

# Lab 04 – Simple SEH Overwrite

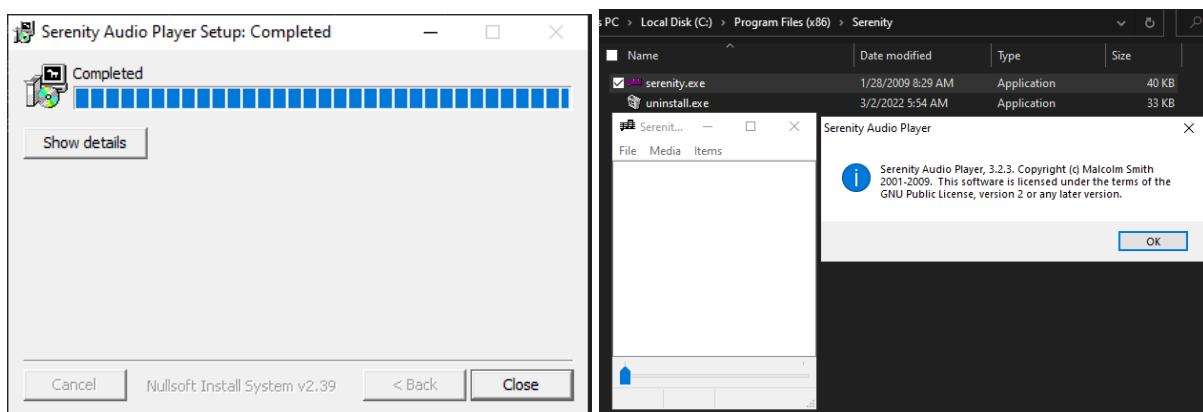
---

Total Points: 18

## Task 1 - Audio:

This is warm up and it corresponds to what we did in our class videos, so try to follow along and then take it a step further.

- Install the software



## Part 1:

- Complete the following in-class activity:
  - Create a working SEH script. **Include the script for points.** Be sure it can run in the Windows 7 IA lab environment. It is fine if you develop it elsewhere, but you may need to adapt it so that, the script can be ran when graded.
  - Do one that utilizes *pop pop ret* **or** one that utilizes *add esp, 8* for the SEH overwrite.
  - Get some shellcode to run to pop a calculator; do not use malicious shellcode at this point.
  - Obtain two screenshots:
    - The first should show the results of !exchain to show your SEH and NSEH overwrites.
    - The second should show your successfully executing shellcode.
  - Hint: Your form of input will be a malicious .m3u file that is loaded into the target application.

```

Command
*** wait with pending attach
Symbol search path is: srv*:Symbols*http://msdl.microsoft.com/d
Executable search path is:
ModLoad: 00400000 00400000 C:\Program Files (x86)\Serenity\seren
(c8b:c5c): Break instruction exception - code 00000003 (first char
eax=7efda000 ebx=00000000 ecx=00000000 edx=793f85a esi=00000000
esp=0040255 esp=002cff5c ebp=002cf8f8 iopl=0 nv up ei pl
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b
ntdll!DbgBreakPoint:
77864015 cc int 3
0:0013 g
(0x498): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00190000 ebx=0018f718 ecx=0018f743 edx=00000008 esi=0018f748
esp=756bc9d7 esp=0018f640 ebp=0018f654 iopl=0 nv up ei pl
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b
msvcrt!_ftbuf+0x1bf:
756bc9d7 8888 mov byte ptr [eax],cl ds:002b:00190000=41
0:0000 lexpath
0:0018fe8c: msvcrt!_except_handler4+0 (75668cd5)
0:0018fe88: 43434343
Invalid exception stack at 42424242

```

Figure 1: Hit EIP with 4-bytes of B's

```

Disassembly
Offset: @$scopeip
0018fe30 41 inc    ecx
0018fe31 41 inc    ecx
0018fe32 41 inc    ecx
0018fe33 41 inc    ecx
0018fe34 41 inc    ecx
0018fe35 41 inc    ecx
0018fe36 41 inc    ecx
0018fe37 41 inc    ecx
0018fe38 eb06 jmp   0018fe40
0018fe3a 90 nop
0018fe3b 90 nop
0018fe3c ff2c40 jmp   fword ptr [eax+eax*2]
0018fe3f 000909090909 add   byte ptr [eax-6F6F6F70h],dl
0018fe45 90 nop
0018fe46 90 nop
0018fe47 90 nop

```

Figure 3: Buffer of A's then jump to the 30 nops from my script

```

(d1.e50): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000041 ebx=41414141 ecx=0018fa40 edx=0018f520 esi=41414141 edi=756b5889
esp=00404255 esp=0018f910 ebp=0018fb64 iopl=0 nv up ei nz na pe nc
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b
serenity+0x4255:
00404255 8b460c mov eax,dword ptr [esi+0Ch] ds:002b:4141414d=???????
0:0000> lexpath
0018fe38: serenity+2cff (00402cff)
0018fe38: Invalid exception stack at 909006eb

```

Figure 2: I used SEH 00402cff in my script shown above after short jmp 6.

