NYU Poly Reverse Engineering Lecture
Session II

Aaron Portnoy
TippingPoint Security Research

Peter Silberman
Mandiant Engineering and Research

Intro to x86

- Contains 8 general purposes registers
  - @eax, ebx, ecx, edx, esi, edi, ebp, esp
  - Consider them temporary variables
- Stack is used to store registers when values need to be saved
  - LIFO (push/pop)

Basic Program Execution Registers

Eight General Purpose Registers

EAX

EBX

ECX

EDX

EBP

ESP

ESI

EDI

Processor Status Flags

EFLAGS

Instruction Pointer

EIP

Six Segment Registers

CS

SS

DS

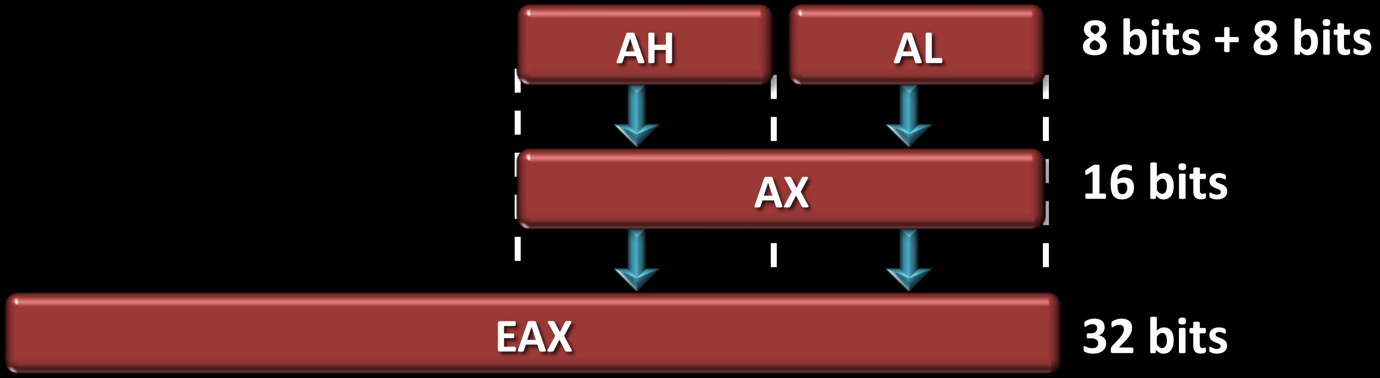
ES

FS

GS

General Purpose Registers

- The 8 general purpose registers are used for arithmetic and data movement
- Each register can be addressed as a 32 bit, 16 bit or 8 bit value



Overlapping Registers

32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		
EDI	DI		
EBP	BP		
ESP	SP		

General-Purpose Registers

- Some of these registers are used by specific instructions
  - EAX is automatically used by multiplication and division operations
  - ECX is used as a counter in several instructions
  - ESI and EDI are as src and dst for copying data in loops
  - EBP and ESP are used to track changes to the stack

- Calling conventions and ABIs define certain registers uses
  - EAX is used to store the return value for function calls
  - ECX is used to store a pointer to the 'this' object in C++

Classes of Instructions

- X86 has a lot of instructions
  - We are only going to cover a select few
- Instructions that:
  - Read
  - Write
  - Compare
  - Branch
  - Perform Arithmetic
    - Add
    - Subtract
    - Multiply
    - Divide
    - Bitmath
    - Floating Point

Read Instructions

- Memory dereferencing is the equivalent of reading

```
unsigned long x = 0;
x = *p;
```

- Where * is dereference assignment
- [ ] is dereference assignment in x86

```
mov reg32, [reg32]  -> mov eax, [ecx]
mov reg32, [imm32] -> mov eax, [04010000]
pop eax
```

Write Instructions

- Consider these a store
  - mov [ebx], 0x20
    - Stores the immediate value 0x20 at the address specified by @ebx
  - mov [ecx+0x14], edx
    - Stores the value of the edx register into the address at ecx plus 0x14 bytes

Stack Operations

- push
  - Syntax: *push src*
- Examples:
  - push eax
  - push 0x100
  - push dword_0x100400

- pop
  - Syntax: *pop dst*
- Examples:
  - pop eax

Arithmetic

- `inc`
  - Syntax: *inc dst*
- Examples:
  - inc edx

- `dec`
  - Syntax: *dec dst*
- Examples:
  - dec eax

Arithmetic (cont.)

- `mul`
  - Syntax: *mul src*
    - *Result is stored in ecx*
- Examples:
  - mul edx

- `div`
  - Syntax: *div src*
    - *Result is stored in eax*
- Examples:
  - div edi

Arithmetic (cont.)

- add
  - Syntax: *add dst, src*
- Examples:
  - add eax, 10
  - add edx, eax

- sub
  - Syntax: *sub dst, src*
- Examples:
  - sub eax, 10
  - sub ecx, edx

Arithmetic (cont.)

- `lea`
  - Syntax: *lea dst, src*
- Examples:
  - lea eax, [eax*4]
  - lea edx, [edi+ecx]

Arithmetic (cont.)

- Bitmath
  - shl, shr
    - Shifts the bits of the operand either to the left or to the right
      - 00000111 << 2 = 00011100
  - Examples:
    - shl eax, 2
    - shr edx, 4

Arithmetic (cont.)

- Floating Point
  - *fbld, fild, fcmovnbe*, …
  - Consult x86 manuals

Comparisons

- `cmp`
  - Syntax: *cmp dst, src*
- Examples:
  - cmp eax, ecx
  - cmp edx, 10

- `test/and`
  - Syntax: *test dst, src*
- Examples:
  - and eax, 10
  - test ecx, edx

Branches

- Used to direct code, frequently based on previous comparison
- jxx
  - Syntax: *jxx dst*
- Examples:
  - jz reg32
  - jnb $-5
  - jnz 0x04010012

Code Execution Transfers

- `Call`
  - Used to call functions
  - Syntax: *call src*
- Examples:
  - call ecx
  - call [40100000]
  - call 0x41c2200c

- Occasionally the jmp instruction will be responsible for transferring execution to another function
  - Examples:
    - jmp 0x41c2200c

Questions?

Vulnerability Classes

- Those we’ll cover
  - Integer Overflows
  - Stack/Heap Based Buffer Overflow
  - Format Strings
- Those we won’t
  - Invalid Free/Double Free
  - Uninitialized Variables
  - Misc. (memory corruption)
    - MS09-028: http://bit.ly/owningMSdirectshow

Integer Wraps

- Integers are able to store a finite size
- Integer wraps due to type conversion
  - Width
    - Unsigned long to short
- Integer wraps due to arithmetic
  - MAX_INT + x
  - 0 - x

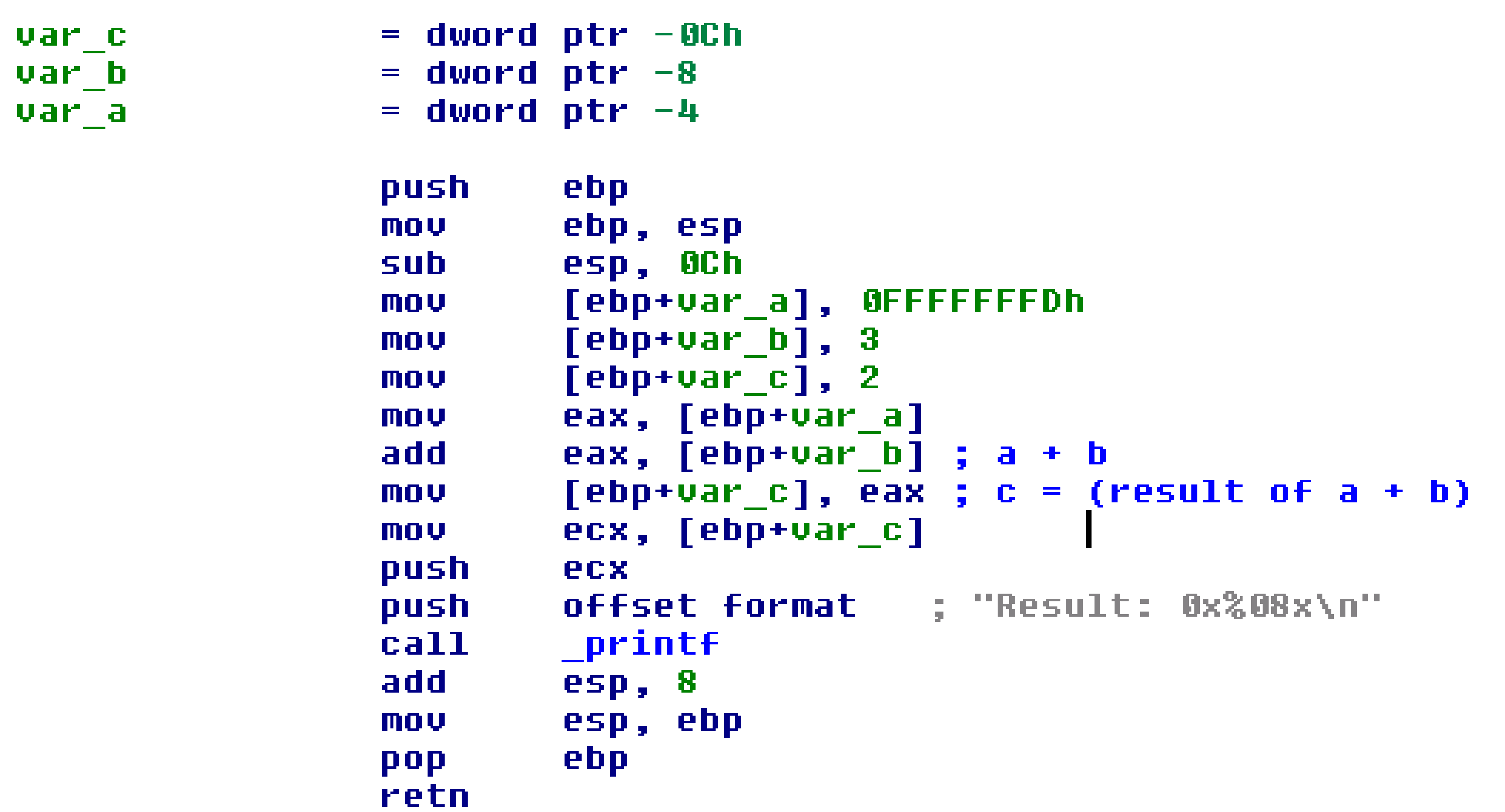
Integer Overflows (Ex 1)

```
unsigned long a = 0xFFFFFFFD;
unsigned long b = 3;
unsigned long c = 2;
c = a + b;
printf("Result: 0x%08x\n", c);
```

- Output:

```
Result: 0x00000000
```

Integer Overflows (Ex1)



```
var_c           = dword ptr -0Ch
var_b           = dword ptr -8
var_a           = dword ptr -4

                push    ebp
                mov     ebp, esp
                sub     esp, 0Ch
                mov     [ebp+var_a], 0FFFFFFFDh
                mov     [ebp+var_b], 3
                mov     [ebp+var_c], 2
                mov     eax, [ebp+var_a]
                add     eax, [ebp+var_b] ; a + b
                mov     [ebp+var_c], eax ; c = (result of a + b)
                mov     ecx, [ebp+var_c]
                push    ecx
                push    offset format   ; "Result: 0x%08x\n"
                call    _printf
                add     esp, 8
                mov     esp, ebp
                pop     ebp
                retn
```

Integer Overflows (Ex2)

```
void SomeFunc(unsigned long user_supplied, char * userbuffer)
    unsigned long a = 0;
    SHORT b = 0;
    char mybuffer[300];

    a = user_supplied;
    b = a;
    printf("Checking %d to make sure it is less than 300\n", b);
    if(b >= 300)
    {
        printf("Thank you come again..\n");
        return;
    }

    printf("Passed my checks, copying %d bytes into buffer of size %d\n", a, sizeof(mybuffer));
    strncpy(mybuffer, userbuffer, a);
```

Integer Overflows (Ex2)

