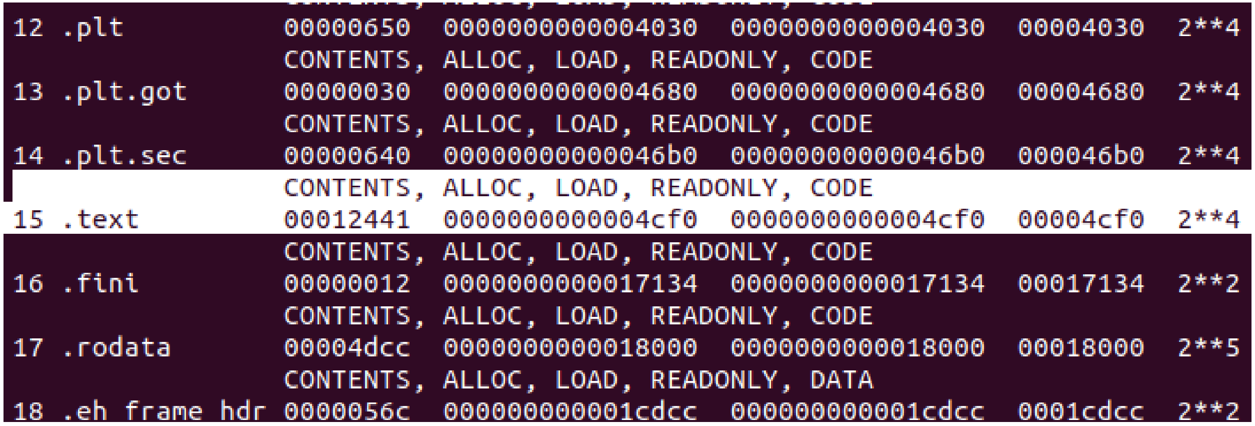
```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ du -sh /bin/ls
136K    /bin/ls
```



```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ ls
cmd.exe  file  kernel32.dll  magic.mgc  ws2_32.dll
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file /bin/ls
/bin/ls: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically l
inked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=897f49cafa98c11d6
3e619e7e40352f855249c13, for GNU/Linux 3.2.0, stripped
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ file -i /bin/ls
/bin/ls: application/x-pie-executable; charset=binary
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ cat -n /bin/ls | awk '{print "Lon
gest line number: " $1 " Length with line number: " length}' | sort -k4 -nr | h
ead -3
Longest line number: 406 Length with line number: 429
Longest line number: 405 Length with line number: 62
Longest line number: 404 Length with line number: 3839
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$
```



```
shane@shane-VirtualBox:~/Desktop/HW1 Reverse$ objdump -x /bin/ls

/bin/ls:     file format elf64-x86-64
/bin/ls
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x0000000000006ab0

Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000000040 paddr 0x00000000000
00040 align 2**3
         filesz 0x00000000000002d8 memsz 0x00000000000002d8 flags r--
  INTERP off    0x0000000000000318 vaddr 0x0000000000000318 paddr 0x00000000000
00318 align 2**0
         filesz 0x000000000000001c memsz 0x000000000000001c flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000000000 paddr 0x00000000000
```



```
 12 .plt          00000650  0000000000004030  0000000000004030  00004030  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .plt.got      00000030  0000000000004680  0000000000004680  00004680  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .plt.sec      00000640  00000000000046b0  00000000000046b0  000046b0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .text         00012441  0000000000004cf0  0000000000004cf0  00004cf0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 16 .fini         00000012  0000000000017134  0000000000017134  00017134  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 17 .rodata       00004dcc  0000000000018000  0000000000018000  00018000  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 18 .eh_frame_hdr 0000056c  000000000001cdcc  000000000001cdcc  0001cdcc  2**2
```

The above information in the screenshots is regarding the /bin/ls executable code. It is similar to the ‘file’ file in that it is in ELF format and can be viewed using readelf and other ‘elf’ commands. The magic number is 7F454C4602010100, the same as ‘file’ since they are both ELF.

Question 2 (10 points) Summarize in your own words the linear sweep disassembly algorithm and recursive
descent disassembly algorithm. Then for each algorithm, give two different scenarios where it will fail to produce
the correct assembly.

A. Linear Sweep Disassembly
Until an illegal instruction is found, the algorithm starts with the first byte in the text (code) section and decodes each byte.

   a. 0x0F 0x85 0xC0…: 0x0F may be considered as a code byte with disassembly 'jne offset' whereas if it is considered a data byte and the following 0x85 is determined to be a code byte then the disassembly error can propagate and subsequent bytes may be interpreted incorrectly as well (https://www.usenix.org/legacy/publications/library/proceedings/usenix03/tech/full_papers/prasad/prasad_html/node5.html#:~:text=There%20are%20two%20main%20classes,an%20illegal%20instruction%20is%20encountered.)

   b. Additional 'push eax' instructions may bring errors since, when the function returns, EIP and ESP cannot be properly restored due to the difference in values (https://resources.infosecinstitute.com/topic/linear-sweep-vs-recursive-disassembling-algorithm/). Due to the jumps and how the algorithm moves linearly, it may use an instruction incorrectly and disassemble the subsequent instructions incorrectly as well.

