Reverse Engineering

Mitchell Adair
1/22/2013

About Me

- Know Owen from our time at Sandia National Labs
- Currently work for Raytheon
- Founded UTDallas's Computer Security Group (CSG) in Spring 2010
- Reversing, binary auditing, fuzzing, exploit dev, pen testing…
- Python

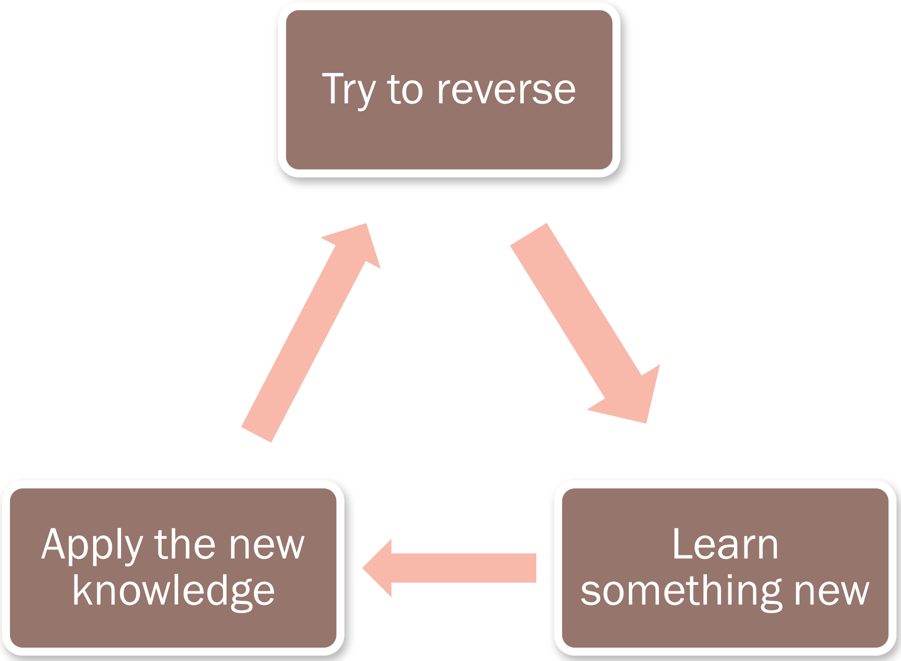
:P

Goal

- At the end of this, you should feel comfortable
  - Being handed a binary
  - Examining a binaries sections, imports, strings
  - Renaming and simplifying the disassembly
  - Converting from assembly to source, where needed
  - Understanding process memory layout
  - Figuring out function arguments and local variables
    - How many and what types
  - Using a debugger to fill in the gaps or manipulate program execution

Outline

- Static vs Dynamic (overview)
- PE and ELF
- Assembly
- Registers
- The Stack
- Functions
- IDA
- Debugging
- Note on Bytecode
- Conclusion



Try to reverse

Learn something new

Apply the new knowledge

Static vs Dynamic

Static vs Dynamic - Overview

- Static
  - Looking at the code, figure things out
  - It’s all there, but possibly more complicated
  - A safer approach
    - Not running the code!
- Dynamic
  - Examine the process during execution
  - Can see the values in real time
    - Registers, memory contents, etc.
  - Allows manipulation of the process
  - Should run in a VM!

Static vs Dynamic - Tools

- Disassemblers are usually the tool of choice for static
  - IDA Pro, objdump, etc.

- Debuggers are used for dynamic analysis
  - Windows
    - WinDBG, Immunity, OllyDBG, IDA
  - Linux
    - GDB

Static vs Dynamic - Tools

- A good disassembler will have several useful features
  - Commenting
  - Renaming variables
  - Changing function prototypes
  - Coloring, grouping and renaming nodes (IDA)
  - …

- A good debugger will have several useful features
  - Set breakpoints
  - Step into / over
  - Show loaded modules, SEH chain, etc.
  - Memory searching
  - …

Static vs Dyamic

- Okay, no more!
- We’ll be going into each of these heavily.
- That was just a high level overview to understand
  - The difference between static and dynamic analysis
  - The general approach taken between the two

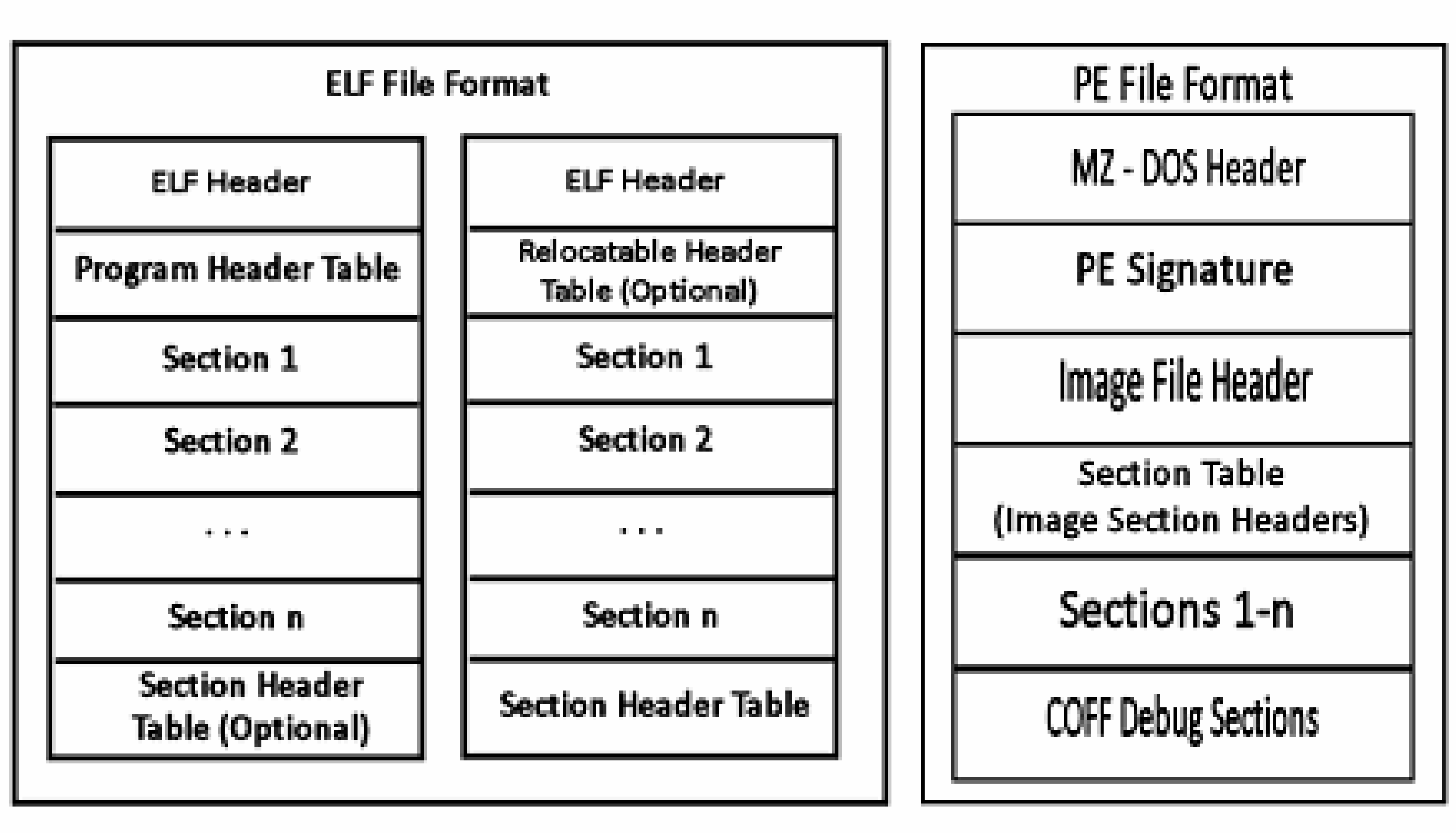
PE and ELF

PE and ELF

- PE (Portable Executable)
  - “File format for executables, object code and DLLs, used in 32-bit and 64-bit versions of **Windows operating systems**” – *wikipedia*

- ELF (Executable and Linkable Format)
  - “A common standard file format for executables, object code, shared libraries, and core dumps” – *wikipedia*
  - Linux, Unix, Apple OS

PE and ELF



ELF File Format

ELF Header

Program Header Table

Section 1

Section 2

. . .

Section n

Section Header Table (Optional)

ELF Header

Relocatable Header Table (Optional)

Section 1

Section 2

. . .

Section n

Section Header Table

PE File Format

MZ - DOS Header

PE Signature

Image File Header

Section Table (Image Section Headers)

Sections 1-n

COFF Debug Sections

Image from http://software.intel.com/sites/default/files/m/d/4/1/d/8/keep-memory-002.gif

PE and ELF

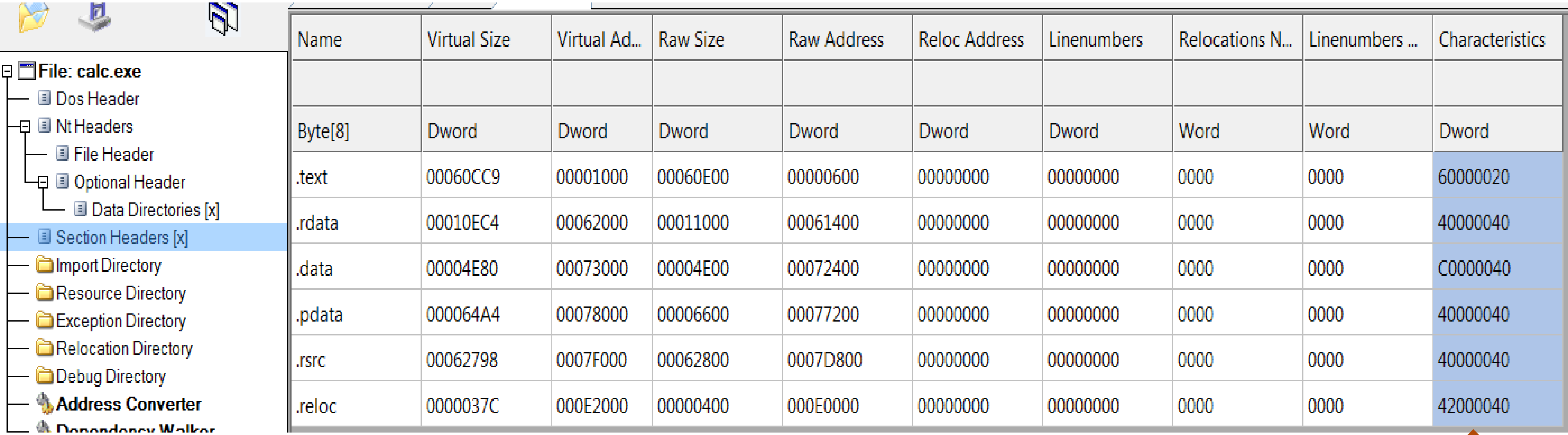
- We could go very, very deep into file formats… but let’s not
- Each format is just a big collection of fields and sections
- Fields will have a particular meaning and hold a particular value
  - Date created, last modified, number of sections, image base, etc.
- A section is, generally, a logical collection of code or data
  - Has permissions (read/write/execute)
  - Has a name (.text, .bss, etc.)

PE and ELF

- Okay, so what? Why is this useful?
- Can get an overview of what the binary is doing
  - Can look at what libraries the binary is loading
  - Can look at what functions are used in a library
    - Find vulns
  - Can parse data sections for strings
    - Very helpful on CTFs
  - Can help determine if a binary is packed
    - Weird section names or sizes, lack of strings, lack of imports
- How do we analyze them?
  - PE : CFF Explorer, IDA, pefile (python library), ...
  - ELF : *readelf*, *objdump*, *file*, ...

PE – CFF Explorer

- This is CFF Explorer looking at calc.exe's sections headers

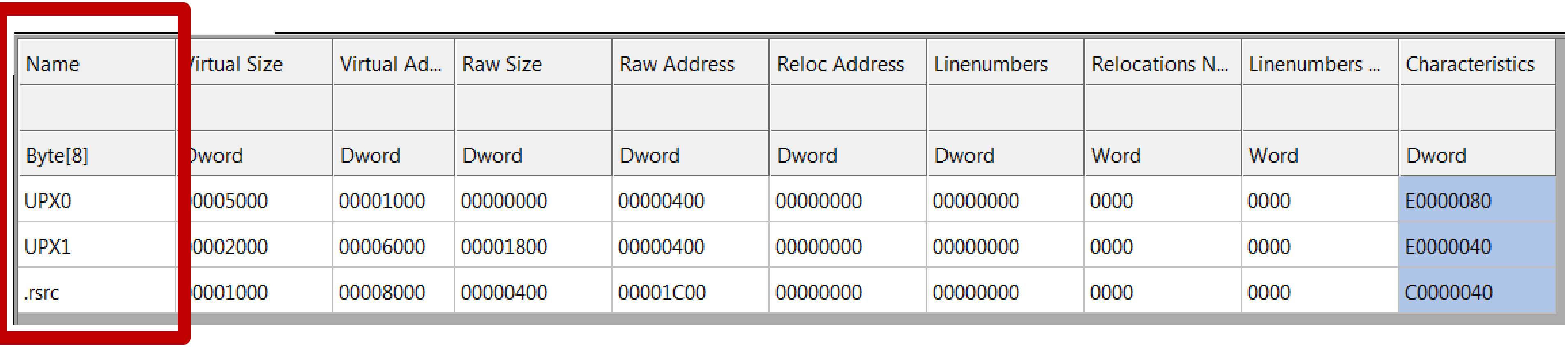


Name	Virtual Size	Virtual Ad...	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00060CC9	00001000	00060E00	00000600	00000000	00000000	0000	0000	60000020
.rdata	00010EC4	00062000	00011000	00061400	00000000	00000000	0000	0000	40000040
.data	00004E80	00073000	00004E00	00072400	00000000	00000000	0000	0000	C0000040
.pdata	000064A4	00078000	00006600	00077200	00000000	00000000	0000	0000	40000040
.rsrc	00062798	0007F000	00062800	0007D800	00000000	00000000	0000	0000	40000040
.reloc	0000037C	000E2000	00000400	000E0000	00000000	00000000	0000	0000	42000040

Represent permissions

PE – CFF Explorer

- This is CFF Explorer looking at a UPX packed executable from a recent CTF

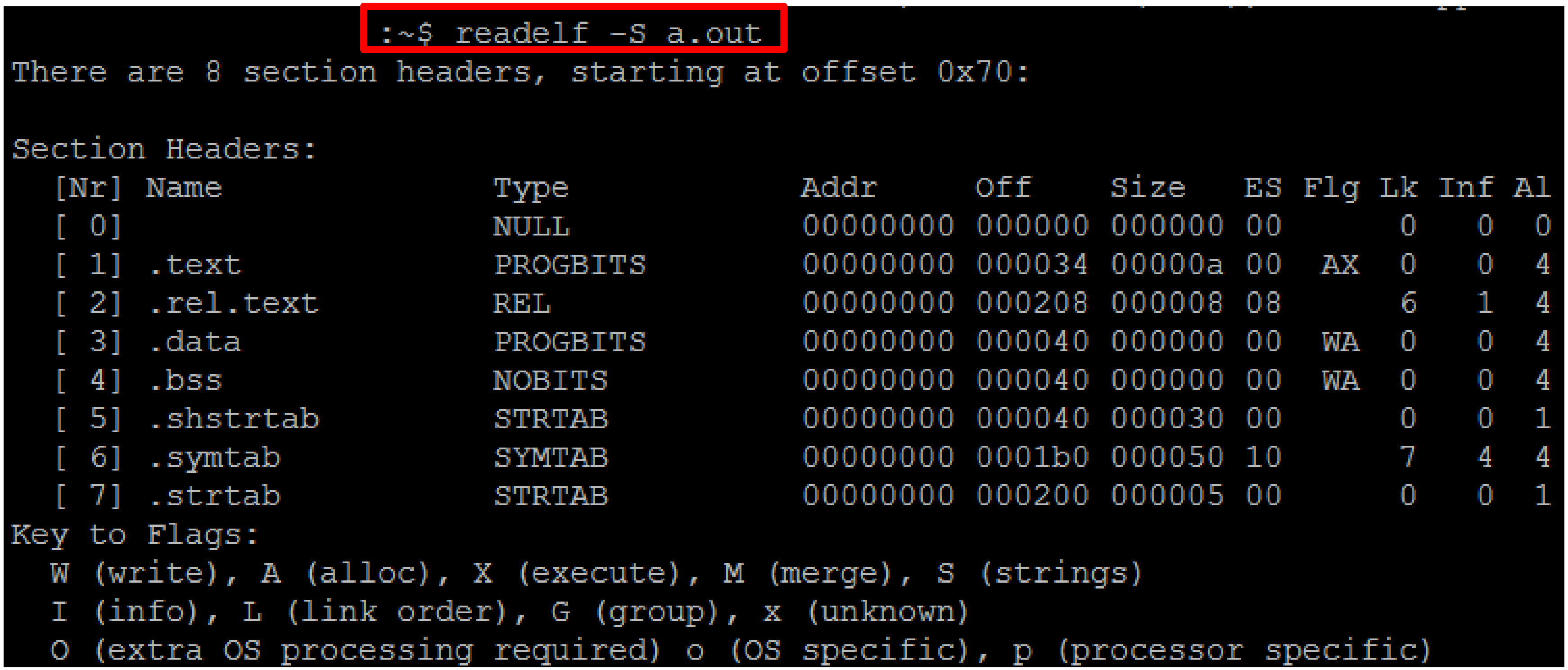


Name	irtual Size	Virtual Ad...	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	word	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
UPX0	0005000	00001000	00000000	00000400	00000000	00000000	0000	0000	E0000080
UPX1	0002000	00006000	00001800	00000400	00000000	00000000	0000	0000	E0000040
.rsrc	0001000	00008000	00000400	00001C00	00000000	00000000	0000	0000	C0000040

- Huge red flag with section names like this

ELF - readelf

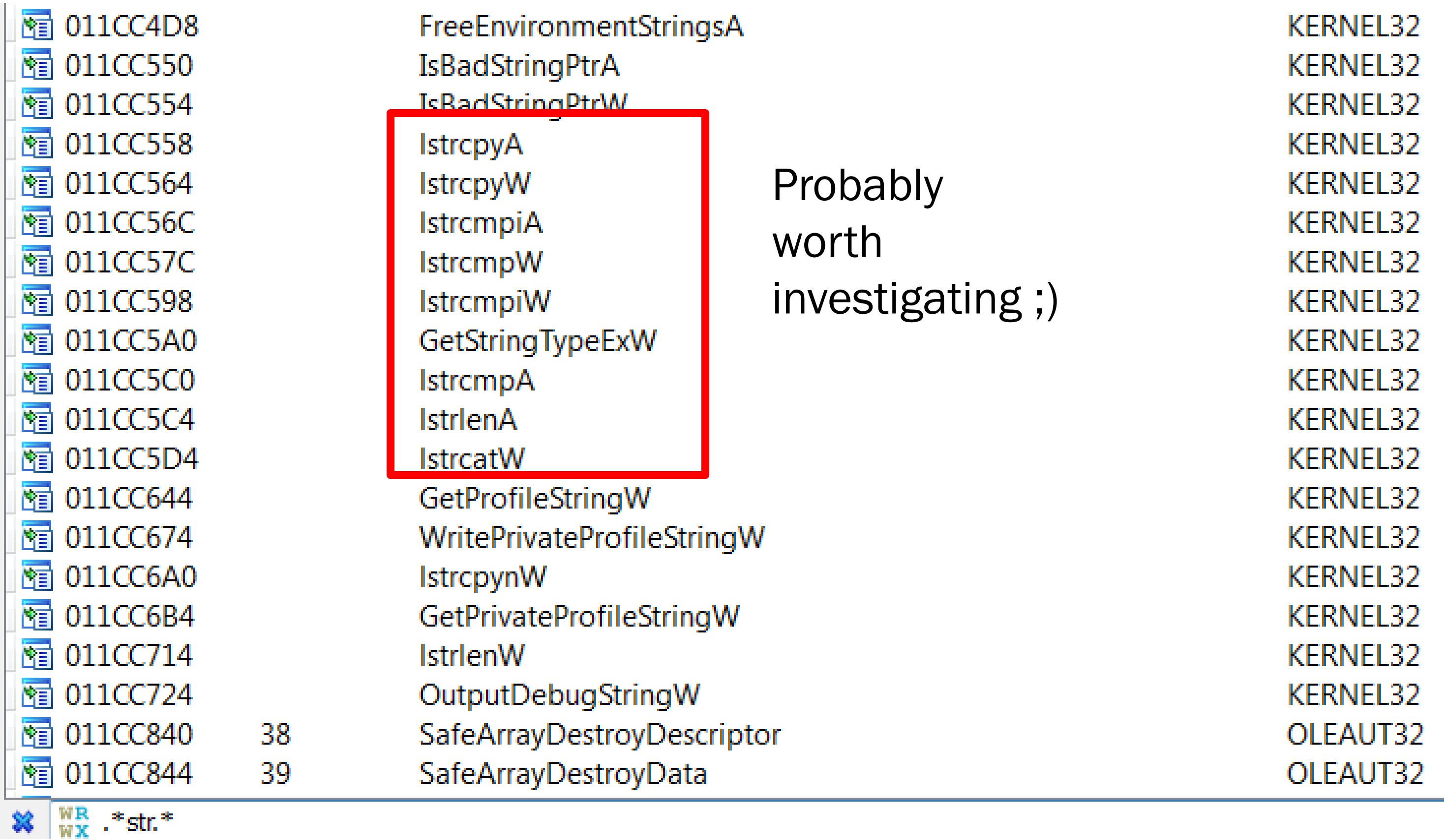
- This is using *readelf* to look at section headers



```
        :~$ readelf -S a.out
There are 8 section headers, starting at offset 0x70:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 00000a 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000208 000008 08      6   1  4
  [ 3] .data             PROGBITS        00000000 000040 000000 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 000040 000000 00  WA  0   0  4
  [ 5] .shstrtab         STRTAB          00000000 000040 000030 00      0   0  1
  [ 6] .symtab           SYMTAB          00000000 0001b0 000050 10      7   4  4
  [ 7] .strtab           STRTAB          00000000 000200 000005 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

PE and ELF - Imports

- This is IDA exemaning what functions are imported
- I have filtered using the regular expression .*str.*



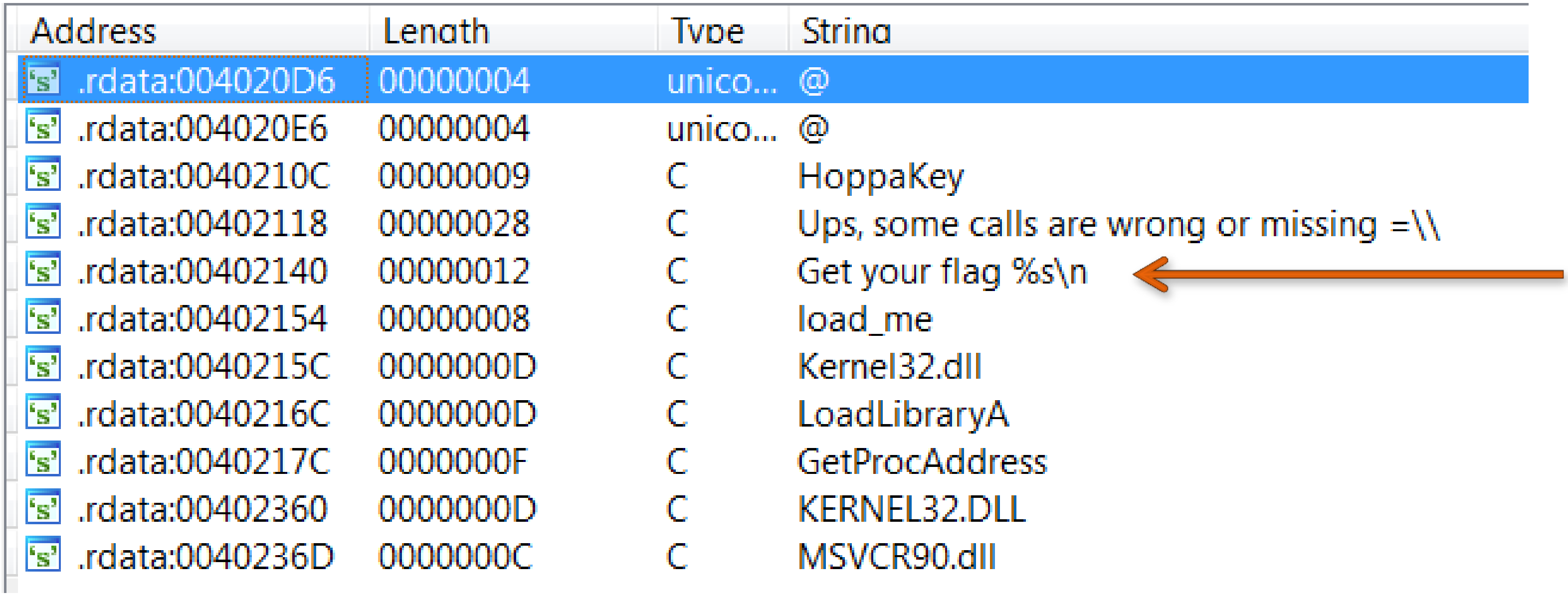
011CC4D8		FreeEnvironmentStringsA	KERNEL32
011CC550		IsBadStringPtrA	KERNEL32
011CC554		IsBadStringPtrW	KERNEL32
011CC558		lstrcpyA	KERNEL32
011CC564		lstrcpyW	KERNEL32
011CC56C		lstrcmpiA	KERNEL32
011CC57C		lstrcmpW	KERNEL32
011CC598		lstrcmpiW	KERNEL32
011CC5A0		GetStringTypeExW	KERNEL32
011CC5C0		lstrcmpA	KERNEL32
011CC5C4		lstrlenA	KERNEL32
011CC5D4		lstrcatW	KERNEL32
011CC644		GetProfileStringW	KERNEL32
011CC674		WritePrivateProfileStringW	KERNEL32
011CC6A0		lstrcpynW	KERNEL32
011CC6B4		GetPrivateProfileStringW	KERNEL32
011CC714		lstrlenW	KERNEL32
011CC724		OutputDebugStringW	KERNEL32
011CC840	38	SafeArrayDestroyDescriptor	OLEAUT32
011CC844	39	SafeArrayDestroyData	OLEAUT32

.*str.*

Probably worth investigating ;)

PE and ELF - Strings

- This is IDA examining strings it has found for a recent CTF problem



Address	Length	Type	String
.rdata:004020D6	00000004	unico...	@
.rdata:004020E6	00000004	unico...	@
.rdata:0040210C	00000009	C	HoppaKey
.rdata:00402118	00000028	C	Ups, some calls are wrong or missing =\\
.rdata:00402140	00000012	C	Get your flag %s\n
.rdata:00402154	00000008	C	load_me
.rdata:0040215C	0000000D	C	Kernel32.dll
.rdata:0040216C	0000000D	C	LoadLibraryA
.rdata:0040217C	0000000F	C	GetProcAddress
.rdata:00402360	0000000D	C	KERNEL32.DLL
.rdata:0040236D	0000000C	C	MSVCR90.dll

- Probably want to start from the “Get your flag %s\n” string and work backwards ;)

PE and ELF – 5 minute exercise

- Open number_checker.exe and number_checker_packed.exe
- Compare these two!
- In CFF Explorer
  - Look at different fields in the PE format
  - Look at sections
  - Just explore
- In IDA
  - Look at strings (shift+f12)
  - Look at imports (view->open subviews->imports)
  - Look at sections (shift+f7)

Assembly

Assembly

- Two syntax options
  - ATT
  - Intel
- ATT
  - instruction source, dest
  - mov %eax, %edx
  - “Move eax into edx”
- Intel
  - instruction dest, source
  - mov edx, eax
  - “Move into edx, eax”

Assembly

- It’s a known fact that Intel’s syntax > ATT’s, so we’ll be using Intels ;)

- mov eax, ecx
  - Move into eax, the contents of ecx

- mov eax, [ecx]
  - Move into eax, the contents of what ecx **points to**
  - The brackets, […], mean dereference the value between them
  - In C, this is like a pointer dereference
  - eax = *ecx

Assembly

- Memory values and immediates can be used as well

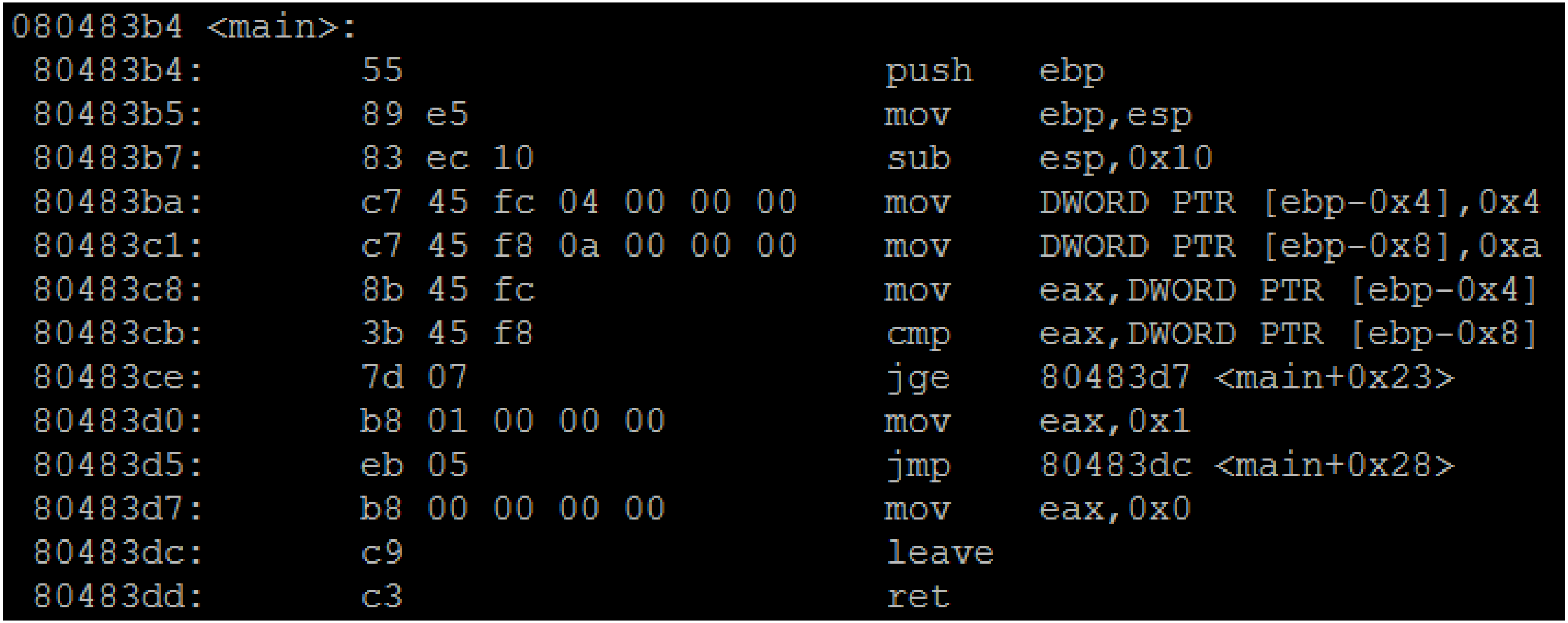
- mov eax, 5
  - Move into eax, the value 5

- mov edx, [0x12345678]
  - Move into edx, what 0x12345678 points to

Assembly

- A very small handful of instructions will get you a long way
  - call, mov, cmp, jmp
- call 0x12345678
  - Call the function at 0x12345678
- cmp eax, 8
  - Compare eax to 8
  - Compare left to right
- jmp 0x12345678
  - Unconditional jump to 0x12345678
- jle 0x12345678
  - Jump to 0x12345678 if eax is less than or equal to 8
- jg 0x12345678
  - Jump to 0x112345678 if eax is greater than 8

Assembly – Example