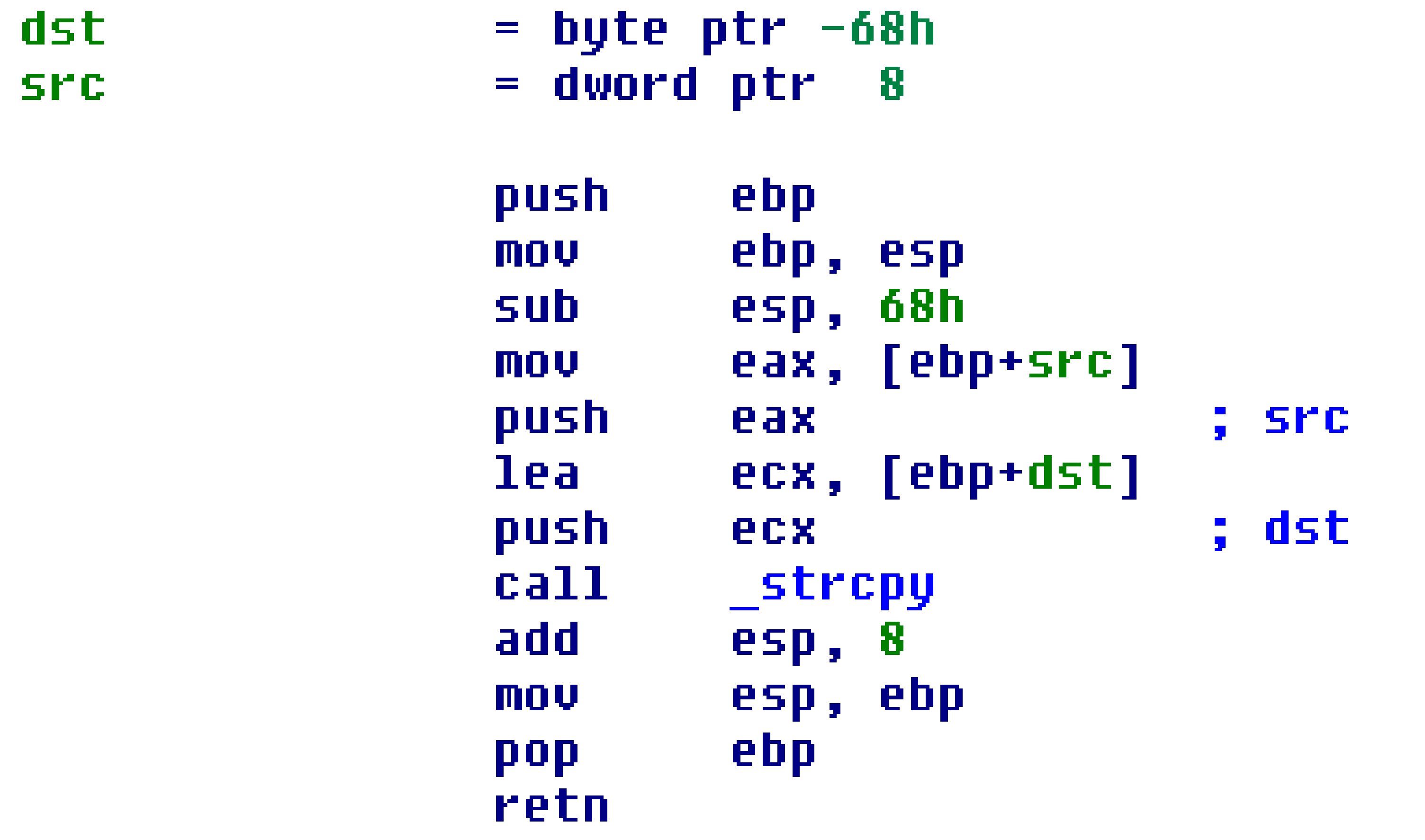
- Output:

```
Checking 0 to make sure it is less
  than 300
Passed my checks, copying 65536 bytes
  into buffer of size 300
```

Stack/Heap Overflows

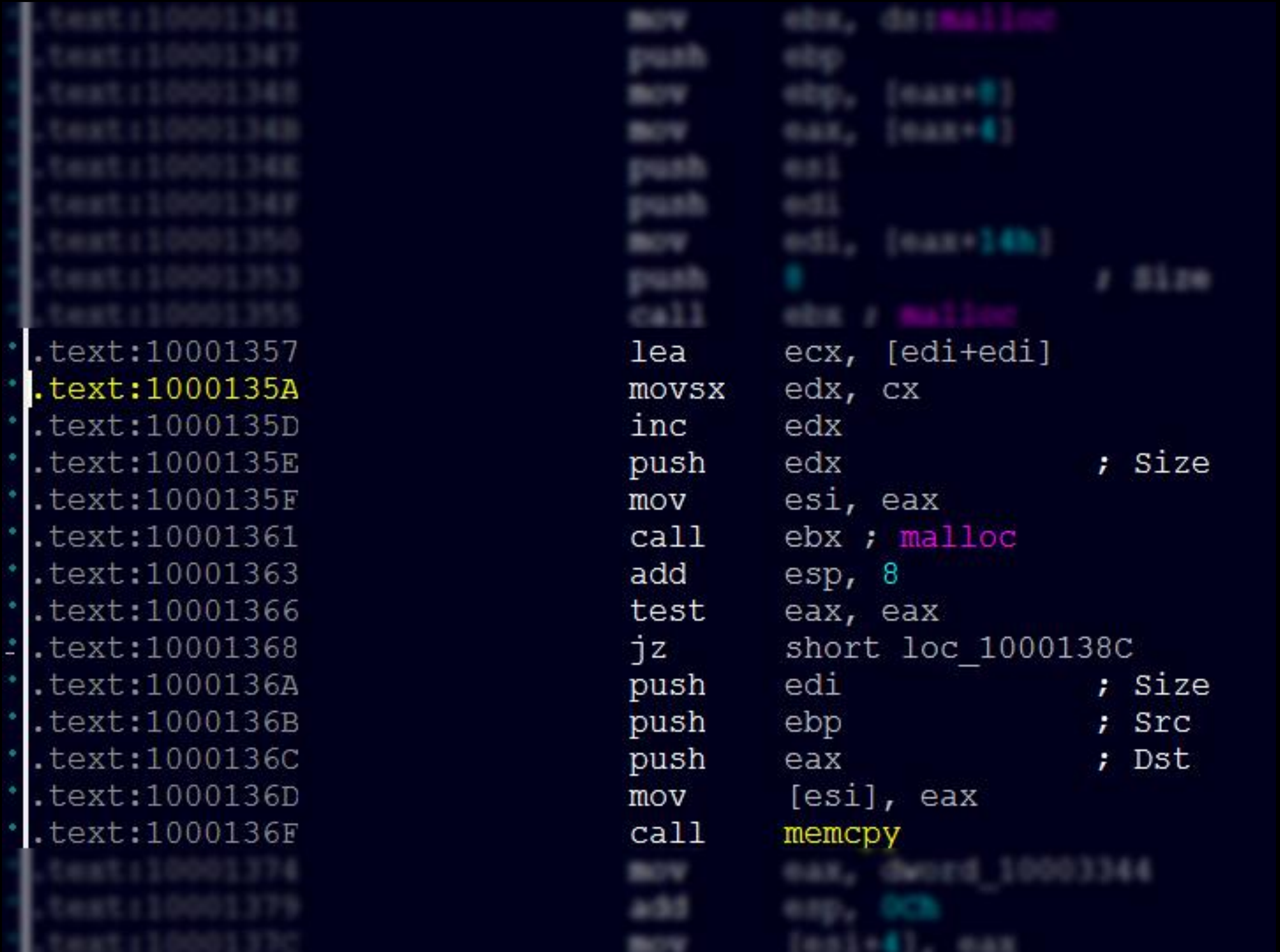
- Stack/Heap overflows are the most common memory mismanagement
  - Smashing the Stack for Fun and Profit, classic: http://www.phrack.org/issues.html?id=14&issue=49
  - Gera's insecure programming examples: http://community.corest.com/~gera/InsecureProgramming/abo1.html
- Essentially, exceeding the bounds of an allocation during a sequence of writes

Stack/Heap Overflows (Ex1)



```
dst             = byte ptr -68h
src             = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 68h
                mov     eax, [ebp+src]
                push    eax                     ; src
                lea     ecx, [ebp+dst]
                push    ecx                     ; dst
                call    _strcpy
                add     esp, 8
                mov     esp, ebp
                pop     ebp
                retn
```

Stack/Heap Overflows (Ex2)



```
.text:10001341                mov     ebx, ds:malloc
.text:10001347                push    ebp
.text:10001348                mov     ebp, [eax+8]
.text:1000134B                mov     eax, [eax+4]
.text:1000134E                push    esi
.text:1000134F                push    edi
.text:10001350                mov     edi, [eax+14h]
.text:10001353                push    8               ; Size
.text:10001355                call    ebx ; malloc
.text:10001357                lea     ecx, [edi+edi]
.text:1000135A                movsx   edx, cx
.text:1000135D                inc     edx
.text:1000135E                push    edx             ; Size
.text:1000135F                mov     esi, eax
.text:10001361                call    ebx ; malloc
.text:10001363                add     esp, 8
.text:10001366                test    eax, eax
.text:10001368                jz      short loc_1000138C
.text:1000136A                push    edi             ; Size
.text:1000136B                push    ebp             ; Src
.text:1000136C                push    eax             ; Dst
.text:1000136D                mov     [esi], eax
.text:1000136F                call    memcpy
.text:10001374                mov     eax, dword_10003344
.text:10001379                add     esp, 0Ch
.text:1000137C                mov     [esi+4], eax
```

A

Format String

- Passing unsanitized user input to a function that accepts a format string
  - Caused by C’s ability to use varargs
  - Enables attackers to read or write data to memory
  - %s, %x, %n

Format String (Ex1)

```
void pretty_print(char * user)
    printf(user);
```

- No sanitization of user input what happens we direct the function to continually pop items off the stack?

Questions?

Automation

Vulnerability Hunting Styles

- *I've been up for 3 days straight, where's my coffee: Cerebral and Successful Method (Aaron)*
  - Fully reverse document a product’s internal workings
  - Pros:
    - Full understanding of the product
    - In the developers head
    - Finding bugs is much easier, think like the developer
  - Cons:
    - Time consuming

Vulnerability Hunting Styles

- *Where's my Ritalin: ADHD Induced Method, Less successful (Peter):*
  - Only reverse points of input:
    - CreateFile, recv, ReadFile, rpc, etc..
  - Spend sometime understanding how to craft input to get most code coverage
  - Pros:
    - Finds bugs faster
    - less time spent reversing
    - Don't have to consume A LOT of coffee
  - Cons:
    - Only knocks off low to medium hanging fruit.
    - May not get full code coverage

P

Vulnerability Hunting Styles

- As much as our styles differ, we both make use of automation
  - Using conditional breakpoints to gather information
  - Analyzing binary code programmatically
  - Instrumenting an application
    - Call unknown() 10000 times with differing args, analyze output
  - Gathering runtime data
    - Fill in cross references with dynamic call information
    - Dump global variable values

P

Automating Binary Analysis

- Binary analysis tasks automated for speed
  - Deobfuscation
  - Control and Data Flow Analysis
    - CF Analysis can lead to identification or programming errors
      - Bad calls, dangerous loops, signed/unsigned compares etc..
    - DF Analysis can be used for type reconstruction or to improve CF analysis
      - User supplied variable + 2 = integer overflow
        - » Knowing its user supplied is the key

Automating Binary Analysis

- Pattern Matching:
  - Can be used to auto comment commonly used instruction sequences
    - Inline strcpy/strcmp/strlen
    - Inline memcpy, memset, memmove
    - FindCrypt plugin (for locating common crypto methods)
  - Find possibly interesting code to audit
    - Arithmetic followed by allocations
    - Format string calls with no format token

IDA + Automated Binary Analysis

- IDA provides its own scripting interface called IDC
- IDAPython gives users access to the SDK and IDC
  - All in python!
  - This is what we will focus on today

- Note we will be using IDAPython 2.5 (compiled from trunk)
  - http://thunkers.net/~deft/misc/Reversing102.zip

Iterating Over Functions

```
for ea_start in Functions(MinEA(), MaxEA()):
    print "%s: 0x%08x" % (GetFunctionName(ea_start), ea_start)
```

- Output:

```
__SEH_epilog: 0x757319fc
__SEH_prolog: 0x75731a0d
?LsapScavengerTrigger@@YGKPAX@Z: 0x75731a4d
?LsapTimerCallback@@YGXPAXE@Z: 0x75731ac1
?LsapDerefScavItem@@YGXPAU_LSAP_SCAVENGER_ITEM@@@Z: 0x75731b46
?LsapScavengerBreak@@YGKPAX@Z: 0x75731b77
_SafeAllocaFreeToHeap@4: 0x75733086
_LsapAllocateLsaHeap@4: 0x757330ab
(..)
```

Iterating Over Function’s Basic Blocks

- Still a little bit buggy
- Only enabled in IDAPython pulled from trunk

A

Iterating Over Function’s Basic Blocks

```
func = get_func(get_screen_ea())
fc = idaapi.FlowChart(func)

for block in fc:
    print "%x - %x [%d]:" % (block.startEA, block.endEA, block.id)

    for succ_block in block.succs():
        print "  %x - %x [%d]:" % (succ_block.startEA,
   succ_block.endEA, succ_block.id)

    for pred_block in block.preds():
        print "  %x - %x [%d]:" % (pred_block.startEA,
   pred_block.endEA, pred_block.id)
```

Iterating Over Function’s Instructions

```
func = get_func(get_screen_ea())
for ea in Heads(func.startEA, func.endEA):
    print “0x%08x: %s” % (ea, GetDisasm(ea))
```

- Output:

```
0x75789146: mov     edi, edi
0x75789148: push    ebp
0x75789149: mov     ebp, esp
0x7578914b: lea     eax, [ebp+arg_0]
0x7578914e: push    eax
0x7578914f: call    _LsapUnregisterAuditEvent@4
0x75789154: pop     ebp
0x75789155: retn    4
```

Tying it all together

- Bad call scanner (I know sooo 2000 and late)
- Identifies dangerous calls to known bad APIs i.e., strcpy, sprintf etc.

Tying it all together

- Finding PE Parsing routines:

```
for start in Functions(MinEA(), MaxEA()):
    for ea in Heads(start, PrevAddr(get_func(start).endEA)):
        disasm = GetDisasm(ea)
        name = GetFunctionName(ea)

        if disasm.lower().find("5a4d") != -1:
            l = "%s => 0x%08x: %s₩n" % (name, ea, disasm)
                msg(l)

        if disasm.lower().find("4550") != -1:
            l = "%s => 0x%08x: %s₩n" % (name, ea, disasm)
            msg(l)
```

Tying it all together

- Reverser’s Cookbook methods
  - find_path, find_all_paths
  - find_instr, find_func
  - enum_switches
  - file_io, net_io
  - …

A

Questions?