B. Recursive Descent Disassembly
Starting again at the main entry point of the program code, the algorithm proceeds linearly until a jump/branch instruction is found. Once this instruction is found, it is followed in depth-first or breadth-first and begin the linear sweep again.

   a. 'mov eax, 2' followed by 'cmp eax, 2' is an instruction that moves the value 2 into the eax register then compares it with the value 2 (obviously a true = true statement) and forcing the je instruction to be evaluated as true, always allowing a jump (https://resources.infosecinstitute.com/topic/linear-sweep-vs-recursive-disassembling-algorithm/). This specific trick can fool some recursive disassemblers like WinDbg in which it becomes confused and does not use the DB statement as it should following the je instruction.

   b. Indirect branch instructions ('call eax' and 'jmp dword[esp + xx]') can make it difficult to construct an accurate control flow of the binary Callback functions and/or exception handlers can also be tricky to recursively disassemble in some cases since it may have no explicit call site. (https://www.usenix.org/legacy/publications/library/proceedings/usenix03/tech/full_papers/prasad/prasad_html/node5.html#:~:text=There%20are%20two%20main%20classes,an%20illegal%20instruction%20is%20encountered.)

Question 3 (15 points) Explain the general problems (i.e., how to pass parameters, how to call the function and
return to the caller, how to use and share the registers, how to use and share the stack, and how to return value(s))
any function calling convention must handle. Then explain how each of the following calling conventions
addresses the problems.

General problems calling conventions must handle may include ordering of parameters and their allocation, how to pass parameters either pushing onto a stack or placing into registers, etc., who cleans up the stack, retaining of values after subroutine calls, and how values are returned. These problems are general for all standardized conventions and are addressed in one way or another.

1) The cdecl calling convention

This convention has the caller clean the arguments from the stack and each function call should have stack cleanup code. It is used by C compilers for x86. Subroutine arguments are passed onto the stack while integer values & memory addresses are returned in the EAX register. EAX, ECX, and EDX registers are caller saved. It passes arguments right to left, and the calling function pops the argument from the stack.

2) The stdcall calling convention

This convention has the callee clean the stack and a function using this call convention should provide a function prototype. Parameters are pushed onto the stack right to left. Registers EAX, ECX, and EDX are used for this convention and the return values are stored in EAX. Called functions pop their own arguments from the stack.

3) The fastcall calling convention

This convention passes the first two DWORD and smaller arguments from left to right to the ECX and EDX registers. The subsequent arguments are passed onto the stack right to left. The called function will pop the arguments from the stack, and the callee performs stack cleanup.

4) The thiscall calling convention

This convention is flexible as it can utilize either caller or callee cleanup. In the case of caller cleanup, this convention becomes very similar to cdecl, where the parameters are passed right to left. The addition of the ‘this’ pointer is pushed onto the stack last acting like a function prototype. Using other compilers, the thiscall ‘this’ pointer will be passed in ECX register and the callee will clean the stack. This makes the convention show similarities to the stdcall convention.

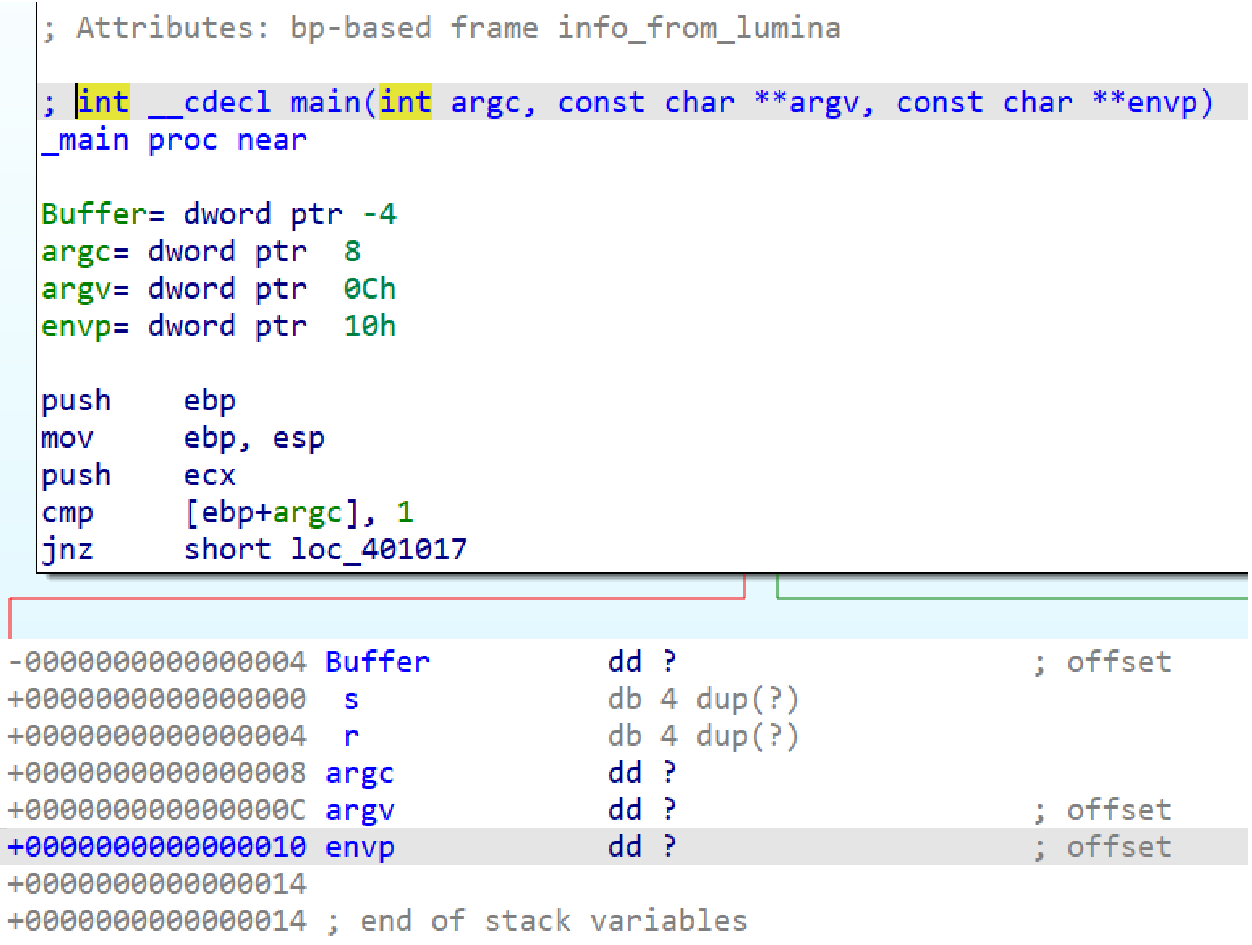
https://en.wikipedia.org/wiki/X86_calling_conventions