```
080483b4 <main>:
 80483b4:       55                      push   ebp
 80483b5:       89 e5                   mov    ebp,esp
 80483b7:       83 ec 10                sub    esp,0x10
 80483ba:       c7 45 fc 04 00 00 00    mov    DWORD PTR [ebp-0x4],0x4
 80483c1:       c7 45 f8 0a 00 00 00    mov    DWORD PTR [ebp-0x8],0xa
 80483c8:       8b 45 fc                mov    eax,DWORD PTR [ebp-0x4]
 80483cb:       3b 45 f8                cmp    eax,DWORD PTR [ebp-0x8]
 80483ce:       7d 07                   jge    80483d7 <main+0x23>
 80483d0:       b8 01 00 00 00          mov    eax,0x1
 80483d5:       eb 05                   jmp    80483dc <main+0x28>
 80483d7:       b8 00 00 00 00          mov    eax,0x0
 80483dc:       c9                      leave
 80483dd:       c3                      ret
```

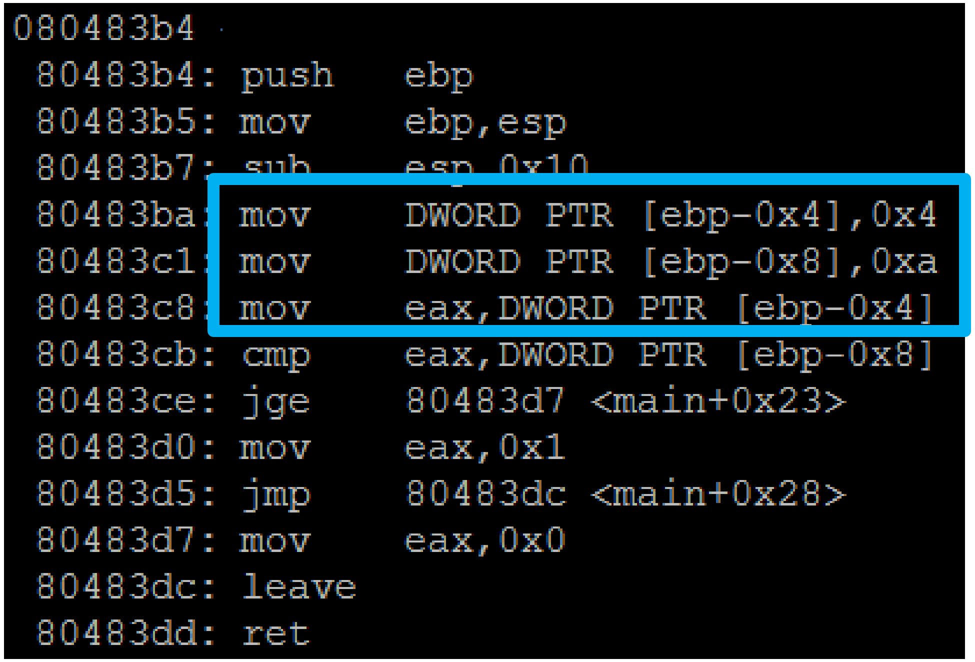
Assembly - Example

- Let’s focus on the instructions we know
  - mov, cmp, jmp, call

Example 1

- [ebp-0x4] = 0x4
- [ebp-0x8] = 0xa
- eax = [ebp-0x4]

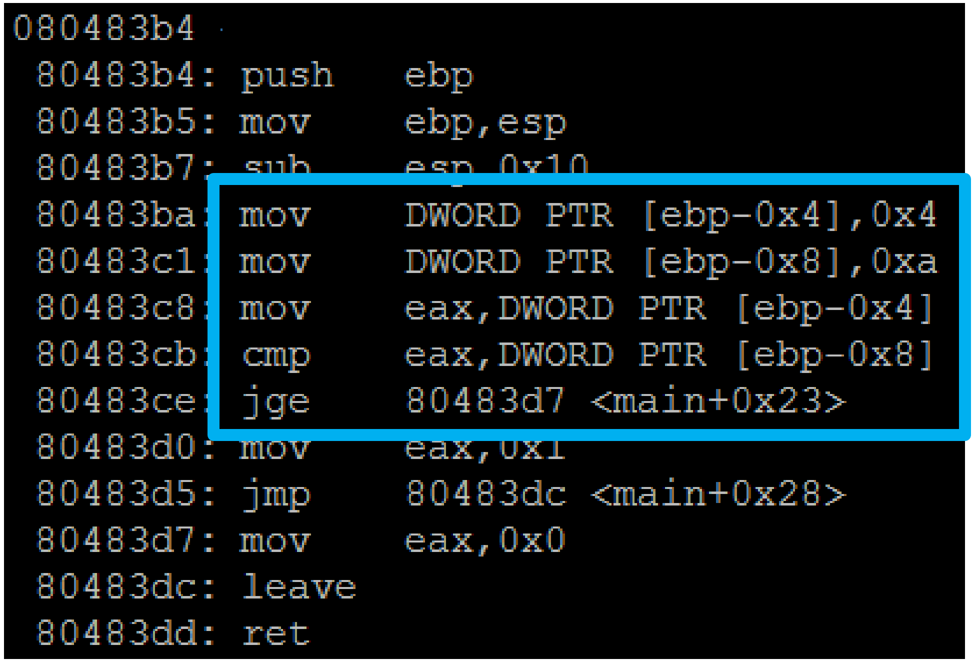
- Two values, relative to the pointer contained in ebp have been assigned values
- One register has been assigned a value



```
080483b4
 80483b4: push   ebp
 80483b5: mov    ebp,esp
 80483b7: sub    esp,0x10
 80483ba: mov    DWORD PTR [ebp-0x4],0x4
 80483c1: mov    DWORD PTR [ebp-0x8],0xa
 80483c8: mov    eax,DWORD PTR [ebp-0x4]
 80483cb: cmp    eax,DWORD PTR [ebp-0x8]
 80483ce: jge    80483d7 <main+0x23>
 80483d0: mov    eax,0x1
 80483d5: jmp    80483dc <main+0x28>
 80483d7: mov    eax,0x0
 80483dc: leave
 80483dd: ret
```

Example 1

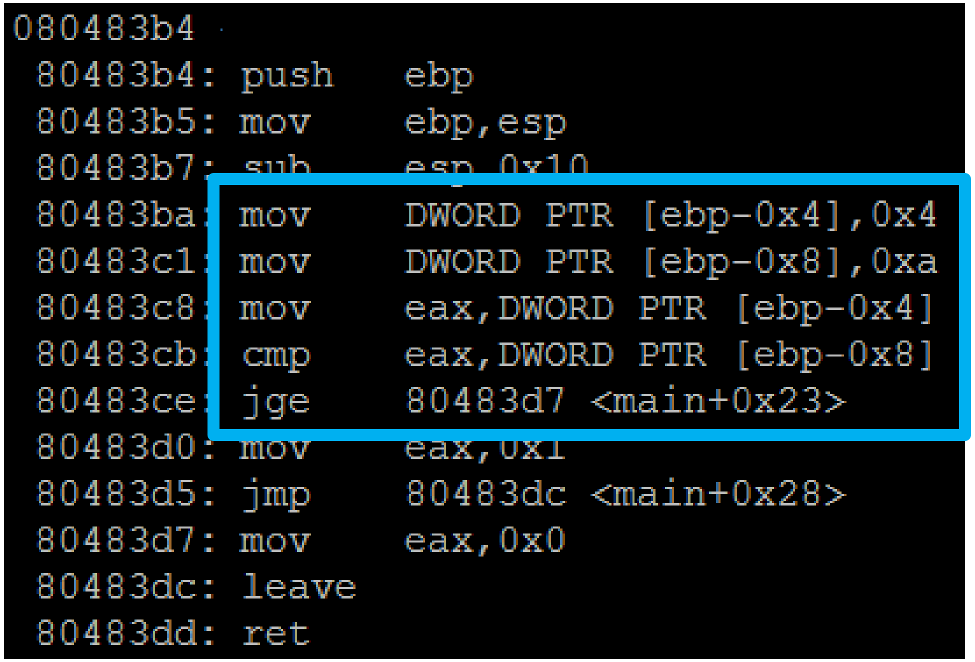
- [ebp-0x4] = 0x4
- [ebp-0x8] = 0xa
- eax = [ebp-0x4]
- cmp eax, [ebp-0x8]
  - eax == [ebp-0x8] ?
  - 4 == 10 ?
- jge 0x80483d7
  - If 4 was >= 10, jmp
  - Else, continue execution



```
080483b4 .
 80483b4: push   ebp
 80483b5: mov    ebp,esp
 80483b7: sub    esp,0x10
 80483ba: mov    DWORD PTR [ebp-0x4],0x4
 80483c1: mov    DWORD PTR [ebp-0x8],0xa
 80483c8: mov    eax,DWORD PTR [ebp-0x4]
 80483cb: cmp    eax,DWORD PTR [ebp-0x8]
 80483ce: jge    80483d7 <main+0x23>
 80483d0: mov    eax,0x1
 80483d5: jmp    80483dc <main+0x28>
 80483d7: mov    eax,0x0
 80483dc: leave
 80483dd: ret
```

Example 1

- [ebp-0x4] = 0x4
- [ebp-0x8] = 0xa
- eax = [ebp-0x4]
- cmp eax, [ebp-0x8]
  - eax == [ebp-0x8] ?
  - 4 == 10 ?
- jge 0x80483d7
  - If 4 was >= 10, jmp
  - Else, continue execution

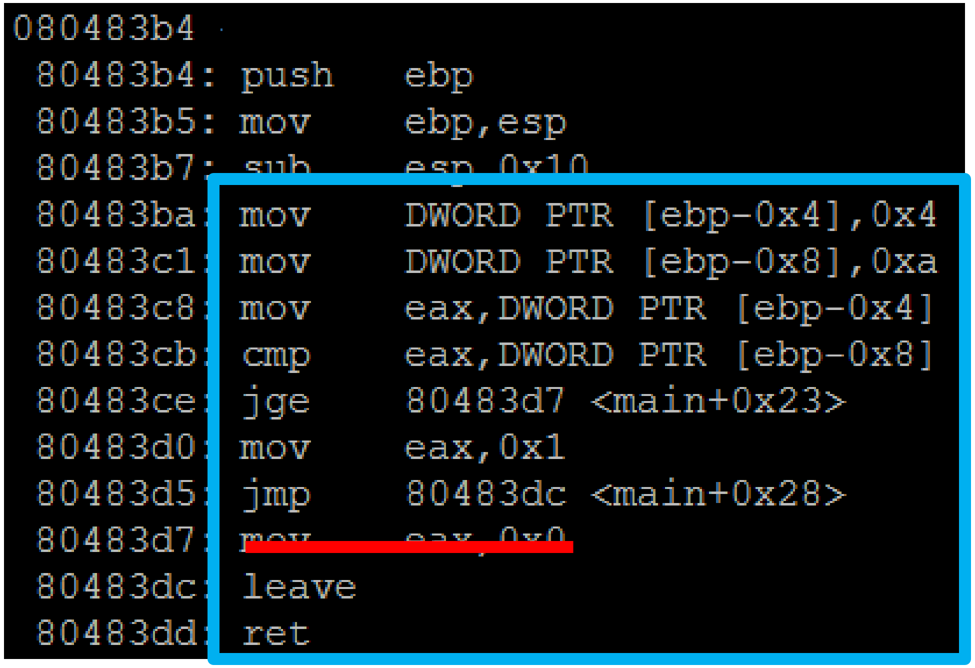


```
080483b4
 80483b4: push   ebp
 80483b5: mov    ebp,esp
 80483b7: sub    esp,0x10
 80483ba: mov    DWORD PTR [ebp-0x4],0x4
 80483c1: mov    DWORD PTR [ebp-0x8],0xa
 80483c8: mov    eax,DWORD PTR [ebp-0x4]
 80483cb: cmp    eax,DWORD PTR [ebp-0x8]
 80483ce: jge    80483d7 <main+0x23>
 80483d0: mov    eax,0x1
 80483d5: jmp    80483dc <main+0x28>
 80483d7: mov    eax,0x0
 80483dc: leave
 80483dd: ret
```

False, so execution just continues to the next instruction

Example 1

- [ebp-0x4] = 0x4
- [ebp-0x8] = 0xa
- eax = [ebp-0x4]
- cmp eax, [ebp-0x8]
- jge 0x80483d7
- mov eax, 0x1
    - eax = 1
- jmp over the mov eax, 0
- leave and return



```
080483b4 
 80483b4: push   ebp
 80483b5: mov    ebp,esp
 80483b7: sub    esp,0x10
 80483ba: mov    DWORD PTR [ebp-0x4],0x4
 80483c1: mov    DWORD PTR [ebp-0x8],0xa
 80483c8: mov    eax,DWORD PTR [ebp-0x4]
 80483cb: cmp    eax,DWORD PTR [ebp-0x8]
 80483ce: jge    80483d7 <main+0x23>
 80483d0: mov    eax,0x1
 80483d5: jmp    80483dc <main+0x28>
 80483d7: mov    eax,0x0
 80483dc: leave
 80483dd: ret
```

Example 1

- So two memory addresses, relative to the pointer contained in ebp, have values. One has 4, one has 10.
- There is a comparison
- If operand 1 >= operand 2, take the jump
- If not, continue execution
- Eax gets assigned the value of 1
- The function returns

Example 1

- Let’s dig deeper
- Everything shown in the disassembly has a purpose
- mov DWORD PTR [ebp-0x4], 0x4
  - What does DWORT PTR mean?
- We know the brackets […] mean get the value held at the dereferenced value between them… but DWORD PTR?

Example 1

- mov DWORD PTR [ebp-0x4], 0x4
- DWORD PTR
  - DWORD = the size
  - PTR = dereference the value, accompanied by the brackets
- We have a few number of sizes allowed

Example 1 – Types and Sizes

Type	Size (bytes)	Size (bits)	ASM	Example
char	1 byte	8 bits	BYTE	char c;
short	2 bytes	16 bits	WORD	short s;
int	4 bytes	32 bits	DWORD	int i;
long long	8 bytes	64 bits	QWORD	long long l;

Example 1

- So…
- mov DWORD PTR [ebp-0x4], 0x4
- The address pointed to by the dereferenced value of [ebp-4] is getting 4 bytes moved into it, with the value of 4.
- [ebp-4] is an int
- So our source code probably has some int value and hard codes a value of 4 to it

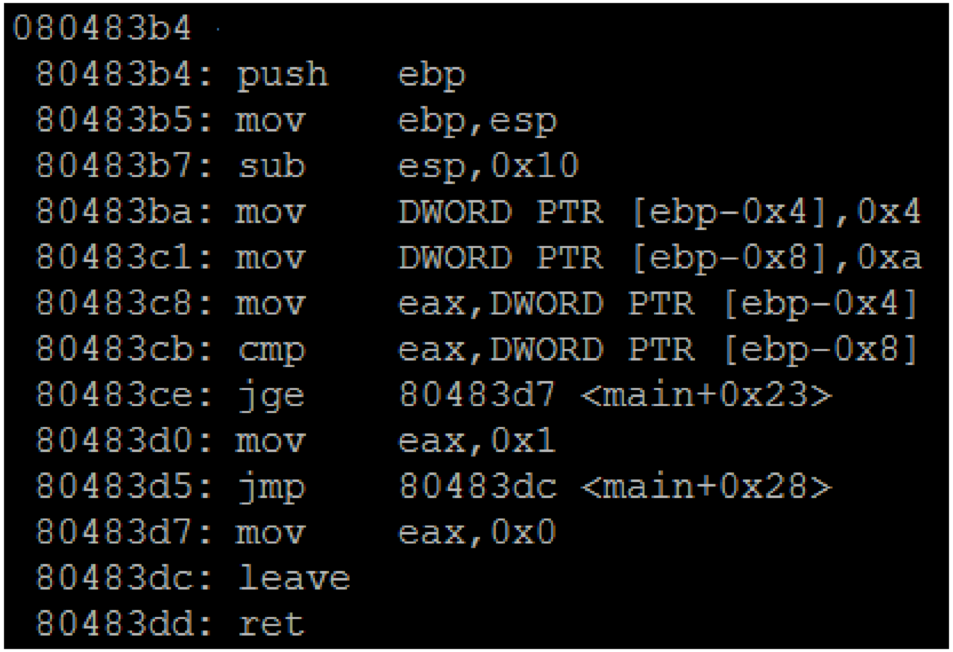
Example 1

- mov DWORD PTR [ebp-0x4], 0x4
- mov DWORD PTR [ebp-0x8], 0xa
- This leaves us with 2 ints being assigned a hard coded value
  - int x = 4;
  - int y = 10;
- Are these locals, globals, static variables???
- We need a little background on process memory layout.

Example 1 – Recap so far

- int x = 4;
- int y = 10;
  - We don’t know where these are declared
- if (4 >= 10)
  - jmp to main+0x23
- eax = 1
- jmp to main+0x28
- main+0x23 :
  - eax = 0
- main+0x28:
  - ret

- We don’t take the jmp as already discussed.
- It’s starting to look like source code!

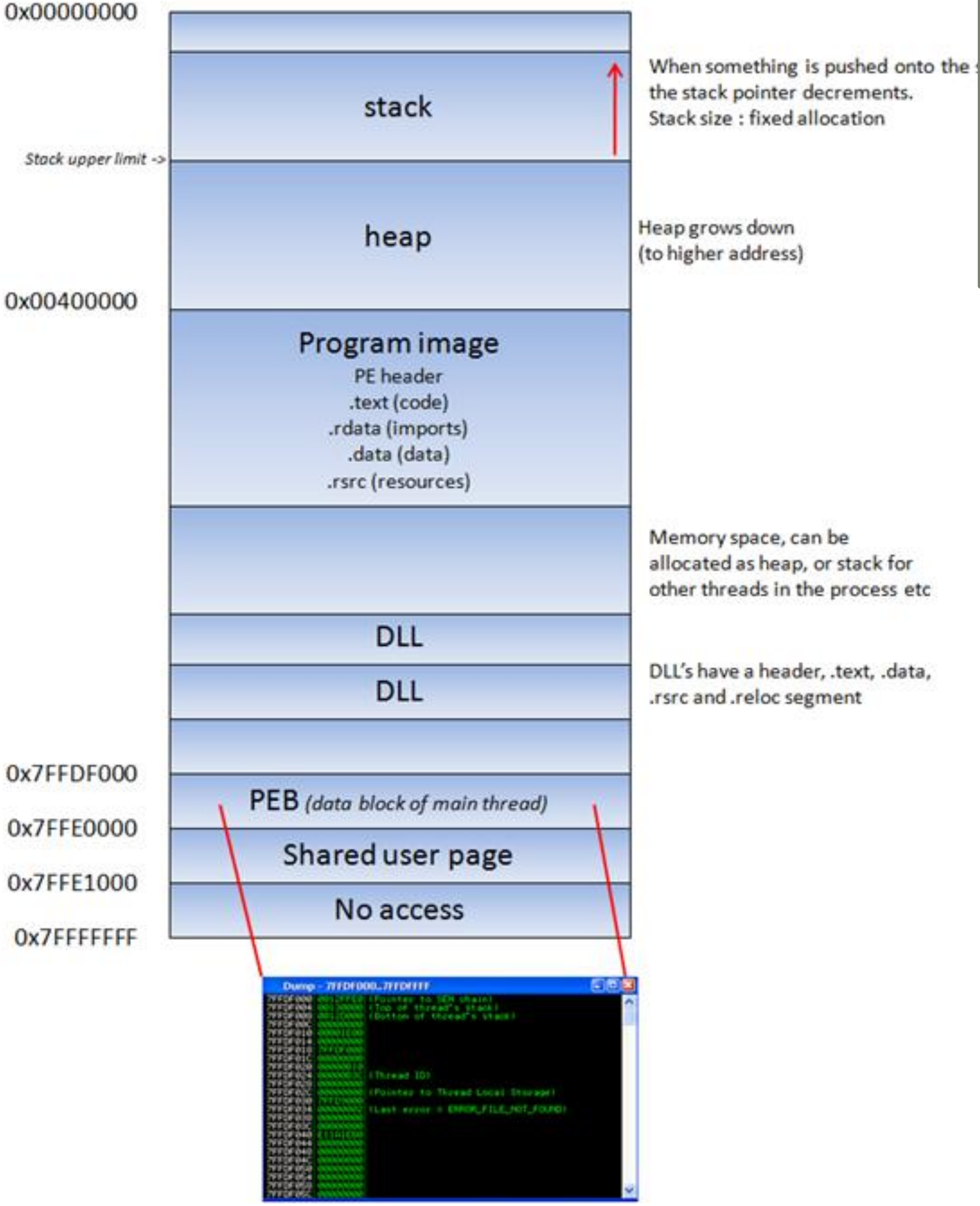


```
080483b4 
 80483b4: push   ebp
 80483b5: mov    ebp,esp
 80483b7: sub    esp,0x10
 80483ba: mov    DWORD PTR [ebp-0x4],0x4
 80483c1: mov    DWORD PTR [ebp-0x8],0xa
 80483c8: mov    eax,DWORD PTR [ebp-0x4]
 80483cb: cmp    eax,DWORD PTR [ebp-0x8]
 80483ce: jge    80483d7 <main+0x23>
 80483d0: mov    eax,0x1
 80483d5: jmp    80483dc <main+0x28>
 80483d7: mov    eax,0x0
 80483dc: leave
 80483dd: ret
```

Process Memory Layout

- Let’s do a quick introduction to process memory layout, then we’ll continue with the first example
- We want to know
  - Why things are relative to esp/ebp?
  - What are the push/pop instructions doing?
  - What about the leave/ret instructions?

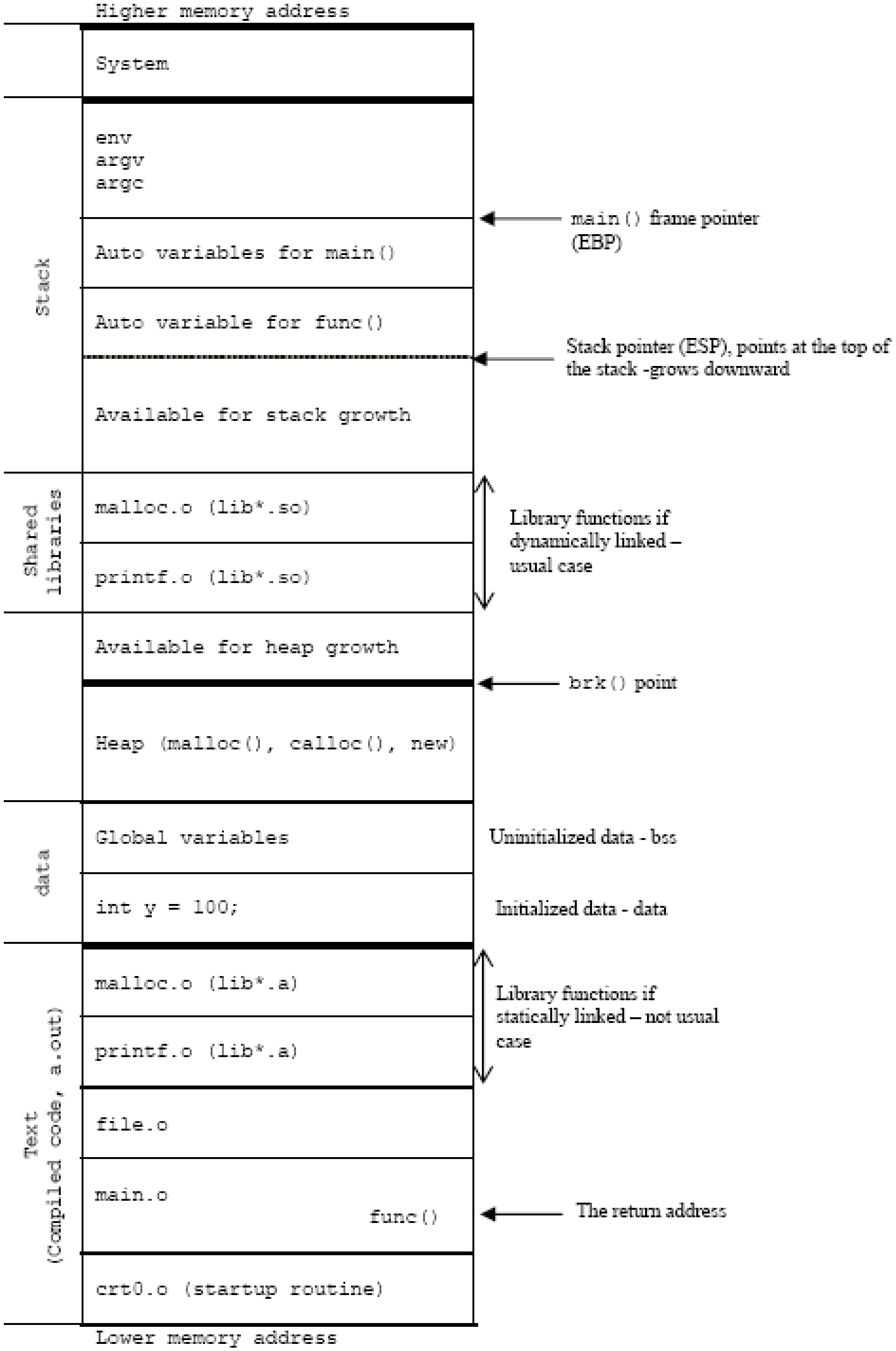
Process Memory Layout - Windows