Debugging

A

Debugging Crashes

- When a crash occurs we are mostly concerned with…
  - Faulting instruction
    - Disassembly around faulting instruction (ub @eip in windbg)
  - Register contents
    - Pointers
      - To code
      - To data
    - Values
      - Simple data types
      - Return values
      - Lengths/Counters
  - Call stack
    - How we got here
  - State of the heap and stack
    - Verify heap integrity (!heap 0 –v)
    - Veryify stack integrity (dd @esp ; !exchain)

Debugging Crashes (cont.)

- VM Debugging
  - – Allows you to snapshot an entire system
    - This allows you to know ahead of time where things are allocated
  - – Revert back to a good state

Clever Breakpoints

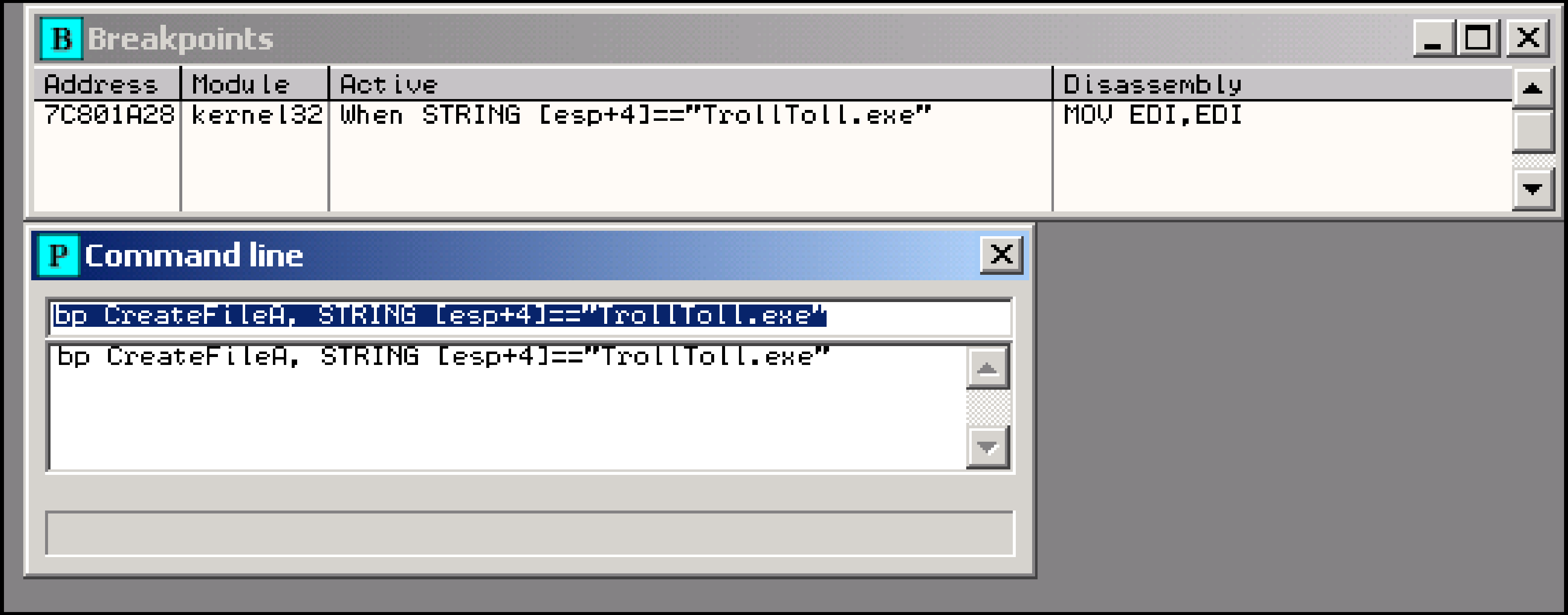
- Breakpoints are great for examining a program
  - Can get an inside peak at what is going on
    - Tracing! (check your log window)
  - Break if loop counter == 0xFFFD
- Memory breakpoints
  - When is a buffer first read from? Written to?
  - Useful for tricks like hunting down the source of an allocation
    - VM debugging helps here

Conditional Breakpoints (WinDbg)

- Conditional Breakpoints
  - Breakpoint is only executed when condition is met:
    - bp myprogram!SomeFunc+0x08 “**j @eax = 0xFFFD '';'gc'"**
      - http://msdn.microsoft.com/en-us/library/cc267482.aspx

Conditional Breakpoints (OllyDbg)

- OllyDbg has a command line plugin
  - Bp CreateFileA, STRING [esp+4] == “TrollToll.exe”



Address	Module	Active	Disassembly
7C801A28	kernel32	When STRING [esp+4]=="TrollToll.exe"	MOV EDI,EDI

```
bp CreateFileA, STRING [esp+4]=="TrollToll.exe"
bp CreateFileA, STRING [esp+4]=="TrollToll.exe"
```

A

Conditional Breakpoints (GDB)

- GDB standard *nix debugger (unfortunately)
  - b *0x00401000 if $eax==5

Useful Debugging Tricks

- GFLAGS
  - Page Heap (awesome)
    - Can obscure vulnerabilities, though
      - Pointer math
    - Enables the debug heap, behaves differently
  - User mode stack trace database
    - Track sources of heap allocations
  - Heap * checking
    - Ensures heap integrity during heap operations
  - Read up: http://technet.microsoft.com/en-us/library/cc738763(WS.10).aspx

Useful Debugging Tricks

- WinDBG
  - !heap
    - Walk the heap (requires symbols)
    - !heap 0 –v
    - !heap –p –a 0xwhatever
  - !exchain
    - Lists the registered exception handles
  - !analyze –v
    - Analyzes the current crash, includes call stack, other useful info
  - !exploitable
    - Not as cool as it sounds
- ImmDbg
    - Has it’s own !heap

Questions?

Hands On Experience: GreenMan

Background

- GreenMan
  - – Compiled for Linux/Mac/Windows
  - – Vulnerable Application (sorry to kill the suspense)
  - – Listens on port 4959
  - – GreenMan has the following vulnerabilities:
    - Integer overflow resulting in a heap overflow
    - Stack overflow
    - Format string

Background

- SweetDee.py
  - – Client for GreenMan
  - – sweetdee.py <host> <port> <opcode> <payload>
    - Host – your ip OR ALL which sends a packet to the ip range:
      - – 192.168.1.0-255
    - Port – should be 4959 unless recompiled
    - Opcode – which vulnerability do you want to trigger?
    - Payload (optional) – used only in opcode 3

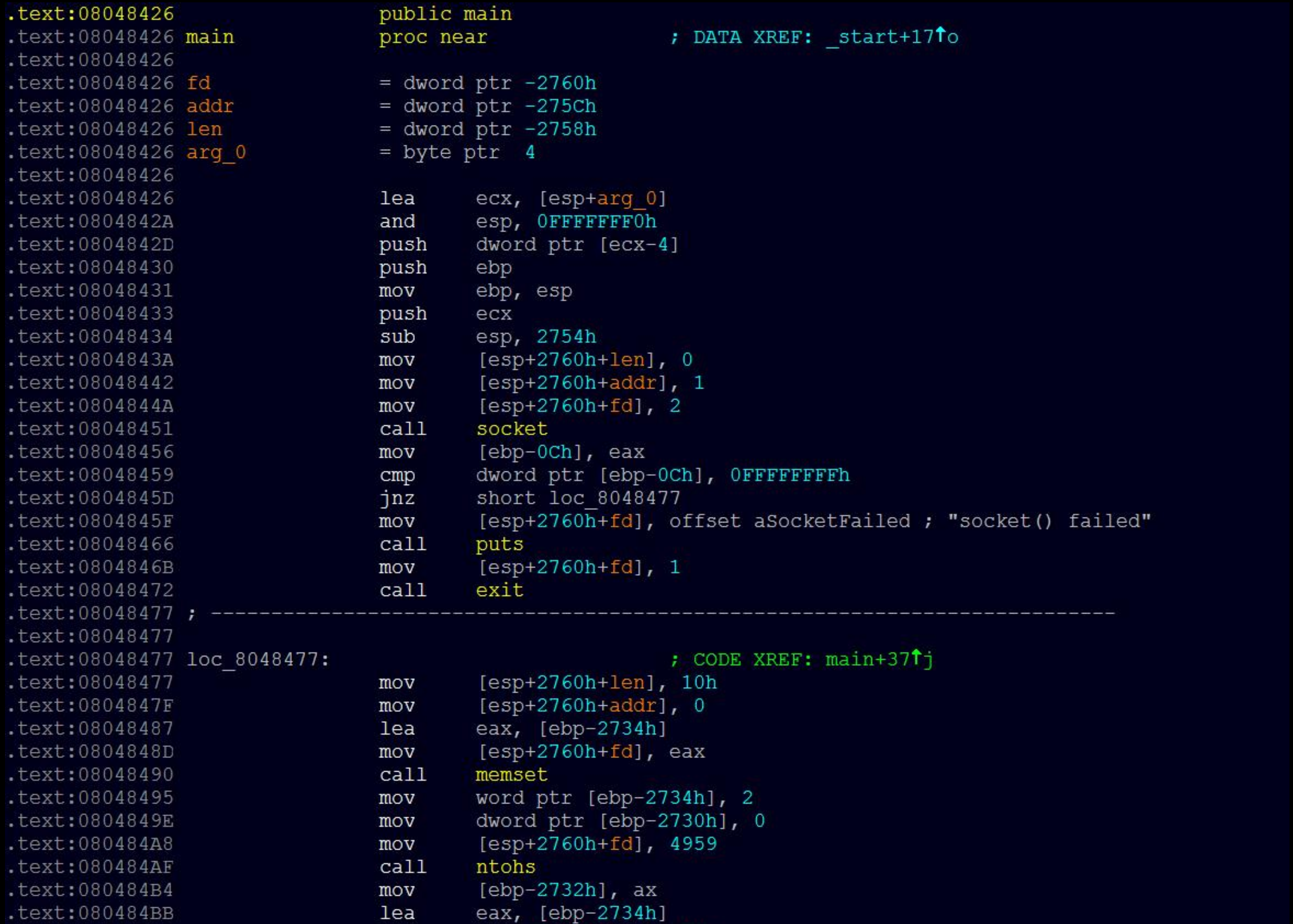
You need to…

- Download
  - http://thunkers.net/~deft/misc/Reversing102.zip
- Please connect to Wireless Access Point
  - VirusNetwork
- Disable your firewalls ☺
- Run the GreenMan application
- Attach your debugger of choice to GreenMan
  - WinDBG: F6
  - OllyDBG:File->Attach
  - GDB: gdb `pidof GreenMan`

Reversing the Binary

- Open up the GreenMan binary in IDA
  - – Windows/Linux/OS X versions will look different
- Check the Imports
  - – This will give you an idea of what functions it uses
  - – However, if it’s statically compiled (linux) it won’t have Imports
    - So check Names subview
  - – Check exports to find main(). Double-click it.

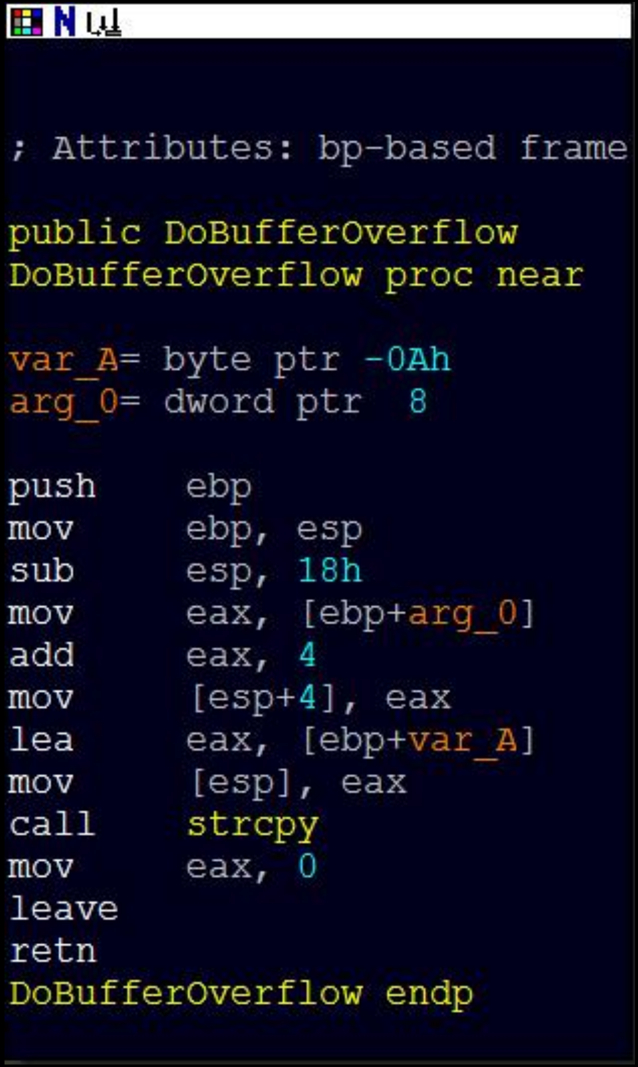
Reversing the Binary (Linux)



```
.text:08048426                 public main
.text:08048426 main            proc near               ; DATA XREF: _start+17↑o
.text:08048426
.text:08048426 fd              = dword ptr -2760h
.text:08048426 addr            = dword ptr -275Ch
.text:08048426 len             = dword ptr -2758h
.text:08048426 arg_0           = byte ptr  4
.text:08048426
.text:08048426                 lea     ecx, [esp+arg_0]
.text:0804842A                 and     esp, 0FFFFFFF0h
.text:0804842D                 push    dword ptr [ecx-4]
.text:08048430                 push    ebp
.text:08048431                 mov     ebp, esp
.text:08048433                 push    ecx
.text:08048434                 sub     esp, 2754h
.text:0804843A                 mov     [esp+2760h+len], 0
.text:08048442                 mov     [esp+2760h+addr], 1
.text:0804844A                 mov     [esp+2760h+fd], 2
.text:08048451                 call    socket
.text:08048456                 mov     [ebp-0Ch], eax
.text:08048459                 cmp     dword ptr [ebp-0Ch], 0FFFFFFFFh
.text:0804845D                 jnz     short loc_8048477
.text:0804845F                 mov     [esp+2760h+fd], offset aSocketFailed ; "socket() failed"
.text:08048466                 call    puts
.text:0804846B                 mov     [esp+2760h+fd], 1
.text:08048472                 call    exit
.text:08048477 ; ---------------------------------------------------------------------------
.text:08048477
.text:08048477 loc_8048477:                            ; CODE XREF: main+37↑j
.text:08048477                 mov     [esp+2760h+len], 10h
.text:0804847F                 mov     [esp+2760h+addr], 0
.text:08048487                 lea     eax, [ebp-2734h]
.text:0804848D                 mov     [esp+2760h+fd], eax
.text:08048490                 call    memset
.text:08048495                 mov     word ptr [ebp-2734h], 2
.text:0804849E                 mov     dword ptr [ebp-2730h], 0
.text:080484A8                 mov     [esp+2760h+fd], 4959
.text:080484AF                 call    ntohs
.text:080484B4                 mov     [ebp-2732h], ax
.text:080484BB                 lea     eax, [ebp-2734h]
```