Reg	Value
eax	1
ecx	752795c1
edi	4
ebx	0
esp	18ef90
ebp	#7
esi	75313ca6
edi	75261072
eip	18fece
cf	0
pt	1

Figure 4: exploit script pops calc.exe

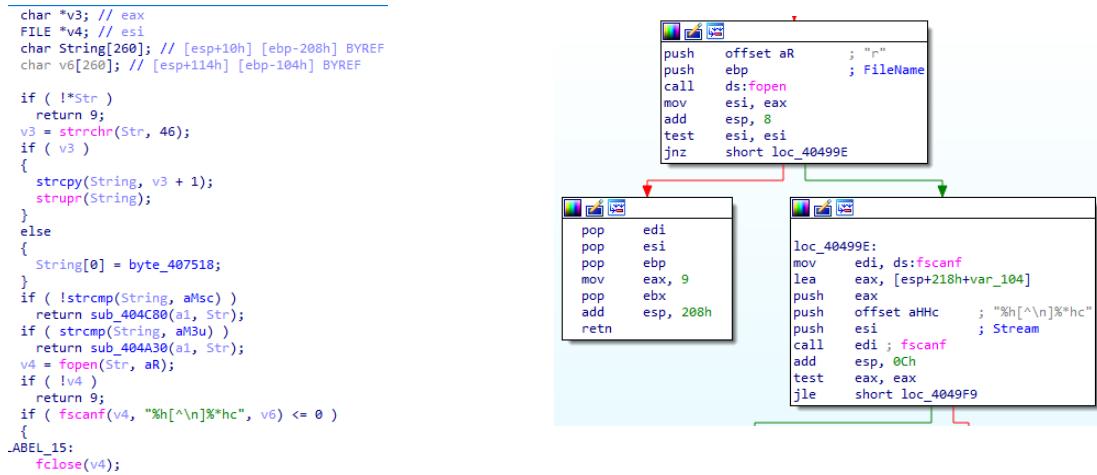
## Part 2:

- What is the cause of the vulnerability? Trace it and find the vulnerable location in code that is exploitable.
  - Explain fully and utilize screenshots.
  - Be sure to trim your screenshots—no screenshots of your entire desktop please.

### Answer:

From NIST: CVE-2009-4097: Stack-based buffer overflow in the MplayInputFile function in Serenity Audio Player 3.2.3 and earlier allows remote attackers to execute arbitrary code via a long URL in an M3U file.

Knowing I am able to overflow it with the file input, I looked for API calls to things related to file read. I came across fscanf() a known vulnerable function. Using IDAPro 7.6 I found calls to this function and looked at cross references (there's only 1). I looked at the function that used fscanf() and it was reading in a file as suspected. I used IDAPro's disassembler and it was quite easy to see. Shortly after this there is a call to GlobalFree where there occurs a memory access situation.



The file pointer `*v4` is what the stream is read into as you can see in the `fscanf()` and it's put into the character array `v6[260]`. From the MSDN `fscanf(pointer (v4), control string, location (v6))`. This is where the overflow occurs.

### Task 2 – Chat Server:

The goal of this task is to manually fuzz and exploit the target binary. Obtain an SEH overwrite and get some shellcode to run. The target lab environment is Windows 7. **Your script should be runnable as is in the IA lab environment, so if you work elsewhere, be sure to adapt it as need be to work in target environment.**

In order to accomplish this, you may wish to interact with the software and observe its normal behavior. You may even try to fuzz by copying and pasting into the console. Eventually, you will want to take a scripted approach, utilizing a Python or Perl script, to interact with the binary. (Note: You can also interact with the binary with just a Python interpreter, sending some of the same commands you might in a script. That could be part of an initial approach of interacting with it.) If you wanted to, you could capture traffic via Wireshark or similar tool and look at what if any commands go across the wire, although for this binary, it is simple enough you may not need to. Make sure to attach your debugger of choice to the binary, so you can capture exceptions and investigate. No debugger = no results.

