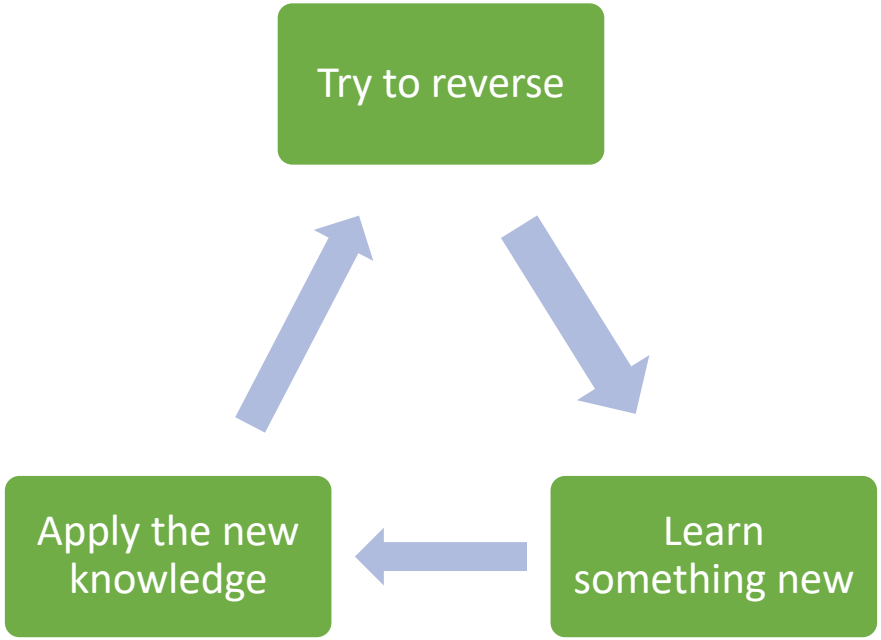
Malware Analysis: Reverse Engineering

Basic Overview

CSCE 465

*Content adopted from hackallthethings.com

A Continuous Cycle



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
  - …

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

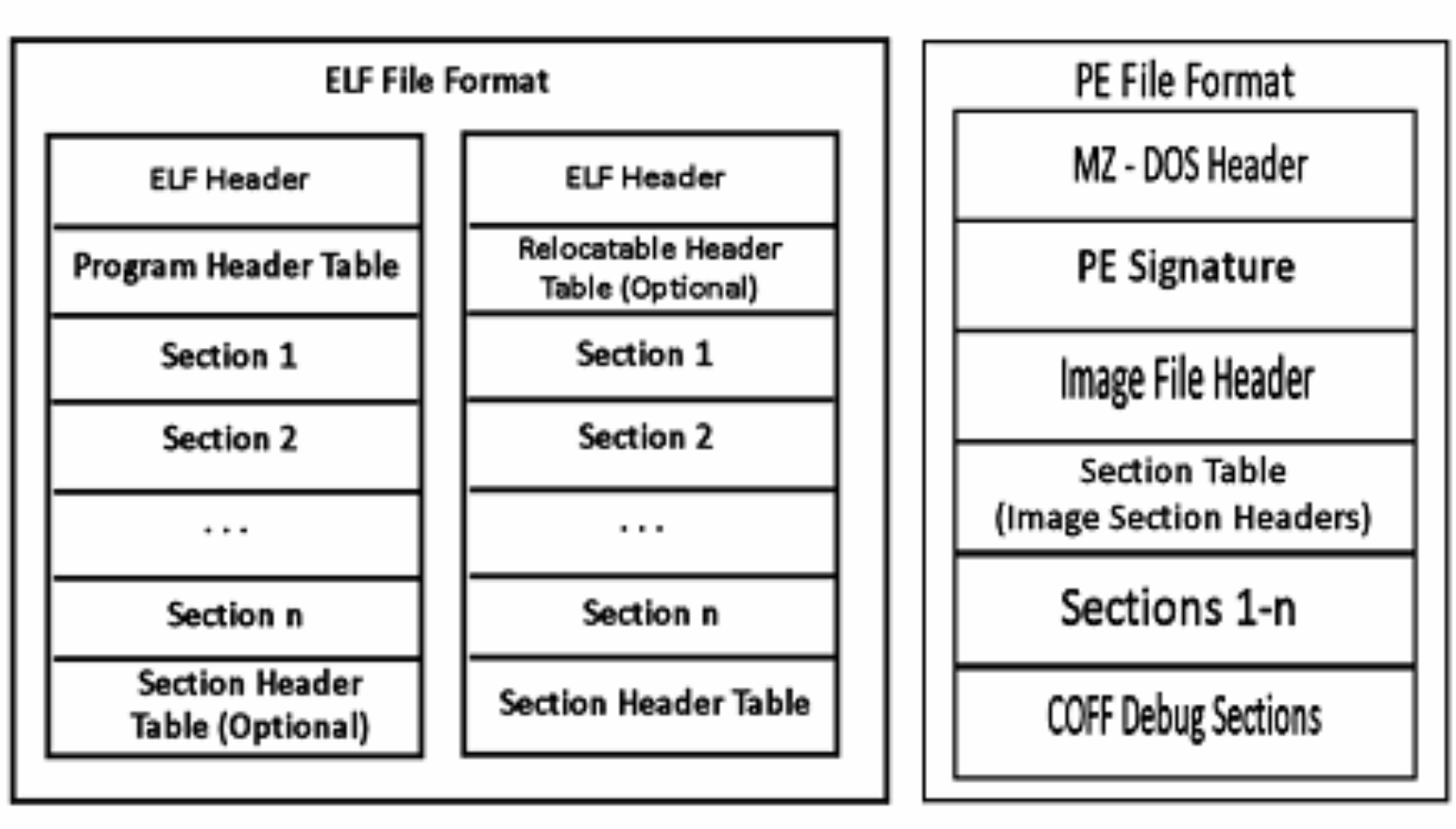


Image from http://software.intel.com/sites/default/files/m/d/4/1/d/8/keep-memory-002.gif

PE and ELF

- 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

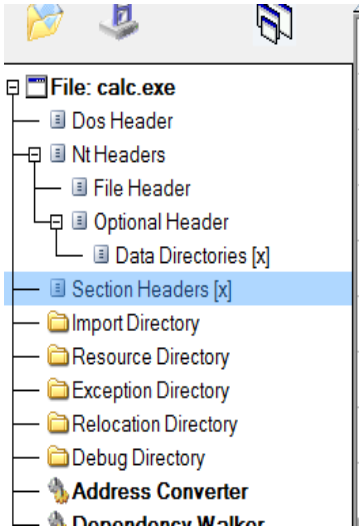
- Okay, so what? Why is this useful?

- 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	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
UPX0	00005000	00001000	00000000	00000400	00000000	00000000	0000	0000	E0000080
UPX1	00002000	00006000	00001800	00000400	00000000	00000000	0000	0000	E0000040
.rsrc	00001000	00008000	00000400	00001C00	00000000	00000000	0000	0000	C0000040