```
7FFDF000 [illegible] (Pointer to SEH chain)
[illegible] [illegible] (Top of thread's stack)
[illegible] [illegible] (Bottom of thread's stack)
[illegible] [illegible] (Thread ID)
[illegible] [illegible] (Pointer to Thread Local Storage)
[illegible] [illegible] (Last error = ERROR_FILE_NOT_FOUND)
```

Image from https://www.corelan.be/wp-content/uploads/2010/08/image_thumb3.png

Process Memory Layout - Linux



Higher memory address

System

Stack

env
argv
argc

Auto variables for main()

main() frame pointer (EBP)

Auto variable for func()

Stack pointer (ESP), points at the top of the stack -grows downward

Available for stack growth

Shared libraries

malloc.o (lib*.so)

printf.o (lib*.so)

Library functions if dynamically linked – usual case

Available for heap growth

brk() point

Heap (malloc(), calloc(), new)

data

Global variables

Uninitialized data - bss

int y = 100;

Initialized data - data

Text (Compiled code, a.out)

malloc.o (lib*.a)

printf.o (lib*.a)

Library functions if statically linked – not usual case

file.o

main.o

func()

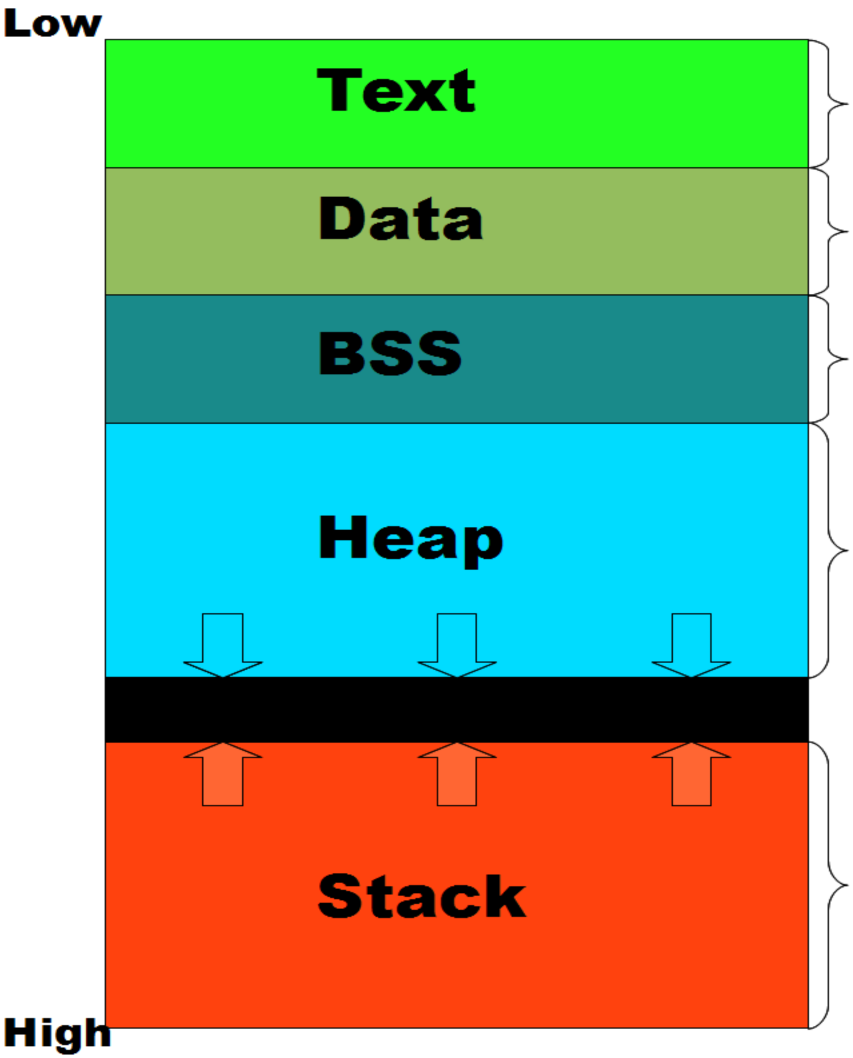
The return address

crt0.o (startup routine)

Lower memory address

Image from
http://www.tenouk.com/Bufferoverflowc/Bufferoverflow1_files/image022.png

Virtual Memory



Code segment, machine instr.

Initialized global and static variables

Uninitialized global and static variables

Dynamic space.
    malloc(…) / free(…)
    new(…) / ~

Program scratch space.
Local variables, pass arguments, etc..

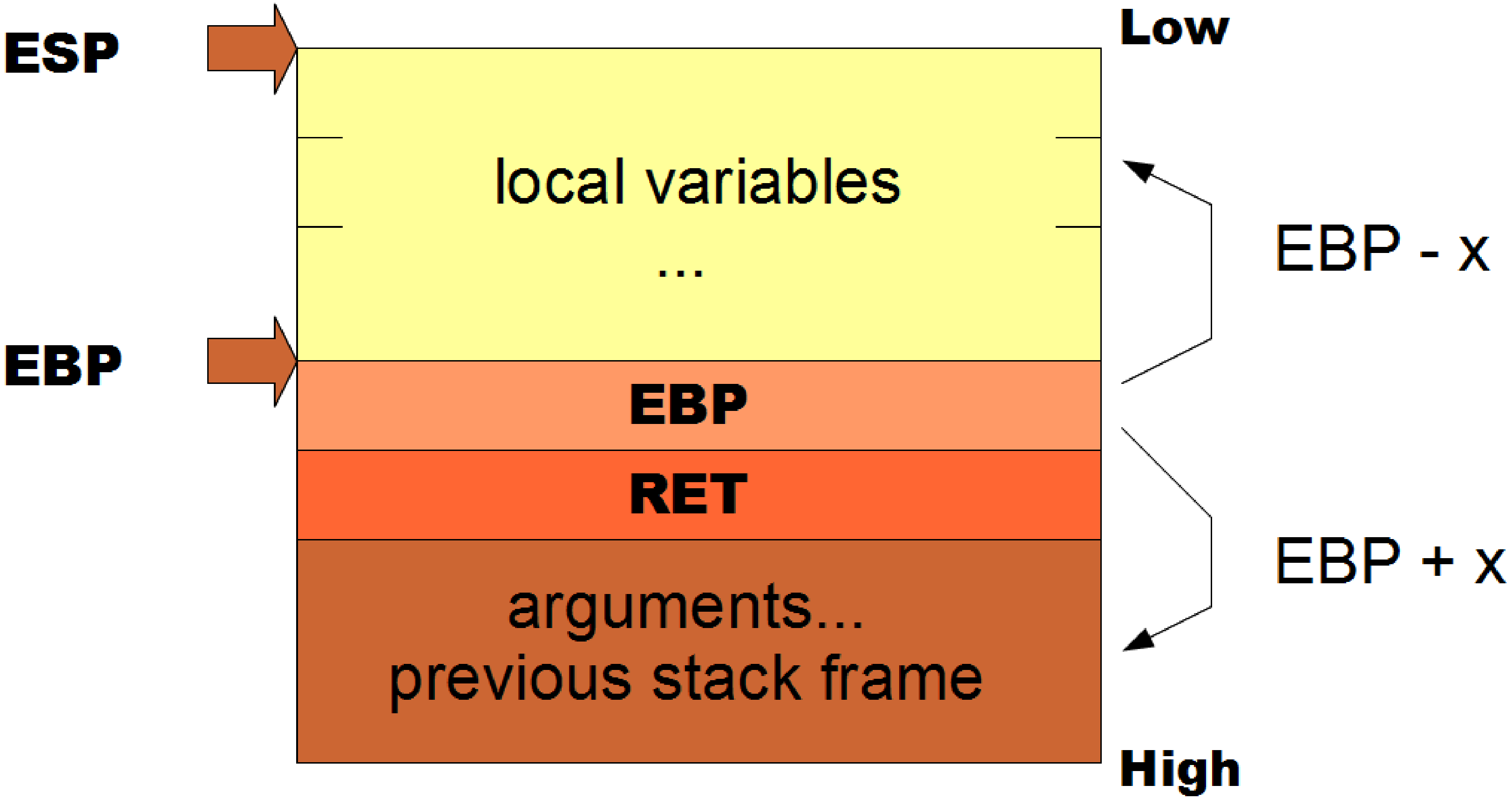
Registers

Registers

Register Name	Description
EIP	Next instruction executed *Want to hijack during exploitation
ESP	Stack pointer
EBP	Base pointer
EAX	Accumulation *Holds the return value, usually.
EBX	Base
ECX	Counter
EDX	Data
ESI	Source index
EDI	Destination index

The Stack

The Stack

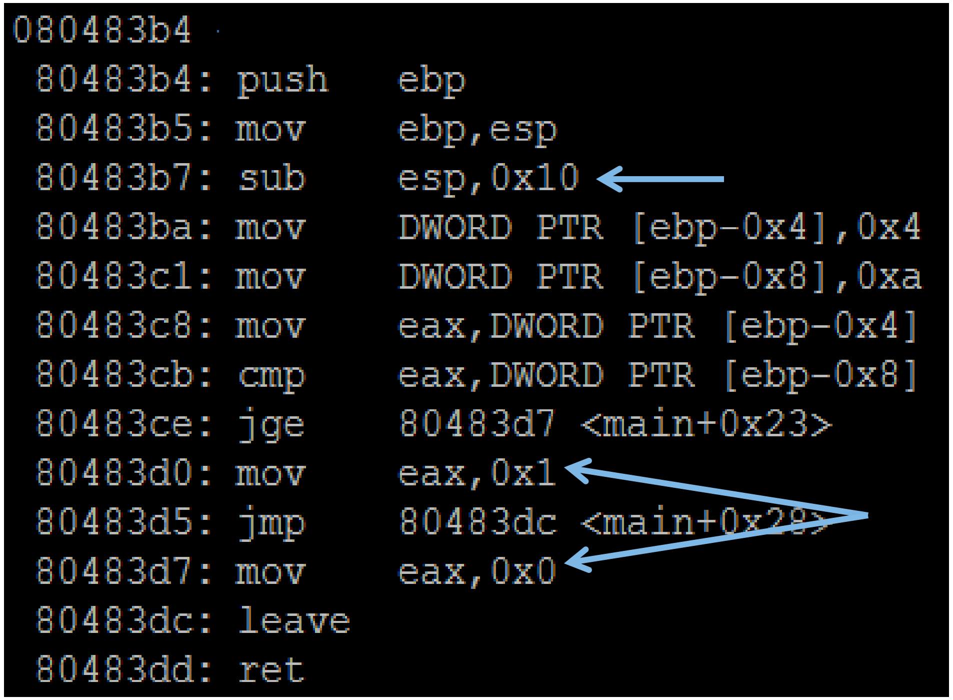

ESP
Low
local variables
...
EBP - x
EBP
EBP
RET
EBP + x
arguments...
previous stack frame
High

Example 1 – Part 2

- Okay, we have some background on the registers, the stack, and process layout
- Let’s try to figure out what this code’s stack layout would look like
- Then, we’ll look back at the code and what we know

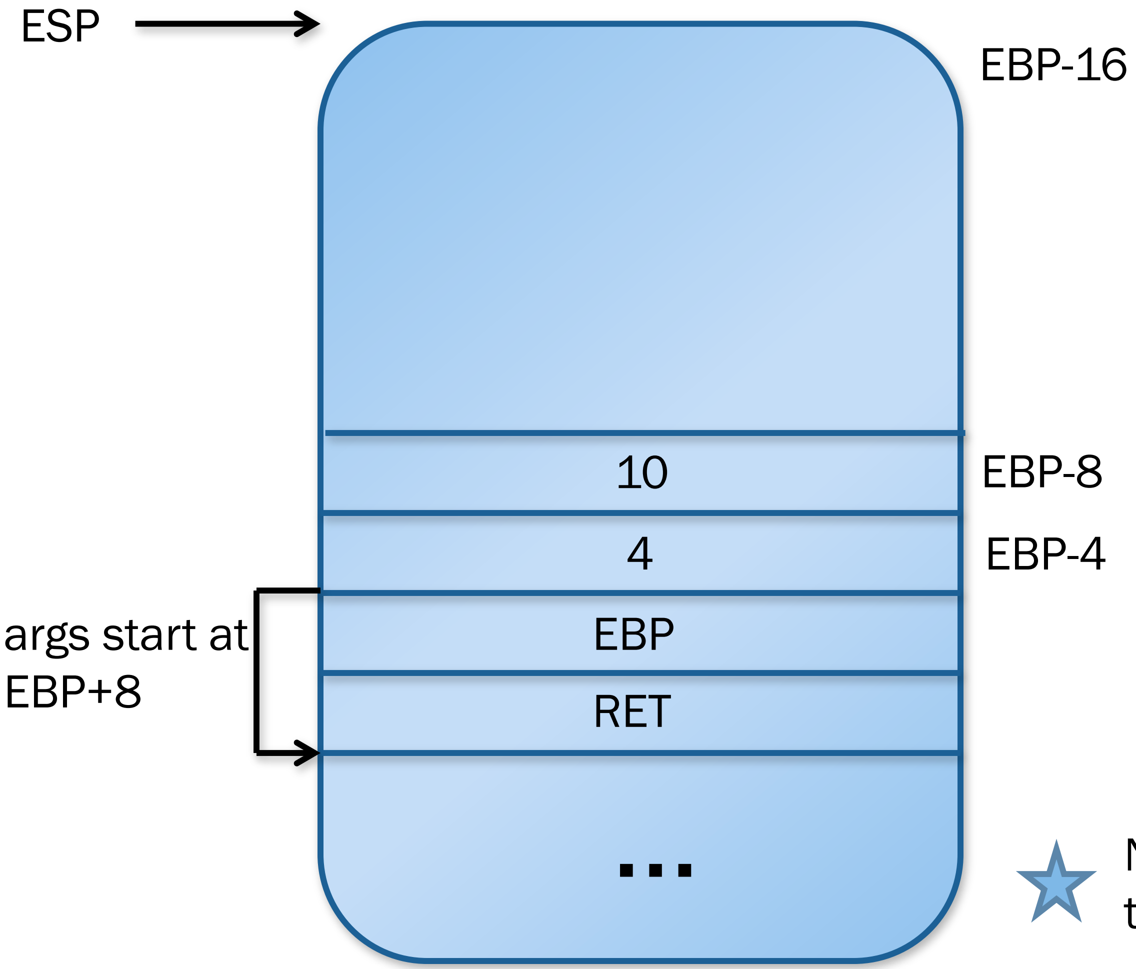
Example 1 – Part 2

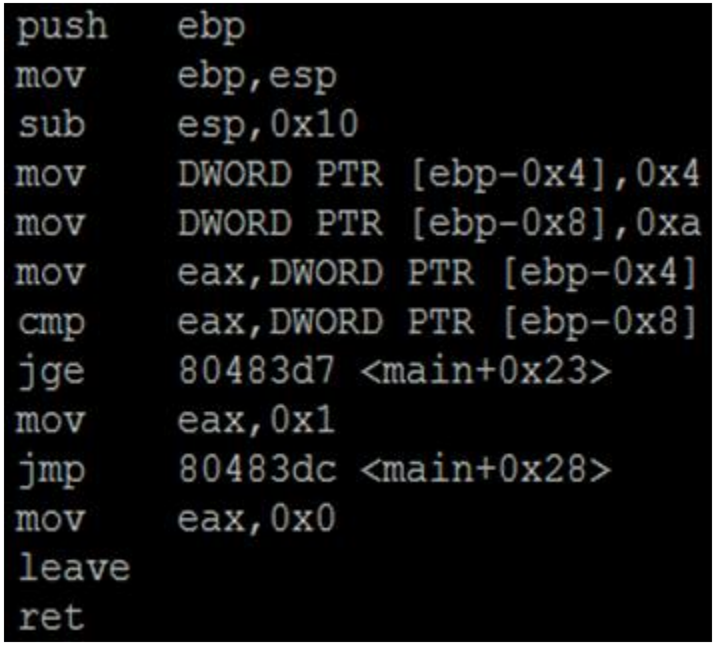
- sub esp, 0x10
  - There is room for 16 bytes of locals, or 4 ints
- [ebp-4] is a local
- [ebp-8] is a local
- Return value, eax, is either 1 or 0 depending on the comparison



```
080483b4
 80483b4: push   ebp
 80483b5: mov    ebp,esp
 80483b7: sub    esp,0x10
 80483ba: mov    DWORD PTR [ebp-0x4],0x4
 80483c1: mov    DWORD PTR [ebp-0x8],0xa
 80483c8: mov    eax,DWORD PTR [ebp-0x4]
 80483cb: cmp    eax,DWORD PTR [ebp-0x8]
 80483ce: jge    80483d7 <main+0x23>
 80483d0: mov    eax,0x1
 80483d5: jmp    80483dc <main+0x28>
 80483d7: mov    eax,0x0
 80483dc: leave
 80483dd: ret
```

Example 1's stack



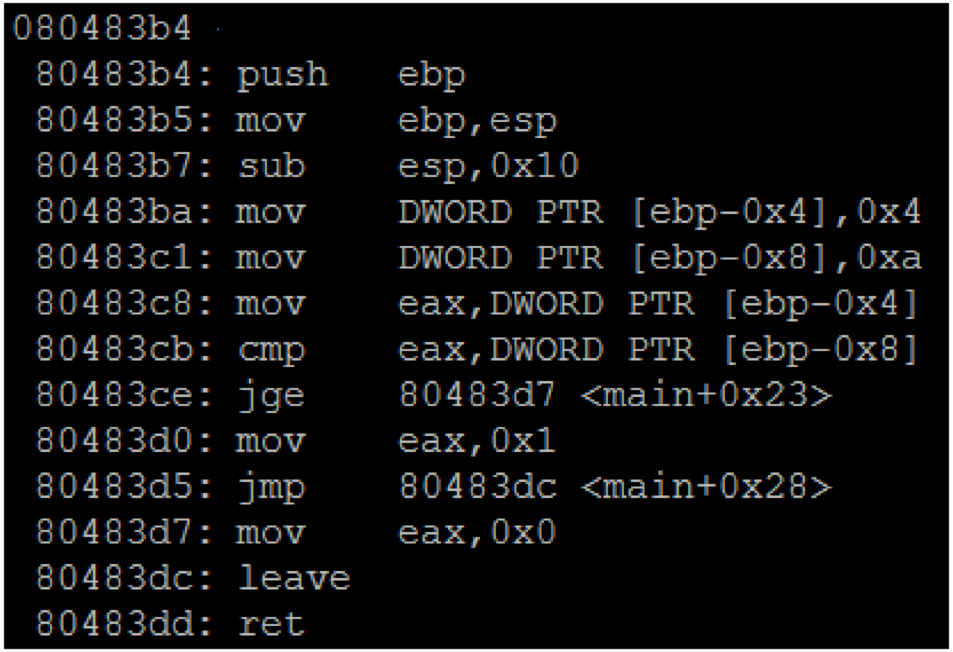


```
push   ebp
mov    ebp,esp
sub    esp,0x10
mov    DWORD PTR [ebp-0x4],0x4
mov    DWORD PTR [ebp-0x8],0xa
mov    eax,DWORD PTR [ebp-0x4]
cmp    eax,DWORD PTR [ebp-0x8]
jge    80483d7 <main+0x23>
mov    eax,0x1
jmp    80483dc <main+0x28>
mov    eax,0x0
leave
ret
```

No [ebp+x], no arguments to the function

Example 1 – Part 2

- int someFunction() {
- int x = 4;
- int y = 10;
- if (4 >= 10)
  - jmp to main+0x23
- eax = 1
- jmp to main+0x28
- main+0x23 :
  - eax = 0
- main+0x28:
  - return



```
080483b4
 80483b4: push   ebp
 80483b5: mov    ebp,esp
 80483b7: sub    esp,0x10
 80483ba: mov    DWORD PTR [ebp-0x4],0x4
 80483c1: mov    DWORD PTR [ebp-0x8],0xa
 80483c8: mov    eax,DWORD PTR [ebp-0x4]
 80483cb: cmp    eax,DWORD PTR [ebp-0x8]
 80483ce: jge    80483d7 <main+0x23>
 80483d0: mov    eax,0x1
 80483d5: jmp    80483dc <main+0x28>
 80483d7: mov    eax,0x0
 80483dc: leave
 80483dd: ret
```

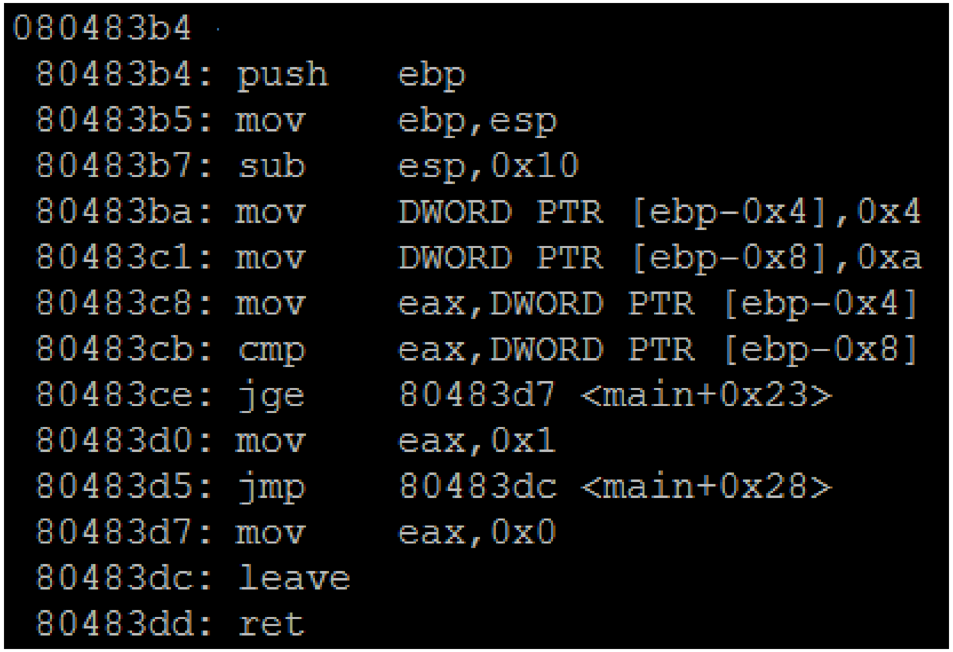
A side note about source to asm

- ‘if’ comparisons get translated opposite from source to assembly
- if x > y
- Will become
  - cmp x, y
  - jle 0x12345678 (jump less than or equal)
  - If some condition is ***not true***, jump over it
- If x <= y
- Will become
  - cmp x, y
  - ja 0x12345678 (jmp above)

Example 1 – Part 2

- int someFunction() {
- int x = 4;
- int y = 10;
- if (4 < 10)
  - Return 1
- Return 0
- }

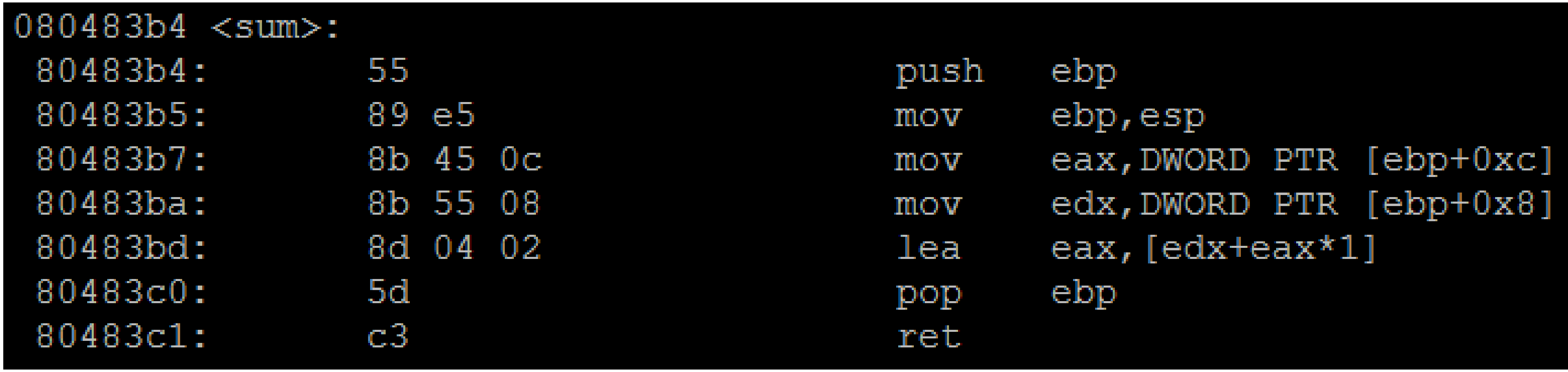
- Hey, that's source code!



```
080483b4
 80483b4: push   ebp
 80483b5: mov    ebp,esp
 80483b7: sub    esp,0x10
 80483ba: mov    DWORD PTR [ebp-0x4],0x4
 80483c1: mov    DWORD PTR [ebp-0x8],0xa
 80483c8: mov    eax,DWORD PTR [ebp-0x4]
 80483cb: cmp    eax,DWORD PTR [ebp-0x8]
 80483ce: jge    80483d7 <main+0x23>
 80483d0: mov    eax,0x1
 80483d5: jmp    80483dc <main+0x28>
 80483d7: mov    eax,0x0
 80483dc: leave
 80483dd: ret
```

5 Minute Exercise

- Produce the source code for the following function

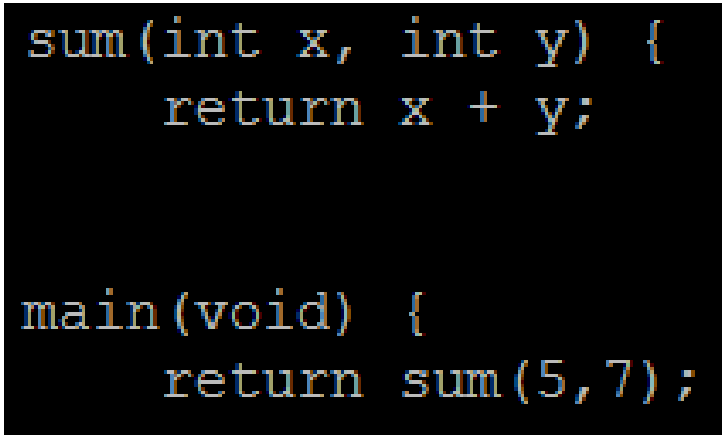


```
080483b4 <sum>:
 80483b4:       55                      push   ebp
 80483b5:       89 e5                   mov    ebp,esp
 80483b7:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 80483ba:       8b 55 08                mov    edx,DWORD PTR [ebp+0x8]
 80483bd:       8d 04 02                lea    eax,[edx+eax*1]
 80483c0:       5d                      pop    ebp
 80483c1:       c3                      ret
```

- How many local variables, how many arguments, what types?
- Hint: lea eax, [edx+eax*1] is the same thing as
  - eax = edx+eax

Exercise 2 - Solution

- What we just saw was the sum function.
- The compiler used lea edx+eax for efficiency
- It could have similarly used the add instruction
- eax contains the return value
- No local variables were used (no [ebp-x]), just arguments ([ebp+x])