Reversing the Binary (Linux)

- Things to note
  - Rather than push, gcc will move the addresses directly to the address of esp+X

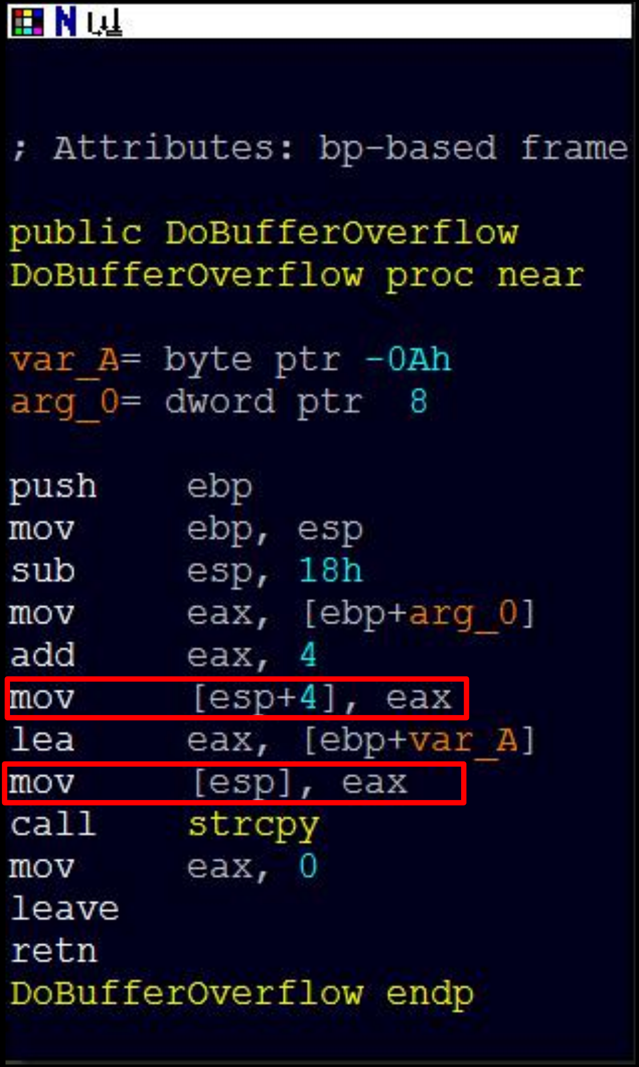


```
; Attributes: bp-based frame

public DoBufferOverflow
DoBufferOverflow proc near

var_A= byte ptr -0Ah
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, [ebp+arg_0]
add     eax, 4
mov     [esp+4], eax
lea     eax, [ebp+var_A]
mov     [esp], eax
call    strcpy
mov     eax, 0
leave
retn
DoBufferOverflow endp
```

A

Reversing the Binary (Linux)

- Things to note
  - Rather than push, gcc will move the addresses directly to the address of esp+X

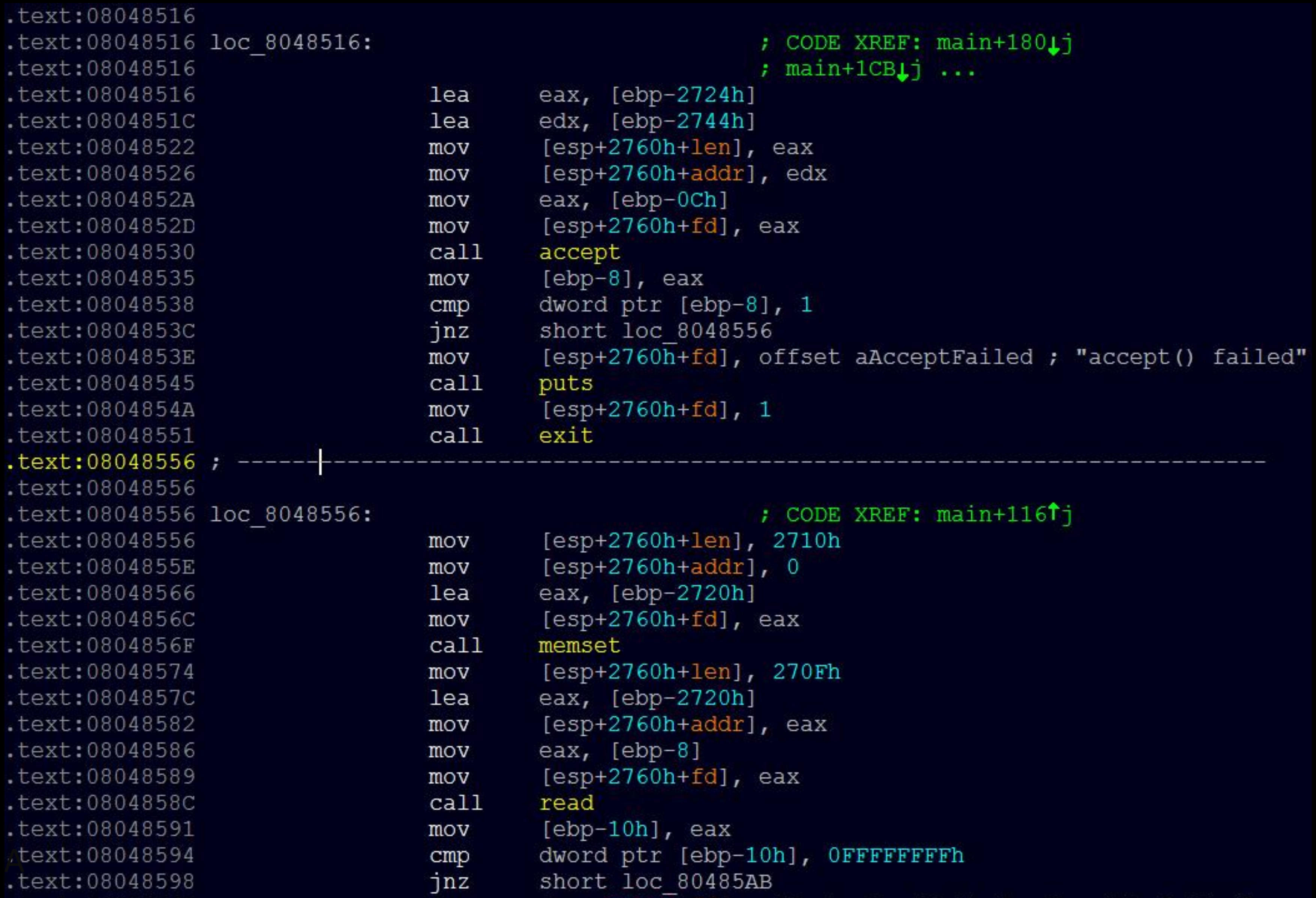


```
; Attributes: bp-based frame

public DoBufferOverflow
DoBufferOverflow proc near

var_A= byte ptr -0Ah
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, [ebp+arg_0]
add     eax, 4
mov     [esp+4], eax
lea     eax, [ebp+var_A]
mov     [esp], eax
call    strcpy
mov     eax, 0
leave
retn
DoBufferOverflow endp
```

Reversing the Binary (Linux)

- Things to note
  - Even though source code uses recv() and printf() the compiler substitutes

Reversing the Binary (Linux)

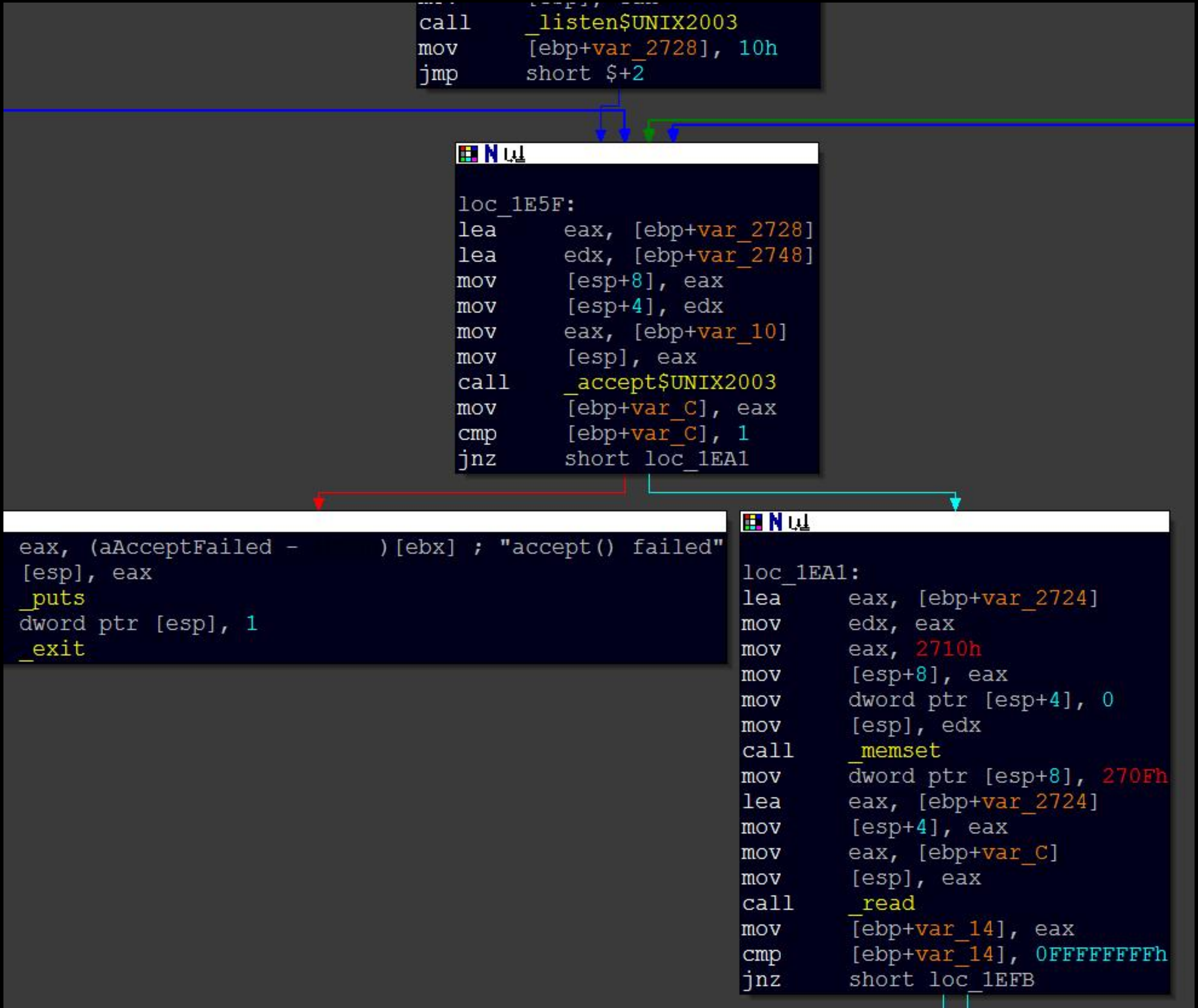


```
.text:08048516
.text:08048516 loc_8048516:                            ; CODE XREF: main+180↓j
.text:08048516                                         ; main+1CB↓j ...
.text:08048516                 lea     eax, [ebp-2724h]
.text:0804851C                 lea     edx, [ebp-2744h]
.text:08048522                 mov     [esp+2760h+len], eax
.text:08048526                 mov     [esp+2760h+addr], edx
.text:0804852A                 mov     eax, [ebp-0Ch]
.text:0804852D                 mov     [esp+2760h+fd], eax
.text:08048530                 call    accept
.text:08048535                 mov     [ebp-8], eax
.text:08048538                 cmp     dword ptr [ebp-8], 1
.text:0804853C                 jnz     short loc_8048556
.text:0804853E                 mov     [esp+2760h+fd], offset aAcceptFailed ; "accept() failed"
.text:08048545                 call    puts
.text:0804854A                 mov     [esp+2760h+fd], 1
.text:08048551                 call    exit
.text:08048556 ; ---------------------------------------------------------------------------
.text:08048556
.text:08048556 loc_8048556:                            ; CODE XREF: main+116↑j
.text:08048556                 mov     [esp+2760h+len], 2710h
.text:0804855E                 mov     [esp+2760h+addr], 0
.text:08048566                 lea     eax, [ebp-2720h]
.text:0804856C                 mov     [esp+2760h+fd], eax
.text:0804856F                 call    memset
.text:08048574                 mov     [esp+2760h+len], 270Fh
.text:0804857C                 lea     eax, [ebp-2720h]
.text:08048582                 mov     [esp+2760h+addr], eax
.text:08048586                 mov     eax, [ebp-8]
.text:08048589                 mov     [esp+2760h+fd], eax
.text:0804858C                 call    read
.text:08048591                 mov     [ebp-10h], eax
.text:08048594                 cmp     dword ptr [ebp-10h], 0FFFFFFFFh
.text:08048598                 jnz     short loc_80485AB
```

