

Tutorial: SEH Based Exploits and the Development Process

Tutorial by Mark Nicholls AKA n1p

The intent of this exploit tutorial is to educate the reader on the use and understanding of vulnerabilities and exploit development. This will hopefully enable readers to gain a better understanding of the use of exploitation tools and what goes on underneath to more accurately assess the risk of discovered vulnerabilities in a computer environment. It is important for security consultants and ethical hackers to understand how buffer overflows actually work, as having such knowledge will improve penetration testing capabilities. It will also give you the tools to more accurately assess the risk of vulnerabilities and develop effective countermeasures for exploits doing the rounds in the wild.

With this in, I am going to focus exclusively on the practical skills needed to exploit Structured Exception Handler buffer overflows. I won't go into too much detail regarding the theory of how they work, or how buffer overflows can be discovered. There are many other resources available on this subject, and I encourage you to research this further

Warning! Please note that this tutorial is intended for educational purposes only, and skills gained here should NOT be used to attack any system for which you don't have permission to access. It is illegal.

del.icio.us

[Discuss in Forums {mos_smf_discuss:/root}](#)

Brief Intro to Structured Exception Handlers (SEH)

An exception handler is a piece of code that is written inside an application with the purpose of dealing with cleanup activities when the application throws an exception error. A typical exception handler looks like this:

```
try {  
  
    line = console.readLine();  
  
} catch {  
  
    (Exception e) {  
  
        console.println("Error: " + e.message());  
  
        }  
  
}
```

When no exception handlers have been coded by a developer, there is a default Structured Exception Handler that is used to handle exceptions within Windows programs. Every process has an OS supplied SEH, and when a Windows program has an exception that it cannot handle itself, control is passed to a SEH address that has code that can be used to show a dialog box explaining that the program has crashed. As seen below:

<http://msdn.microsoft.com/en-us/library/ms679270%28v=VS.85%29.aspx>

<http://www.microsoft.com/msj/0197/exception/exception.aspx>

This default handler is seen at 0xFFFFFFFF and viewable in a debugger as such in the Stack window below. This is the end of the Stack Chain and should always be hit if the program cannot successfully handle crashes.

The SEH chain is essentially a linked list that is laid out in a structure similar to the chain below with the default OS handler at the end.

Each code block has its own stack frame, and the pointer to the exception handler is part of this stack frame. Information

about the exception handler is stored in an `exception_registration` structure on the stack. Each record then has the following info:

- A pointer to the next SEH record
- Pointer to address of the exception handler (SE Handler)

Ok, that's enough theory. References are provided at the end for further reading and learning. On to the practical stuff...

Choosing the Target

The target chosen is the Yahoo! Media Player. It has a known SEH overflow vulnerability. The best way to learn about exploiting such holes is to search for an application on a site like ExploitDB or Milw0rm. The available exploits will provide information as to whether they are remote, stack or seh type overflows that lead to crash / DOS / code execution. Rather than looking at the exploit code, I download the software and attempt to exploit it with only the type of vulnerability being known to me.

This provides a more realistic scenario and allows me to create the exploit from scratch, whilst also avoiding the time consuming task of fuzzing a dead end. If you do get stuck, then the exploit code is there for some hints. However, the code available may also not work or be complete. This is usually done to prevent script kiddies from copying code directly. Additionally, the code may not work due to changes in the OS, such as an update, service pack, etc. This provides a more challenging task to those that want to learn about exploit development, and, as a result, it can be more rewarding when a correct exploit is generated.

A quick search for "exploitdb movie player" returns a hit that provides enough information to progress with — Yahoo Player v1.0 (.m3u/.pls/.ypl) Buffer Overflow (SEH) (This may change). So with simple inference, we know that we have a target that incorrectly handles playlist files and results in a SEH overflow. So, let's begin.

Crashing the Media Player

When performing a regular stack-based buffer overflow, we overwrite the return address (EIP) and make the application jump to our shellcode. When doing a SEH overflow, we will continue overwriting the stack after overwriting EIP, so we can overwrite the default exception handler as well. This will provide the launch pad into our shellcode that will simply launch the Windows Calculator.