- Huge red flag with section names like this

ELF - readelf

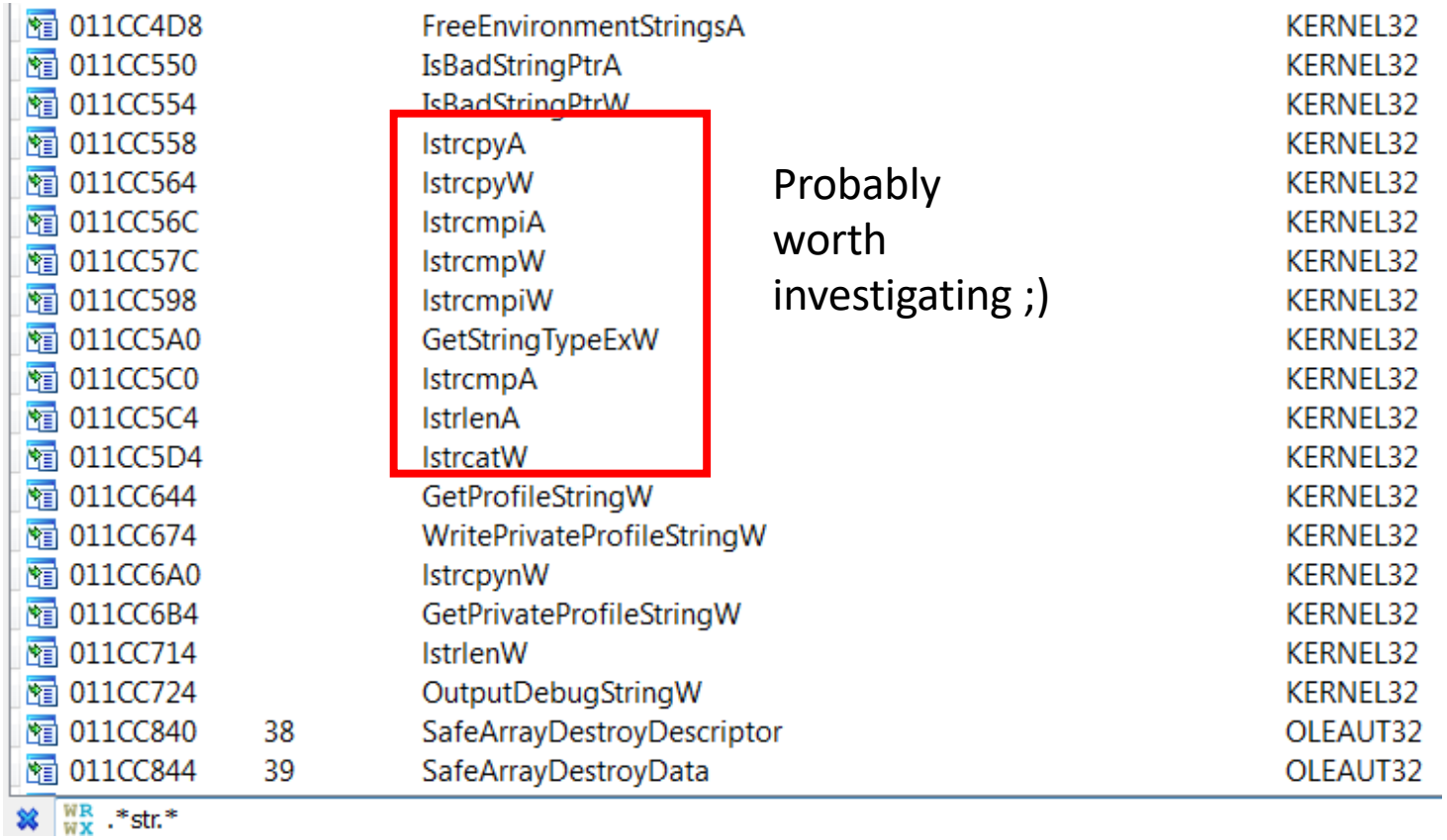
- This is using *readelf* to look at section headers

```
:~$ 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)
```