Reversing the Binary (OS X)

- Things to note
  - IDA identifies it as a MACH-O binary
  - Code looks similar to linux binary

Reversing the Binary (OS X)



```
call    _listen$UNIX2003
mov     [ebp+var_2728], 10h
jmp     short $+2
```

```
loc_1E5F:
lea     eax, [ebp+var_2728]
lea     edx, [ebp+var_2748]
mov     [esp+8], eax
mov     [esp+4], edx
mov     eax, [ebp+var_10]
mov     [esp], eax
call    _accept$UNIX2003
mov     [ebp+var_C], eax
cmp     [ebp+var_C], 1
jnz     short loc_1EA1
```

```
eax, (aAcceptFailed -      )[ebx] ; "accept() failed"
[esp], eax
_puts
dword ptr [esp], 1
_exit
```

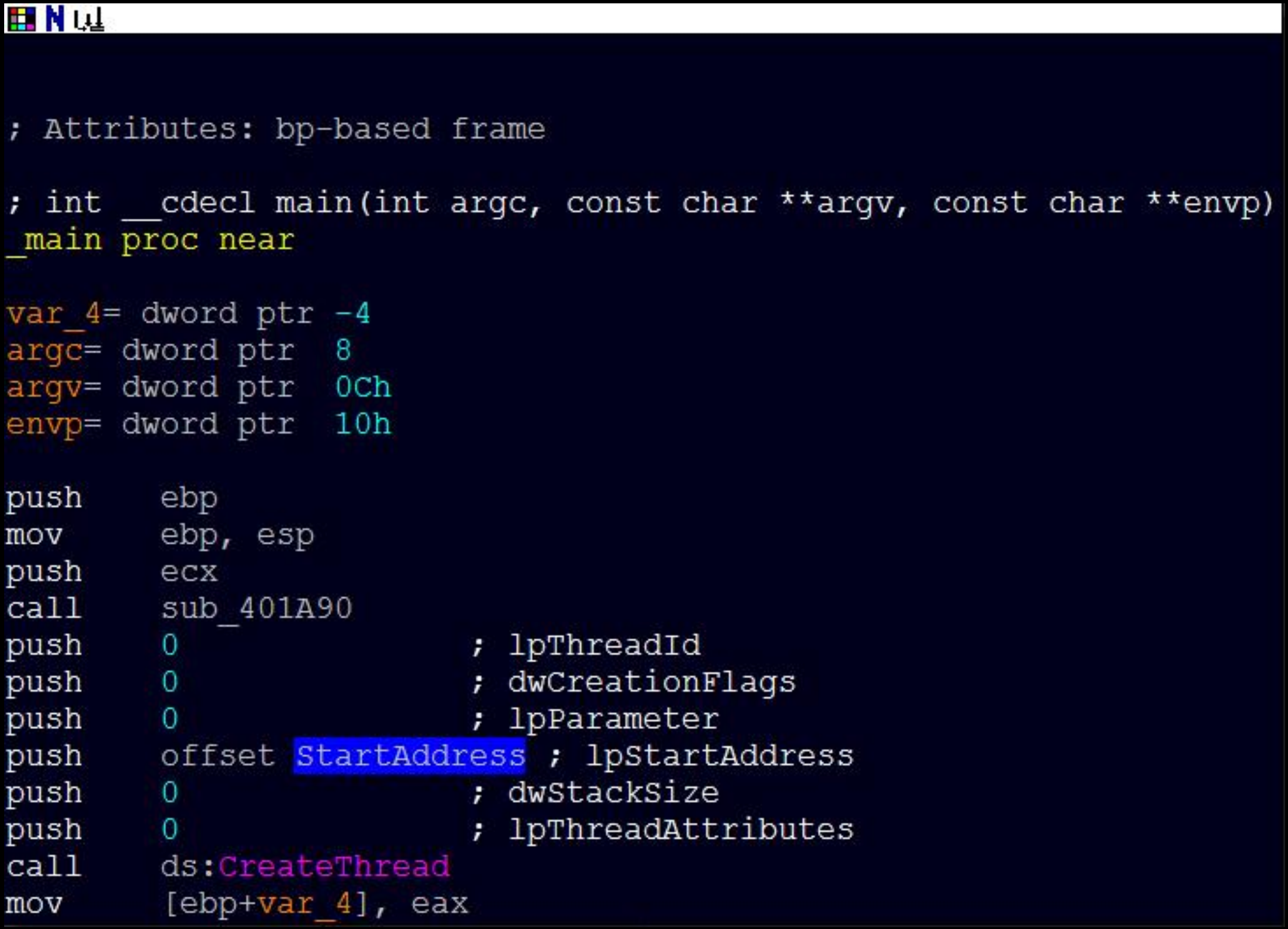
```
loc_1EA1:
lea     eax, [ebp+var_2724]
mov     edx, eax
mov     eax, 2710h
mov     [esp+8], eax
mov     dword ptr [esp+4], 0
mov     [esp], edx
call    _memset
mov     dword ptr [esp+8], 270Fh
lea     eax, [ebp+var_2724]
mov     [esp+4], eax
mov     eax, [ebp+var_C]
mov     [esp], eax
call    _read
mov     [ebp+var_14], eax
cmp     [ebp+var_14], 0FFFFFFFFh
jnz     short loc_1EFB
```

A

Reversing the Binary (Windows)

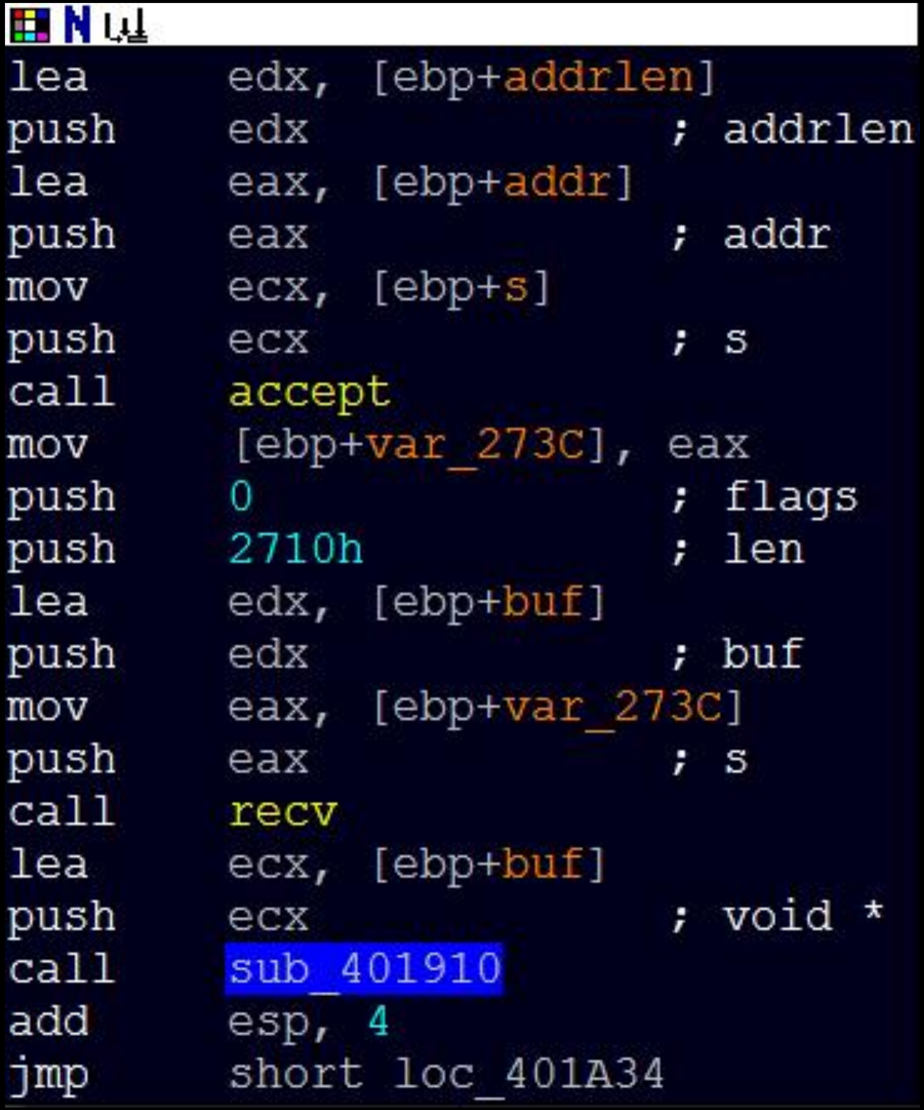
- Things to note
  - Uses threads
  - Symbol support (type information)

Reversing the Binary (Windows)



```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
push    ecx
call    sub_401A90
push    0               ; lpThreadId
push    0               ; dwCreationFlags
push    0               ; lpParameter
push    offset StartAddress ; lpStartAddress
push    0               ; dwStackSize
push    0               ; lpThreadAttributes
call    ds:CreateThread
mov     [ebp+var_4], eax
```

Reversing the Binary (Windows)



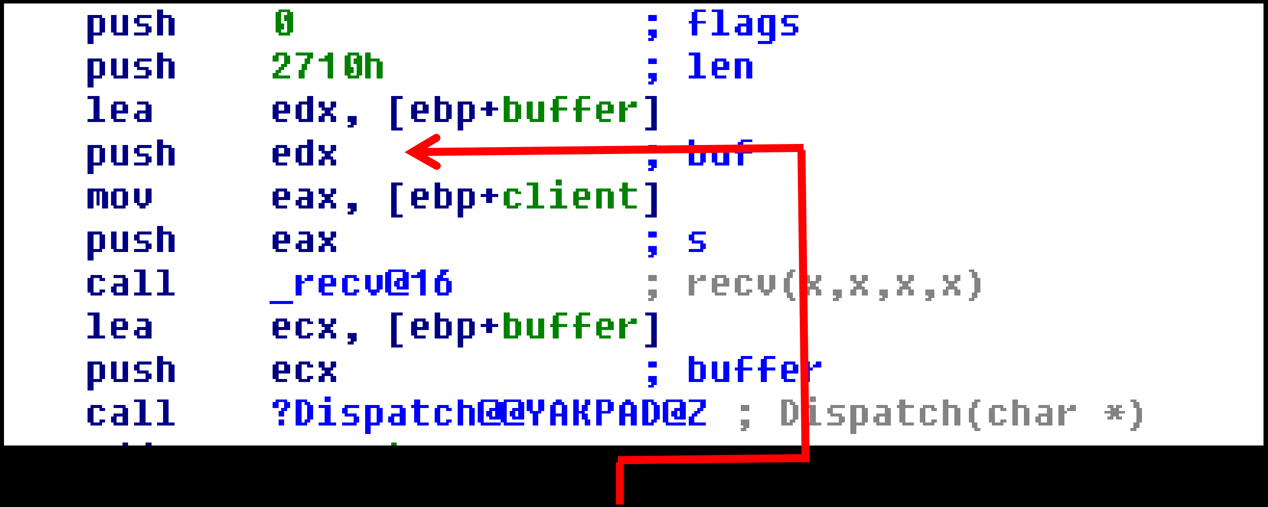
```
lea     edx, [ebp+addrlen]
push    edx                     ; addrlen
lea     eax, [ebp+addr]
push    eax                     ; addr
mov     ecx, [ebp+s]
push    ecx                     ; s
call    accept
mov     [ebp+var_273C], eax
push    0                       ; flags
push    2710h                   ; len
lea     edx, [ebp+buf]
push    edx                     ; buf
mov     eax, [ebp+var_273C]
push    eax                     ; s
call    recv
lea     ecx, [ebp+buf]
push    ecx                     ; void *
call    sub_401910
add     esp, 4
jmp     short loc_401A34
```

A

Reversing the Protocol

- Must identify where network program “recvs” network input
  - How the network input is manipulated or understood (protocol parsing)
    - Use cross references to identify network input points

