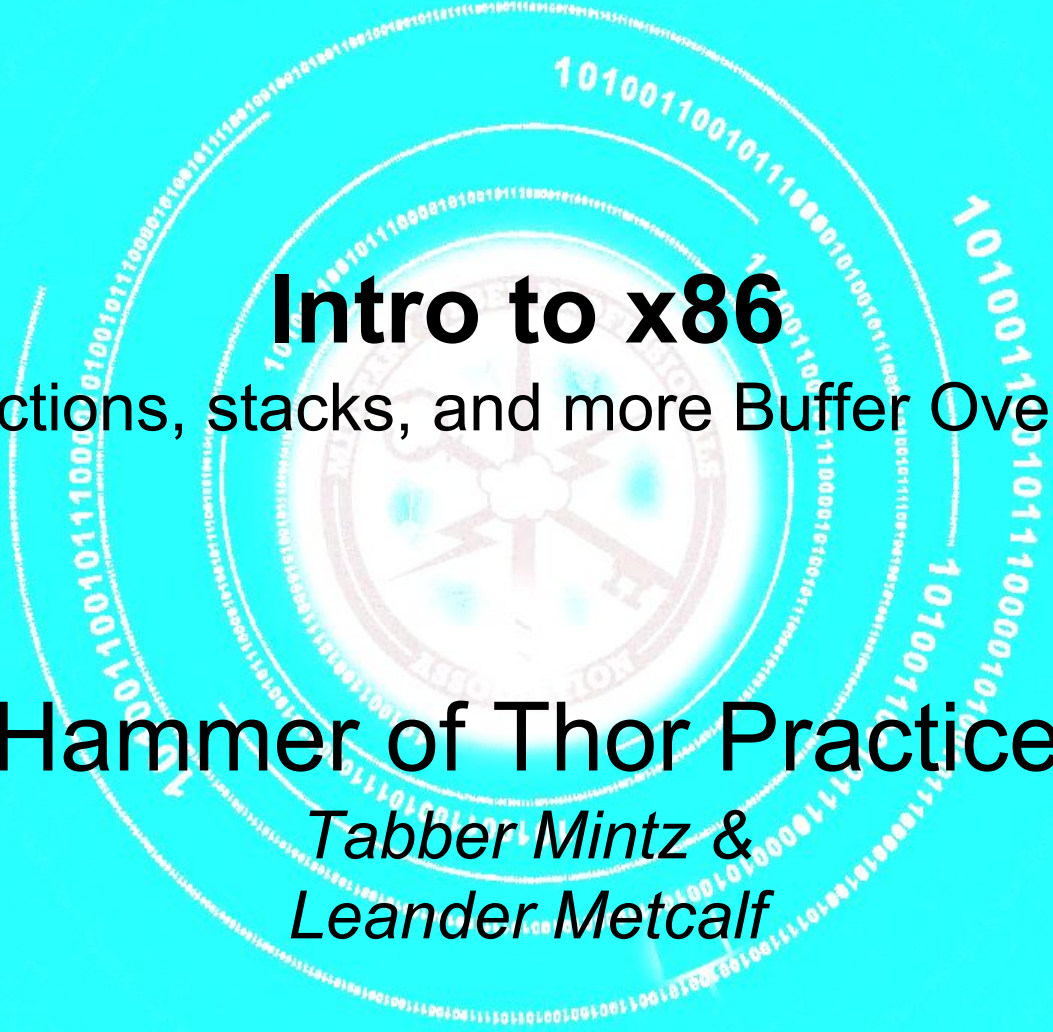

MILITARY CYBER PROFESSIONALS
ASSOCIATION



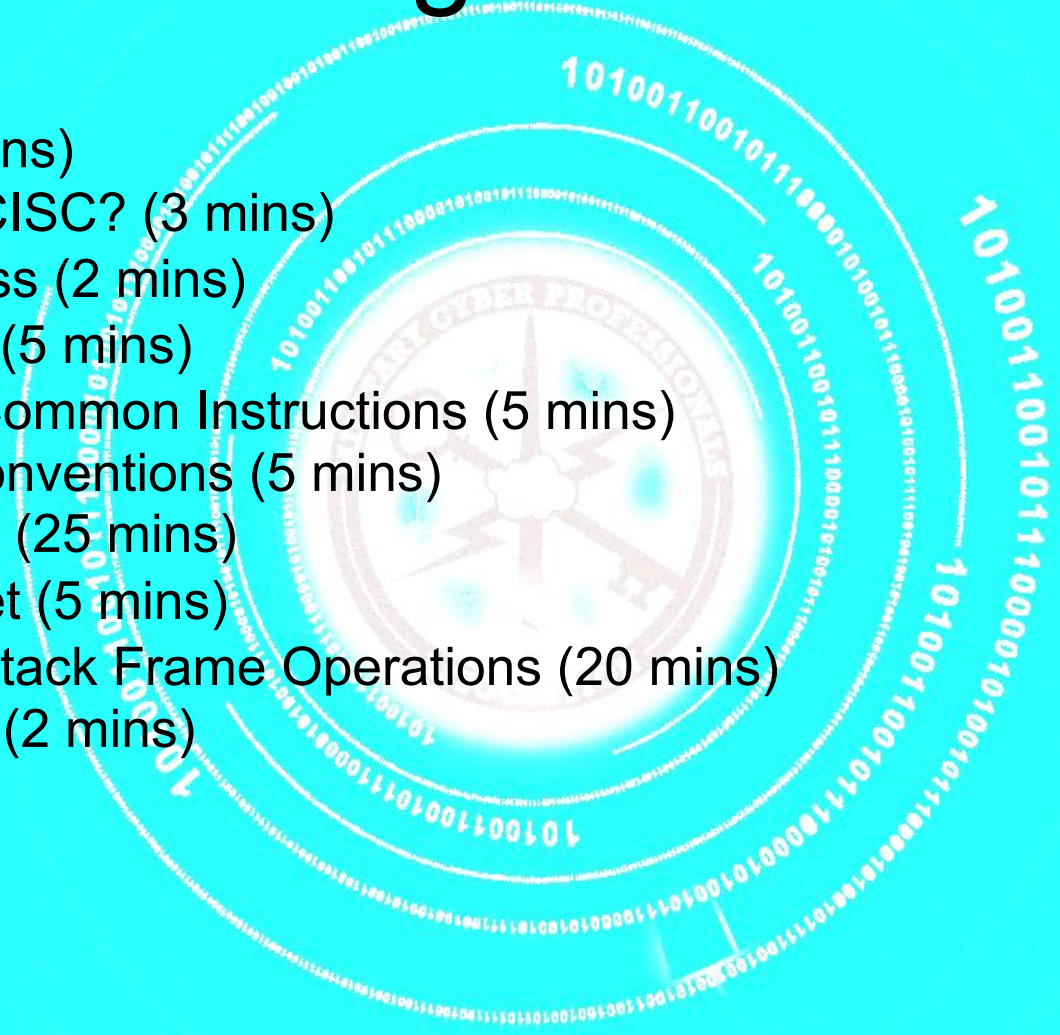
Intro to x86

Instructions, stacks, and more Buffer Overflows

Hammer of Thor Practice

Tabber Mintz &
Leander Metcalf

Agenda



- Intro (3 mins)
- RISC vs CISC? (3 mins)
- Endianness (2 mins)
- Registers (5 mins)
- 10 Most Common Instructions (5 mins)
- Calling Conventions (5 mins)
- The Stack (25 mins)
- call and ret (5 mins)
- General Stack Frame Operations (20 mins)
- Summary (2 mins)
- Questions