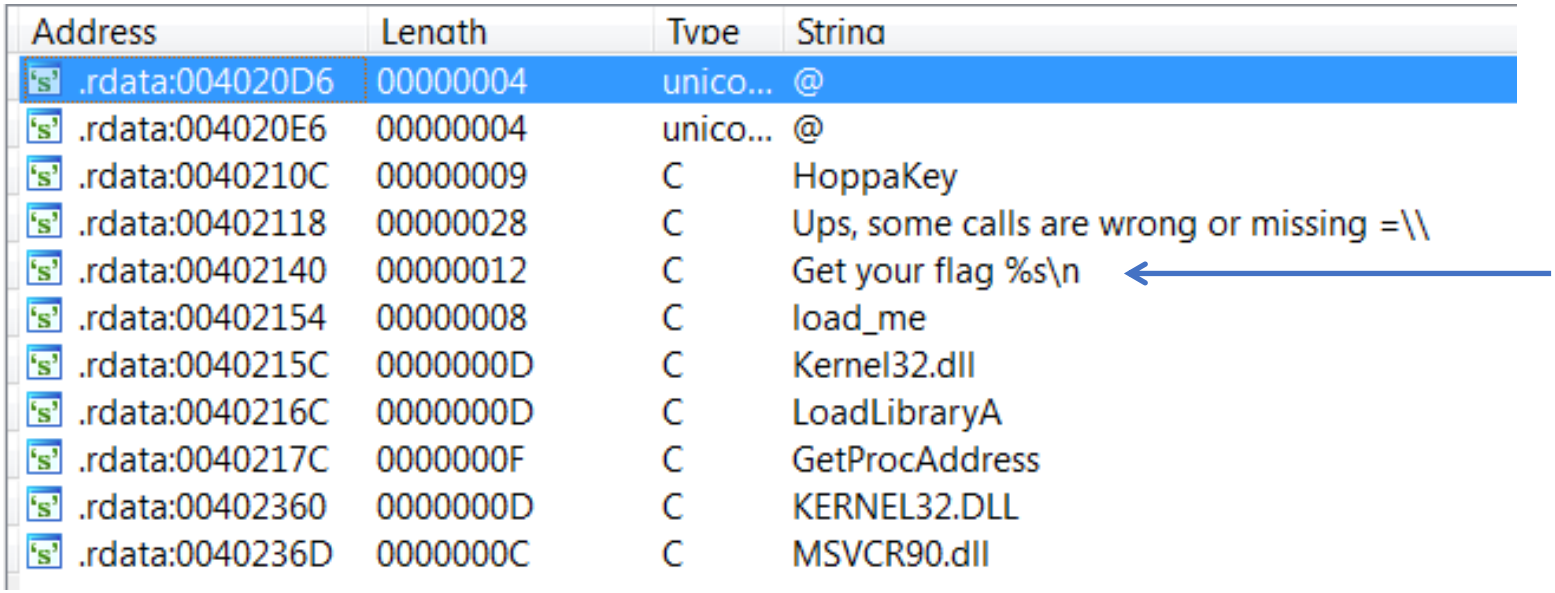
PE and ELF - Imports

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



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 ;)

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

- Intel's syntax most prevalent

- 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

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

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

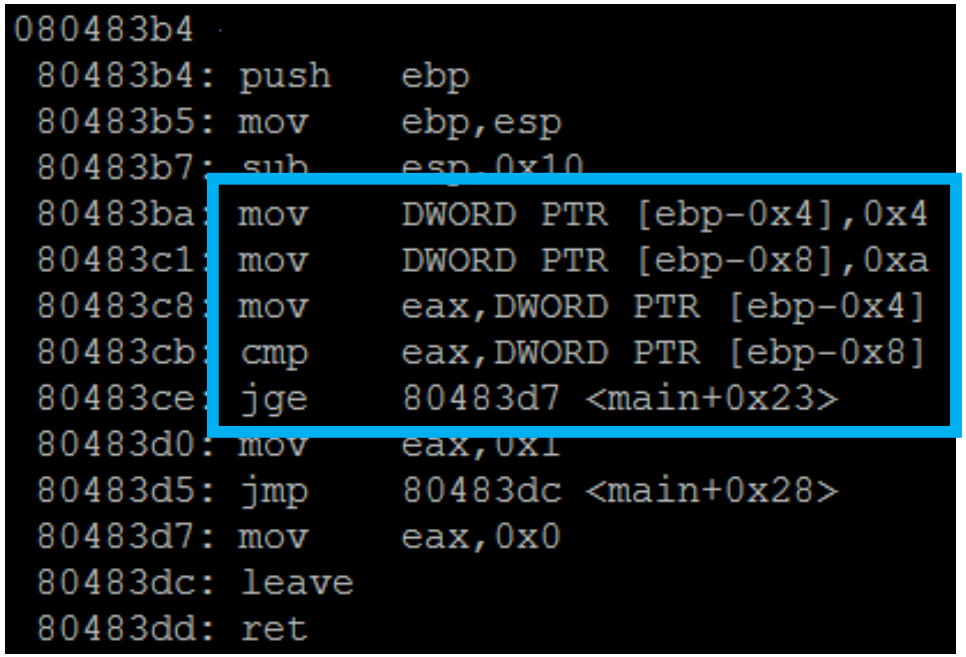
- [ebp-0x4] = 0x4
- [ebp-0x8] = 0xa
- eax = [ebp-0x4]

- 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



Example 1

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

- cmp eax, [ebp-0x8]
  - eax == [ebp-0x8] ?
  - 4 == 10 ?