https://docs.microsoft.com/en-us/cpp/cpp/calling-conventions?view=msvc-170

The following questions (4 - 6) require the following binary file that can be found at http://www.cs.fsu.edu/~liux/courses/reversing/assignments/reverse_homework_exe. There are six phrases and an additional hidden phase, where each phase requires an input string to go on to the next phase or finish the program successfully. As this file has been updated recently, make sure that you download the current version using the link. This program runs on a Windows machine only; you can run it on the Windows machines in the Majors Lab or install a Windows virtual machine if you do not have a Windows one already.

Question 4 (15 points) Load the binary file into GHIDRA/IDA. Answer the following questions:

1) Analyze how the stack and parameters are used by function _main (starting at address 0x00401000). Using the prototype "int __cdecl main(int argc, const char **argv, const char **envp)", explain how you would access envp[5][4]. You can only use the values in the registers already defined in function _main. (Hint: note that envp is the third parameter to main; also make sure that you understand the difference between an address and the value in it.)

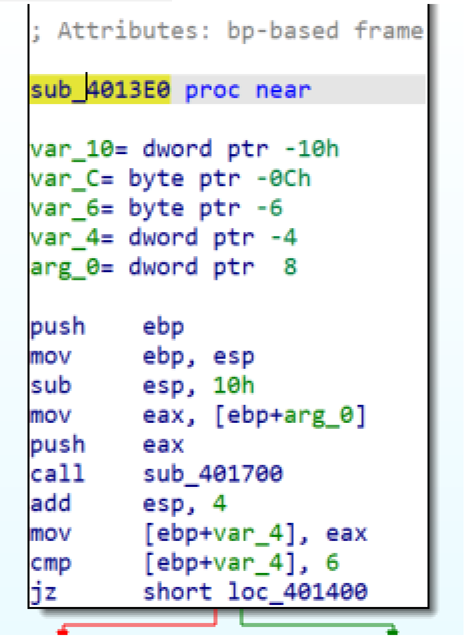


```
; Attributes: bp-based frame info_from_lumina

; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

Buffer= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
push    ecx
cmp     [ebp+argc], 1
jnz     short loc_401017
```

```
-0000000000000004 Buffer          dd ?                    ; offset
+0000000000000000  s              db 4 dup(?)
+0000000000000004  r              db 4 dup(?)
+0000000000000008 argc            dd ?
+000000000000000C argv            dd ?                    ; offset
+0000000000000010 envp            dd ?                    ; offset
+0000000000000014
+0000000000000014 ; end of stack variables
```

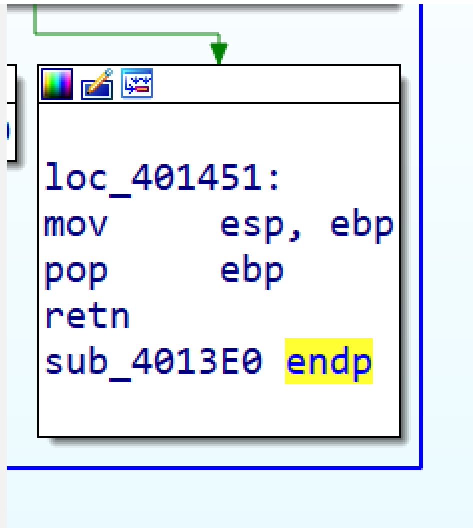
The main function uses three parameters: argc, argv, and envp. These each are given their offsets 08, 0Ch, and 10h respectively. Ebp is first pushed onto the stack, where then esp is pushed onto ebp and ecx is further pushed above that. Ebp and argc is then compared to 1 and if the outcome is not zero (jnz), the jump is performed to the location specified (short loc_401017). It appears that accessing envp is adding an offset of 0000000000000010 when pointing on the stack. Similar to the jack king and queen example recently in Offensive Computer Security, a command line input that targets the offset at 10h should be able to access this envp parameter. This command should be a vulnerable one, and would have to be written in the same endian as the stack.

2) Analyze how the stack, parameters, and local variables are used by the function starting at address 0x004013E0. Then explain the calling convention and the prototype of the function.



```
; Attributes: bp-based frame

sub_4013E0 proc near

var_10= dword ptr -10h
var_C= byte ptr -0Ch
var_6= byte ptr -6
var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 10h
mov     eax, [ebp+arg_0]
push    eax
call    sub_401700
add     esp, 4
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 6
jz      short loc_401400
```