In WinDbg, after you get an exception that you think is from your malicious input, run the command `!exchain` to view the SEH handler and NSEH, as these are what you will need to overwrite. Craft your input to overwrite the SEH and NSEH. Once you have overwritten these, replace them with real values, and then use a nop sled and ride it down to your shellcode. As to the shellcode, you are left to your own devices. At this stage, anything is acceptable, so it could be a benign shellcode or something generated by msfvenom (Metasploit). Note: you can search the internet for shellcode for Windows 7. You are not graded on what your shellcode does, so long as it does something. Something benign like popping a calculator is acceptable. **Note: If your shellcode is malicious, please identify it as such.**

This application is inherently flawed, so it also does not run as correctly as it could. Focus on trying to cause the SEH overwrite.

To complete the lab, provide the following tasks.

**Part 1:**

- Provide a narrative of how your exploit worked (limited to a maximum of one page);

**Answer:**

The exploit worked by overflowing a buffer in the vulnerable server.exe code. It was accomplished by running the server and fuzzing it through a socket connection on the listening port. I used A's, B's, and C's for the initial payload. I set B's to 4 bytes and C's to 1000 and just changed the A's using intuition to guess where it should be. It took me ~8-10 times of changing the A values before hitting the 4 bytes of B's exact. Then using some of my previous code from question 1 I set the NSEH to short jmp 6 or \xeb\x06\x90\x90. This is the pointer to the next SEH record and then used mona in windbg to find a valid pop pop ret I could use. The SHE gets triggered with the A's and you need to overwrite both parts (next record and pointer to current records exception handler) of the SEH record. When sending all this at the same time you are overwriting the SEH (ESP+8) with the short jmp 6, the pop pop ret. The pop pop ret pops 8 bytes off the stack and returns execution to the top of the stack which leaves the pointer to the next SHE at the top of the stack. This happens to be where the shellcode, nor nops then shellcode, sits you can ride in on your shellcode which in my case was calc.exe.

- Identify the gadget you used in your SEH overwrite. For instance, if this is pop pop ret or add esp, 0x8, then label the memory address for the gadget that does this.

**Answer:**

I used the pop pop ret at address 0040FE76 which was the first one that mona.py returned.

```
- Number of pointers of type 'pop ecx # pop ebp # ret 0x0c' : 1
[+] Results :
0x0040fe76 | 0x0040fe76 : pop eax # pop ebp # ret 0x1C | startnull {PAGE_EXECUTE_READ} [server.exe] ASLR: False, Rebase: False, SafeSEH: False
```

- Describe the process of finding gadgets to use for your SEH overwrite in the tools you choose to use. (You are taught to use some software, but if you know others, you are free to work with the tools of your choosing.) For instance, would you use WinDbg and Mona—what would you do with them, and what would your workflow be?

**Answer:**

I used mona.py with windbg as shown in the class but I've used immunity debugger before with mona in Offsec OSCP but want to get better with windbg so I stuck with that. After fuzzing and once I controlled the 4 bytes in EIP I ran mona in windbg to find an appropriate SEH with pop pop ret instructions to use. I chose the first one from the list (0040FE76) and it worked successfully.

- A screenshot of successful execution, showing it in the debugger and execution of your shellcode, e.g. popping a calc. The images must accurately depict what is requested.
  - Show this via a screenshot with !exchain in WinDbg

```

0:003> g
(d84.a7c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00050050 ecx=00000078 edx=00000001b esi=0239efc0 edi=0239ffc0
eip=0042b7da esp=0239ed74 ebp=0239ff0 iopl=0 nv up ei pl nz na pe cy
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b ef1=00010207
server+0xb7d0:
0042b7da 660f7f6740 movdq xmmword ptr [edi+40h],xmm4 ds:002b:023a0000=?????????????????????????????
0:003> lexchain
0239fed0: 43434343
Invalid exception stack at 42424242

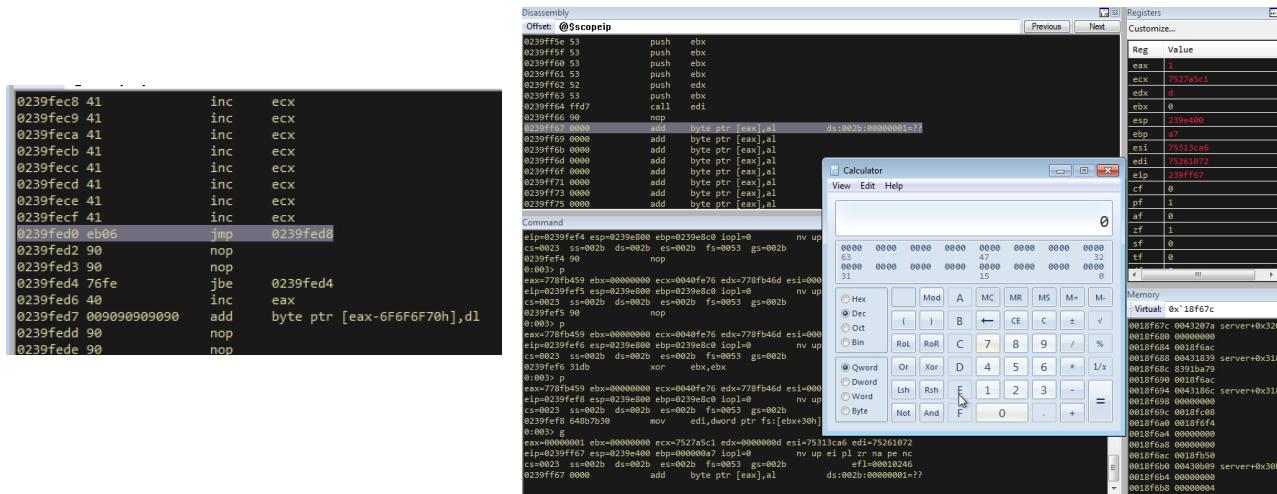
```

```

WARNING: Inaccessible path: 'C:\Users\burlly\Documents\Visual Studio 2010\Projects'
*** wait with pending attach
Symbol search path is: srv*C:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
eax=7efaf000 ebx=00000000 ecx=00000000 edi=00000000
eip=0042b7da esp=0239ff05c ebp=0239ff88 iopl=0 nv up ei pl nz na pe cy
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b ef1=00000246
ntdll!DbgBreakPoint:
77800000 int 3
0:003> g
eax=00000000 ebx=005d0058 ecx=00000078 edx=00000001b esi=0239efc0 edi=0239ffc0
eip=0042b7da esp=0239ed74 ebp=0239fee0 iopl=0 nv up ei pl nz na pe cy
cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b ef1=00010207
server+0xb7d0:
0042b7da 660f7f6740 movdq xmmword ptr [edi+40h],xmm4 ds:002b:023a0000=?????
0:003> !exchain
0239fed0: server+fe76 (0040fe76)
invalid exception stack at 909000eb