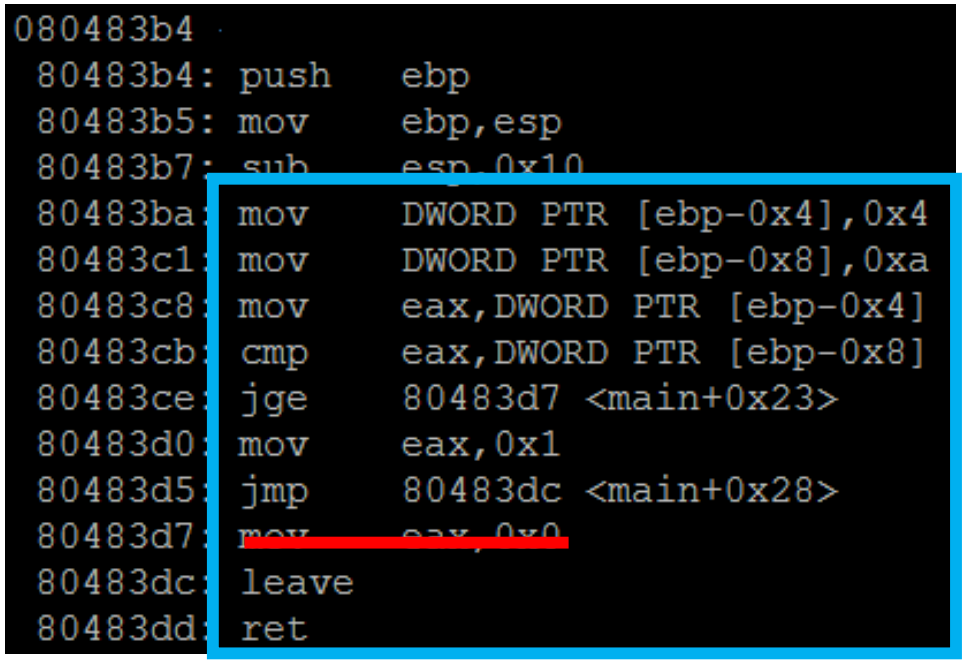
- jge 0x80483d7
  - If 4 was >= 10, jmp
  - Else, continue execution

```
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



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