Using python (or language of choice), we can create a script to generate files that will hopefully crash the application. Firstly, we create the script below that will generate a ".m3u" playlist file. In this file a number of "\x41" characters ("A" in hex) will be placed. The purpose of this is to show us where our file gets loaded in memory. With a successful overflow, a large amount of "\x41" characters will be visible in registers or, in this case, exception handlers. The script is below:

Using this, we create files with increments of 1000 "A" characters and continue this until a crash. Using an offset we are met with the default Windows error handler and... Voila... the program crashes. To investigate further attach a debugger (OllyDBG/Immunity). To do this either run the program and attach it to the process, let it breakpoint and then hit run. Alternatively just open it as an executable from the debugger menu.

With the application running, select File->Open location/playlist and select the newly created file that contains 1000 "\x41" characters.

We can see that only the EBP register has been overwritten with some "A" characters in the yxga dll but not much else. The application is also handled by the default windows handler.

Let us continue with another 1000 characters. This time using 2000 characters, the application breaks, and, looking at the SEH chain window using the debugger (View->SEH Chain), we can see one exception record.

This record points to 0x001AC88. Looking closely at the register window, the EBX register contains our A*2000 characters. EBX contains 0x0010A888 which points to the start of our A buffer which continues to 0x0010AC56. Looking at our SEH record again, we can see that the end of the buffer is close to this address. It seems highly likely that larger buffers will successfully overwrite this address and as a result overwrite the SEH record. From this, we could also work out the correct buffer size with some simple calculation, but what would we learn? So, we try a larger file of 3000 characters.

Using this file, the application crashes again and viewing our SEH chain using the debugger (View->SEH Chain), we can see that the record and associated handler have indeed been overwritten with our crafted file data. At this point, if we use the Shift and F9 keys to pass the exception to the application, we will see that this results in an access violation when executing 41414141 or AAAA, and our EIP register now points to 41414141.

The SE handler has been overwritten, and now it becomes interesting. When the exception is handled, EIP will be overwritten with the address in the SE Handler. Since we can control the value in the handler, we can control and therefore redirect execution to shellcode to launch calc.exe.

Find Offsets and Create Exploit

Whilst we have now crashed the application and identified a vulnerability, we do not have the correct offsets required to allow execution of our shellcode payload. To do this, we must use some additional tools. There are a range of tools available for this purpose, and you can even code your own. For this tutorial I prefer to use Metasploit's range of applications. These are:

- Pattern_create.rb
- Pattern_offset.rb
- Msfencode
- Msfpayload

The first two ruby scripts are used to identify the correct size of the buffer to send to the application. This will allow us to correctly align or exploit code and shellcode for successful execution. So firstly, we need to identify the buffer length to cause the overflow. Using the pattern_create.rb script, we generate a unique string of length 3000, and plug that into our buffer string which is currently `'\x41'*3000`. So fire up Metasploit in Backtrack or whatever distro you use. Navigate to the metasploit tools folder and output the pattern into a file or stdout.

With the pattern created, we can copy this unique string into our offset code in our python script. So rather than 3000 A characters, we will have this string which will overwrite the SEH record. So generate the new file and re-run the media

player in the debugger.

Opening the file in the debugger and viewing the resultant crash, we can once again look at our overwritten SEH handler. Rather than 41414141, the value is now 71433471 or 71344371 (qC4q from our string), remembering little-endian architecture.

We can also view the memory in the stack window by right-clicking on the SE handler and selecting 'Follow in Address Stack' to see our unique string now filling the stack and our SEH Pointer and Handler are easily viewable.

We now use Metasploit's `pattern_offset.rb` to get out offset. Seen below, we enter 71433471 as the value (Metasploit helpfully takes endianness into account :P):

Our buffer size is 2053! So we use this offset in our python script with some additional characters to definitively map out our buffer and structure of our exploit. So let's update the script and execute it again in the media player.

Pic 19

Once we hit the crash, hit Shift+F9 to hit the handler and pass the exception to the application. From the SEH window, follow the address in the stack again, and we can see that our value of CCCCCCCC and the last 4 bytes of our offset buffer form the SEH record. The EIP register also now contains our latest value. So our exploit looks like this so far:

```
[ Junk ][ Next SEH][SEH Handler]
[ A*2053 ][ \xcc*4 ]
```

The code in EIP is just an address and not executable opcodes. We need to continue! So, we will further break it down into distinctive parts – Junk, SEH Pointer, SEH Handler, and Shellcode. See the updated code below along with

resultant debugger information.