```
sum(int x, int y) {
    return x + y;


main(void) {
    return sum(5,7);
```

Functions

Functions

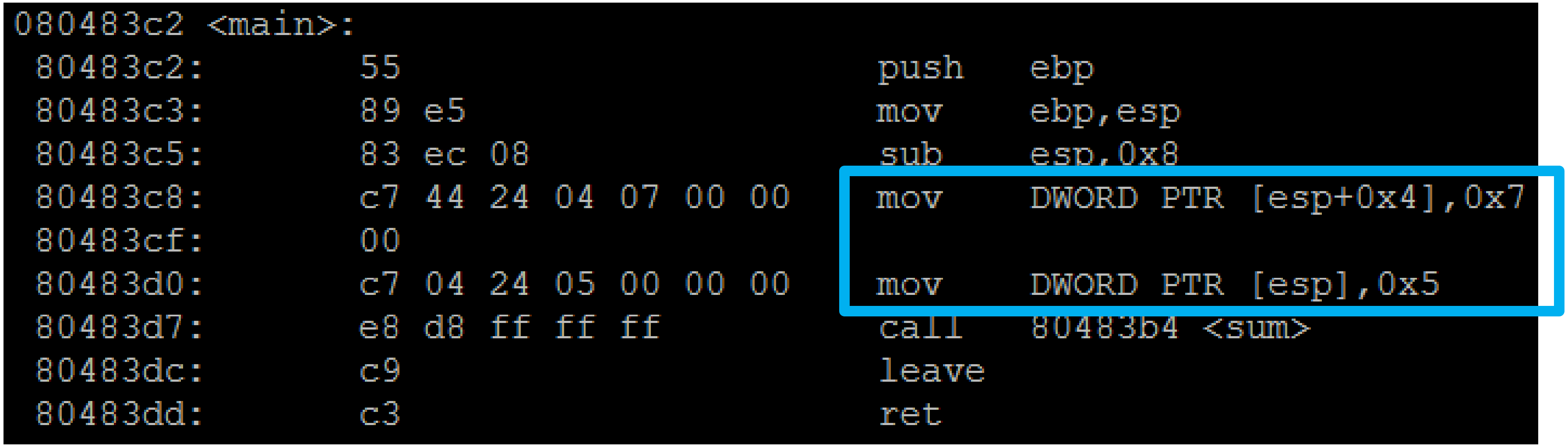
- Looking at the previous exercise introduces a question about how function calls are handled
- We know
  - eax holds the return value
  - Arguments (from the functions point of view) begin at ebp+8
- But how do those arguments get there, and how are they removed?

Functions – Calling Conventions

- Two main calling conventions are commonly used
- CDECL
  - Originates from C
  - Args pushed on the stack, right to left (reverse)
  - **Calling function cleans up**
- STDCall
  - Orignates from Microsoft
  - Args pushed on the stack, right to left (reverse)
  - **Called function cleans up**
    - **Must know how many bytes ahead of time**

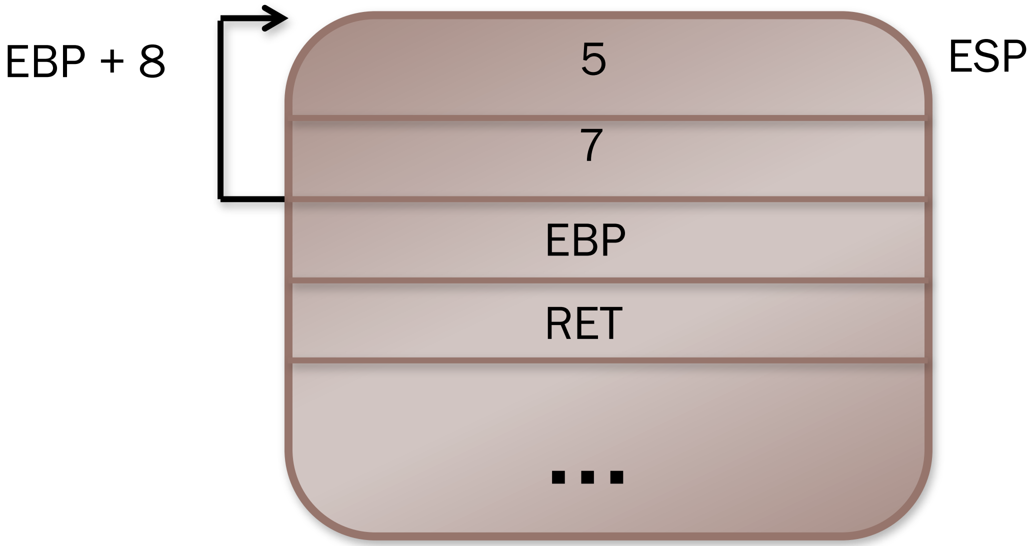
Functions – Exercise 2's main

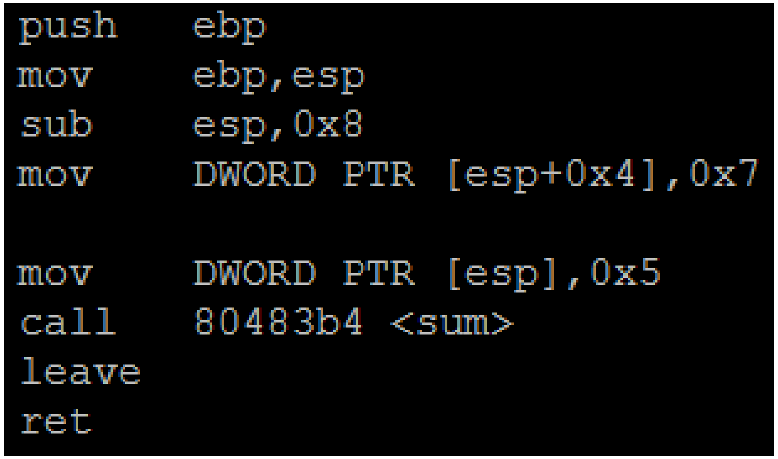
- GCC tends to use : move [esp+x], arg
- Visual studio tents to use : push arg
- Regardless, we're putting args on top of the stack



```
080483c2 <main>:
 80483c2:       55                      push   ebp
 80483c3:       89 e5                   mov    ebp,esp
 80483c5:       83 ec 08                sub    esp,0x8
 80483c8:       c7 44 24 04 07 00 00    mov    DWORD PTR [esp+0x4],0x7
 80483cf:       00
 80483d0:       c7 04 24 05 00 00 00    mov    DWORD PTR [esp],0x5
 80483d7:       e8 d8 ff ff ff          call   80483b4 <sum>
 80483dc:       c9                      leave
 80483dd:       c3                      ret
```

Functions – Exercise 2’s main





```
push    ebp
mov     ebp,esp
sub     esp,0x8
mov     DWORD PTR [esp+0x4],0x7

mov     DWORD PTR [esp],0x5
call    80483b4 <sum>
leave
ret
```

- Now that the stack is setup, sum is called

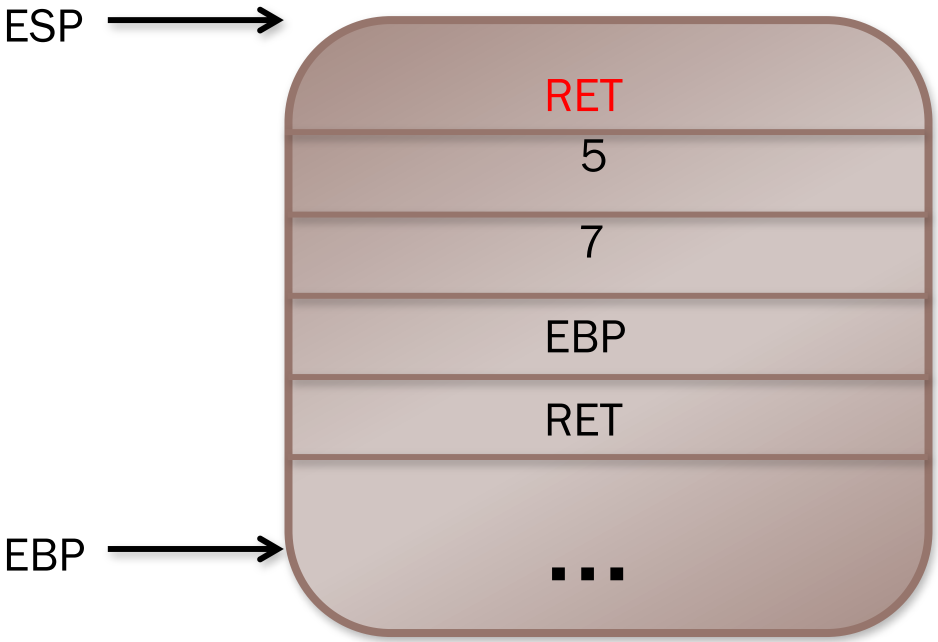
Stack Frames

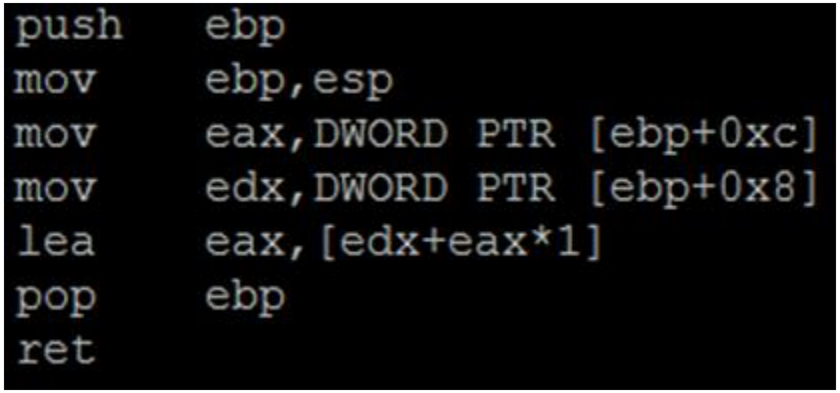
- Functions reference local variables and arguments via their stack frame pointers, esp and ebp
- So, every function has it’s own prolog and epilog to adjust esp and ebp to contain the correct values

Stack Frames

- Prolog – push ebp to save it on the stack, then move ebp to the top of the stack, then make room for locals
  - Push ebp
  - mov ebp, esp
  - sub esp, x
- Epilog – move esp back to ebp, pop the top of the stack into ebp, return to the address on top of the stack
  - add esp, x
  - pop ebp
  - ret
- Epilog 2 – leave is equivalent to : mov esp, ebp; pop ebp
  - leave
  - ret

Stack Frames – Exercise 2

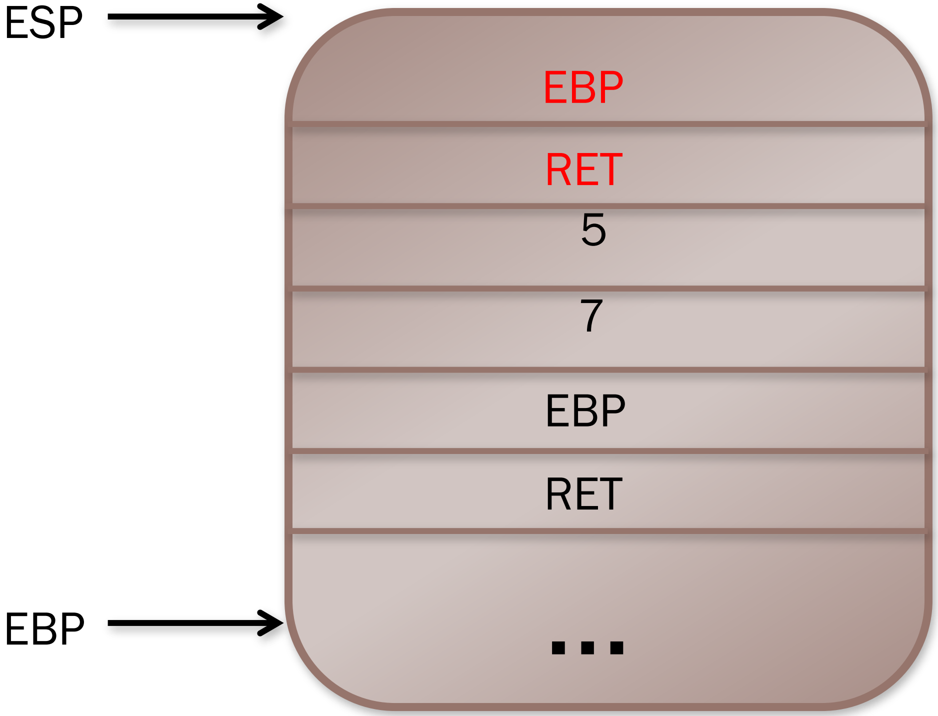


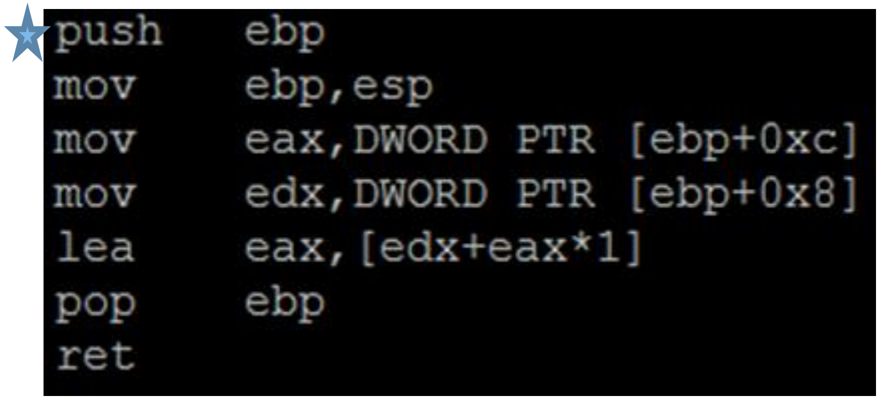


```
push    ebp
mov     ebp,esp
mov     eax,DWORD PTR [ebp+0xc]
mov     edx,DWORD PTR [ebp+0x8]
lea     eax,[edx+eax*1]
pop     ebp
ret
```

- The call instruction pushes EIP onto the stack

Stack Frames – Exercise 2

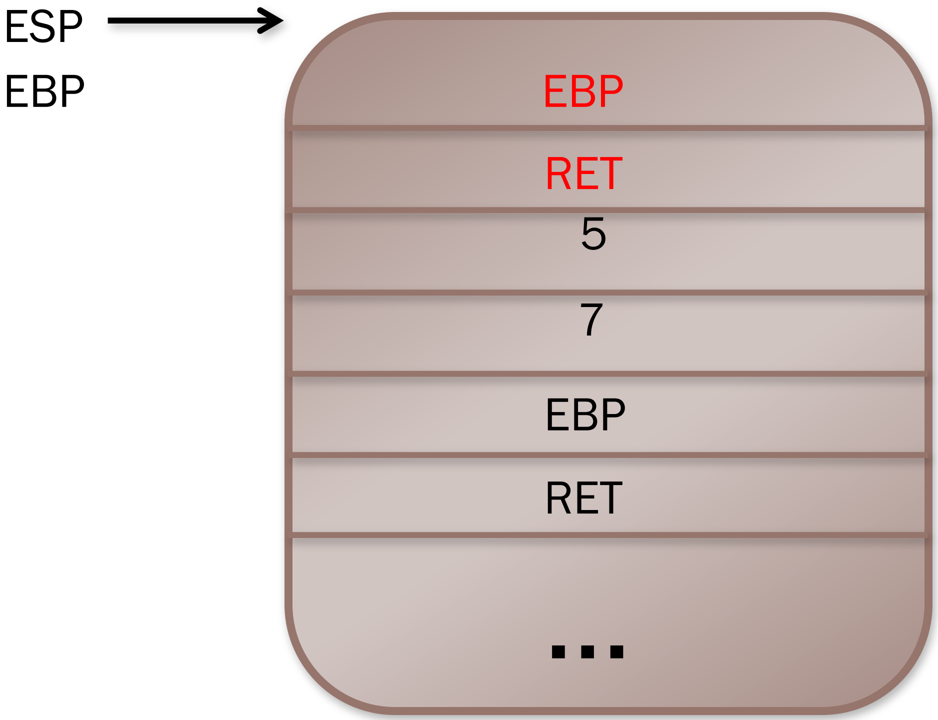


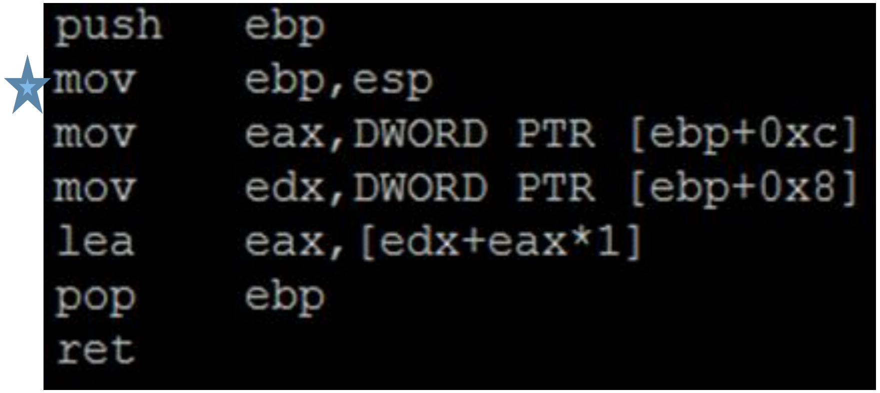


```
push    ebp
mov     ebp,esp
mov     eax,DWORD PTR [ebp+0xc]
mov     edx,DWORD PTR [ebp+0x8]
lea     eax,[edx+eax*1]
pop     ebp
ret
```

- EBP is saved

Stack Frames – Exercise 2

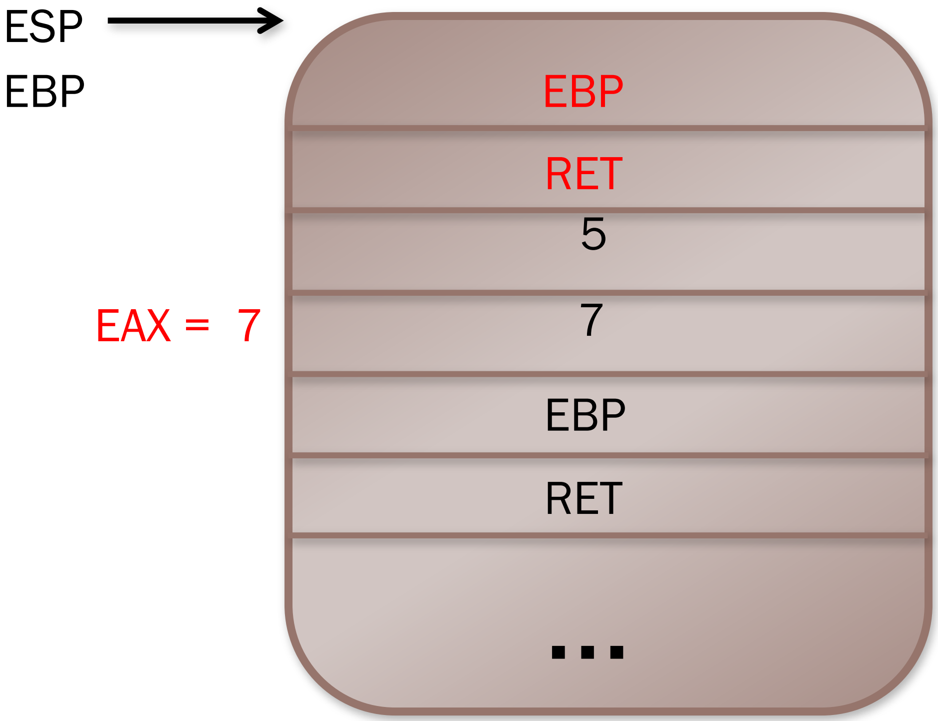


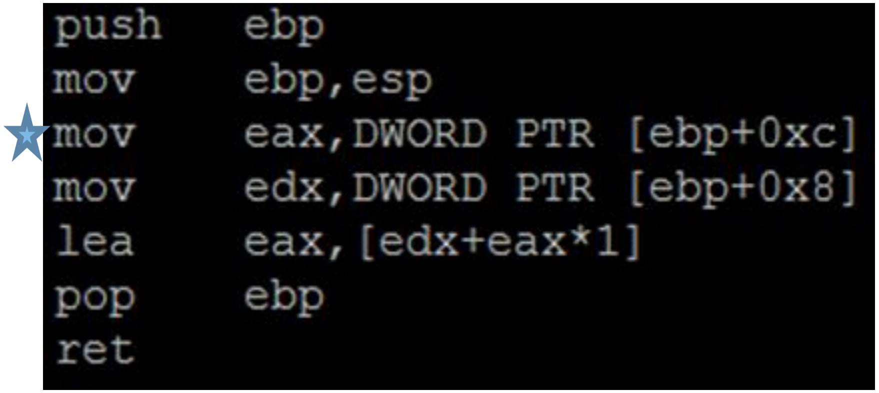


```
push    ebp
mov     ebp,esp
mov     eax,DWORD PTR [ebp+0xc]
mov     edx,DWORD PTR [ebp+0x8]
lea     eax,[edx+eax*1]
pop     ebp
ret
```

- EBP has the same value as ESP now

Stack Frames – Exercise 2

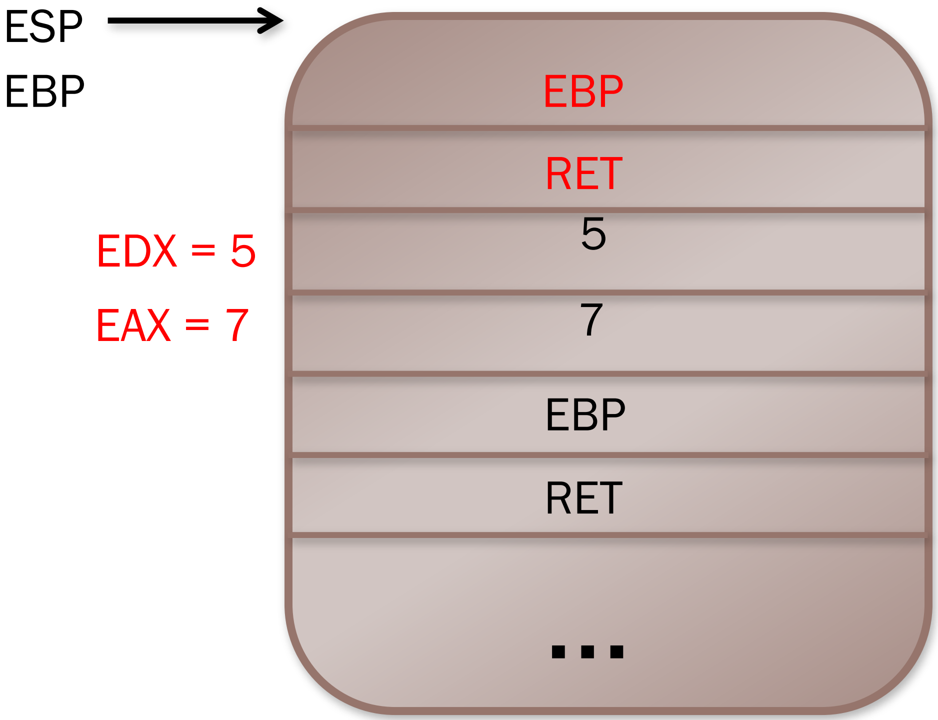


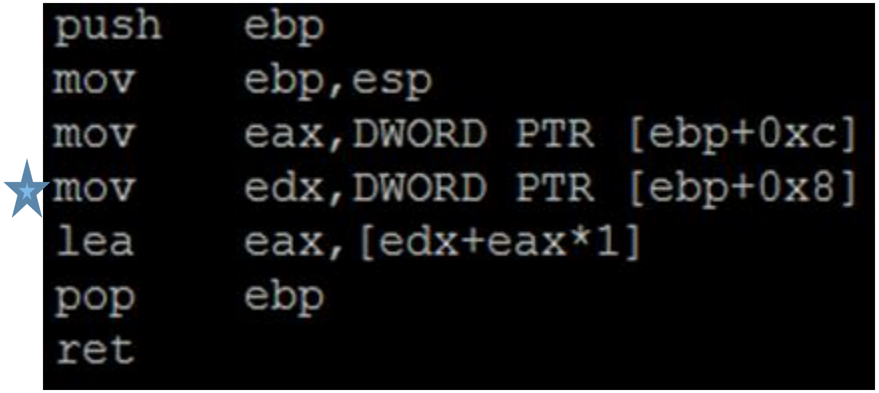


```
push    ebp
mov     ebp,esp
mov     eax,DWORD PTR [ebp+0xc]
mov     edx,DWORD PTR [ebp+0x8]
lea     eax,[edx+eax*1]
pop     ebp
ret
```

- EAX gets the value of arg 2

Stack Frames – Exercise 2

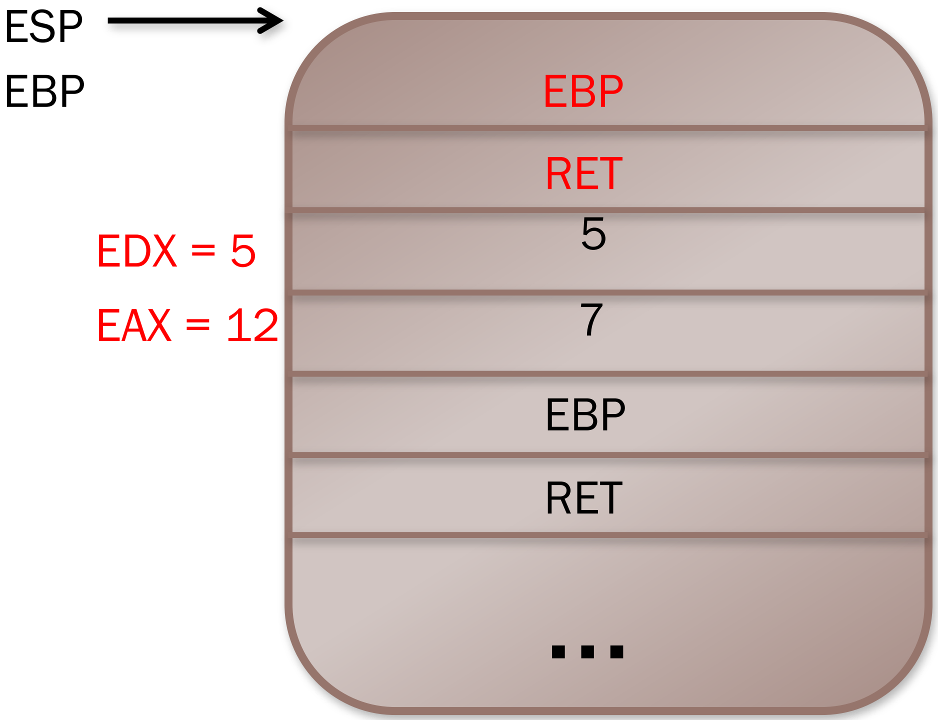




```
push    ebp
mov     ebp,esp
mov     eax,DWORD PTR [ebp+0xc]
mov     edx,DWORD PTR [ebp+0x8]
lea     eax,[edx+eax*1]
pop     ebp
ret
```

- EDX gets the value of arg 1

Stack Frames – Exercise 2



ESP

EBP

EDX = 5

EAX = 12

EBP

RET

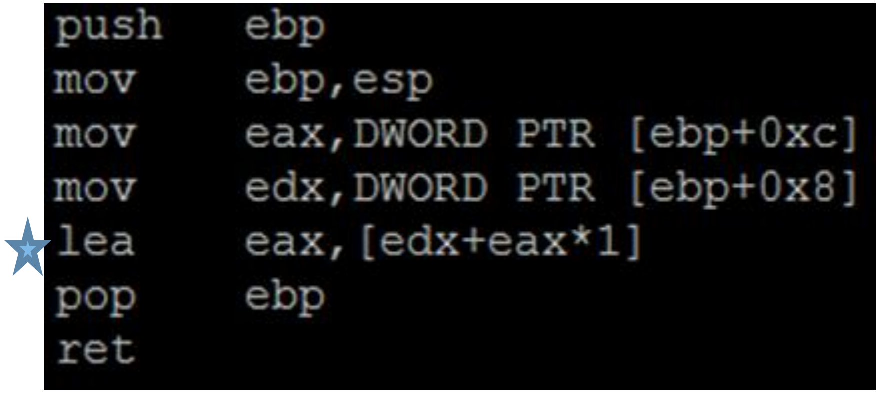
5

7

EBP

RET

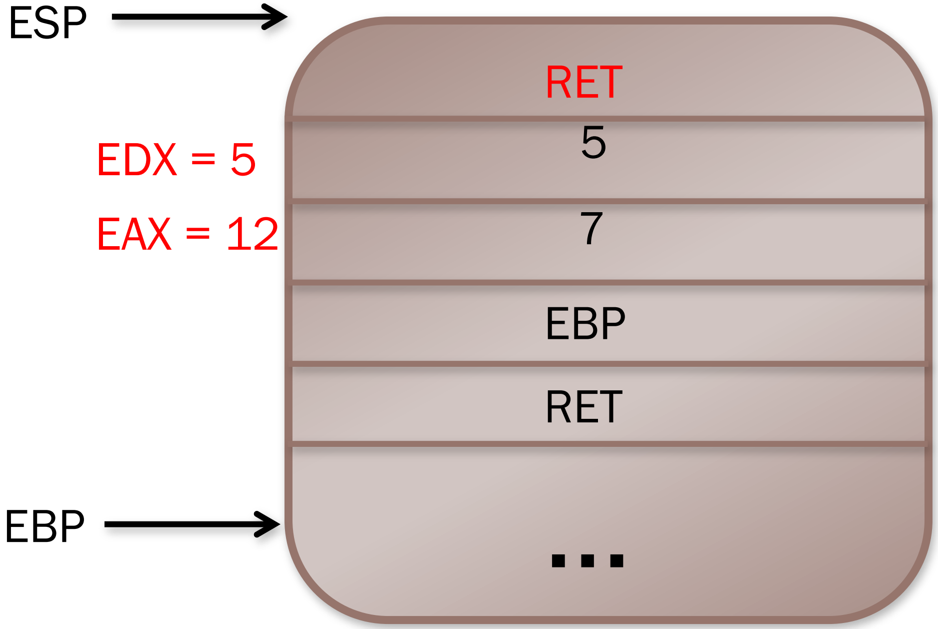
...