```
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
```

Example 1

- 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???

- Think back to our process memory layout.

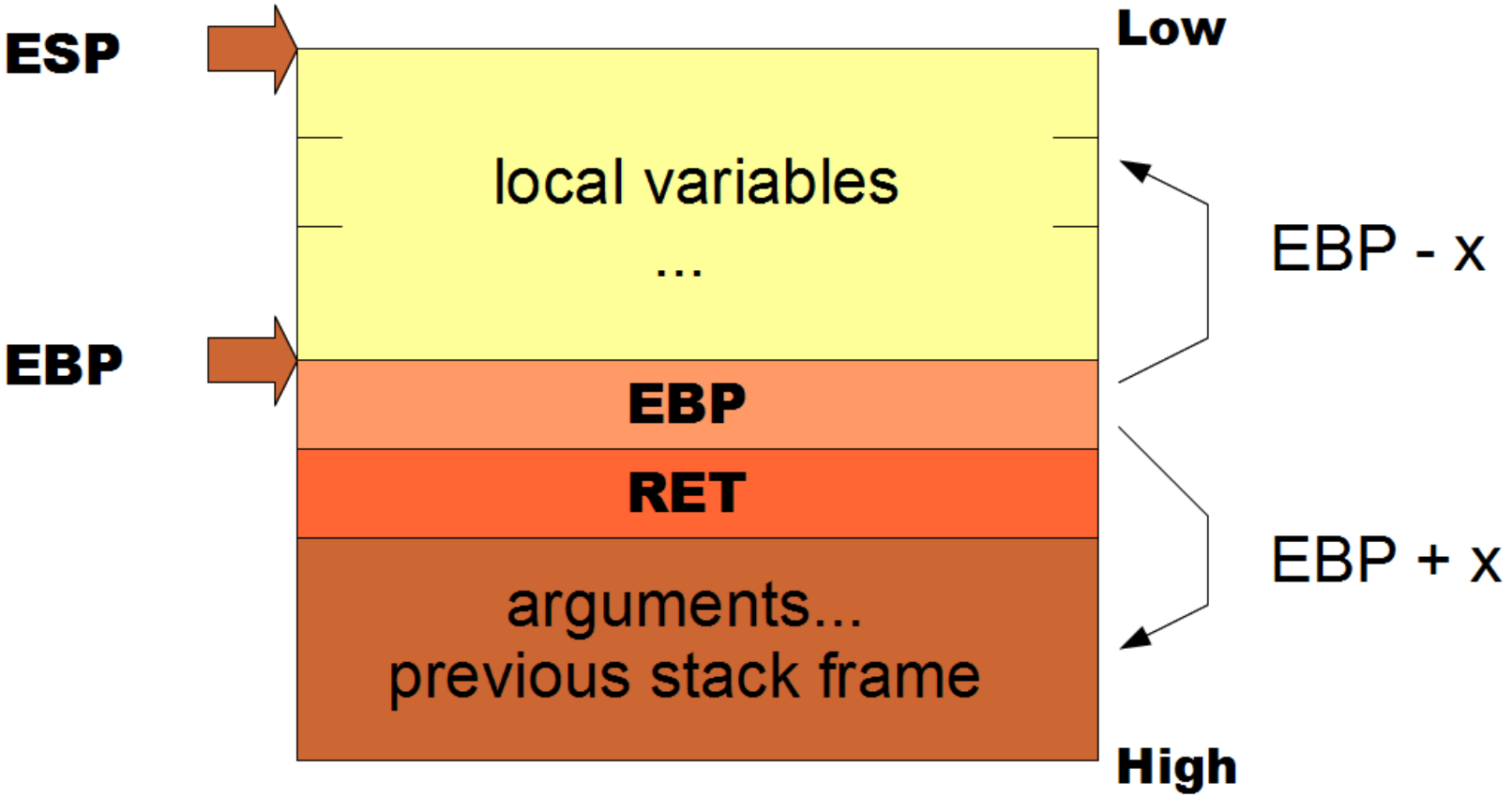
Example 1 – Recap so far

- 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!

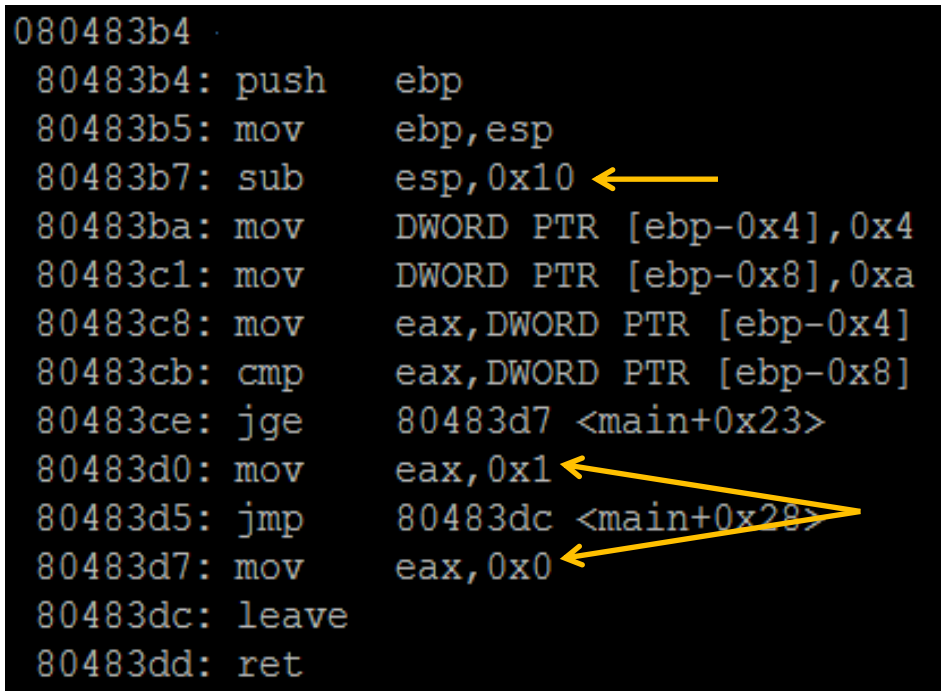
```
080483b4
 80483b4: push   ebp
 80483b5: mov    ebp,esp
 80483b7: sub    esp,0x10
 80483ba: mov    DWORD PTR [ebp-0x4],0x4
 80483c1: mov    DWORD PTR [ebp-0x8],0xa
 80483c8: mov    eax,DWORD PTR [ebp-0x4]
 80483cb: cmp    eax,DWORD PTR [ebp-0x8]
 80483ce: jge    80483d7 <main+0x23>
 80483d0: mov 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