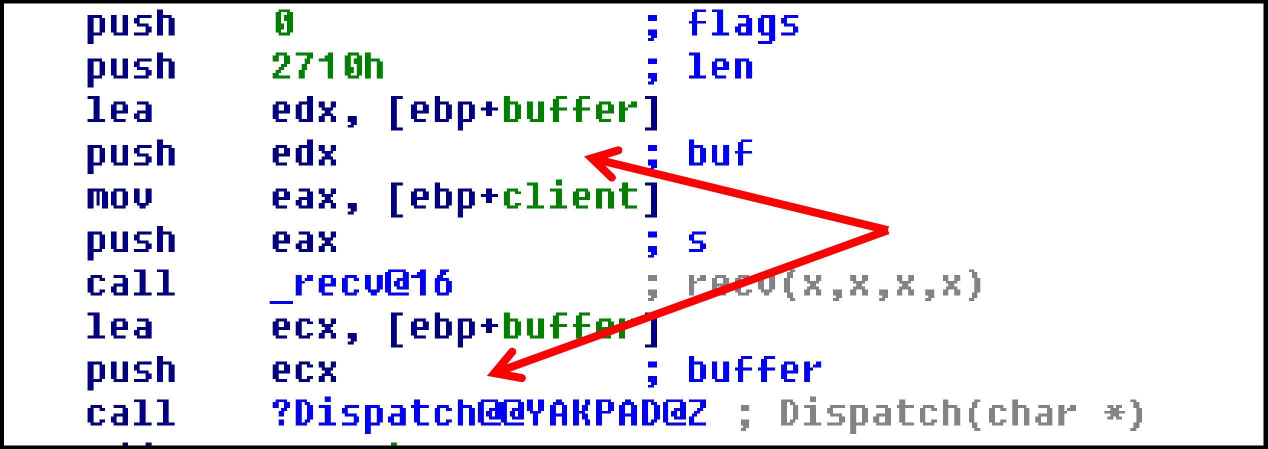
Reversing the Protocol



```
push    0                   ; flags
push    2710h               ; len
lea     edx, [ebp+buffer]
push    edx                 ; buf
mov     eax, [ebp+client]
push    eax                 ; s
call    _recv@16            ; recv(x,x,x,x)
lea     ecx, [ebp+buffer]
push    ecx                 ; buffer
call    ?Dispatch@@YAKPAD@Z ; Dispatch(char *)
```

- recv stores result in buffer

Reversing the Protocol

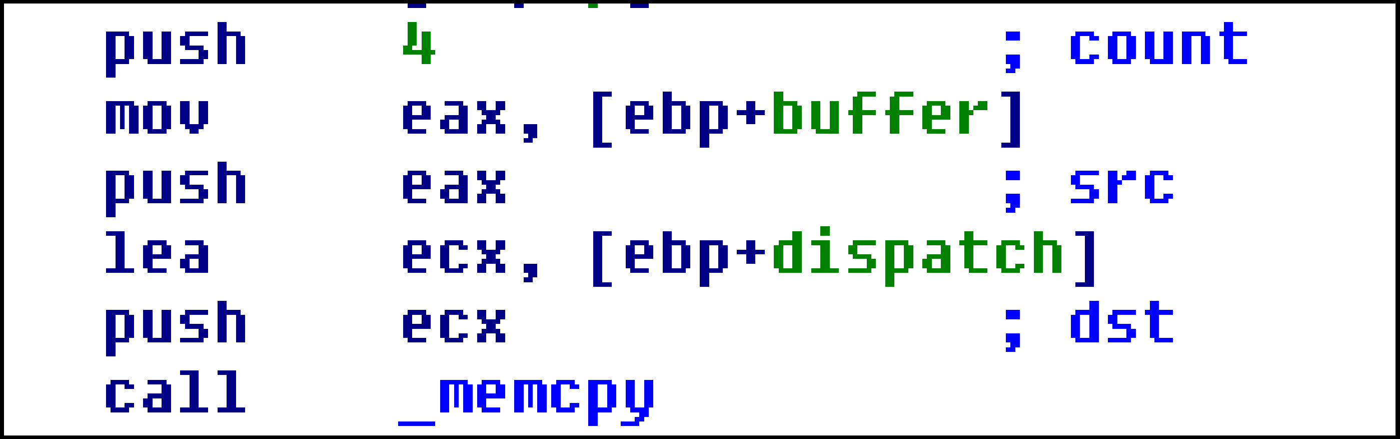


```
push    0                   ; flags
push    2710h               ; len
lea     edx, [ebp+buffer]
push    edx                 ; buf
mov     eax, [ebp+client]
push    eax                 ; s
call    _recv@16            ; recv(x,x,x,x)
lea     ecx, [ebp+buffer]
push    ecx                 ; buffer
call    ?Dispatch@@YAKPAD@Z ; Dispatch(char *)
```

- recv stores result in buffer
- buffer is passed to Dispatch function
  - Dispatch takes only one parameter

Reversing the Protocol

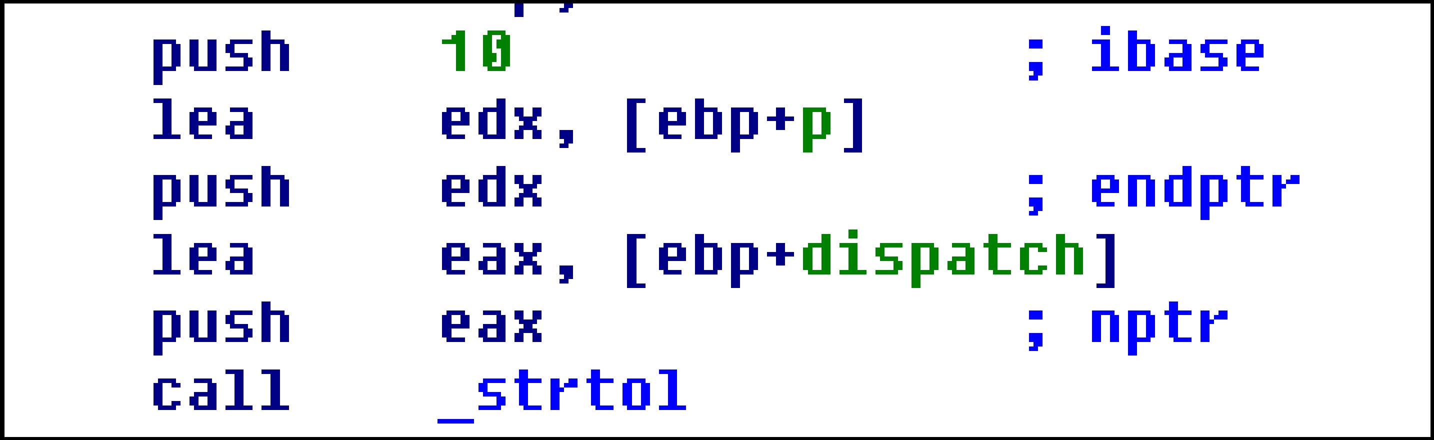
- Dispatch
  - Copies the first four bytes from the buffer into a separate buffer



```
push    4                       ; count
mov     eax, [ebp+buffer]
push    eax                     ; src
lea     ecx, [ebp+dispatch]
push    ecx                     ; dst
call    _memcpy
```

Reversing the Protocol

- Dispatch
  - Calls strtol on first four bytes of packet
    - long int strtol(const char * str, char ** endptr, int base);
      - Parses the C string *str* interpreting its content as an integral number of the specified *base*, which is returned as a long int value.



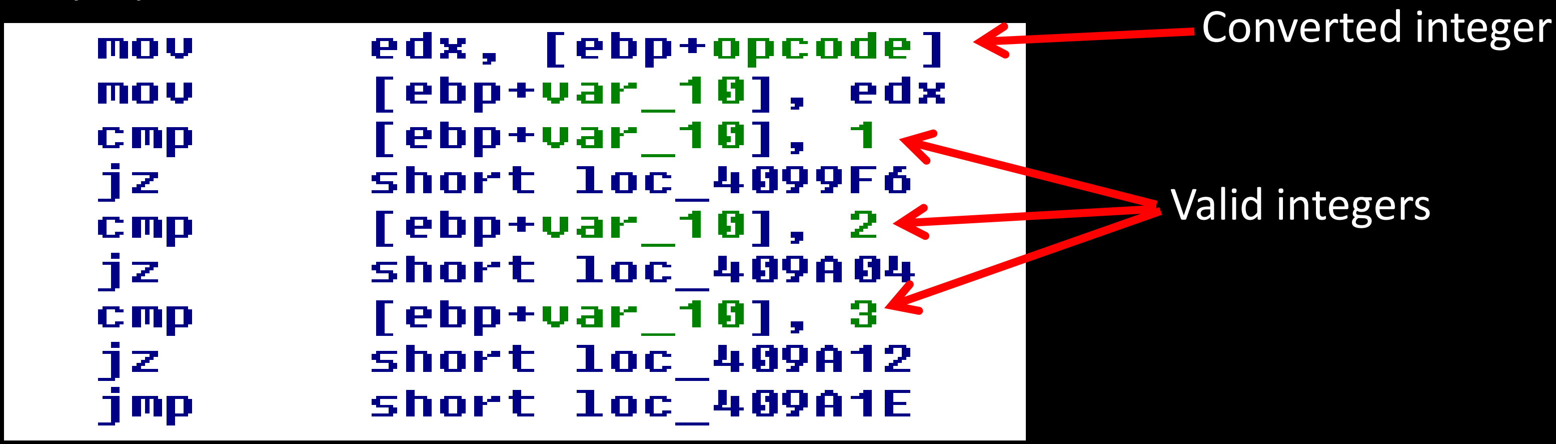
```
push    10                  ; ibase
lea     edx, [ebp+p]
push    edx                 ; endptr
lea     eax, [ebp+dispatch]
push    eax                 ; nptr
call    _strtol
```

Reversing the Protocol

- Dispatch
  - Recap:
    - Takes recv'd buffer
    - Parses out first four bytes
    - Converts first four bytes to long int
    - **Then enters a switch of valid integers**

Reversing the Protocol

- Dispatch
  - Protocol accepts the following integers:
    - 1, 2, 3



```
mov     edx, [ebp+opcode]
mov     [ebp+var_10], edx
cmp     [ebp+var_10], 1
jz      short loc_4099F6
cmp     [ebp+var_10], 2
jz      short loc_409A04
cmp     [ebp+var_10], 3
jz      short loc_409A12
jmp     short loc_409A1E
```

Converted integer

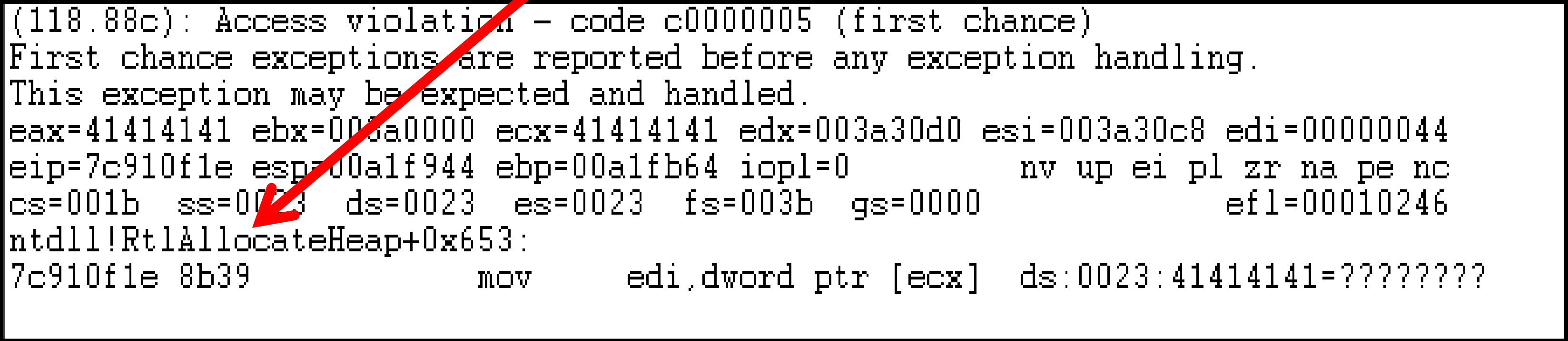
Valid integers

Opcode: 1

- Where do you want to set a breakpoint?
- Payload:
  [opcode][size][string]
- Judging from disassembly, payload and crash what do we suspect the vulnerability to be?

Crash…

- Where did it crash:

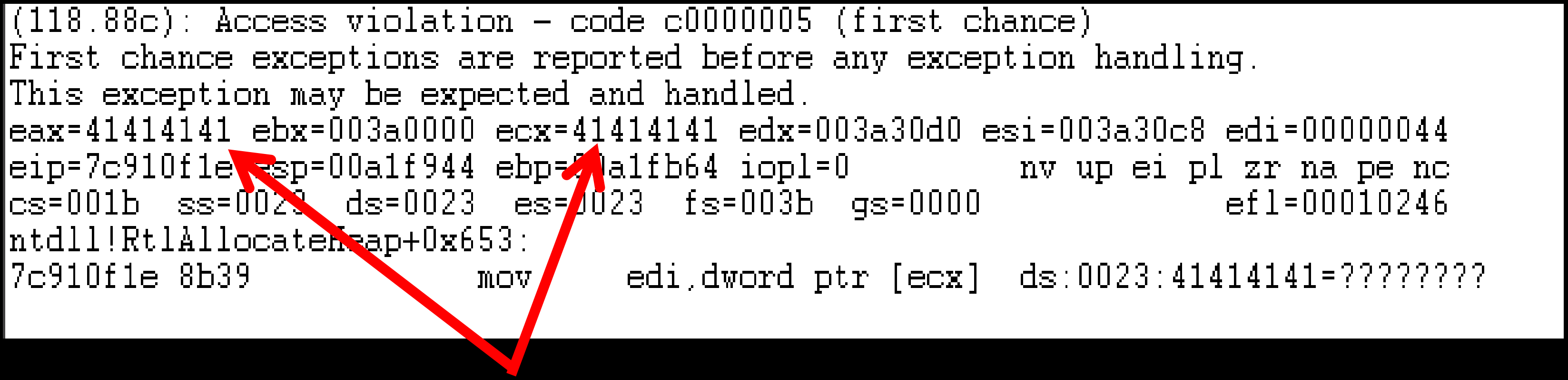


```
(118.88c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=003a0000 ecx=41414141 edx=003a30d0 esi=003a30c8 edi=00000044
eip=7c910f1e esp=00a1f944 ebp=00a1fb64 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
ntdll!RtlAllocateHeap+0x653:
7c910f1e 8b39            mov     edi,dword ptr [ecx]  ds:0023:41414141=????????
```

- Heap Related?