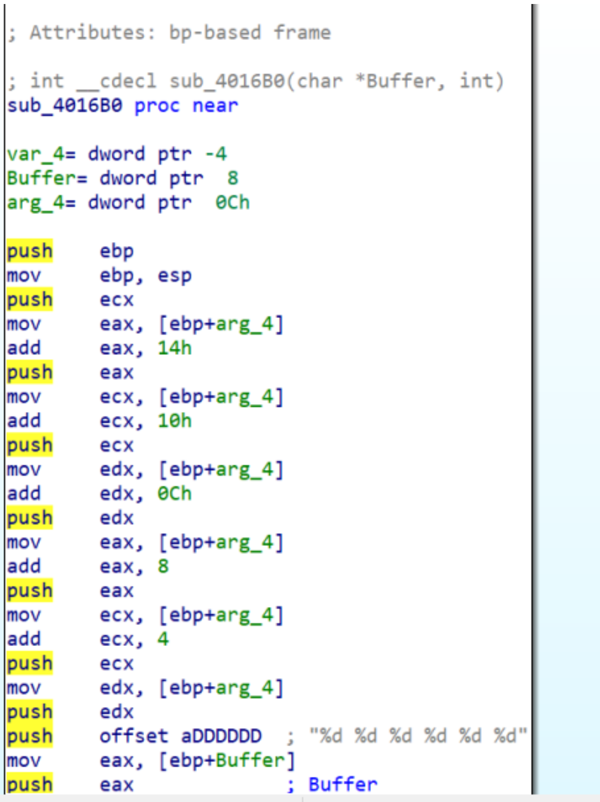
Here is the snippet of the function at address 0x004013E0. Multiple variables and an arg value are provided locally with offsets. Firstly, the stack pushes ebp then moves esp to ebp. Esp is subtracted from 10h (which should be at var_10 or close to it) and ebp+arg_0 is moves to eax. Eax is pushed onto the stack and sub_401700 is called. After the call, 4 is added to esp, and eax is moved to ebp+var_4. 6 is compared to ebp+var_4. Jz (jump if zero) is then commenced (if zero) and jumps the function to the address (short loc_401400). If the comparison is not zero, sub_401960 is called. I feel this is cdecl calling convention because it cannot be fastcall (due to the fact that the registers are getting values assigned to them) and for it to be stdcall, the callee should be cleaning the stack. Values are being assigned to the registers too, so this leads me to believe it is cdecl convention. At the end of the control flow, this snippet is found:



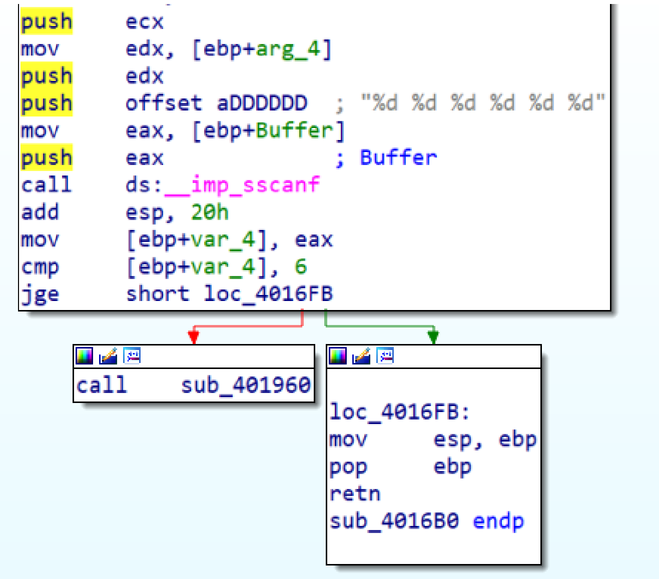
```
loc_401451:
mov     esp, ebp
pop     ebp
retn
sub_4013E0 endp
```

From my understandings, stdcall ‘ret’ should be followed by a parameter or address/pointer where here, it is not.

3) Illustrate how the stack is used by the function starting at address 0x 004016B0. Based on your analysis, give the correct prototype (including the return value type, the calling convention, the parameters and their types) and a brief explanation. (Hint: the prototype “int __cdecl sub_4016B0(char *Src, int)” given by IDA Pro is incorrect)