```
push    ebp
mov     ebp,esp
mov     eax,DWORD PTR [ebp+0xc]
mov     edx,DWORD PTR [ebp+0x8]
lea     eax,[edx+eax*1]
pop     ebp
ret
```

- EAX contains a new value now, not what was in arg2

Stack Frames – Exercise 2



ESP

RET

EDX = 5

5

EAX = 12

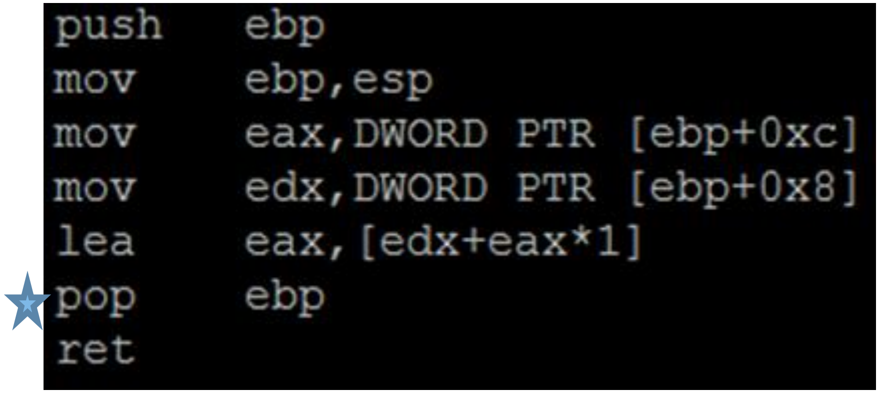
7

EBP

RET

EBP

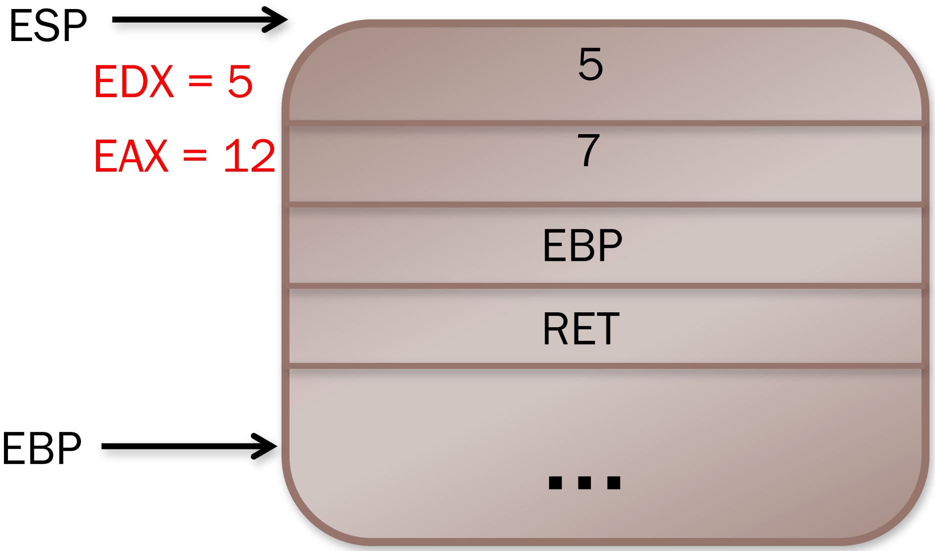
...

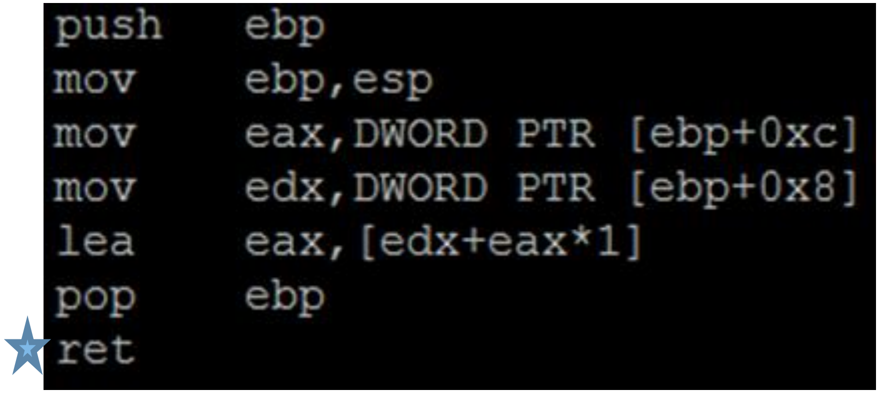


```
push    ebp
mov     ebp,esp
mov     eax,DWORD PTR [ebp+0xc]
mov     edx,DWORD PTR [ebp+0x8]
lea     eax,[edx+eax*1]
pop     ebp
ret
```

- In the epilog now, set EBP back to the callers value

Stack Frames – Exercise 2



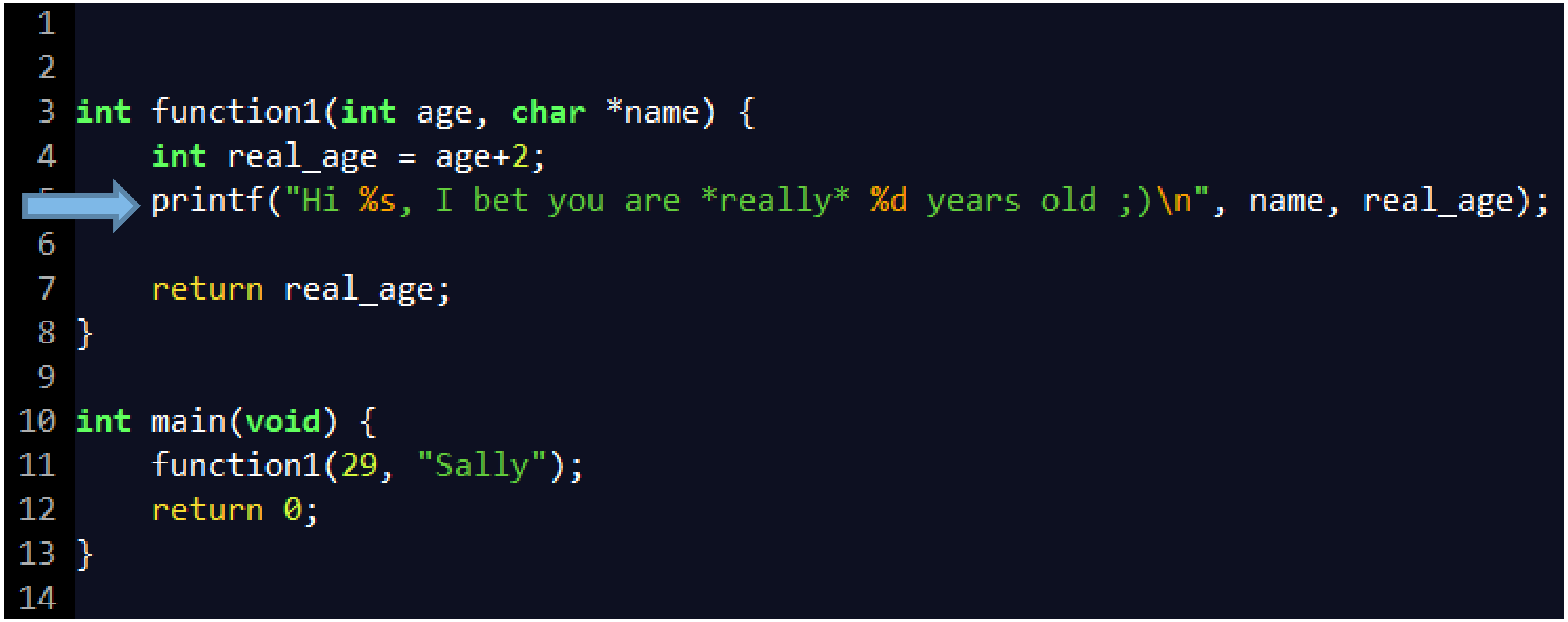


```
push    ebp
mov     ebp,esp
mov     eax,DWORD PTR [ebp+0xc]
mov     edx,DWORD PTR [ebp+0x8]
lea     eax,[edx+eax*1]
pop     ebp
ret
```

- Ret is the same as : pop EIP
- Control flow returns to the next instruction in the caller

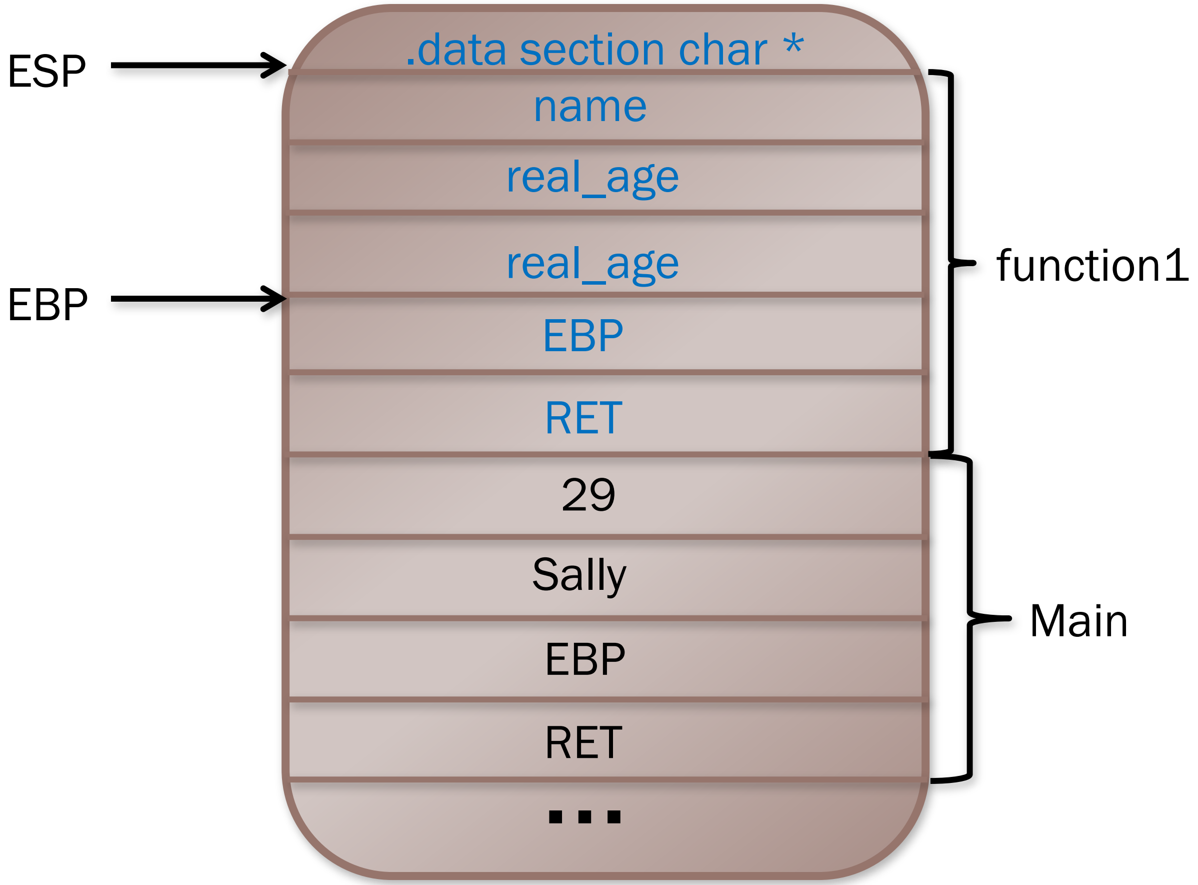
Quick Exercise – 5 minutes

- What is the stack going to look like at the printf call?



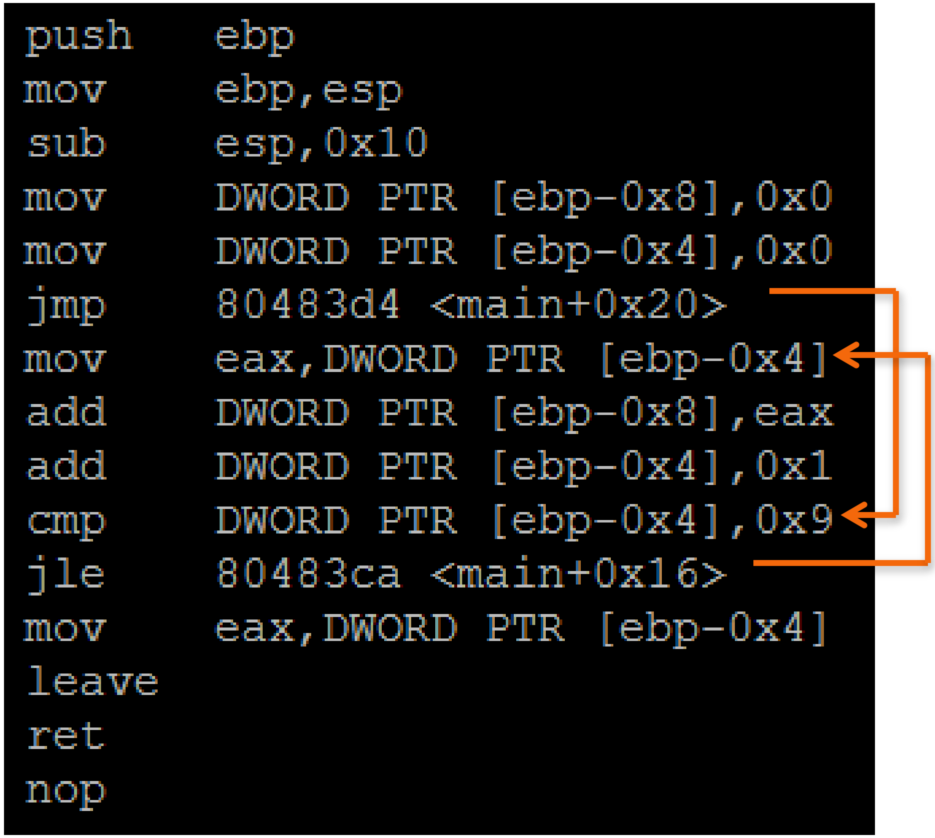
```c


int function1(int age, char *name) {
    int real_age = age+2;
    printf("Hi %s, I bet you are *really* %d years old ;)\n", name, real_age);

    return real_age;
}

int main(void) {
    function1(29, "Sally");
    return 0;
}

```

Solution

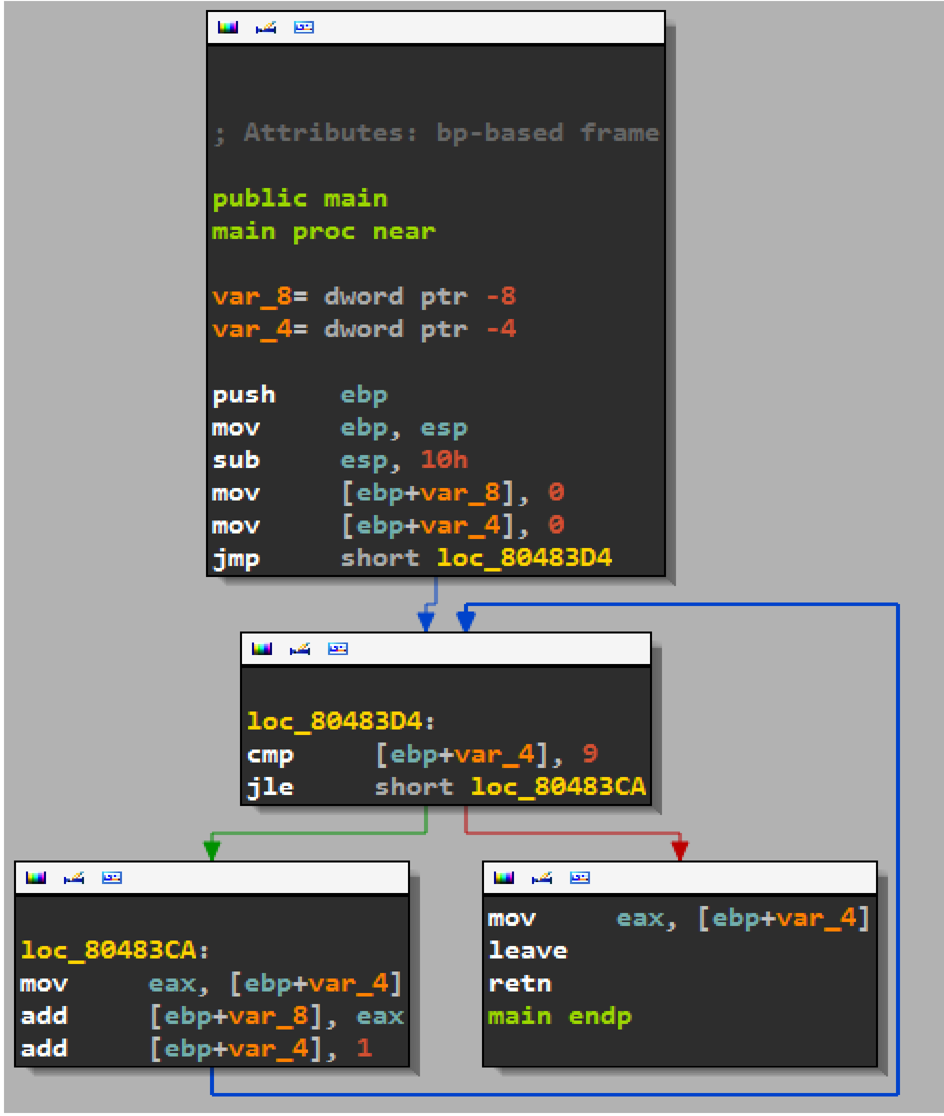

ESP
EBP
.data section char *
name
real_age
real_age
EBP
RET
29
Sally
EBP
RET
...
function1
Main

Recognizing Patterns

- for(i = 0; i < 10; i++)



```
push    ebp
mov     ebp,esp
sub     esp,0x10
mov     DWORD PTR [ebp-0x8],0x0
mov     DWORD PTR [ebp-0x4],0x0
jmp     80483d4 <main+0x20>
mov     eax,DWORD PTR [ebp-0x4]
add     DWORD PTR [ebp-0x8],eax
add     DWORD PTR [ebp-0x4],0x1
cmp     DWORD PTR [ebp-0x4],0x9
jle     80483ca <main+0x16>
mov     eax,DWORD PTR [ebp-0x4]
leave
ret
nop
```



```
; Attributes: bp-based frame

public main
main proc near

var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 10h
mov     [ebp+var_8], 0
mov     [ebp+var_4], 0
jmp     short loc_80483D4
```

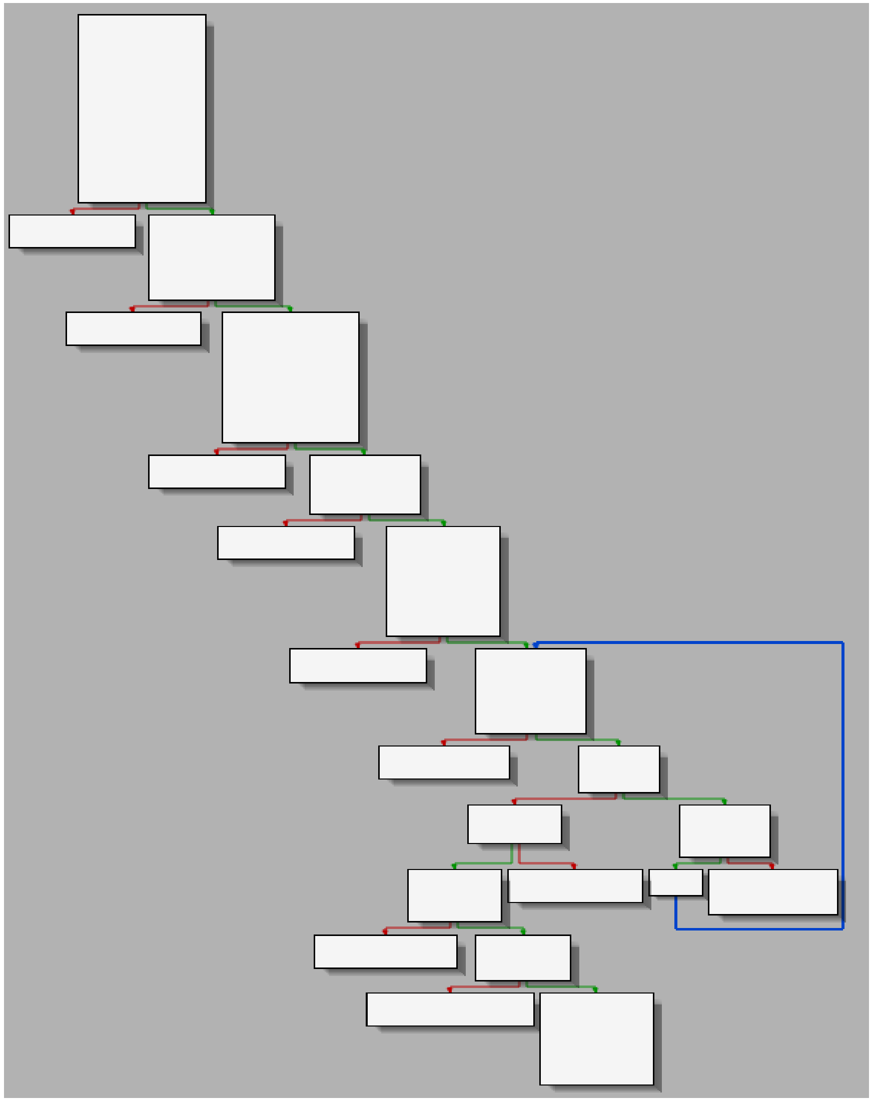
```
loc_80483D4:
cmp     [ebp+var_4], 9
jle     short loc_80483CA
```

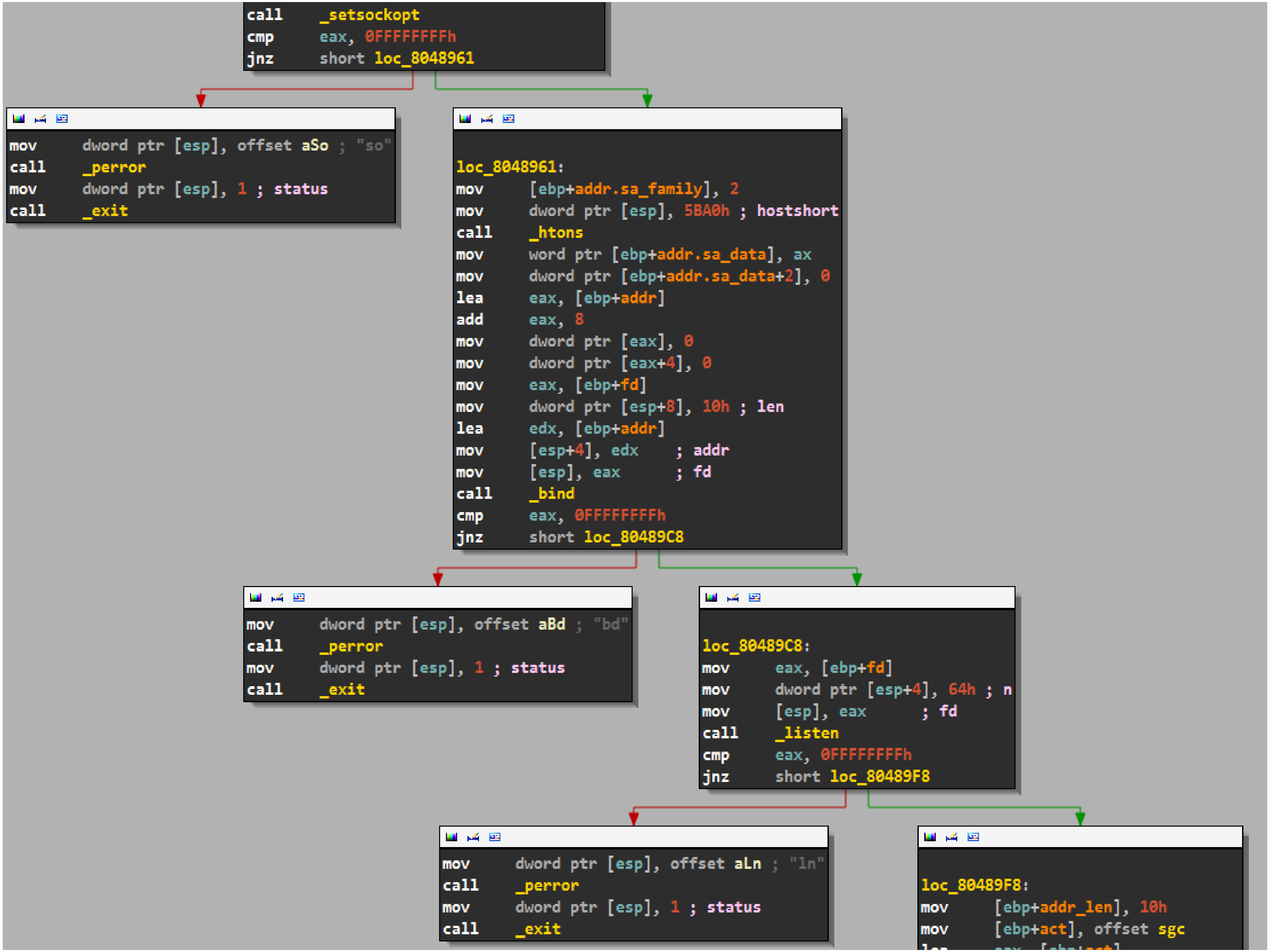
```
loc_80483CA:
mov     eax, [ebp+var_4]
add     [ebp+var_8], eax
add     [ebp+var_4], 1
```

```
mov     eax, [ebp+var_4]
leave
retn
main endp
```

Recognizing Patterns

- Without a single instruction, it's clear what is happening at a high level here
- This common "stair step" graph structure is a series of calls/checks that error out on failure





```
call    _setsockopt
cmp     eax, 0FFFFFFFFh
jnz     short loc_8048961
```

```
mov     dword ptr [esp], offset aSo ; "so"
call    _perror
mov     dword ptr [esp], 1 ; status
call    _exit
```

```
loc_8048961:
mov     [ebp+addr.sa_family], 2
mov     dword ptr [esp], 5BA0h ; hostshort
call    _htons
mov     word ptr [ebp+addr.sa_data], ax
mov     dword ptr [ebp+addr.sa_data+2], 0
lea     eax, [ebp+addr]
add     eax, 8
mov     dword ptr [eax], 0
mov     dword ptr [eax+4], 0
mov     eax, [ebp+fd]
mov     dword ptr [esp+8], 10h ; len
lea     edx, [ebp+addr]
mov     [esp+4], edx    ; addr
mov     [esp], eax      ; fd
call    _bind
cmp     eax, 0FFFFFFFFh
jnz     short loc_80489C8
```

```
mov     dword ptr [esp], offset aBd ; "bd"
call    _perror
mov     dword ptr [esp], 1 ; status
call    _exit
```