Code Execution & SafeSEH Protection

In Windows XP SP1 and later a number of protections were added to prevent this sort of exploit. These include SafeSEH and XORing of registers to 0x00000000, so that they can't be used for shellcode execution. There are plenty of tutorials on the theory of this and will not be covered in this practical tutorial. References are provided at the end for further reading. But essentially SafeSEH is a compiler function and is used to maintain a list of registered exception handlers, failing when they are not used.

The main mechanism for bypassing these protections is the use of POP, POP, RETURN instructions. Performing two POP instructions removes the top two entries from the stack and the RETURN instruction will then take the memory address that is now at the top of the stack and execute the instructions at that address.

In our case, the next SEH pointer address that we overwrite is also located at ESP+8 when the handler is called. Now if we overwrite the handler with the address of an external POP, POP, RETURN (some other loaded dll) sequence, we will take the +8 from ESP and the return address will now be the next SEH and placed in EIP. This is what we control and in this we will place a jump into our shellcode, which is executed!

This may well cause confusion, so viewing it in the debugger may help. The diagrams here - <http://www.corelan.be:8800/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/> will also help.

So firstly, let's find a POP POP Return in a module that is not SafeSEH compiled. Again, there are a number of tools for this. However, I use the Immunity Debugger plugin `!safeseh`; With the application loaded in the debugger, type `!safeseh` in the command window and hit enter.

This will bring up a SEH table window. Navigate to View->Logs to view allow loaded modules that the plugin has processed and scanned for the presence of SafeSEH. The modules without the protection are marked as unprotected and will be used to locate a suitable Pop, Pop, Return instruction. Using an application DLL is more reliable, as this will not be dependent on service packs etc.

We also need to ensure that the address does not contain and /x00 nulls that would terminate our exploit string. Try using the search plugin with Immunity and enter the command !search pop r32\npop r32\nret to search for pop, pop, returns. Looking through the resulting log, we can see that all of the media player DLLs are at low memory address and contain problematic \x00s. We need to look at some system DLLs.

For this I used 0x30010136 which was found as result of Immunity's plugin. We can also use Metasploits msfpescan with the P flag to identify a POP POP RETN.

So taking this address we need to place it in our script. Our script now looks like the following:

Before we can actually run this exploit, we want to set a breakpoint on our SEH overwrite address to confirm that it is being hit. Right-click in the CPU pane and select Go to->Expression, then enter the address of your SEH overwrite instruction and hit OK. Then set a breakpoint on it with F2.

Run the exploit file again and view the SEH chain in the debugger. The breakpoint should be visible and highlighted.

Then use the Shift + F9 keys to pass the exception to the program. The exception handler should kick in and the breakpoint should be hit. To see the POP instructions working, we need to step through the code. We can now use the F7 to step. Keep an eye on the Stack window to watch the stack change.

The addresses get popped off in the stack with the next address to be removed being 0x0010A0C4. With one more pop, the RETN address becomes 0x001AC88, as does EIP. We can also change the absolute address to display it as relative to ESP to see it move from ESP+8 to ESP (Right-click: Address->Relative to ESP).

Continuing with one more F7, we jump to the return address which is 0x001AC88. EIP is also set to this address. We can see that our `int3` is now interpreted as executable code, so we now see them as INT3 instructions. Close by we can also see the NOPs we added at the end. This is where we need to get to when our shellcode resides there.

In the way of our code execution however is our sehhandler pop pop return code that we used earlier, however it looks mangled as it is being interpreted as opcodes. So, we simply need to replace our `int3` instructions with a small jump to our NOPs and shellcode.

Let's put that in our python script. So first we add the jump and try a 16 byte jump. We also add a NOP sled to aim for and some `int3` to act as shellcode. This way, if we hit the `int3` our debugger will break, and we will know our jump is correct. Running the code to our SEH breakpoint and stepping, we can trace to the JMP and see that it works. We could even shorten it if necessary. Now we can add our real shellcode.