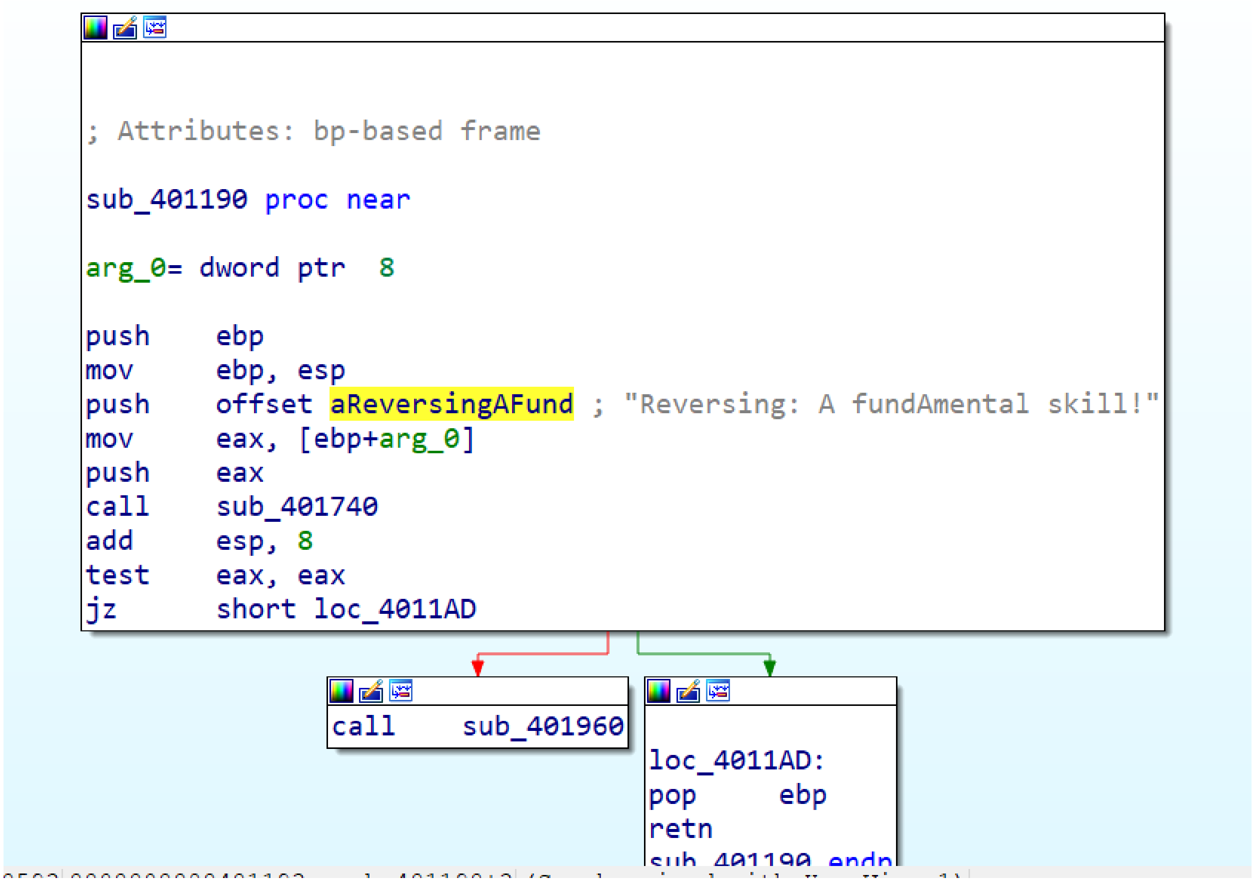
```
; Attributes: bp-based frame

; int __cdecl sub_4016B0(char *Buffer, int)
sub_4016B0 proc near

var_4= dword ptr -4
Buffer= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
push    ecx
mov     eax, [ebp+arg_4]
add     eax, 14h
push    eax
mov     ecx, [ebp+arg_4]
add     ecx, 10h
push    ecx
mov     edx, [ebp+arg_4]
add     edx, 0Ch
push    edx
mov     eax, [ebp+arg_4]
add     eax, 8
push    eax
mov     ecx, [ebp+arg_4]
add     ecx, 4
push    ecx
mov     edx, [ebp+arg_4]
push    edx
push    offset aDDDDDD  ; "%d %d %d %d %d %d"
mov     eax, [ebp+Buffer]
push    eax             ; Buffer
```



```
push    ecx
mov     edx, [ebp+arg_4]
push    edx
push    offset aDDDDDD  ; "%d %d %d %d %d %d"
mov     eax, [ebp+Buffer]
push    eax             ; Buffer
call    ds:__imp_sscanf
add     esp, 20h
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 6
jge     short loc_4016FB

call    sub_401960

loc_4016FB:
mov     esp, ebp
pop     ebp
retn
sub_4016B0 endp
```

The return value is a near return seen at the end of the function. The calling convention appears to be cdecl again because values are being assigned (mov) onto registers at eax, ecx, and edx. The parameters in this function include var_4, Buffer, and arg_4 which are ‘dword ptr -4’, ‘dword ptr 8’, and ‘dword ptr 0Ch’ respectively. After manipulating the stack, the fundction compares 6 to [ebp+var_4]. Following this, a ‘jge’ instruction is given (signed cmp jump after cmp). As seen in the screenshot, if the condition is met, the function jumps to loc_4016FB where ebp is moved to esp, popped, and returned. If the condition is not met, sub_401960 is called (this “blows up the bomb”).

Question 5 (20 points) Find the input for the binary file so that it will pass Phase 1 using only static analysis. Give an explanation with screenshots of your analysis process in addition to the solved password.