Crash…

- Where did it crash:

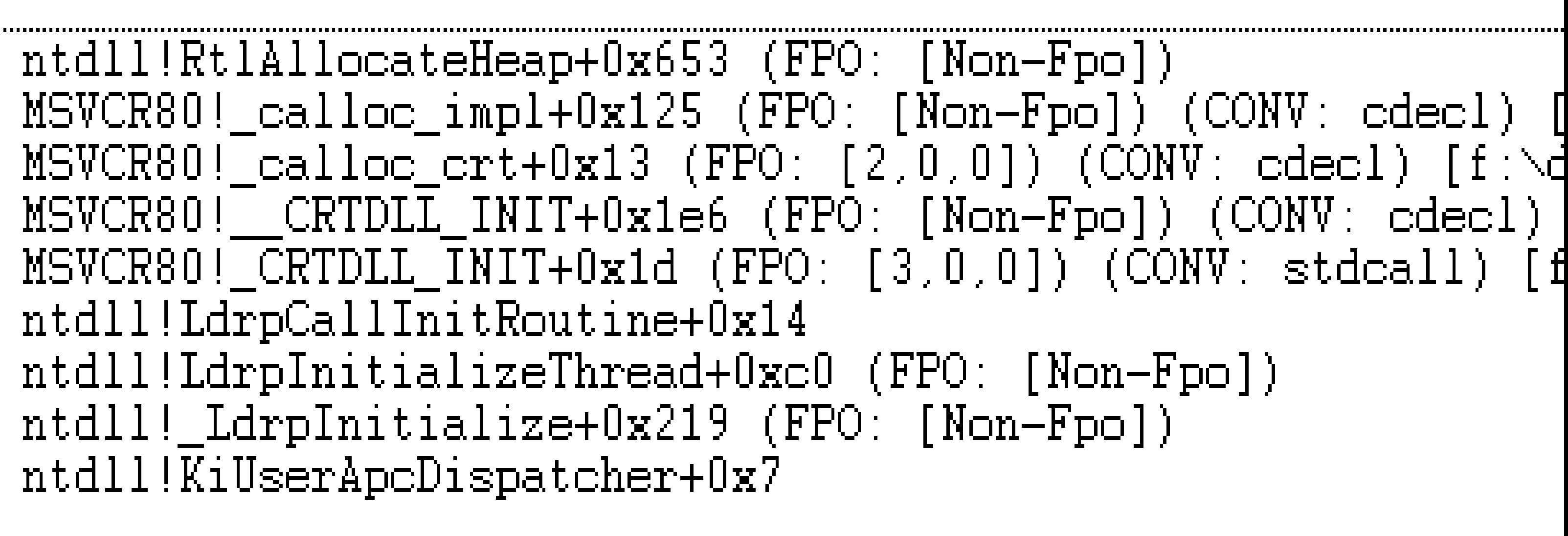


```
(118.88c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=003a0000 ecx=41414141 edx=003a30d0 esi=003a30c8 edi=00000044
eip=7c910f1e esp=00a1f944 ebp=00a1fb64 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
ntdll!RtlAllocateHeap+0x653:
7c910f1e 8b39            mov     edi,dword ptr [ecx]  ds:0023:41414141=????????
```

- What do our registers contain?

Crash…

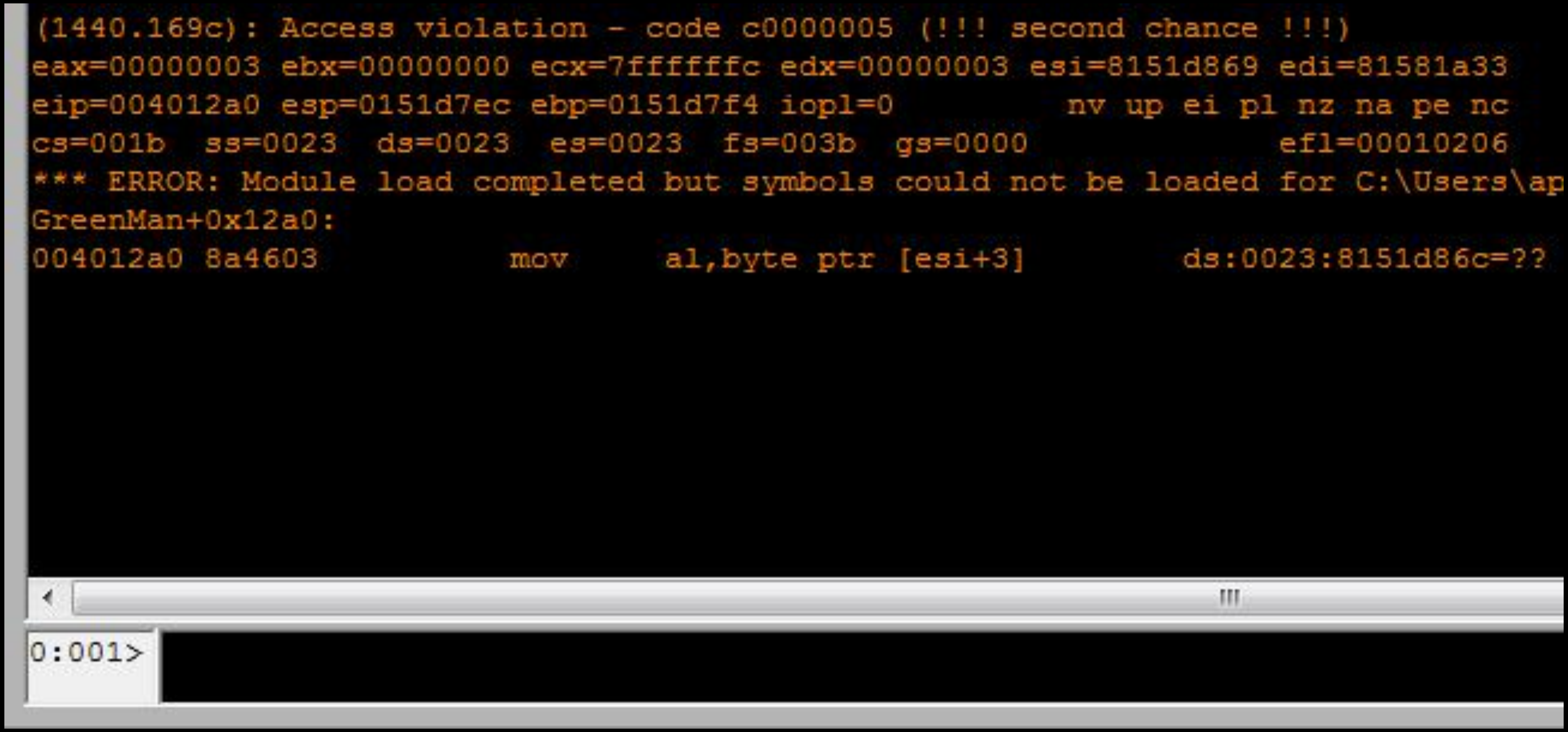
- Check the call stack for hints



```
ntdll!RtlAllocateHeap+0x653 (FPO: [Non-Fpo])
MSVCR80!_calloc_impl+0x125 (FPO: [Non-Fpo]) (CONV: cdecl) [
MSVCR80!_calloc_crt+0x13 (FPO: [2,0,0]) (CONV: cdecl) [f:\d
MSVCR80!__CRTDLL_INIT+0x1e6 (FPO: [Non-Fpo]) (CONV: cdecl)
MSVCR80!_CRTDLL_INIT+0x1d (FPO: [3,0,0]) (CONV: stdcall) [f
ntdll!LdrpCallInitRoutine+0x14
ntdll!LdrpInitializeThread+0xc0 (FPO: [Non-Fpo])
ntdll!_LdrpInitialize+0x219 (FPO: [Non-Fpo])
ntdll!KiUserApcDispatcher+0x7
```

Crash…

- Depending on heap state and frequency of heap operations, different crashes can occur

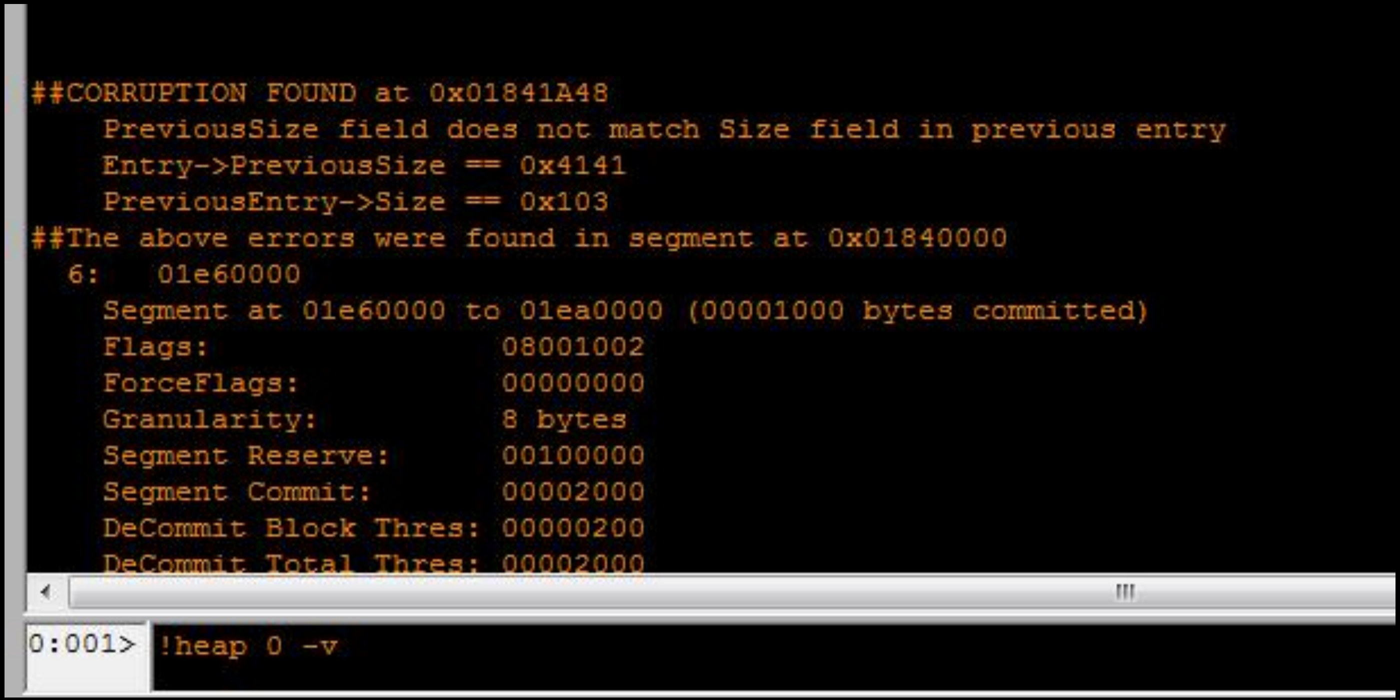


```
(1440.169c): Access violation - code c0000005 (!!! second chance !!!)
eax=00000003 ebx=00000000 ecx=7ffffffc edx=00000003 esi=8151d869 edi=81581a33
eip=004012a0 esp=0151d7ec ebp=0151d7f4 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
*** ERROR: Module load completed but symbols could not be loaded for C:\Users\ap
GreenMan+0x12a0:
004012a0 8a4603          mov     al,byte ptr [esi+3]        ds:0023:8151d86c=??
```

```
0:001>
```

Crash…

- On Windows, we can check the heap integrity…
  - !heap 0 -v



```
##CORRUPTION FOUND at 0x01841A48
    PreviousSize field does not match Size field in previous entry
    Entry->PreviousSize == 0x4141
    PreviousEntry->Size == 0x103
##The above errors were found in segment at 0x01840000
  6:   01e60000
    Segment at 01e60000 to 01ea0000 (00001000 bytes committed)
    Flags:                08001002
    ForceFlags:           00000000
    Granularity:          8 bytes
    Segment Reserve:      00100000
    Segment Commit:       00002000
    DeCommit Block Thres: 00000200
    DeCommit Total Thres: 00002000
```

```
0:001> !heap 0 -v
```