```
loc_80489C8:
mov     eax, [ebp+fd]
mov     dword ptr [esp+4], 64h ; n
mov     [esp], eax      ; fd
call    _listen
cmp     eax, 0FFFFFFFFh
jnz     short loc_80489F8
```

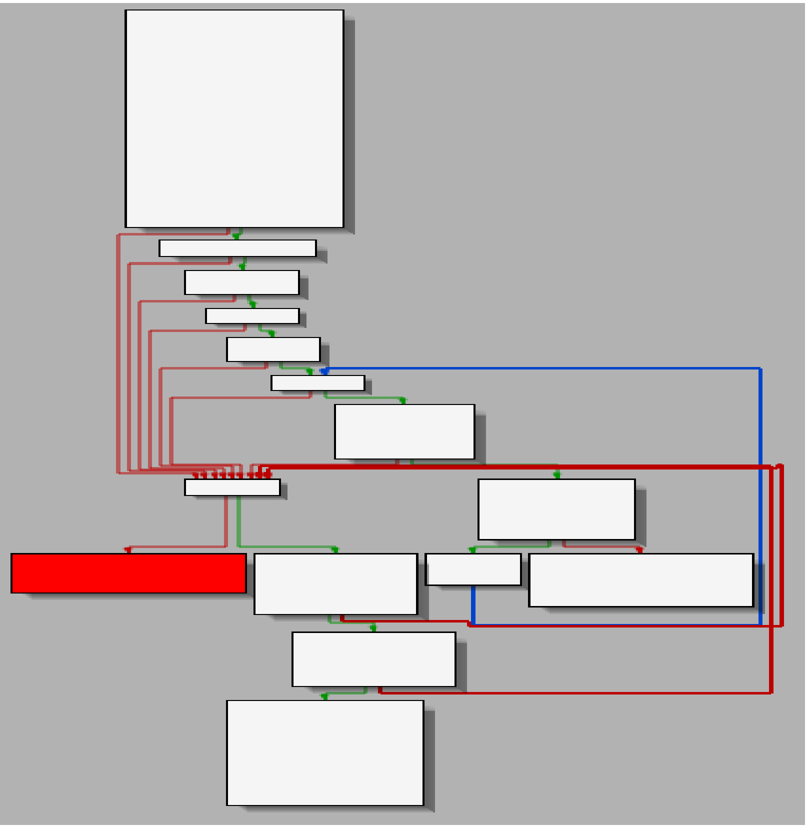
```
mov     dword ptr [esp], offset aLn ; "ln"
call    _perror
mov     dword ptr [esp], 1 ; status
call    _exit
```

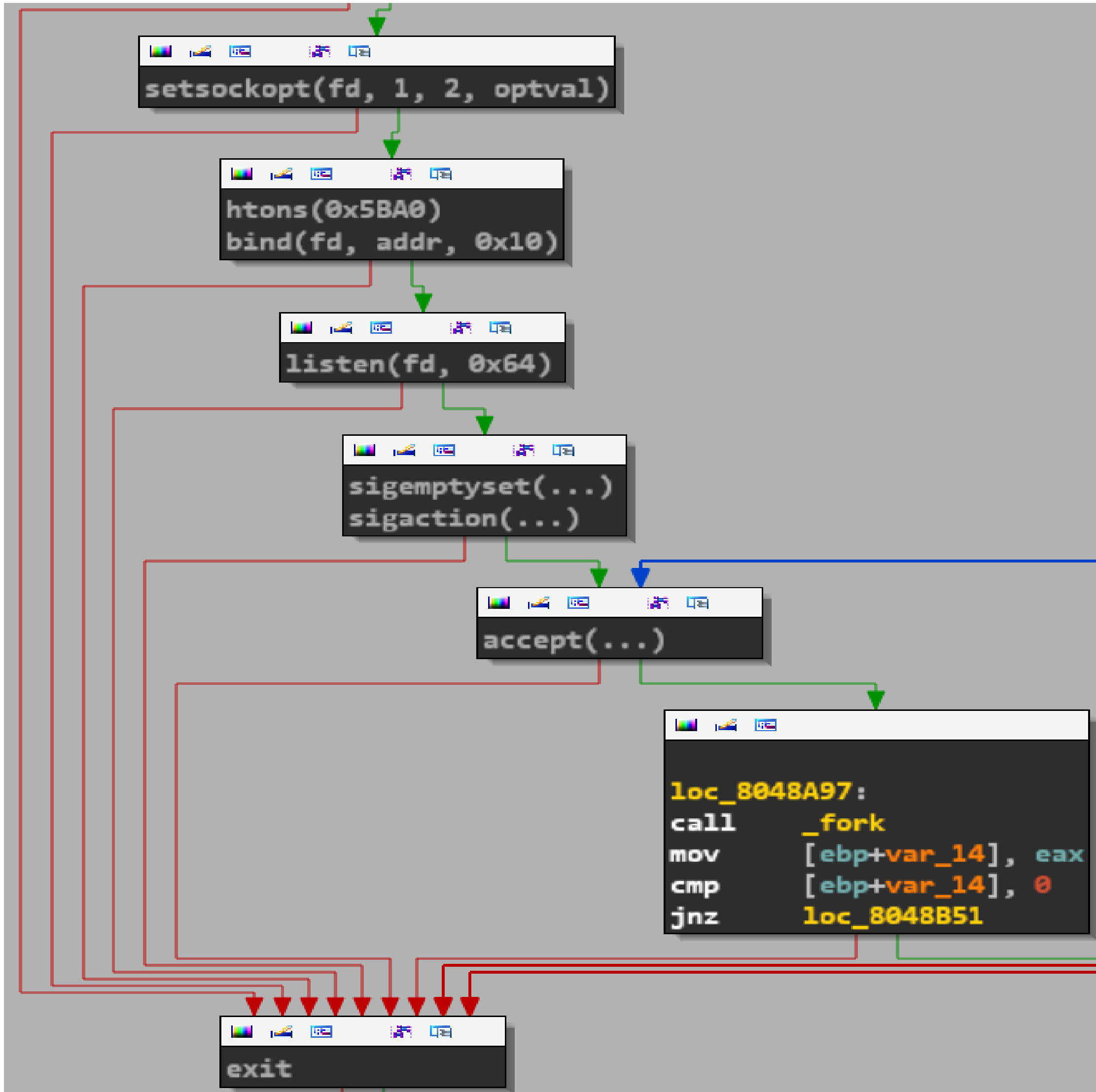
```
loc_80489F8:
mov     [ebp+addr_len], 10h
mov     [ebp+act], offset sgc
lea     eax, [ebp+act]
```

IDA

- IDA rocks…
- We can do many things, including grouping a set of nodes, color coding them, and renaming them
- Knowing that all these checks error out on failure we can simplify the graph

IDA – Simplifying the graph





```
setsockopt(fd, 1, 2, optval)
```

```
htons(0x5BA0)
bind(fd, addr, 0x10)
```

```
listen(fd, 0x64)
```

```
sigemptyset(...)
sigaction(...)
```

```
accept(...)
```

```
loc_8048A97:
call    _fork
mov     [ebp+var_14], eax
cmp     [ebp+var_14], 0
jnz     loc_8048B51
```

```
exit
```

IDA

- I could spend on all day on IDA, too much information to put into slides without making it a pure IDA talk
- *Live demo goes here*
  - How to use IDA
  - Go over variable renaming, function protocol modification, comments, coloring, grouping, sections, string, imports, etc.

Exercise 3

- Can you figure out the correct input to get the key program to print the key?
- Use the executable number_checker.exe

Debugging

Debugging

- Everything covered so far has been static analysis
- Now we’ll cover dynamic analysis through debugging

Debugging

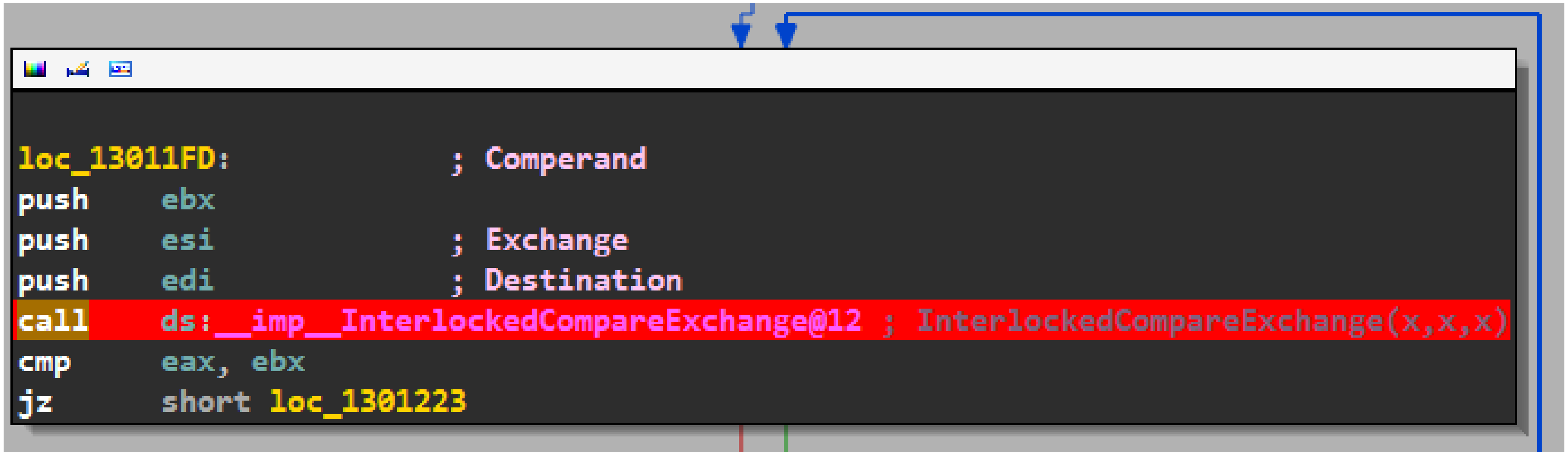
- Remember
- A good debugger will have several useful features
  - Set breakpoints
  - Step into / over
  - Show loaded modules, SEH chain, etc.
  - Memory searching
  - …
- WinDBG, OllyDBG, Immunity, IDA, GDB, etc. are good debuggers

Dynamic Analysis – Quick Note

- Keep in mind…
- *You control everything!*
- If you want to skip over an instruction, or a function call, do it!
- If you want to bypass the “authentication” method or make it return true… you can!
- You can change register contents and memory values, whatever you want.
- You can even patch programs (make changes and save it to a new executable).

Dynamic Analysis - IDA

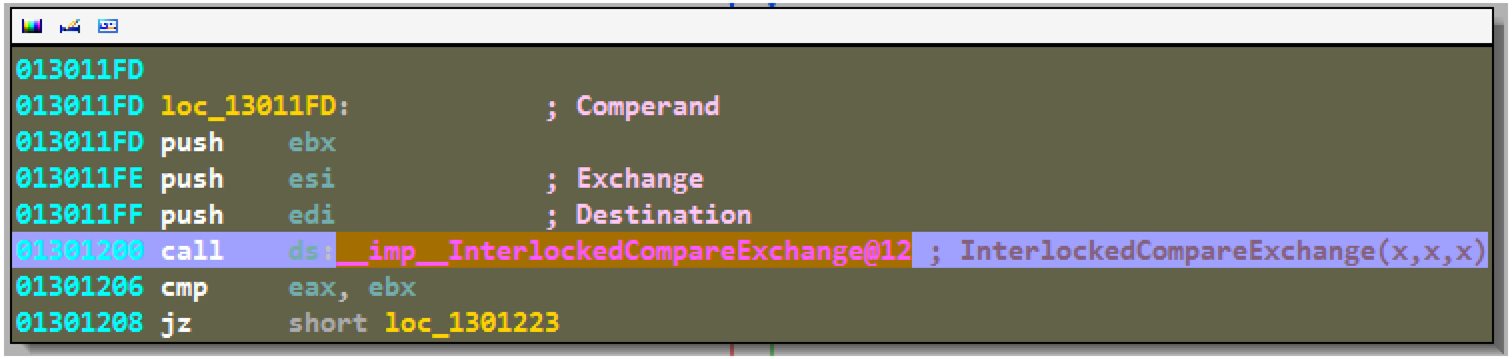
- F2 will set a breakpoint in IDA, Olly, Immunity



```
loc_13011FD:                    ; Comperand
push    ebx
push    esi                     ; Exchange
push    edi                     ; Destination
call    ds:__imp__InterlockedCompareExchange@12 ; InterlockedCompareExchange(x,x,x)
cmp     eax, ebx
jz      short loc_1301223
```

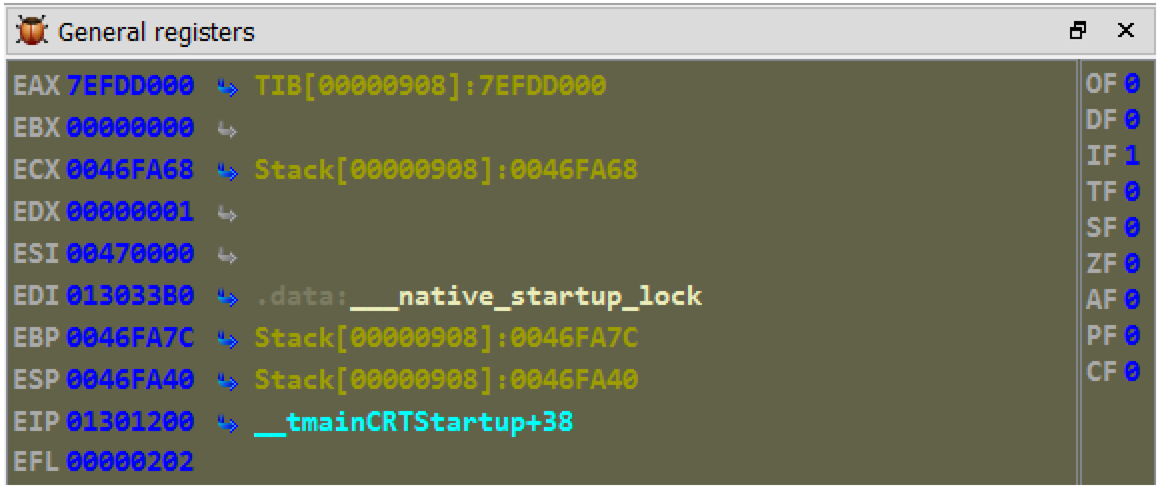
Dynamic Analysis - IDA

- The breakpoint has been hit, execution is stopped



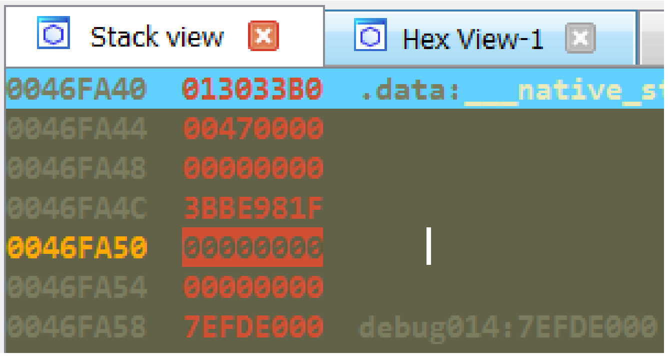
```
013011FD
013011FD loc_13011FD:                    ; Comperand
013011FD push    ebx
013011FE push    esi                     ; Exchange
013011FF push    edi                     ; Destination
01301200 call    ds:__imp__InterlockedCompareExchange@12 ; InterlockedCompareExchange(x,x,x)
01301206 cmp     eax, ebx
01301208 jz      short loc_1301223
```

- The registers



```
EAX 7EFDD000 ↳ TIB[00000908]:7EFDD000          OF 0
EBX 00000000 ↳                                 DF 0
ECX 0046FA68 ↳ Stack[00000908]:0046FA68        IF 1
EDX 00000001 ↳                                 TF 0
ESI 00470000 ↳                                 SF 0
EDI 013033B0 ↳ .data:___native_startup_lock    ZF 0
EBP 0046FA7C ↳ Stack[00000908]:0046FA7C        AF 0
ESP 0046FA40 ↳ Stack[00000908]:0046FA40        PF 0
EIP 01301200 ↳ __tmainCRTStartup+38            CF 0
EFL 00000202
```

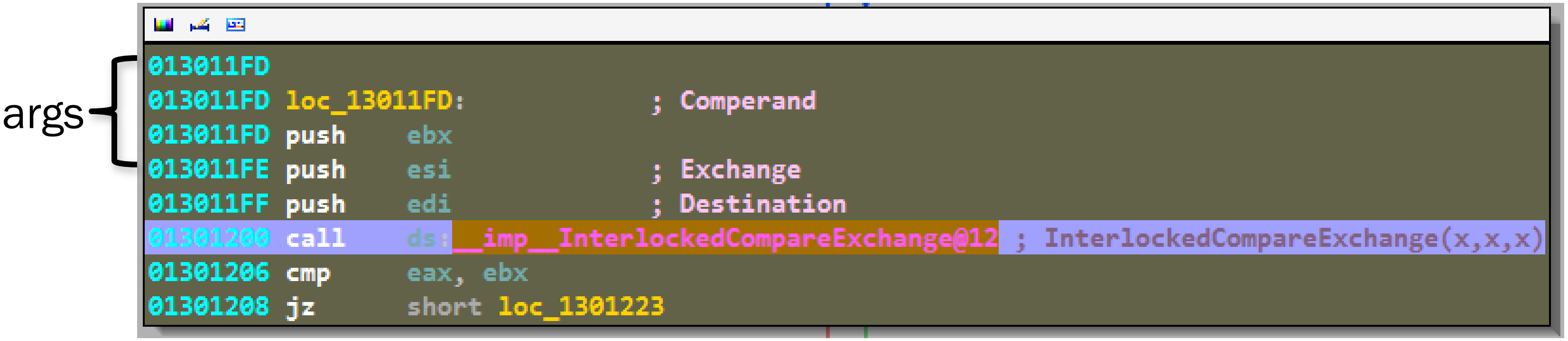
- The stack



```
0046FA40 013033B0 .data:___native_st
0046FA44 00470000
0046FA48 00000000
0046FA4C 3BBE981F
0046FA50 00000000
0046FA54 00000000
0046FA58 7EFDE000 debug014:7EFDE000
```

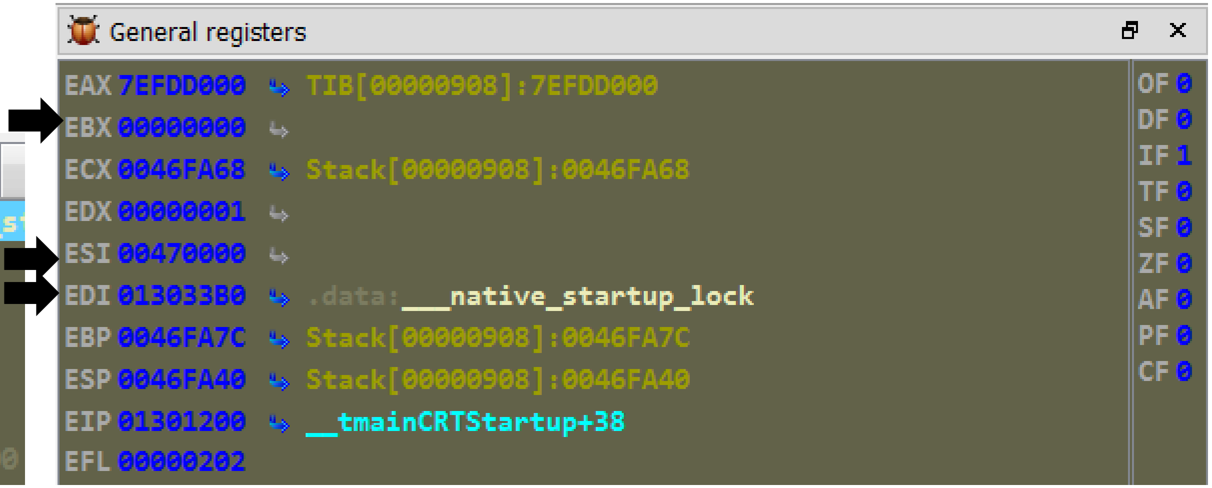
Dynamic Analysis - IDA

- The breakpoint has been hit, execution is stopped



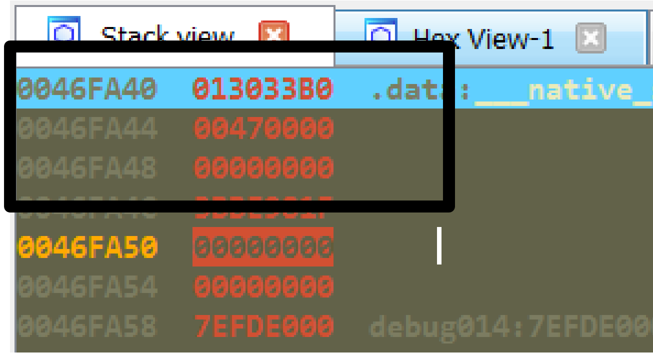
```
013011FD
013011FD loc_13011FD:                   ; Comperand
013011FD push    ebx
013011FE push    esi                    ; Exchange
013011FF push    edi                    ; Destination
01301200 call    ds:__imp__InterlockedCompareExchange@12 ; InterlockedCompareExchange(x,x,x)
01301206 cmp     eax, ebx
01301208 jz      short loc_1301223
```

- The registers



```
EAX 7EFDD000 ↳ TIB[00000908]:7EFDD000          OF 0
EBX 00000000 ↳                                  DF 0
ECX 0046FA68 ↳ Stack[00000908]:0046FA68         IF 1
EDX 00000001 ↳                                  TF 0
ESI 00470000 ↳                                  SF 0
EDI 013033B0 ↳ .data:___native_startup_lock     ZF 0
EBP 0046FA7C ↳ Stack[00000908]:0046FA7C         AF 0
ESP 0046FA40 ↳ Stack[00000908]:0046FA40         PF 0
EIP 01301200 ↳ __tmainCRTStartup+38             CF 0
EFL 00000202
```

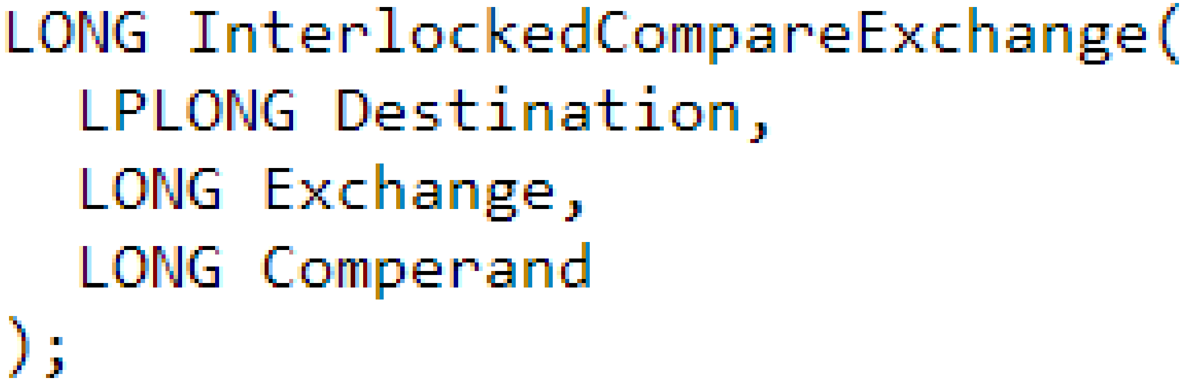
- The stack



```
0046FA40  013033B0  .data:___native_st
0046FA44  00470000
0046FA48  00000000
0046FA4C  [illegible]
0046FA50  00000000
0046FA54  00000000
0046FA58  7EFDE000  debug014:7EFDE000
```

Dynamic Analysis - IDA

- We can now see the function call is
- InterlockedCompareExchange(__native_startup_lock, 0x47000, 0)
- Looking at the MSDN site for the prototype :



```
LONG InterlockedCompareExchange(
  LPLONG Destination,
  LONG Exchange,
  LONG Comperand
);
```

Dynamic Analysis - IDA

- Knowing the data types of the parameters, we can trace back up through the program where the values in ebx, esi and edi came from
- Then we can rename those values to something useful
- Just looking at calls, figuring out their arguments, and tracing back to fill in the data types can **really** help figure out most of the functions

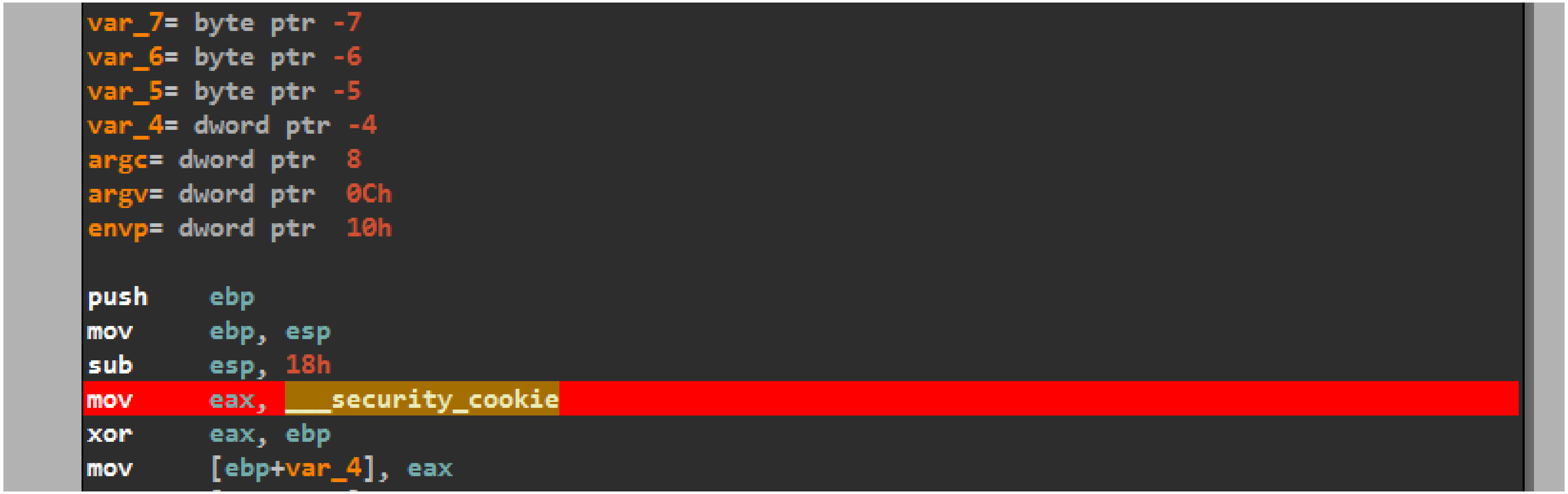
Exercise 4

- We’ll again use the number_checker.exe binary for this exercise
- Can you bypass the key check entirely?

- In CTFs a lot of times we can see where the key get’s printed, and we’ll try to just jump directly to that function, or make checks return True/False depending on where we want to go.
  - Usually can get a quick low point problem this way ;)

Exercise 4 - Solution

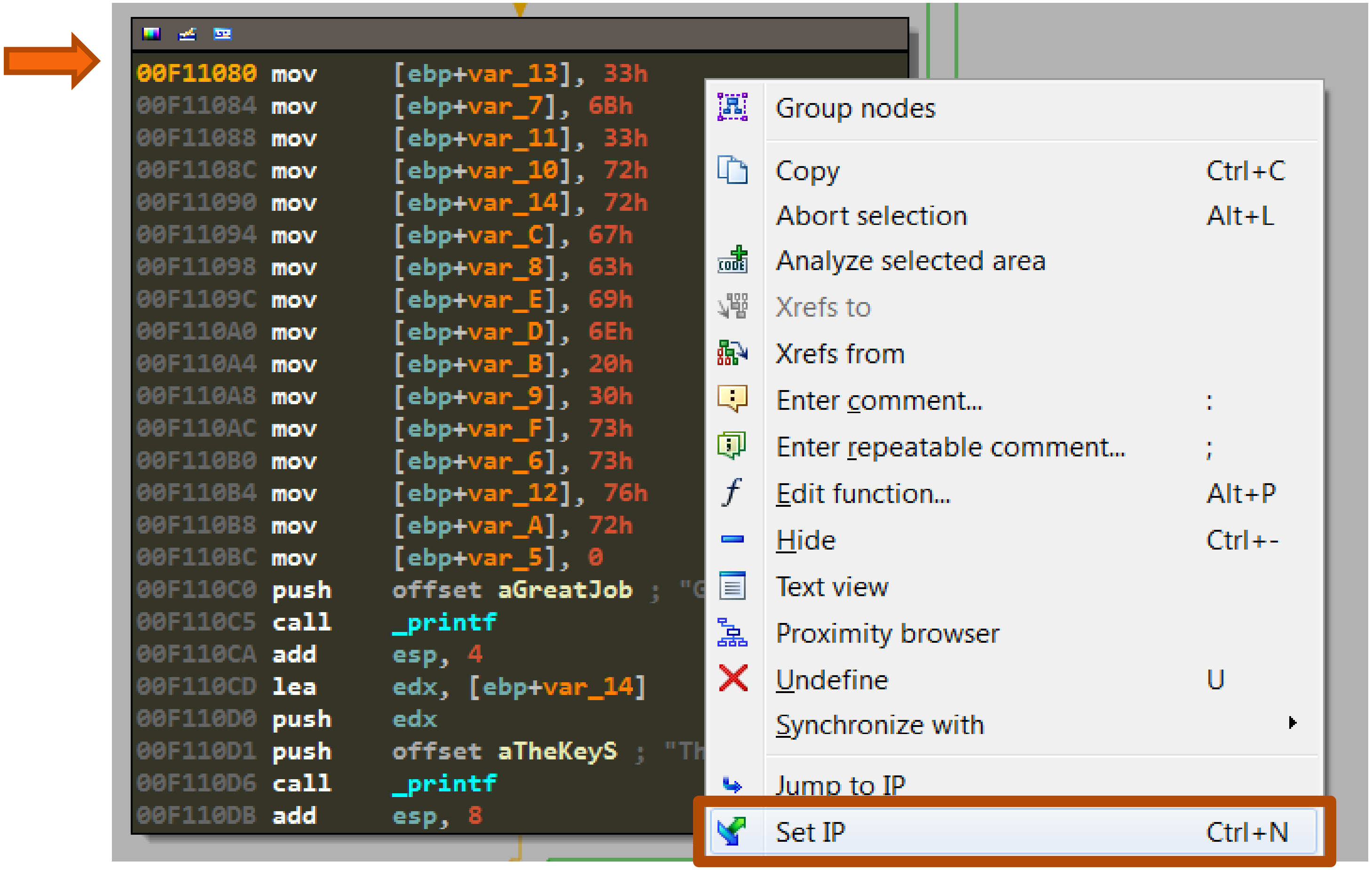
- Set a breakpoint at the beginning of the function (f2)