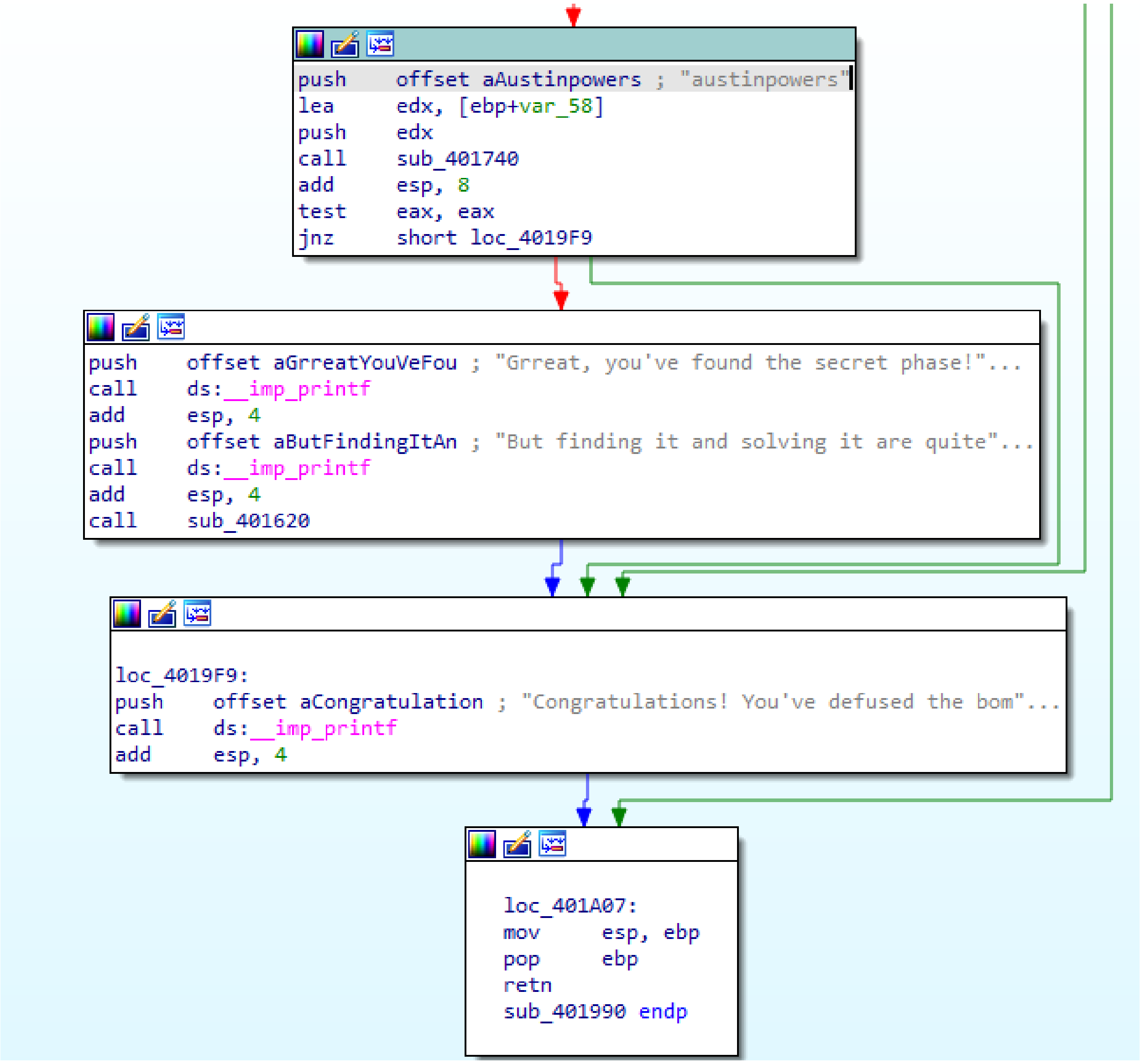
```
; Attributes: bp-based frame

sub_401190 proc near

arg_0= dword ptr  8

push    ebp
mov     ebp, esp
push    offset aReversingAFund ; "Reversing: A fundAmental skill!"
mov     eax, [ebp+arg_0]
push    eax
call    sub_401740
add     esp, 8
test    eax, eax
jz      short loc_4011AD
```

```
call    sub_401960
```

```
loc_4011AD:
pop     ebp
retn
sub_401190 endp
```

After scouring the disassembly, I believe this is the input for the binary file to pass Phase 1. ‘Reversing: A fundAmental skill!’. This is the only phrase I have found aside from ‘**austinpowers’**, however, the austin phrase leads me to believe it is a secret phrase and not the actual key to Phase 1. In the above screenshot, the reversing phrase input leads to two different outcomes, one where the bomb may blow up, and one where it presumably doesn’t and awaits further input. When looking at the control flow of this program, this phrase is placed above the phrase which I believe unlocks phase 2, so that is why I believe it is the input for phase 1.

Here is the austinpowers phrase mentioned above:



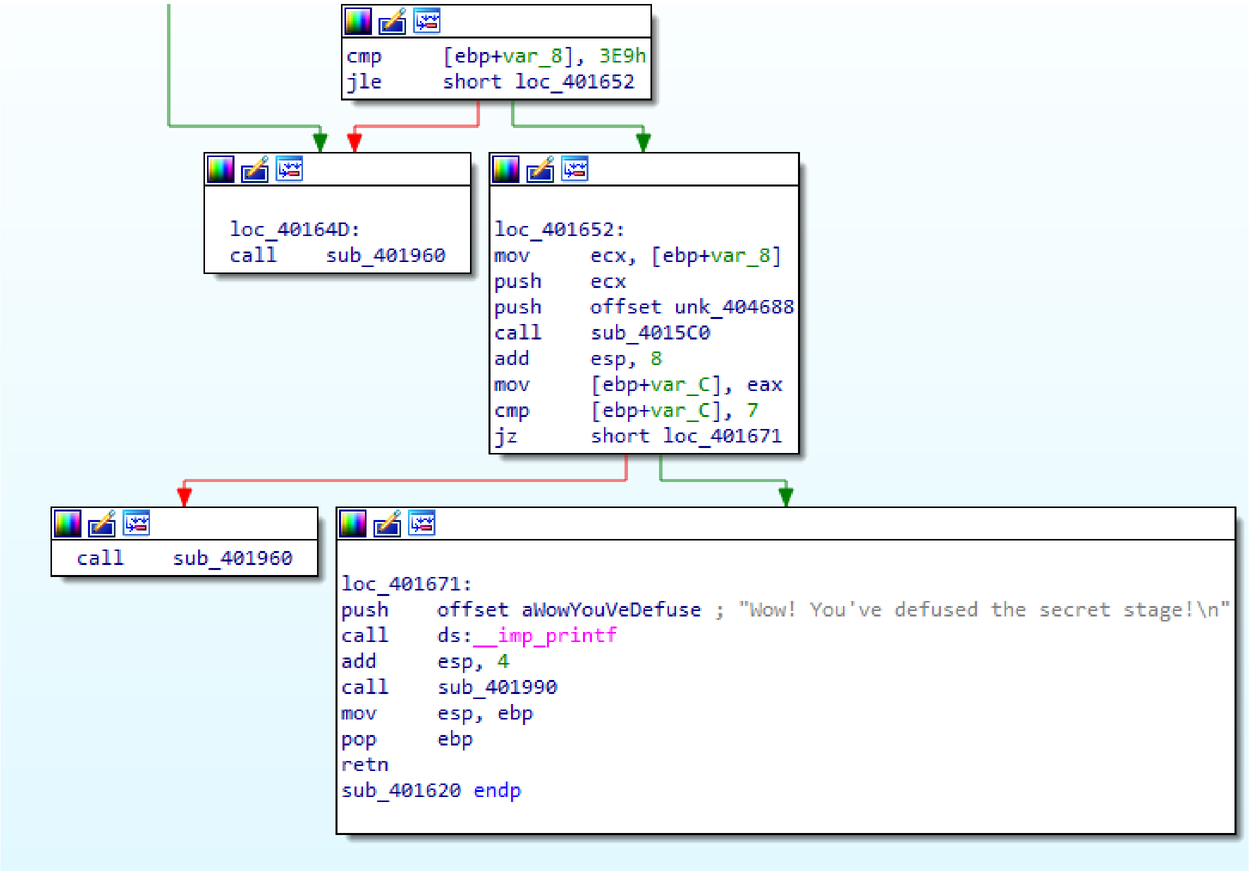
```
push    offset aAustinpowers ; "austinpowers"
lea     edx, [ebp+var_58]
push    edx
call    sub_401740
add     esp, 8
test    eax, eax
jnz     short loc_4019F9
```

```
push    offset aGrreatYouVeFou ; "Grreat, you've found the secret phase!"...
call    ds:__imp_printf
add     esp, 4
push    offset aButFindingItAn ; "But finding it and solving it are quite"...
call    ds:__imp_printf
add     esp, 4
call    sub_401620
```

```

loc_4019F9:
push    offset aCongratulation ; "Congratulations! You've defused the bom"...
call    ds:__imp_printf
add     esp, 4
```

```

loc_401A07:
mov     esp, ebp
pop     ebp
retn
sub_401990 endp
```