```

- Show your shellcode being executed.



- Your Python script must be included. I will look at and may run your scripts. Again, it should be written to run in the IA lab Windows 7 environment, so if you work elsewhere, be sure to adapt it.

**Answer:**

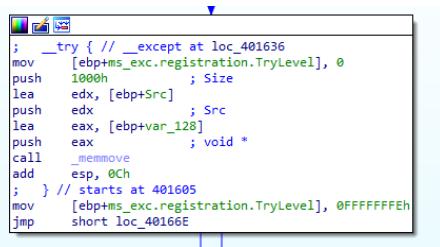
Requested scripts are attached to assignment.

## Part 2:

- Provide a narrative explaining the core vulnerability that caused this. What is it?
  - Find it in IDA Pro or Ghidra, along with your debugger—this is your evidence.
    - Confirm your suspicions. Don't just assume something is the cause because you saw it in static analysis; figure out how to confirm it in the debugger and present your evidence. (A one sentence description of the vulnerability with no evidence is not adequate.)
  - Show evidence in the form of screenshots and written descriptions—**both are expected**. Use coherent paragraph(s) while avoiding unnecessary, irrelevant digressions.
  - Explain the vulnerability. What can be done to fix it? Provide recommendations for remediation.
    - You must identify the vulnerability. If you do not know how to remediate it, then you can skip this.

### Answer:

The core vulnerability is the function `memmove()`. The destination is smaller than the source. It causes an exception exactly on the call to `memmove`.



The screenshot shows a debugger interface with two panes. The left pane displays assembly code with some comments and labels. The right pane shows the corresponding C-like pseudocode. The assembly code includes instructions like `push 1000h`, `lea edx, [ebp+Src]`, `push edx`, `lea eax, [ebp+var_128]`, `push eax`, `call _memmove`, `add esp, 0Ch`, and `jmp short loc_40166E`. The pseudocode on the right is:

```
char v19[72]; // [esp+101Ch] [ebp-128h] BYREF
memmove(v19, Src, 0x1000u);
```

In the disassembly and decompilation above you can see that size `1000h` is specified and read into an array size `72h`. `Memcpy` doesn't do bounds check so it's vulnerable to buffer overflows in this fashion. To fix this you should use `memcompy_s` as this "secure" version incorporates bounds checking or what is called parameter validation to prevents a destination buffer from being overflowed.

### Deliverable:

- **No zip files.** Attach each separately
- Your Python scripts.
- A Word Document or PDF that provides the requested tasks.
  - This is limited to no more than 10 pages, although it likely could be done in a lot less. Answers with a few screenshots and 1 or 2 sentences though will be inadequate and loss points. You can be brief, but you must explain fully.

- Do not need 15 pages showing images of varying amounts of A's and B's as input. 😊 It is fine to retain those for your personal notes, but that should not be a part of the submission.