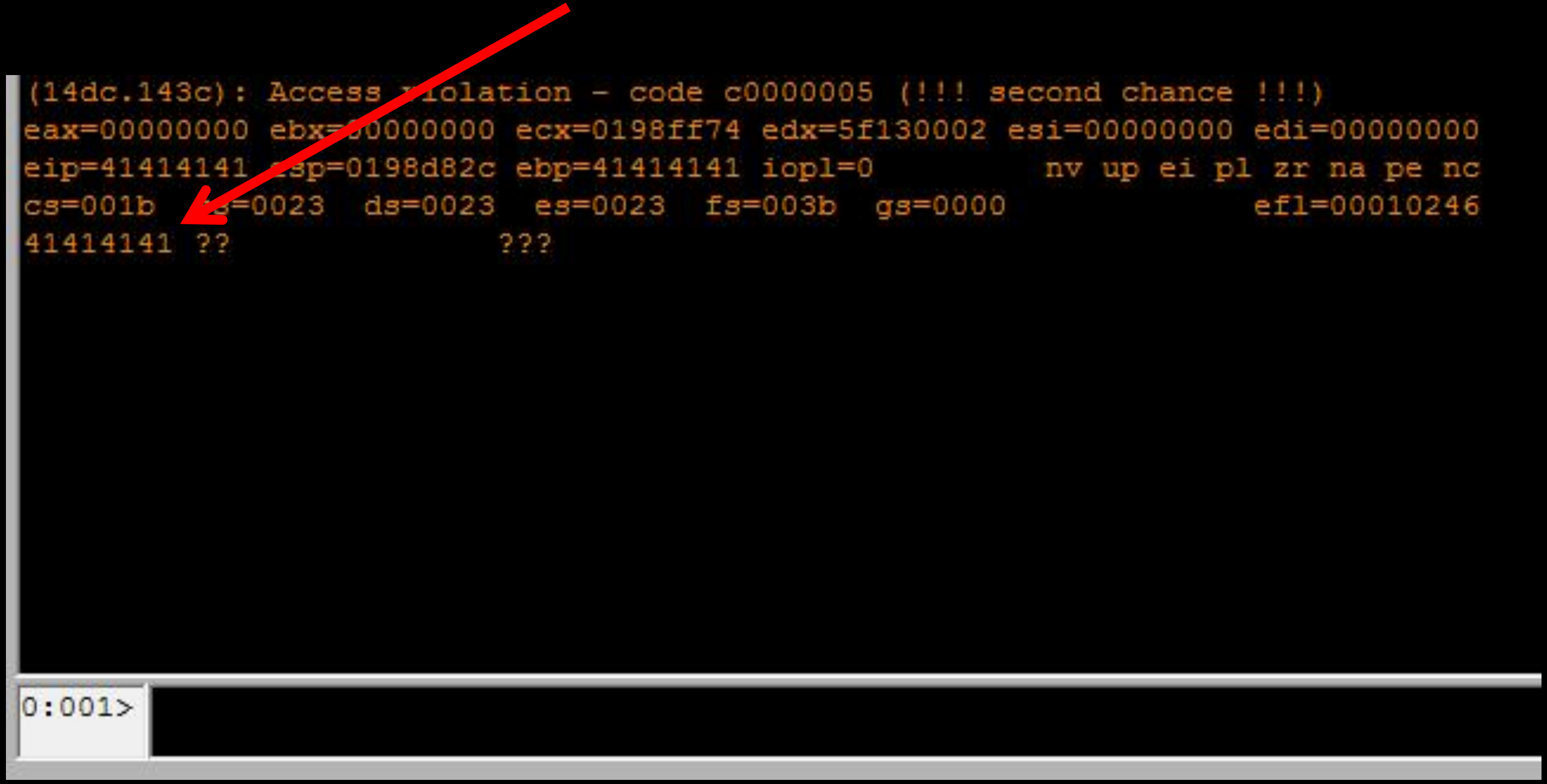
Opcode: 2

- Payload:
  [opcode] [string]

- Judging from disassembly, payload and crash what do we suspect the vulnerability to be?

Crash…

- Where did it crash:

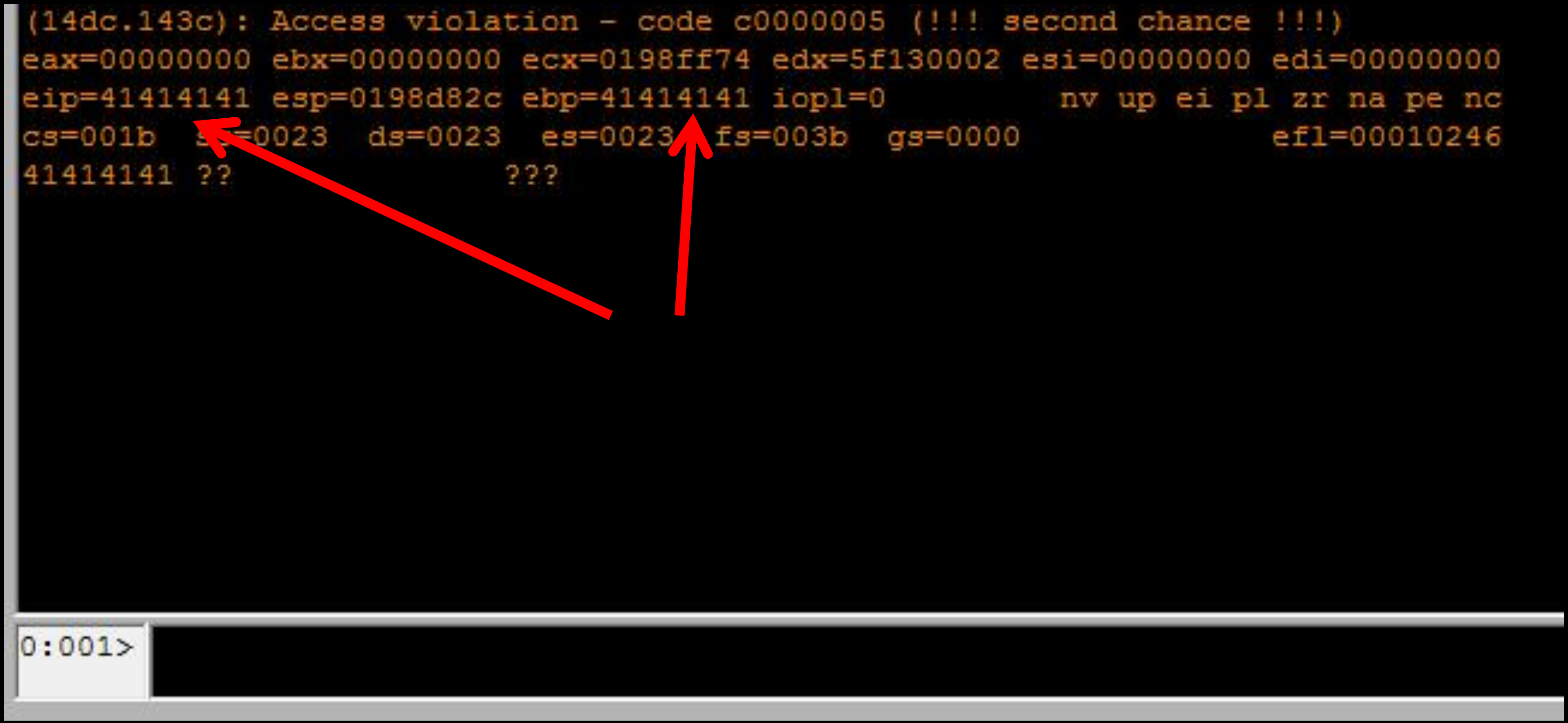


```
(14dc.143c): Access violation - code c0000005 (!!! second chance !!!)
eax=00000000 ebx=00000000 ecx=0198ff74 edx=5f130002 esi=00000000 edi=00000000
eip=41414141 esp=0198d82c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

```
0:001>
```

Crash…

- What do our registers contain?

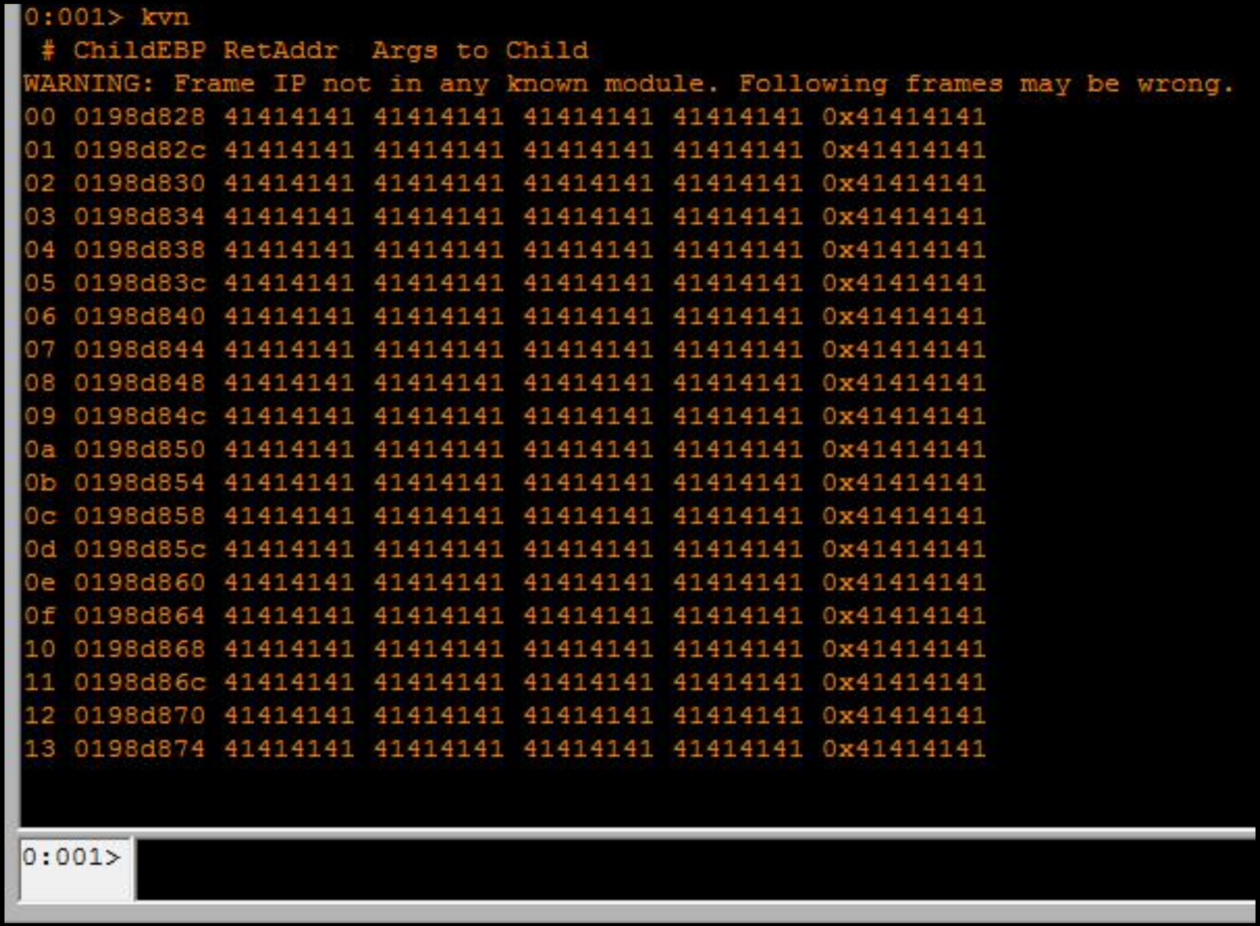


```
(14dc.143c): Access violation - code c0000005 (!!! second chance !!!)
eax=00000000 ebx=00000000 ecx=0198ff74 edx=5f130002 esi=00000000 edi=00000000
eip=41414141 esp=0198d82c ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

```
0:001>
```

Crash…

- What does our call stack look like



```
0:001> kvn
 # ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 0198d828 41414141 41414141 41414141 41414141 0x41414141
01 0198d82c 41414141 41414141 41414141 41414141 0x41414141
02 0198d830 41414141 41414141 41414141 41414141 0x41414141
03 0198d834 41414141 41414141 41414141 41414141 0x41414141
04 0198d838 41414141 41414141 41414141 41414141 0x41414141
05 0198d83c 41414141 41414141 41414141 41414141 0x41414141
06 0198d840 41414141 41414141 41414141 41414141 0x41414141
07 0198d844 41414141 41414141 41414141 41414141 0x41414141
08 0198d848 41414141 41414141 41414141 41414141 0x41414141
09 0198d84c 41414141 41414141 41414141 41414141 0x41414141
0a 0198d850 41414141 41414141 41414141 41414141 0x41414141
0b 0198d854 41414141 41414141 41414141 41414141 0x41414141
0c 0198d858 41414141 41414141 41414141 41414141 0x41414141
0d 0198d85c 41414141 41414141 41414141 41414141 0x41414141
0e 0198d860 41414141 41414141 41414141 41414141 0x41414141
0f 0198d864 41414141 41414141 41414141 41414141 0x41414141
10 0198d868 41414141 41414141 41414141 41414141 0x41414141
11 0198d86c 41414141 41414141 41414141 41414141 0x41414141
12 0198d870 41414141 41414141 41414141 41414141 0x41414141
13 0198d874 41414141 41414141 41414141 41414141 0x41414141
```

```
0:001>
```

Opcode: 3

- Payload:
  [opcode] [specific string]

- Judging from disassembly, payload and crash what do we suspect the vulnerability to be?

Conclusion of Session II

Questions?

E-mail the mailing list if you have additional questions

We are subscribed as well

Alternatively

Our gmail usernames are aportnoy and petersilberman

Thanks!