```
var_7= byte ptr -7
var_6= byte ptr -6
var_5= byte ptr -5
var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 18h
mov     eax, ___security_cookie
xor     eax, ebp
mov     [ebp+var_4], eax
```

Exercise 4 - Solution

- When execution is stopped, find where you want to jump to, and right click -> set ip



```
00F11080 mov     [ebp+var_13], 33h
00F11084 mov     [ebp+var_7], 6Bh
00F11088 mov     [ebp+var_11], 33h
00F1108C mov     [ebp+var_10], 72h
00F11090 mov     [ebp+var_14], 72h
00F11094 mov     [ebp+var_C], 67h
00F11098 mov     [ebp+var_8], 63h
00F1109C mov     [ebp+var_E], 69h
00F110A0 mov     [ebp+var_D], 6Eh
00F110A4 mov     [ebp+var_B], 20h
00F110A8 mov     [ebp+var_9], 30h
00F110AC mov     [ebp+var_F], 73h
00F110B0 mov     [ebp+var_6], 73h
00F110B4 mov     [ebp+var_12], 76h
00F110B8 mov     [ebp+var_A], 72h
00F110BC mov     [ebp+var_5], 0
00F110C0 push    offset aGreatJob ; "G
00F110C5 call    _printf
00F110CA add     esp, 4
00F110CD lea     edx, [ebp+var_14]
00F110D0 push    edx
00F110D1 push    offset aTheKeyS ; "Th
00F110D6 call    _printf
00F110DB add     esp, 8
```

Dynamic Analysis - Debuggers

- Most of the Windows debuggers are similar
  - Same windows, same hotkeys, etc.
  - Except WinDBG, WinDBG is more GDB like
- GDB is similar, but is command line
- We'll cover some simple GDB usage

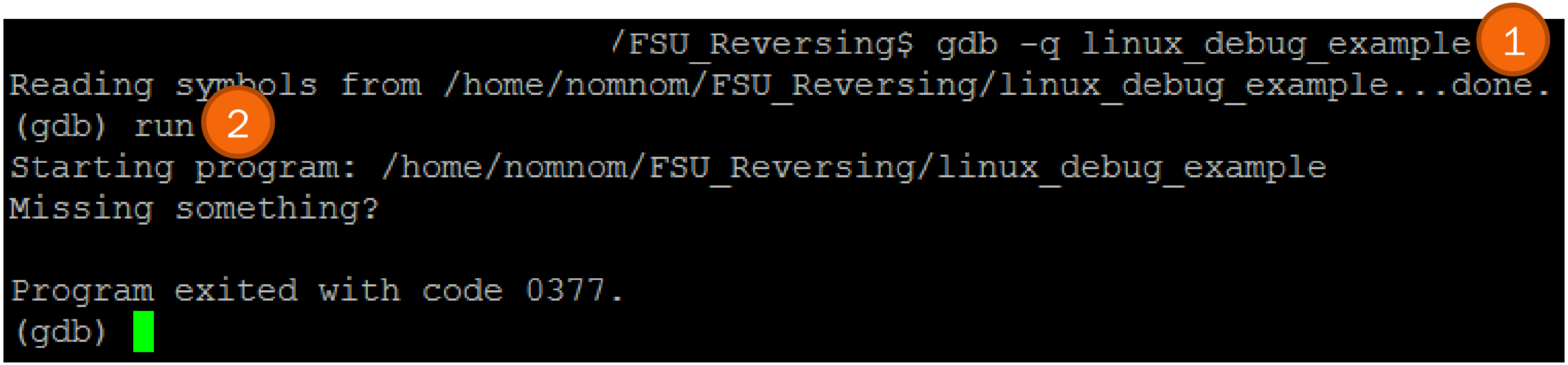
Dynamic Analysis - GDB

- Starting GDB and launching the application
  - With and without arguments

Command	Description
gdb ./my_program	Launch gdb, debug my_program
gdb --args ./my_program arg1 arg2	Launch gdb, debug my_program, passing two arguments
run	Run the application
run arg1 arg2	Run the application, pass two args
run $(python –c “print ‘A’*1000”)	Run the application, pass one arg, just like regular shell execution

Dynamic Analysis - GDB

- 1. Launch GDB with the program we want to debug
- 2. Run it

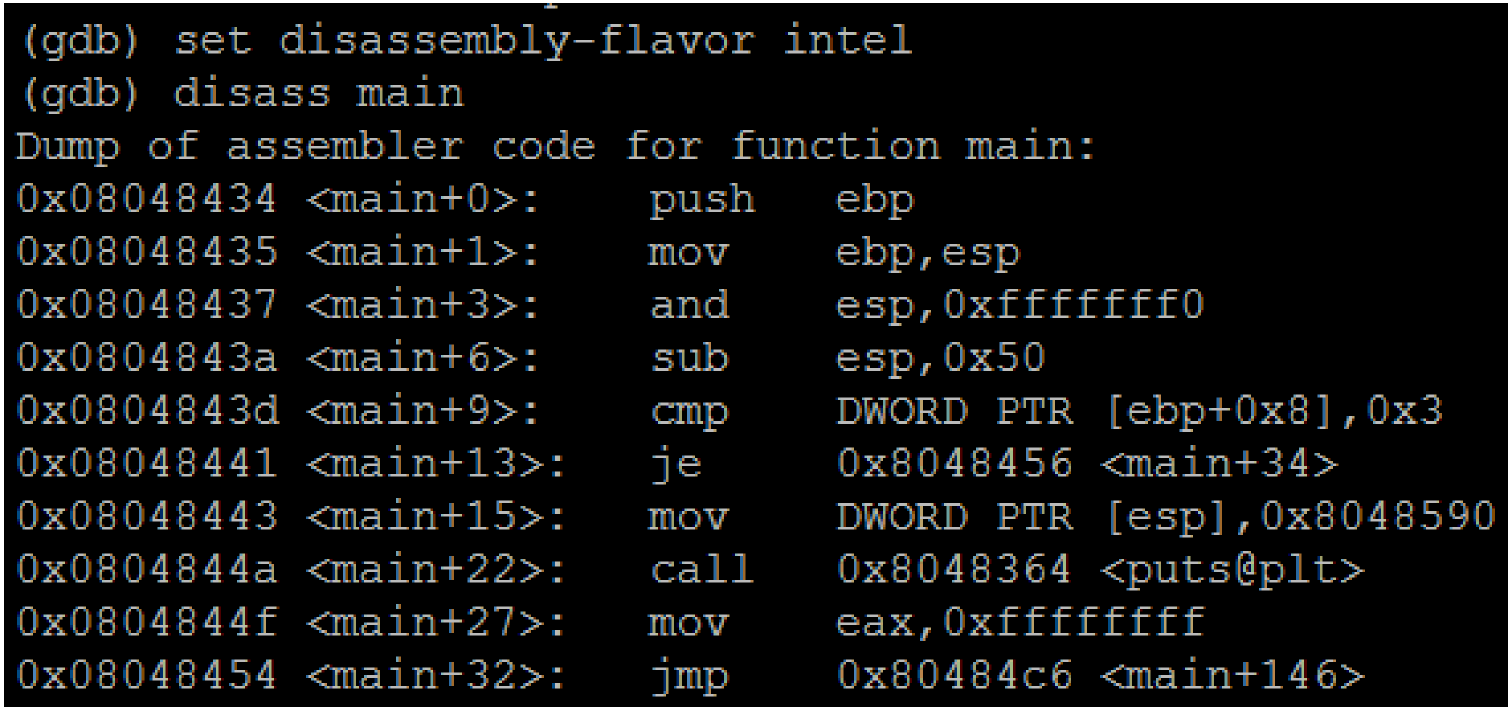


```
                                /FSU_Reversing$ gdb -q linux_debug_example
Reading symbols from /home/nomnom/FSU_Reversing/linux_debug_example...done.
(gdb) run
Starting program: /home/nomnom/FSU_Reversing/linux_debug_example
Missing something?

Program exited with code 0377.
(gdb) 
```

- Hmm… we need more information
  - (I would just open it in IDA, but we’re trying to learn GDB here!)

Dynamic Analysis - GDB

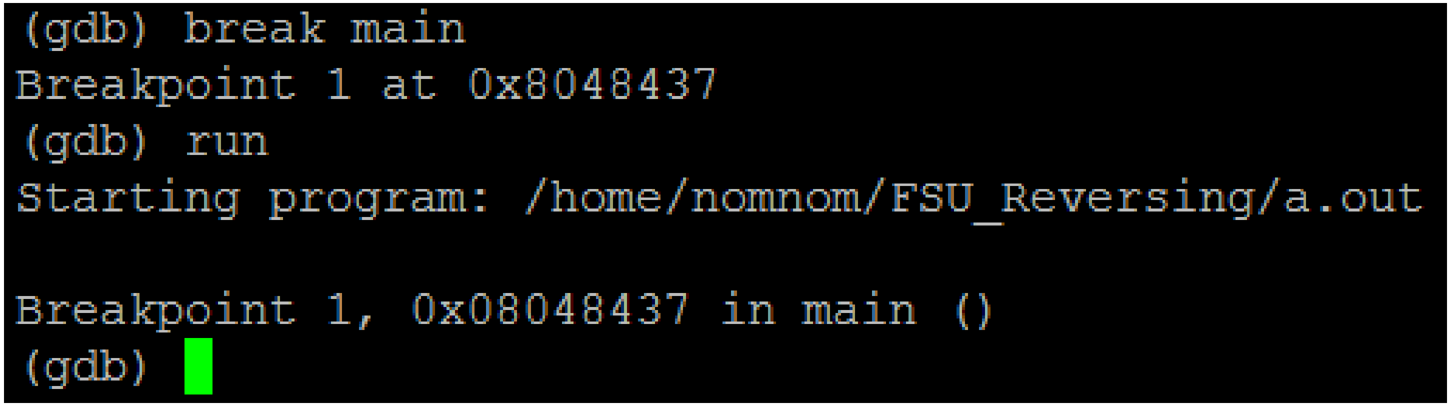
Command	Description
set disassembly-flavor intel	Use Intel syntax
disas [function_name]	Disassemple the chosen function



```
(gdb) set disassembly-flavor intel
(gdb) disass main
Dump of assembler code for function main:
0x08048434 <main+0>:    push   ebp
0x08048435 <main+1>:    mov    ebp,esp
0x08048437 <main+3>:    and    esp,0xfffffff0
0x0804843a <main+6>:    sub    esp,0x50
0x0804843d <main+9>:    cmp    DWORD PTR [ebp+0x8],0x3
0x08048441 <main+13>:   je     0x8048456 <main+34>
0x08048443 <main+15>:   mov    DWORD PTR [esp],0x8048590
0x0804844a <main+22>:   call   0x8048364 <puts@plt>
0x0804844f <main+27>:   mov    eax,0xffffffff
0x08048454 <main+32>:   jmp    0x80484c6 <main+146>
```

Dynamic Analysis - GDB

Command	Description
break main	Set a breakpoint on the function “main”
break *0x12345678	Set a breakpoint on the address 0x…
info breakpoints	Show information regarding breakpoints
delete breakpoint 2	Delete breakpoint 2
delete breakpoints	Delete all breakpoints

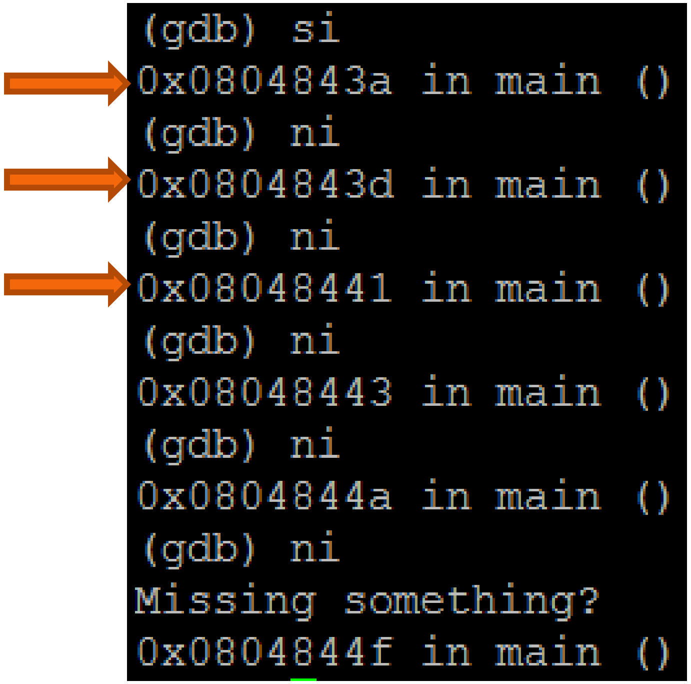


```
(gdb) break main
Breakpoint 1 at 0x8048437
(gdb) run
Starting program: /home/nomnom/FSU_Reversing/a.out

Breakpoint 1, 0x08048437 in main ()
(gdb) █
```

Dynamic Analysis - GDB

Commands	Description
si	Step Instruction. Execute to next instruction, go *into* functions
ni	Next Instruction. Execute to next instruction, go *over* functions

- Look at the addresses
- We're manually stepping through the instructions

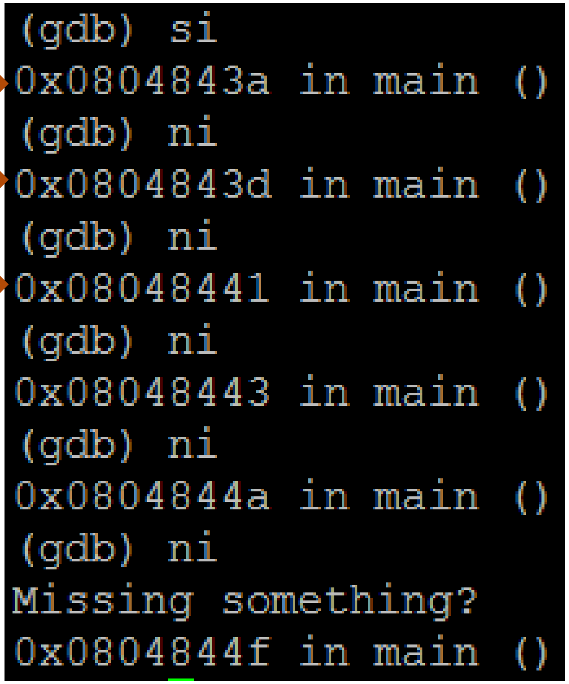


```
(gdb) si
0x0804843a in main ()
(gdb) ni
0x0804843d in main ()
(gdb) ni
0x08048441 in main ()
(gdb) ni
0x08048443 in main ()
(gdb) ni
0x0804844a in main ()
(gdb) ni
Missing something?
0x0804844f in main ()
```

Dynamic Analysis - GDB

Commands	Description
si	Step Instruction. Execute to next instruction, go *into* functions
ni	Next Instruction. Execute to next instruction, go *over* functions

- Look at the addresses
- We’re manually stepping through the instructions



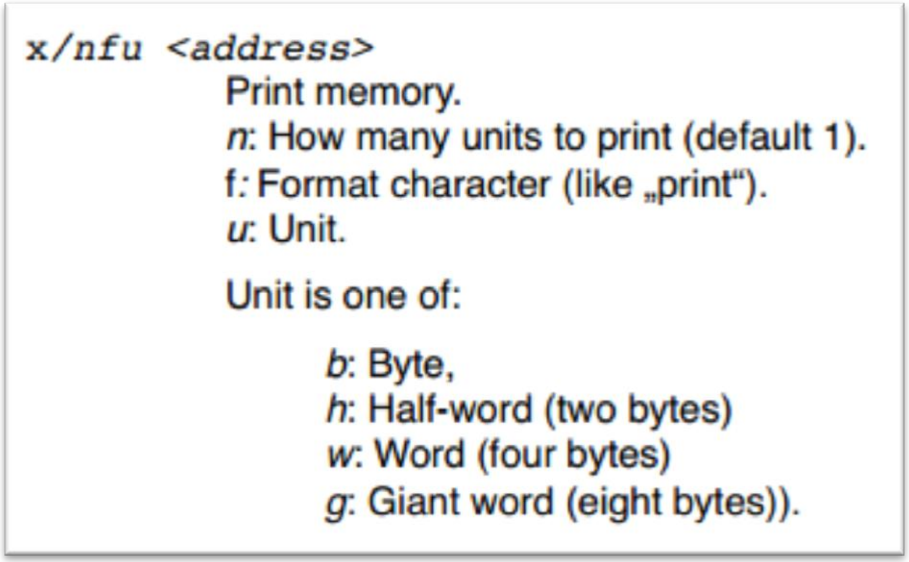
```
(gdb) si
0x0804843a in main ()
(gdb) ni
0x0804843d in main ()
(gdb) ni
0x08048441 in main ()
(gdb) ni
0x08048443 in main ()
(gdb) ni
0x0804844a in main ()
(gdb) ni
Missing something?
0x0804844f in main ()
```

This still isn’t helping us though!

Dynamic Analysis - GDB

- We can disassemble, set breakpoints, and step through the program… but
- We need to
  - See the contents of registers
  - See the contents of memory
  - Modify (if desired)

Dynamic Analysis - GDB



```
x/nfu <address>
```

Print memory.
n: How many units to print (default 1).
f: Format character (like „print“).
u: Unit.

Unit is one of:

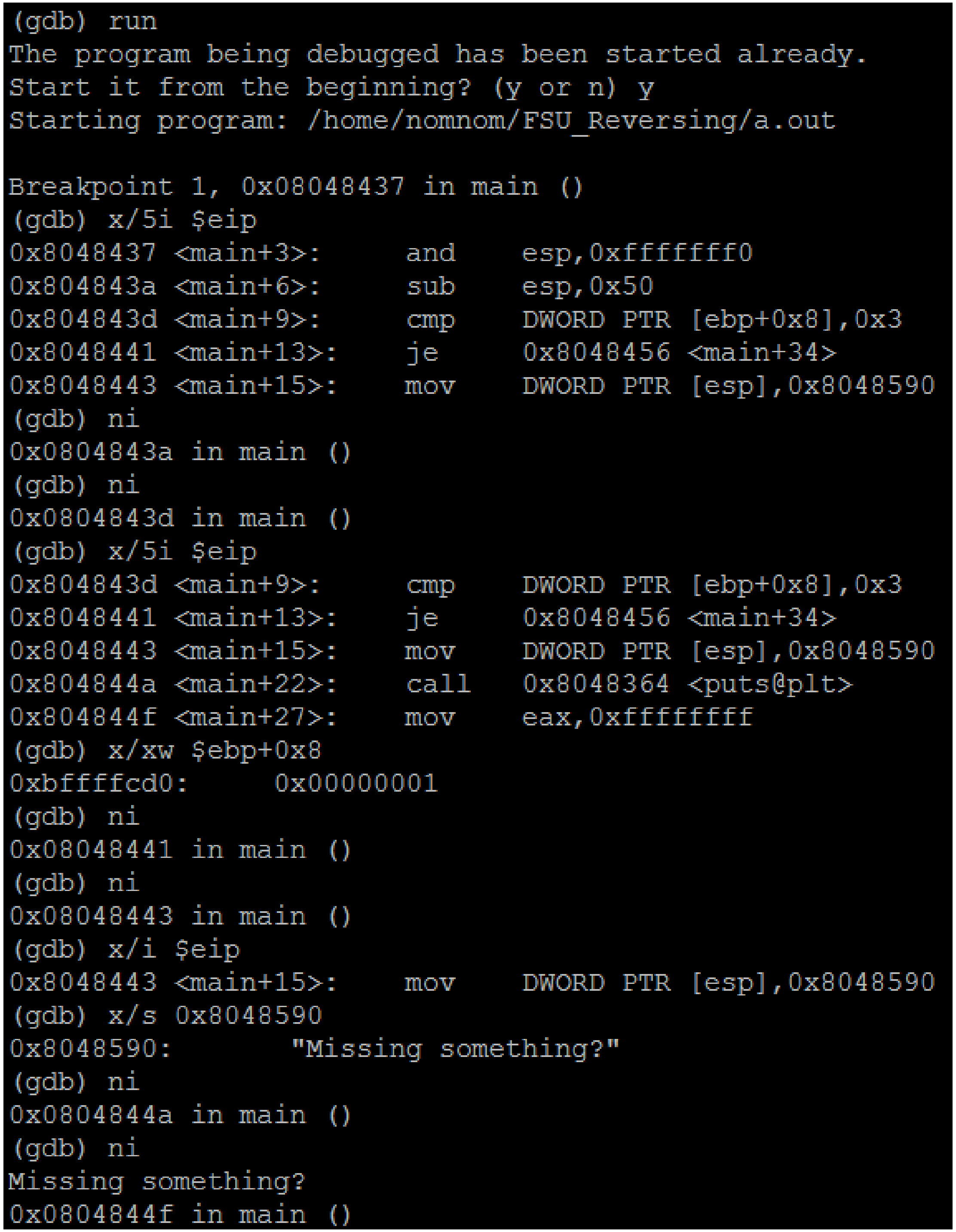
b: Byte,
h: Half-word (two bytes)
w: Word (four bytes)
g: Giant word (eight bytes)).

Image from http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

Dynamic Analysis - GDB

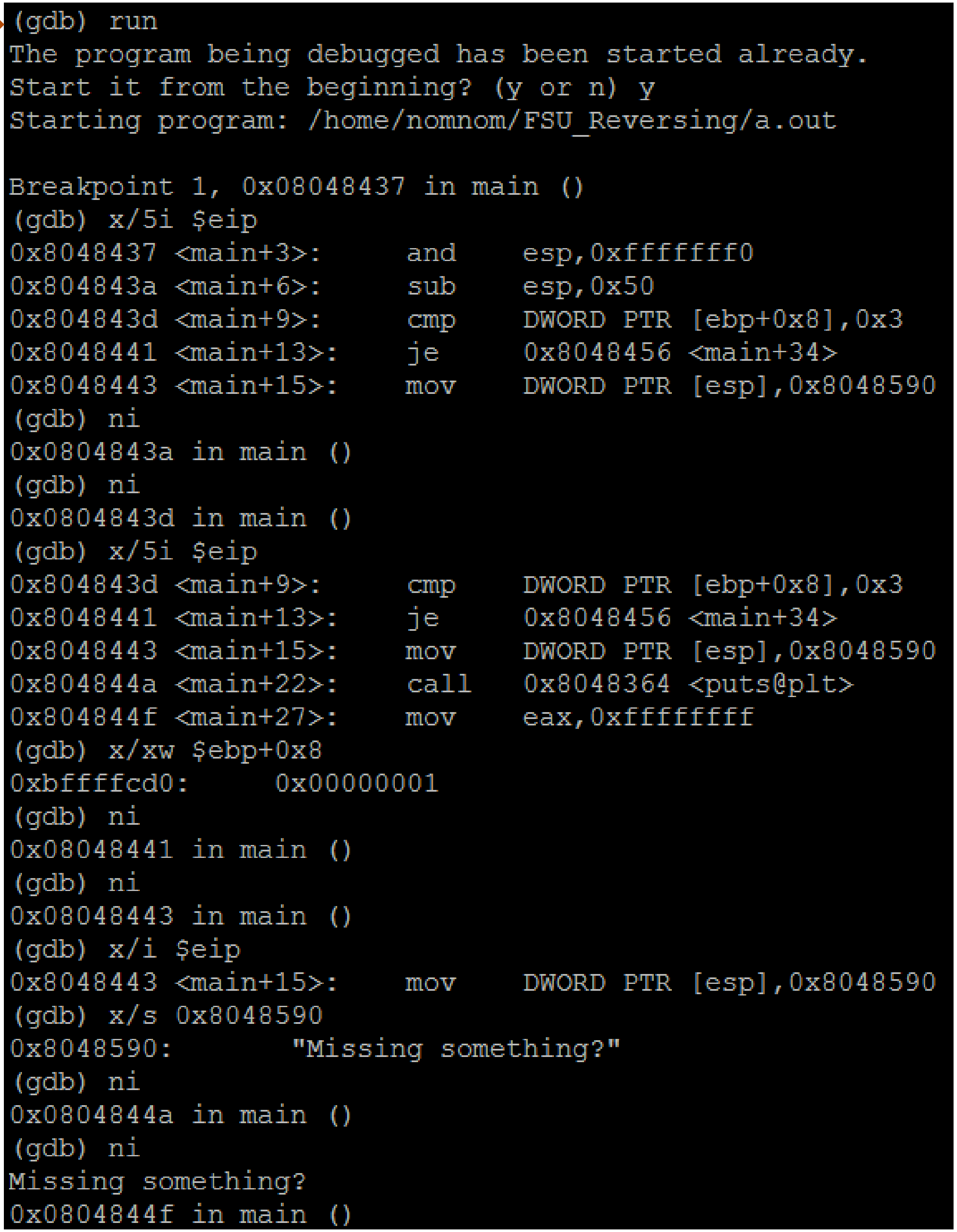
```
x/nfu <address|register>
```

Command	Description
x/5i $eip	Examine 5 instructions at EIP
x/4xw $esp	Examine 4 hex words at ESP
x/s 0x12345678	Examine the string at 0x12345678
x/5b $ecx	Examine 5 bytes at ECX
i r	“info register”, show the values of all registers
i r esp ebp ecx	Show the values of registers ESP, EBP, and ECX



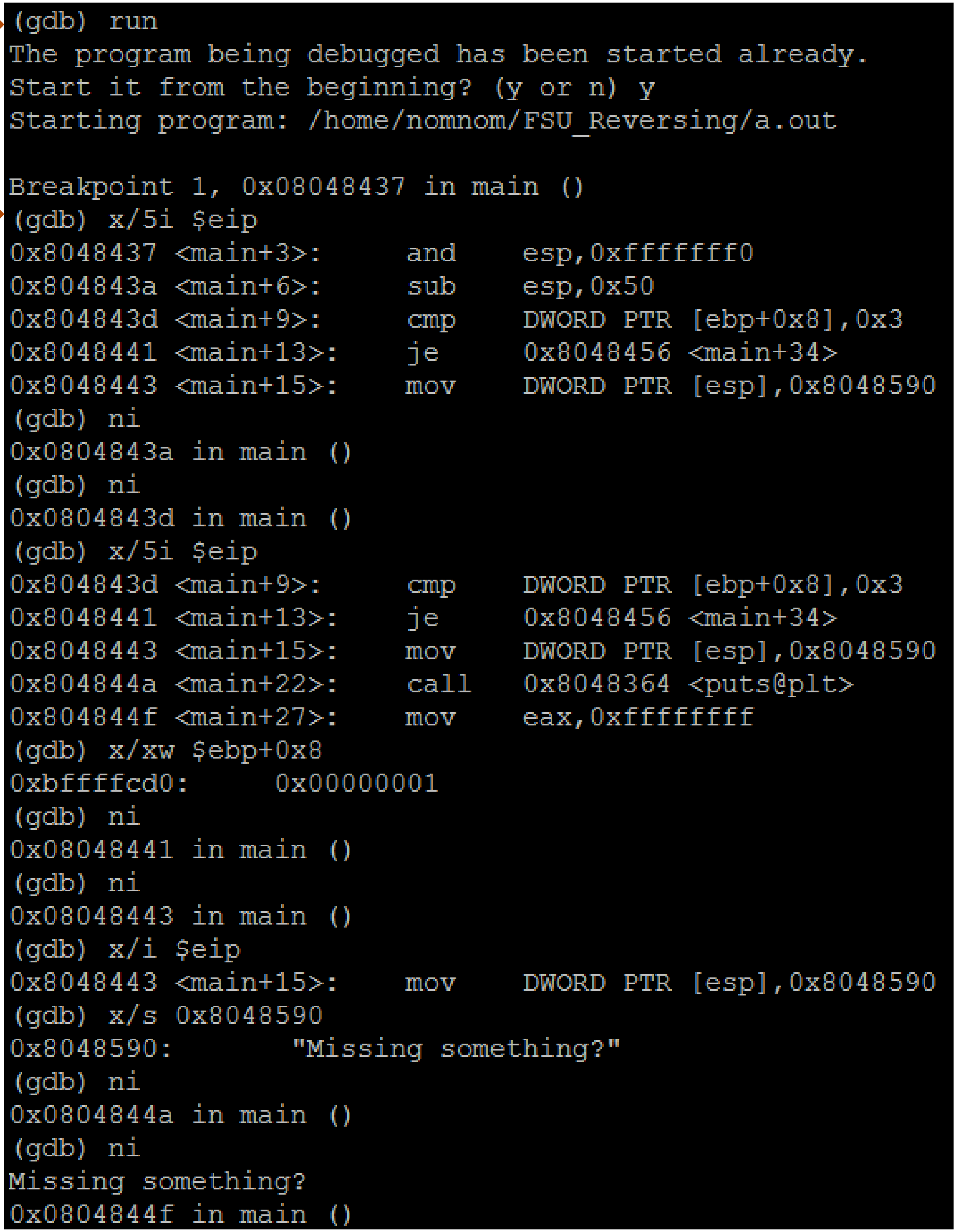
```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nomnom/FSU_Reversing/a.out