The Stack

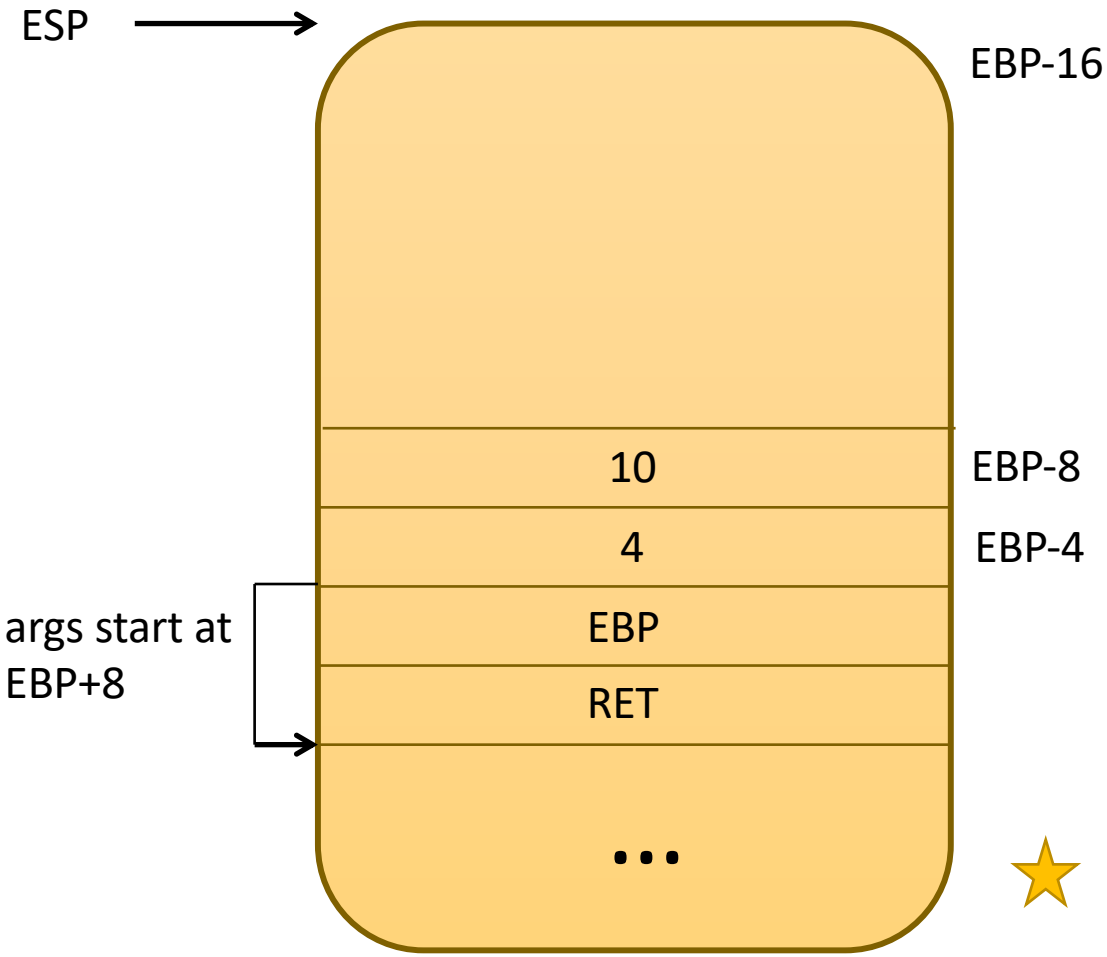

ESP
EBP
Low
local variables
...
EBP
RET
arguments...
previous stack frame
High
EBP - x
EBP + x

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



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 comparisons

- ‘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 – 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
```

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

  Return 0

  }

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;
```