This phrase sends us here which ends with:



```
cmp     [ebp+var_8], 3E9h
jle     short loc_401652
```

```
loc_40164D:
call    sub_401960
```

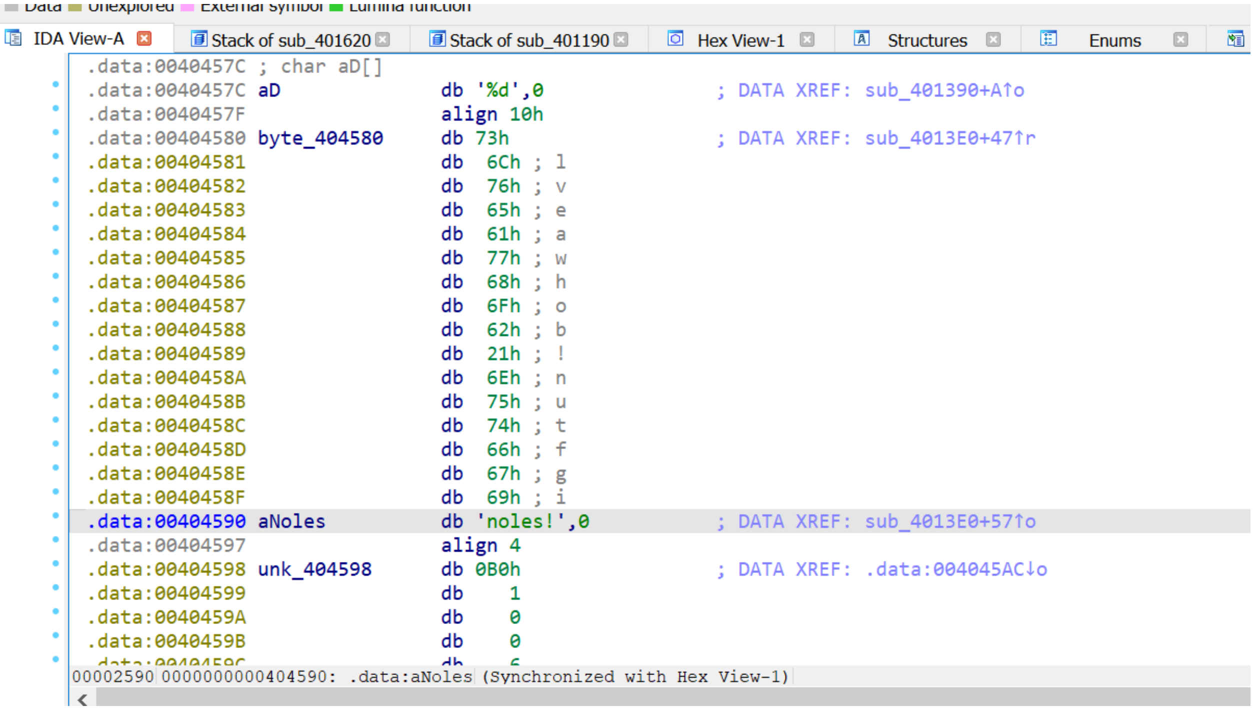
```
loc_401652:
mov     ecx, [ebp+var_8]
push    ecx
push    offset unk_404688
call    sub_4015C0
add     esp, 8
mov     [ebp+var_C], eax
cmp     [ebp+var_C], 7
jz      short loc_401671
```

```
call    sub_401960
```

