

Cracking SP2 SEH

Lin0xx
Noxusfiles.com

Overview

- How to exploit SEH overwrites
 - And how to do it with style
- How to smash SP2 SEH specifically

What is SEH?

- Structured Exception Handling
- What is an exception?
 - Catches bad stuff
 - Allows you to fix it
 - Divide by zeros / access violations
- MS and Linux have different methods of implementing this

Win32 vs. Linux

- Linux typically uses signal handlers

```
typedef void (*sighandler_t)(int);
```

```
sighandler_t signal(int signum, sighandler_t handler);
```

- Somewhat ugly.

- Win32

- Uses the thread environment block (TEB) to store a chain of handlers

- Once something bad happens, it walks the chain until someone can fix it

- Very nice.

Why Use SEH Overwrites?

- Reliability
 - You don't have to worry about access violations after you get your SEH structure pwn'd.
 - In fact, you **want** an access violation so your handler executes.
- Also bypasses certain protections on former versions of MSVC++.

How This Looks in Memory

+0x000 Next : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 Handler : Ptr32

Win32 SEH Assembly

```
push fs:[0]           ;previous SEH
push 0x3133790       ;address of new SEH
mov fs:[0], esp      ;set SEH at top of list
```

- When something bad happens, 0x3133790 gets called
- If it can't handle it, the exception is passed on
- In C-esque languages, they look like this:

```
__try { /*do bad stuff*/ }
__except { /*fix it or cleanup*/ }
```

Notice Something...

- That SEH structure is on the stack.
- Can you say 'stack based buffer overflow'?
- If you can overwrite this exception handler and call an exception...
 - You win.

How to Exploit

- First, we need to know some internal data structures.

```
typedef EXCEPTION_DISPOSITION (*ExceptionHandler)(  
    IN EXCEPTION_RECORD ExceptionRecord,  
    IN PVOID EstablisherFrame,  
    IN PCONTEXT ContextRecord,  
    IN PVOID DispatcherContext);
```

- Why is this important? Well, EstablisherFrame points to OUR EXCEPTION_REGISTRATION structure...
- EstablisherFrame is at [esp+8]

Continued...

- So.. if you can shift the stack up then jump to the current stack pointer...
 - Your code executes.
- This is ONLY 4 bytes though, so just jump over the other part of the structure and jump into your shellcode.
- Queue the diagram I stole from skape to make things pretty.

SP2 SEH

- Microsoft ***did*** try to do something about this.
- Somewhat half-assed though, but still causes us problems.
- Logic – if the SEH structure is on the stack or if it's inside a binary – don't use it.
- But if it's inside of something that isn't an image (dll / exe / etc), go.

How to Crack This

- Dump the entire memory space and get searching.
- Of course, metasploit has an answer to this.
 - memdump.exe
- Output can then be used with msfpescan to find pop / pop / ret sequences.
- Look at the addresses, then see if they're inside an image.

When You Just Can't Fit It In...

- <Queue Taylor's joke>
- You've got 0x44 bytes (68 decimal). This is NOT enough to shovel a shell.
- What if...
 - Your shellcode was copied to somewhere else, but you don't know where?
 - Happens a lot in different software (could be one of the reasons there was a BOF in the first place ;)
 - Or, you could just store your code in a completely different place in memory.

Egg Hunting

- What if... You could have shellcode that would search for shellcode?
- Put a unique sequence of bytes in front of your shellcode, then search for them, and jump right into the buffer.
- These bytes must be executable, since it takes more code to jump past them.
- However, there are some complications with this.

Tripping The VAS Fantastic

- Okay, searching through process memory is dangerous.
 - What if you access violate?
 - That will kill the app.
- There has to be a way to know you're not searching through unmapped memory.
- Skape wrote a whitepaper on this.

Method 1: SEH

- The same thing we exploited can be used to help us search as well.
- Methodology:
 - Setup an SEH to catch any access violations.
 - Then, just walk the entire VAS – you won't die.
- This method will work on ALL versions of windows, including 9x.
- Downside: it is very large – about 60 bytes.

Method 2: IsBadPtr

- Basically the same method as number 1, but this is implemented as an API call in kernel32.dll
- REALLY big downside: you have to know the address of IsBadPtr.
 - You can either hard code it and risk crashing the program.
 - Or... You can do the position independent GetProcAddress implementation, which is a lot of work.
- 37 bytes.

Method 3: NtDisplayString

- You know when you have a bluescreen?
 - Messages are displayed on it via NtDisplayString.
- Pass it the address you want to validate, and if it's invalid, you get the return value of 0xc0000005.
- Uses a syscall, with a number that ***hasn't*** changed across any NT based version of windows.
- Really robust, really small at 32 bytes.

Code Demo

- Let's pwn that `#*&$()`.