Breakpoint 1, 0x08048437 in main ()
(gdb) x/5i $eip
0x8048437 <main+3>:     and    esp,0xfffffff0
0x804843a <main+6>:     sub    esp,0x50
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) ni
0x0804843a in main ()
(gdb) ni
0x0804843d in main ()
(gdb) x/5i $eip
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
0x804844a <main+22>:    call   0x8048364 <puts@plt>
0x804844f <main+27>:    mov    eax,0xffffffff
(gdb) x/xw $ebp+0x8
0xbffffcd0:     0x00000001
(gdb) ni
0x08048441 in main ()
(gdb) ni
0x08048443 in main ()
(gdb) x/i $eip
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) x/s 0x8048590
0x8048590:       "Missing something?"
(gdb) ni
0x0804844a in main ()
(gdb) ni
Missing something?
0x0804844f in main ()
```

1



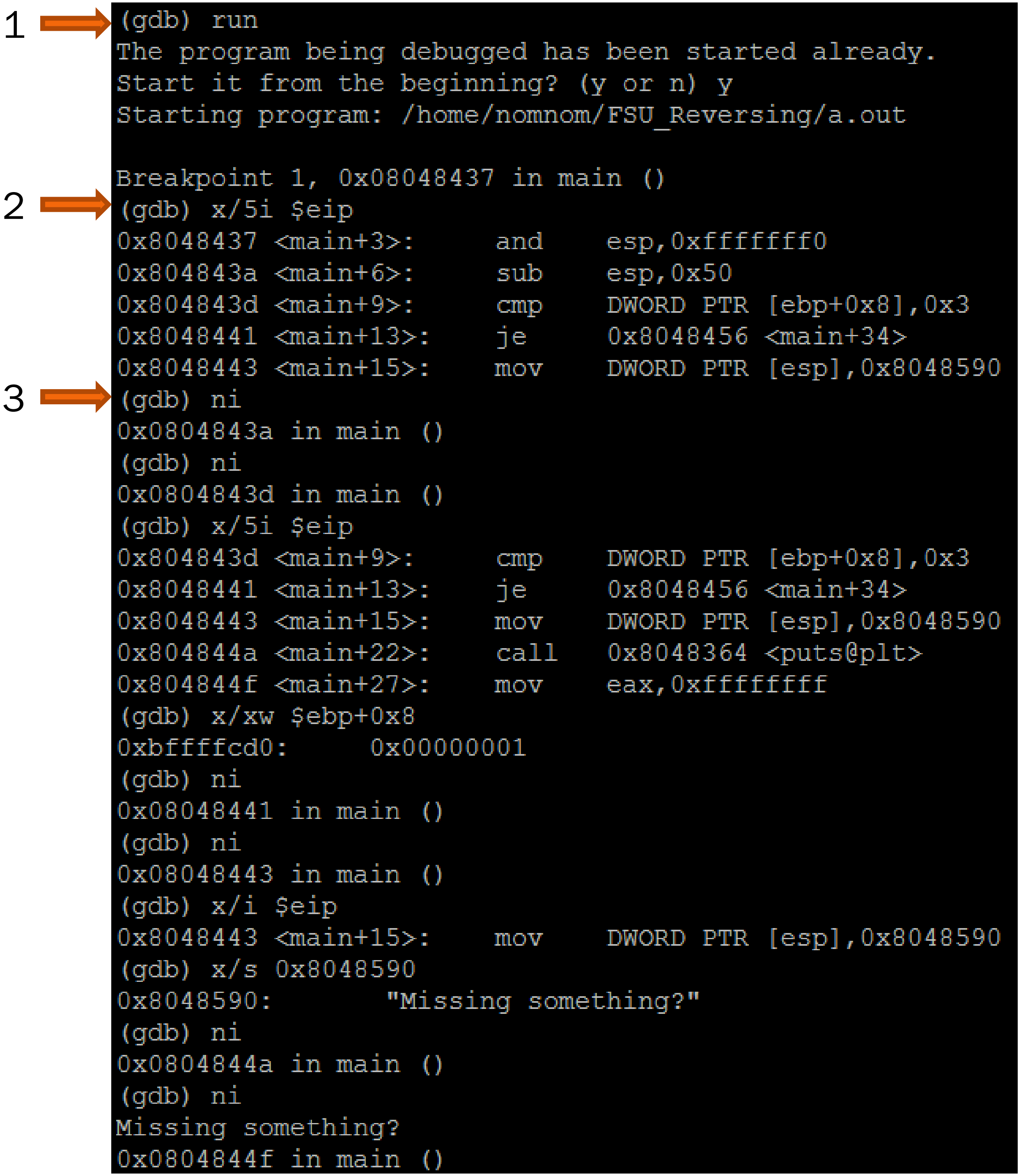
```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nomnom/FSU_Reversing/a.out

Breakpoint 1, 0x08048437 in main ()
(gdb) x/5i $eip
0x8048437 <main+3>:     and    esp,0xfffffff0
0x804843a <main+6>:     sub    esp,0x50
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) ni
0x0804843a in main ()
(gdb) ni
0x0804843d in main ()
(gdb) x/5i $eip
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
0x804844a <main+22>:    call   0x8048364 <puts@plt>
0x804844f <main+27>:    mov    eax,0xffffffff
(gdb) x/xw $ebp+0x8
0xbffffcd0:     0x00000001
(gdb) ni
0x08048441 in main ()
(gdb) ni
0x08048443 in main ()
(gdb) x/i $eip
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) x/s 0x8048590
0x8048590:       "Missing something?"
(gdb) ni
0x0804844a in main ()
(gdb) ni
Missing something?
0x0804844f in main ()
```

1. Run the program



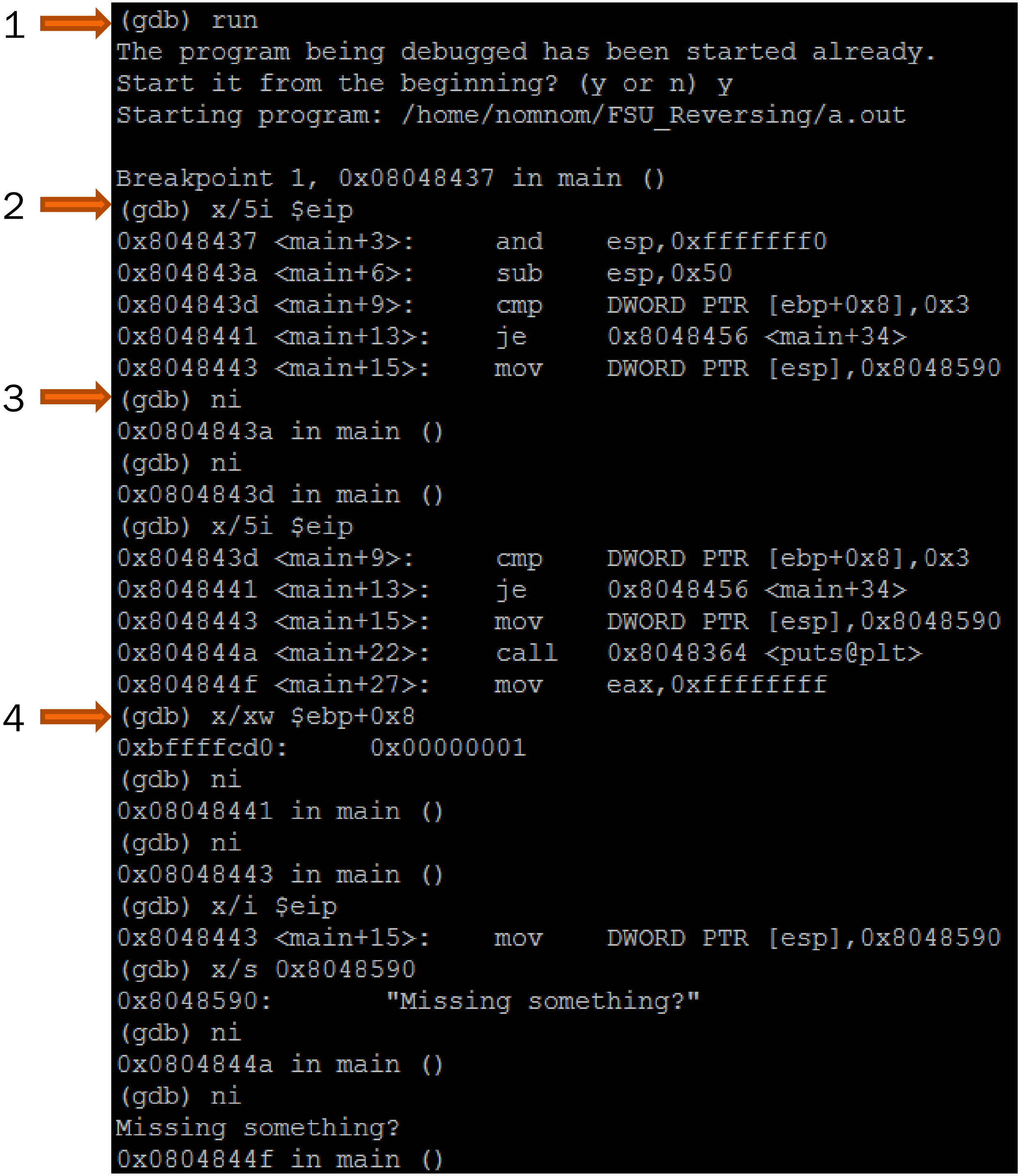
```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nomnom/FSU_Reversing/a.out

Breakpoint 1, 0x08048437 in main ()
(gdb) x/5i $eip
0x8048437 <main+3>:     and    esp,0xfffffff0
0x804843a <main+6>:     sub    esp,0x50
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) ni
0x0804843a in main ()
(gdb) ni
0x0804843d in main ()
(gdb) x/5i $eip
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
0x804844a <main+22>:    call   0x8048364 <puts@plt>
0x804844f <main+27>:    mov    eax,0xffffffff
(gdb) x/xw $ebp+0x8
0xbffffcd0:     0x00000001
(gdb) ni
0x08048441 in main ()
(gdb) ni
0x08048443 in main ()
(gdb) x/i $eip
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) x/s 0x8048590
0x8048590:       "Missing something?"
(gdb) ni
0x0804844a in main ()
(gdb) ni
Missing something?
0x0804844f in main ()
```

1. Run the program
2. Where are we? Check out EIP



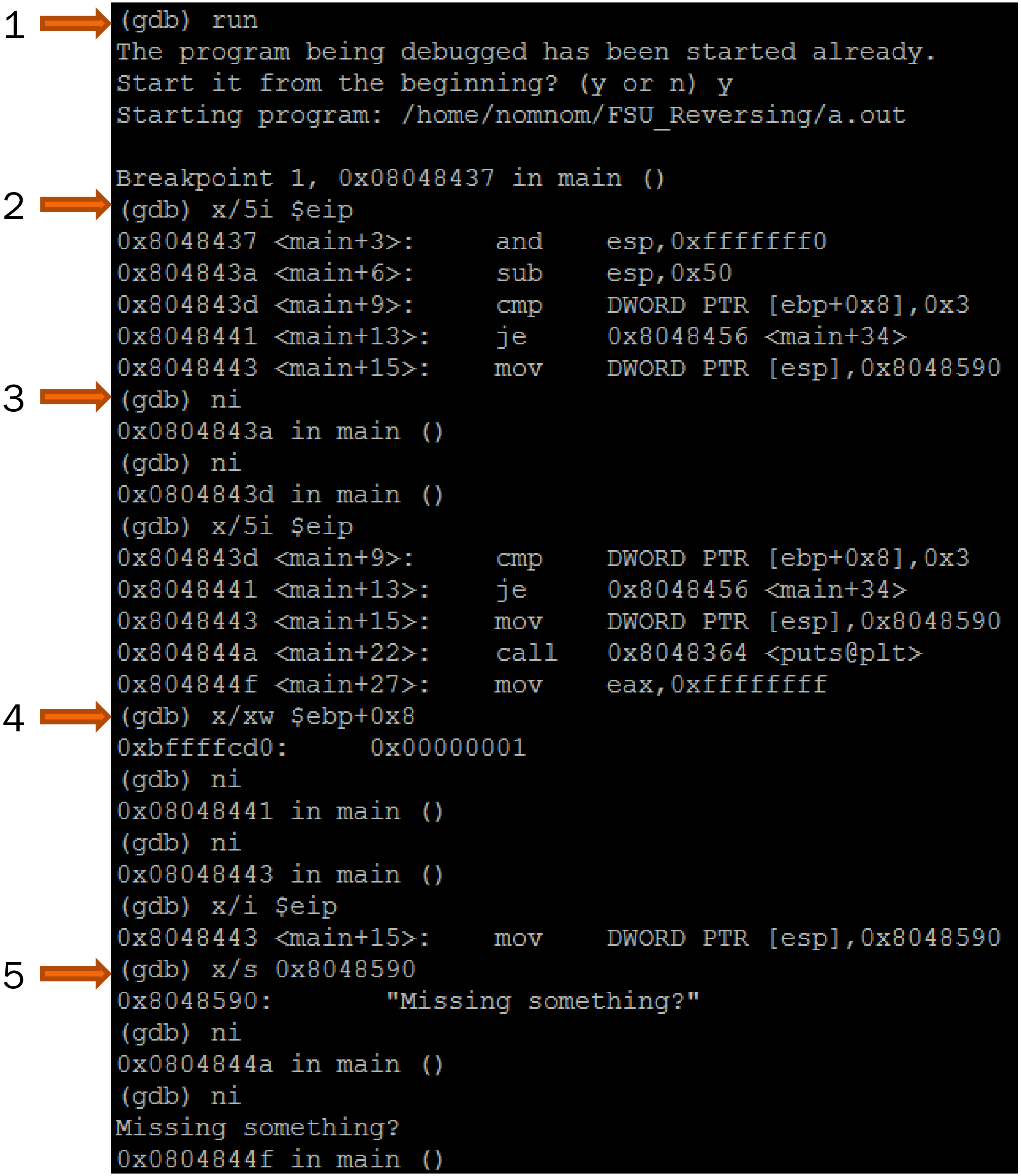
```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nomnom/FSU_Reversing/a.out

Breakpoint 1, 0x08048437 in main ()
(gdb) x/5i $eip
0x8048437 <main+3>:     and    esp,0xfffffff0
0x804843a <main+6>:     sub    esp,0x50
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) ni
0x0804843a in main ()
(gdb) ni
0x0804843d in main ()
(gdb) x/5i $eip
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
0x804844a <main+22>:    call   0x8048364 <puts@plt>
0x804844f <main+27>:    mov    eax,0xffffffff
(gdb) x/xw $ebp+0x8
0xbffffcd0:     0x00000001
(gdb) ni
0x08048441 in main ()
(gdb) ni
0x08048443 in main ()
(gdb) x/i $eip
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) x/s 0x8048590
0x8048590:       "Missing something?"
(gdb) ni
0x0804844a in main ()
(gdb) ni
Missing something?
0x0804844f in main ()
```

1. Run the program
2. Where are we? Check out EIP
3. Continue until we hit an instruction of interest



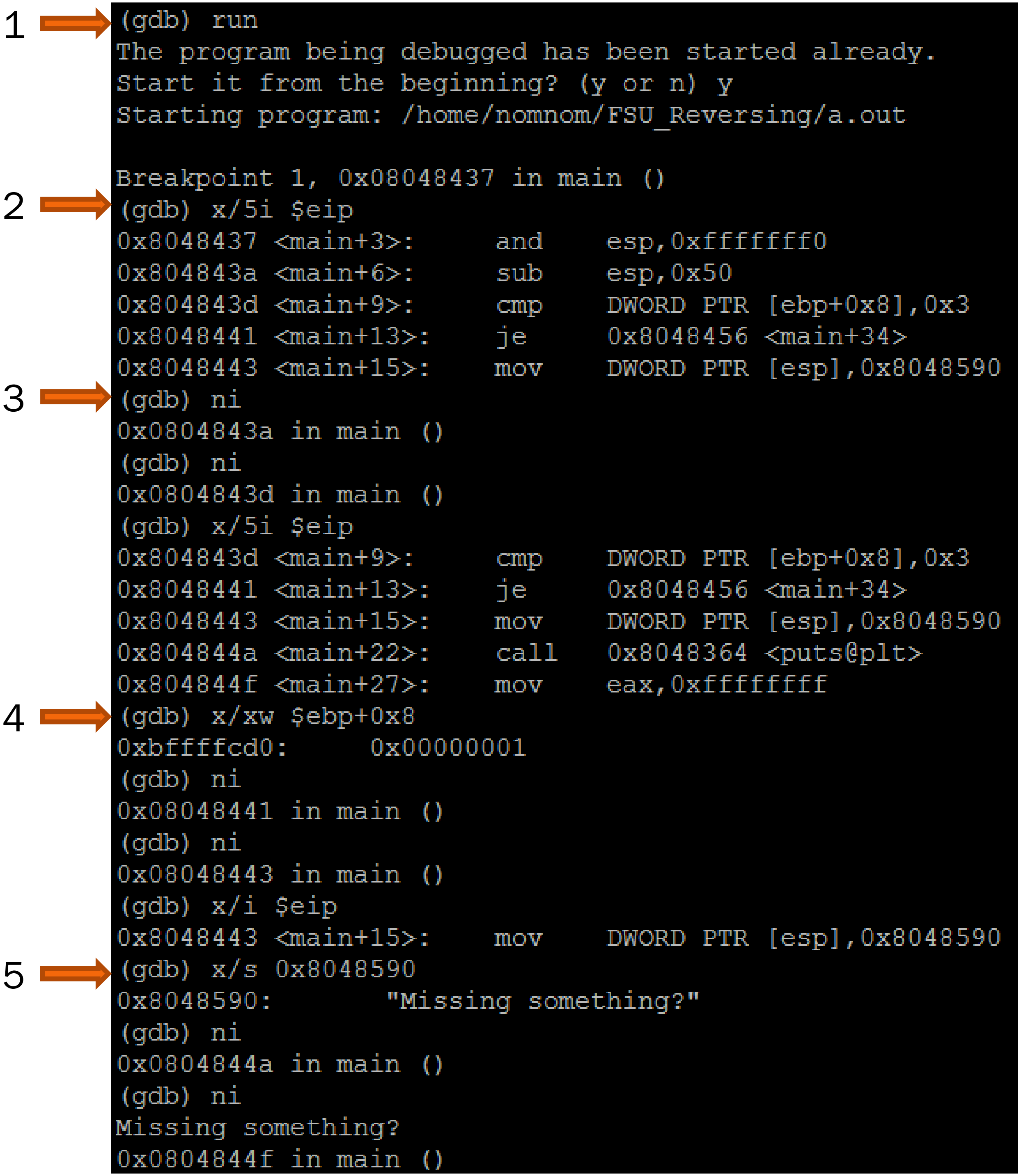
```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nomnom/FSU_Reversing/a.out

Breakpoint 1, 0x08048437 in main ()
(gdb) x/5i $eip
0x8048437 <main+3>:     and    esp,0xfffffff0
0x804843a <main+6>:     sub    esp,0x50
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) ni
0x0804843a in main ()
(gdb) ni
0x0804843d in main ()
(gdb) x/5i $eip
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
0x804844a <main+22>:    call   0x8048364 <puts@plt>
0x804844f <main+27>:    mov    eax,0xffffffff
(gdb) x/xw $ebp+0x8
0xbffffcd0:     0x00000001
(gdb) ni
0x08048441 in main ()
(gdb) ni
0x08048443 in main ()
(gdb) x/i $eip
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) x/s 0x8048590
0x8048590:       "Missing something?"
(gdb) ni
0x0804844a in main ()
(gdb) ni
Missing something?
0x0804844f in main ()
```

1. Run the program
2. Where are we? Check out EIP
3. Continue until we hit an instruction of interest
4. Let’s see what’s being compared – we can see this jump is not taken



```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nomnom/FSU_Reversing/a.out

Breakpoint 1, 0x08048437 in main ()
(gdb) x/5i $eip
0x8048437 <main+3>:     and    esp,0xfffffff0
0x804843a <main+6>:     sub    esp,0x50
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) ni
0x0804843a in main ()
(gdb) ni
0x0804843d in main ()
(gdb) x/5i $eip
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
0x804844a <main+22>:    call   0x8048364 <puts@plt>
0x804844f <main+27>:    mov    eax,0xffffffff
(gdb) x/xw $ebp+0x8
0xbffffcd0:     0x00000001
(gdb) ni
0x08048441 in main ()
(gdb) ni
0x08048443 in main ()
(gdb) x/i $eip
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) x/s 0x8048590
0x8048590:       "Missing something?"
(gdb) ni
0x0804844a in main ()
(gdb) ni
Missing something?
0x0804844f in main ()
```

1. Run the program
2. Where are we? Check out EIP
3. Continue until we hit an instruction of interest
4. Let's see what's being compared – we can see this jump is not taken
5. Check out the argument passed to puts



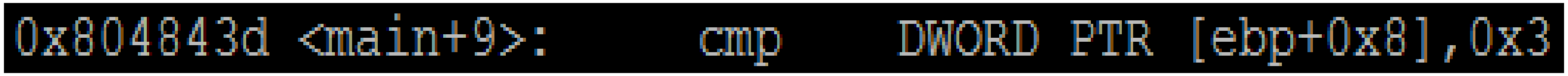
```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/nomnom/FSU_Reversing/a.out

Breakpoint 1, 0x08048437 in main ()
(gdb) x/5i $eip
0x8048437 <main+3>:     and    esp,0xfffffff0
0x804843a <main+6>:     sub    esp,0x50
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) ni
0x0804843a in main ()
(gdb) ni
0x0804843d in main ()
(gdb) x/5i $eip
0x804843d <main+9>:     cmp    DWORD PTR [ebp+0x8],0x3
0x8048441 <main+13>:    je     0x8048456 <main+34>
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
0x804844a <main+22>:    call   0x8048364 <puts@plt>
0x804844f <main+27>:    mov    eax,0xffffffff
(gdb) x/xw $ebp+0x8
0xbffffcd0:     0x00000001
(gdb) ni
0x08048441 in main ()
(gdb) ni
0x08048443 in main ()
(gdb) x/i $eip
0x8048443 <main+15>:    mov    DWORD PTR [esp],0x8048590
(gdb) x/s 0x8048590
0x8048590:       "Missing something?"
(gdb) ni
0x0804844a in main ()
(gdb) ni
Missing something?
0x0804844f in main ()
```

1. Run the program
2. Where are we? Check out EIP
3. Continue until we hit an instruction of interest
4. Let’s see what’s being compared – we can see this jump is not taken
5. Check out the argument passed to puts

Aha! We don’t satisfy the compare (1 != 3), and call puts, then exit!

Dynamic Analysis - GDB

- Think about the function protocol for main
    - int main (int argc, char *argv[])
- In main, [ebp+8] would reference the first argument, **argc**

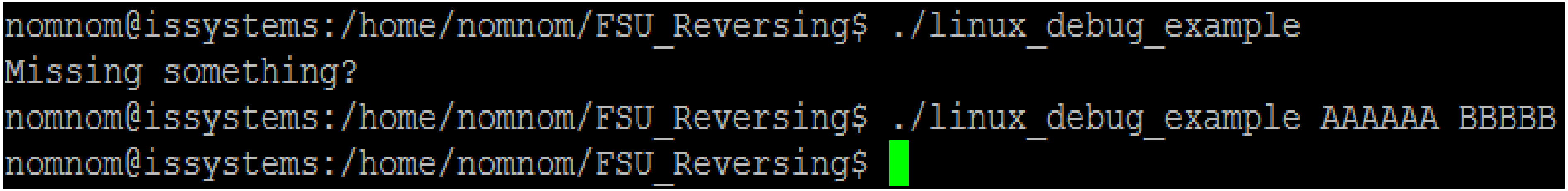


```
0x804843d <main+9>:     cmp     DWORD PTR [ebp+0x8],0x3
```

- We aren’t passing any arguments, besides argv[0], the program name, hence why [ebp+8] has the value 1

Dynamic Analysis - GDB

- Haha, passing the program 2 more arguments (3 total) does in fact satisfy the first cmp instruction

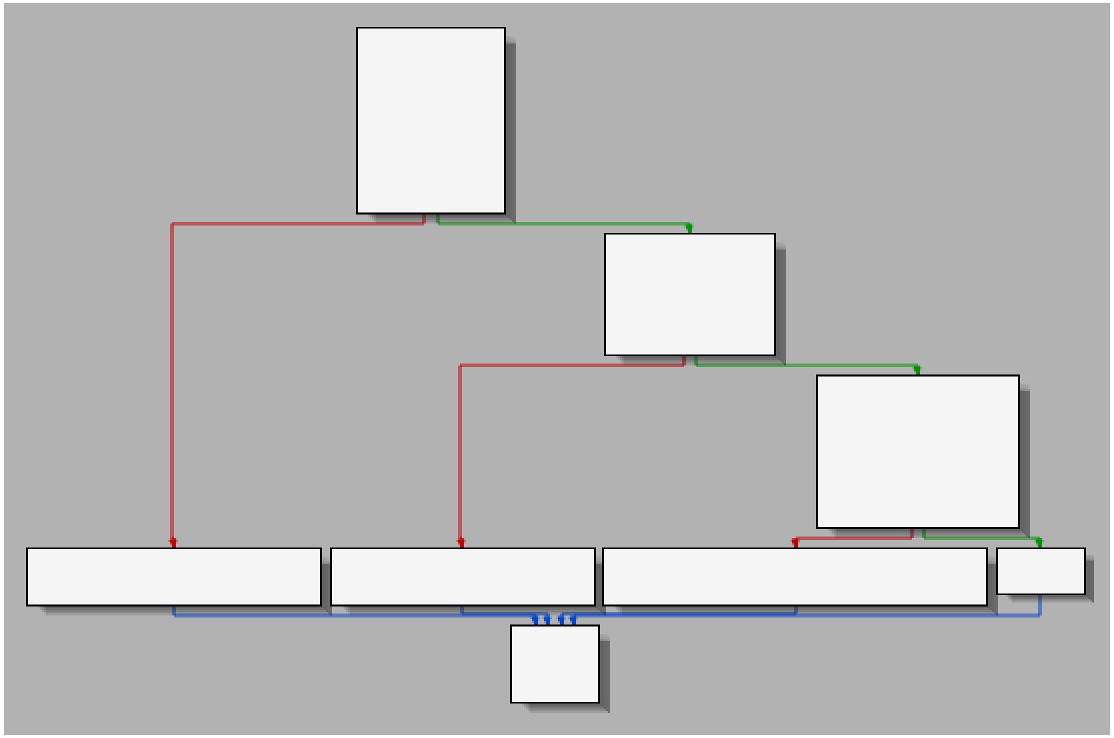


```
nomnom@issystems:/home/nomnom/FSU_Reversing$ ./linux_debug_example
Missing something?
nomnom@issystems:/home/nomnom/FSU_Reversing$ ./linux_debug_example AAAAAA BBBBB
nomnom@issystems:/home/nomnom/FSU_Reversing$ 
```

- A new code path is taken!

Exercise 5

- Try to figure out the correct input that will cause the program to print message, "Congrats, you did it!“
- Use IDA **and** GDB!



- Hey, we’ve seen this graph pattern before!

Dynamic Vs. Static

- Everyone has their own preferences
- But the combination of the two will undoubtedly yield the best results
- IDA, WinDBG, Immunity, GDB all have scripting
  - In fact, they all use Python except WinDBG*
  - There are awesome scripts that will import results from debuggers into IDA's view, filling in all the registers/operands for each instruction.

Last Exercise (homework?)

- key_checker.exe or
- We'll do a real crackme
- Crackme at
  - http://www.woodmann.com/RCE-CD-SITES/Quantico/mib/crackme2.zip
- This might be a little tricky, that's okay.

One quick note

- What about bytecode?
  - .NET applications, java, python, etc.
- Just download a disassembler
- You’ll get near complete source code back
- It’s really that easy…

Conclusion

- Hopefully you feel comfortable
  - Opening up and examining a binary and looking at it’s sections to get a feel for it
  - Renaming and simplifying the disassembly
  - Converting back to source code where needed
  - Using a debugger to fill in the gaps or manipulate program execution

Conclusion

- Fantastic books
  - Reversing: The secrets of reverse engineering
  - The IDA Pro book
  - The Art of Exploitation
- Challenges
  - Crackmes.de
  - Woodmann.com
  - Smashthestack.org (plenty of debugging involved ;))
- Links
  - CSG : csg.utdallas.edu and irc.oftc.net #utdcsg (everyone is welcome)
  - IDA : hex-rays.com
  - CFF Explorer : ntcore.com/exsuite.php
  - Immunity Debugger : immunityinc.com