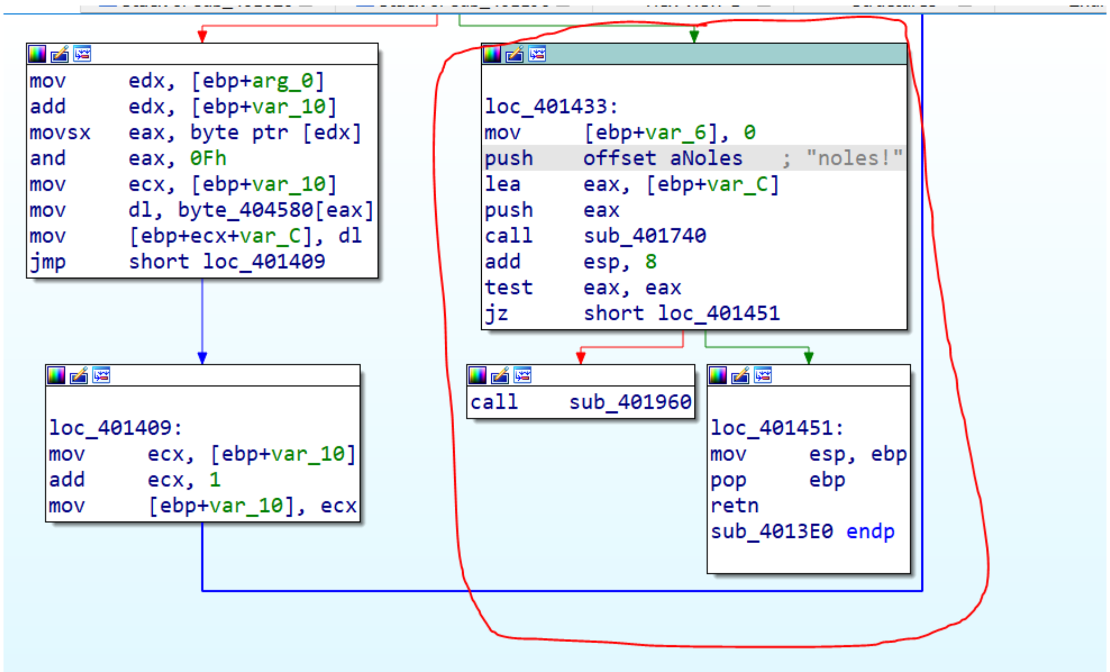
```
loc_401671:
push    offset aWowYouVeDefuse ; "Wow! You've defused the secret stage!\n"
call    ds:__imp_printf
add     esp, 4
call    sub_401990
mov     esp, ebp
pop     ebp
retn
sub_401620 endp
```

This appears to be a secret stage. The user has the chance to defuse the secret stage here, however, a wrong input can call 401960, which detonates the bomb (similar to the other phases).

Question 6 (30 points) Find the input for the binary file so that it will pass Phase 2 using only static analysis. Incorporate Graph View, Function Renaming, and Parameter Renaming in your analysis. Give an explanation with screenshots of your analysis process in addition to the solved password.



```
.data:0040457C ; char aD[]
.data:0040457C aD              db '%d',0               ; DATA XREF: sub_401390+A↑o
.data:0040457F                 align 10h
.data:00404580 byte_404580     db 73h                  ; DATA XREF: sub_4013E0+47↑r
.data:00404581                 db  6Ch ; l
.data:00404582                 db  76h ; v
.data:00404583                 db  65h ; e
.data:00404584                 db  61h ; a
.data:00404585                 db  77h ; w
.data:00404586                 db  68h ; h
.data:00404587                 db  6Fh ; o
.data:00404588                 db  62h ; b
.data:00404589                 db  21h ; !
.data:0040458A                 db  6Eh ; n
.data:0040458B                 db  75h ; u
.data:0040458C                 db  74h ; t
.data:0040458D                 db  66h ; f
.data:0040458E                 db  67h ; g
.data:0040458F                 db  69h ; i
.data:00404590 aNoles          db 'noles!',0           ; DATA XREF: sub_4013E0+57↑o
.data:00404597                 align 4
.data:00404598 unk_404598      db 0B0h                 ; DATA XREF: .data:004045AC↓o
.data:00404599                 db    1
.data:0040459A                 db    0
.data:0040459B                 db    0
.data:0040459C                 db    6
00002590 0000000000404590: .data:aNoles (Synchronized with Hex View-1)
```



```
mov     edx, [ebp+arg_0]
add     edx, [ebp+var_10]
movsx   eax, byte ptr [edx]
and     eax, 0Fh
mov     ecx, [ebp+var_10]
mov     dl, byte_404580[eax]
mov     [ebp+ecx+var_C], dl
jmp     short loc_401409
```

```
loc_401409:
mov     ecx, [ebp+var_10]
add     ecx, 1
mov     [ebp+var_10], ecx
```

```
loc_401433:
mov     [ebp+var_6], 0
push    offset aNoles   ; "noles!"
lea     eax, [ebp+var_C]
push    eax
call    sub_401740
add     esp, 8
test    eax, eax
jz      short loc_401451
```

```
call    sub_401960
```

```
loc_401451:
mov     esp, ebp
pop     ebp
retn
sub_4013E0 endp
```

After looking through the program, I believe ‘noles!’ is the input for the binary file that passes phase 2. This is because it was found after the phase 1 phrase and leads to outcomes involving the bomb detonating and the function returning depending on the test. Since in the disassembler the phase 1 phrase was just before this one, I am led to believe this is the phrase that unlocks phase two. This took some digging as not many values pointed to this phrase. However, as stated above, this phrase ‘noles!’ does appear after the input I believe to be the key to phase one, so I believe this is the key to phase 2. Once again, I found the **‘austinpowers’** phrase, but this, to my understanding, is a secret phase. In the first screenshot, there is also a long list of single characters that suspiciously looks like a scrambled passcode; however, I am unsure if it is related since, although there is an XREF, it points to the same structure where the input ‘noles!’ is appearing.