Shellcode Generation

Again, the task of exploitation is made all the more easy with the use of Metasploit. For shellcode generation we will use `msfpayload` and `msfencode`. `Msfpayload` generates the shellcode, whilst the `msfencode` encodes (obviously) the raw shellcode to remove bad characters such as null terminators. Our shellcode will simply display the windows calculator. There are numerous payloads in `Msfpayload`, and I encourage you to take a look at how each works. The Metasploit Unleashed tutorials cover this in much greater detail (link in references).

To generate the shellcode, enter the command shown below. This windows/exec payload takes EXITFUNC and CMD as arguments. The EXITFUNC variable controls how the payload will clean up after itself once it accomplishes its task and CMD is obviously the command to run. The R flag outputs the shellcode in raw format for the encoder.

The encoder arguments `'e'` selects the encoder which in this instance is Alpha Upper. The `'b'` flag is to remove bad characters. It is not necessary to use it here, as the Alpha Upper encoder will not use them. The `'t'` flag is used to select the output format, which is C in this case. So run the command to get our shellcode for calculator.

Then copy and paste this code into your exploit python script and run it. This will hopefully crash the application and run the calculator.

Finally, there we have our executed shellcode delivering the payload. Not very exciting I know. I would also suggest that you now take a look at the 0-day exploit code that is already available on exploitDB and see that our exploit is in fact different! We have used different offsets and safeSEH modules to get the shellcode to run successfully.

So, there we have it. That is the process for developing an SEH overflow exploit. We have taken a vulnerable app with an SEH overflow and without any prior knowledge, other than the existence of a vulnerability, we have generated our own working exploit.

Prevention

So we have looked at the vulnerability associated with Structured Exception Handlers and methods to exploit such issues. What are the mechanisms in place to prevent this sort of exploit?

SafeSEH

The main countermeasure offered by Microsoft is the SafeSEH compiler flag, but as we can see this is easily bypassed through the use of POP/POP/Return instructions to access modules that are not compiled with SafeSEH. The problem with this protection is that every application dll/module/executable would need to be compiled with this flag to include this protection. There is no real protection to prevent the Handler from being overwritten with our code as demonstrated earlier.

Stack Cookies / Canaries

The Stack Cookie is a compiler option that will add some code to a function's prologue and epilogue code in order to prevent successful abuse of typical stack-based overflows. When an application starts, a cookie is randomly generated

and saved in the function prologue. This cookie is copied to the stack and placed before the saved EBP and EIP which is located between the local variables and the return addresses. At function exit, this cookie is compared again with the initial cookie, and, if it is different (as a result of overflow), the application terminates. There are a number of methods to bypass these, and these are covered in a number of articles.

<http://blogs.technet.com/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx>

<http://www.corelan.be:8800/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr>

<http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf>

SEH Overwrite Protection

Recently in Vista, Win 7 and Server 2008, Microsoft has implemented a new security feature known as SEH Overwrite Protection. SEHOP is an improvement on the concept of Structured Exception Handling and implements more stringent security checks on SEH structure/chains. The core feature of SEHOP checks the chaining of all SEH structures present on the stack. Focus is on the last handler in the chain which should have the handler value pointing to our default handler 0xFFFFFFFF in NTDLL seen earlier in this tutorial. It is quite robust protection, but a POC that demonstrates a bypass mechanism is covered here:

http://www.sysdream.com/articles/sehop_en.pdf

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

Additional mechanisms that attempt to remedy the issue of buffer overflows are also available. These are Data Execution Prevention (DEP) and Address Space Layout Randomisation (ASLR). These significantly increase the difficulty of exploit development in Windows environments. However, a number of bypass mechanisms exist to defeat these protections as well. I may demonstrate these protection schemes and their use in later articles.

References

<http://msdn.microsoft.com/en-us/library/ms680657%28VS.85%29.aspx>

<http://www.corelan.be:8800/index.php>

<http://grey-corner.blogspot.com/>

<http://www.uninformed.org/>

<http://www.offensive-security.com/metasploit-unleashed/>

<http://www.metasploit.com/>

<http://www.ethicalhacker.net>

Be sure to stop by Mark Nicholls security blog, isolated-threat, that focuses on malware, shellcode and exploits. After visiting, if there's any tutorials you'd like to see from Mark, feel free to ask.