Intro

```
pc@host:~$ ./hello
Hello World!
pc@host:~$
```

```
start:
message_on_stack:
   push   dword 0xa
   push   dword '!'
   push   dword 'orld'
   push   dword 'o, W'
   push   dword 'Hell'
   push   esp
write_message:
   push   dword 17
   push   dword [esp+4]
   push   dword 1
   mov    eax, 4
   sub    esp, 4
   int    0x80
exit:
   xor    ebx, ebx
   mov    eax, 1
   int    0x80
```

```
#include <stdio.h>
int main(){
   printf("Hello World!\n");
   return 0;
}
```

```
1 #!/usr/bin/python
2
3 print 'Hello World!'
```

```
_main:
    push        rbp
    mov         rbp, rsp
    sub         rsp, 0x10
    lea         rdi, qword [ds:0x100000f8a]
    mov         dword [ss:rbp+var_4], 0x0
    mov         al, 0x0
    call        imp___stubs__printf
    xor         ecx, ecx
    mov         dword [ss:rbp+var_8], eax
    mov         eax, ecx
    add         rsp, 0x10
    pop         rbp
    ret
```

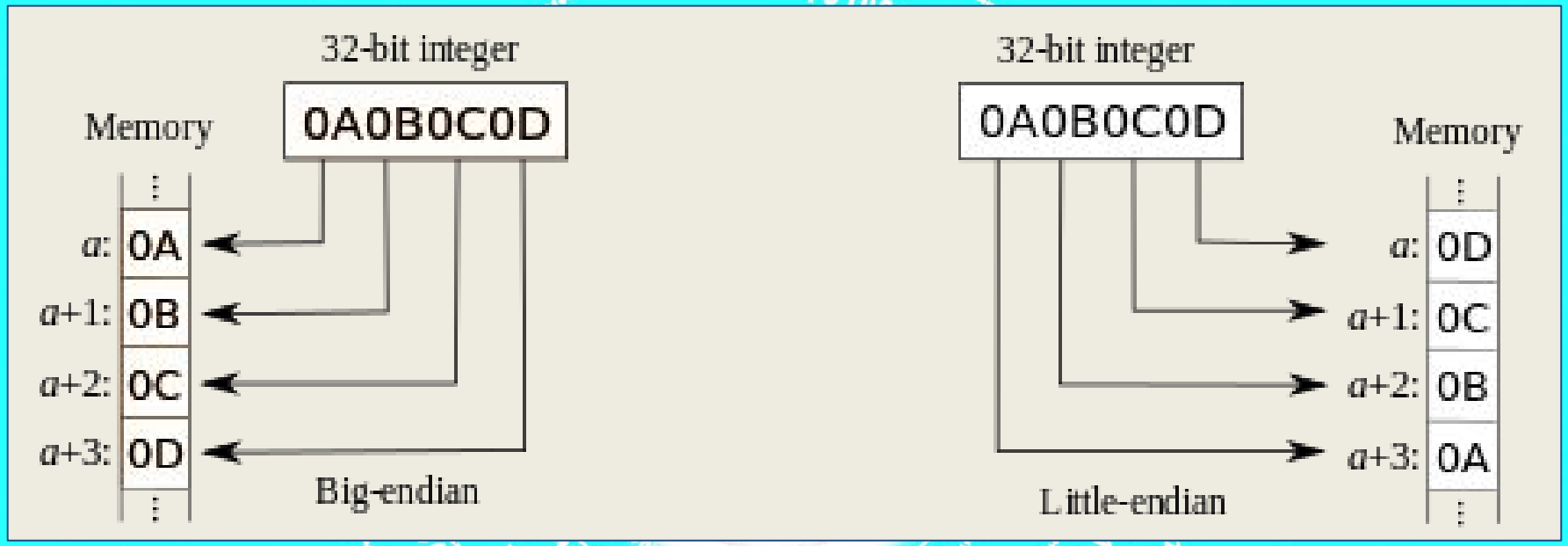
RISC vs CISC?

Reduced Instruction Set Computing (RISC) reduces the number of computations required in a CPU cycle, but sophisticated code requires numerous simple instructions.

Complex Instruction Set Computing (CISC) sacrifices speed to accomplish more in a single instruction decreasing the amount of code to create a program.

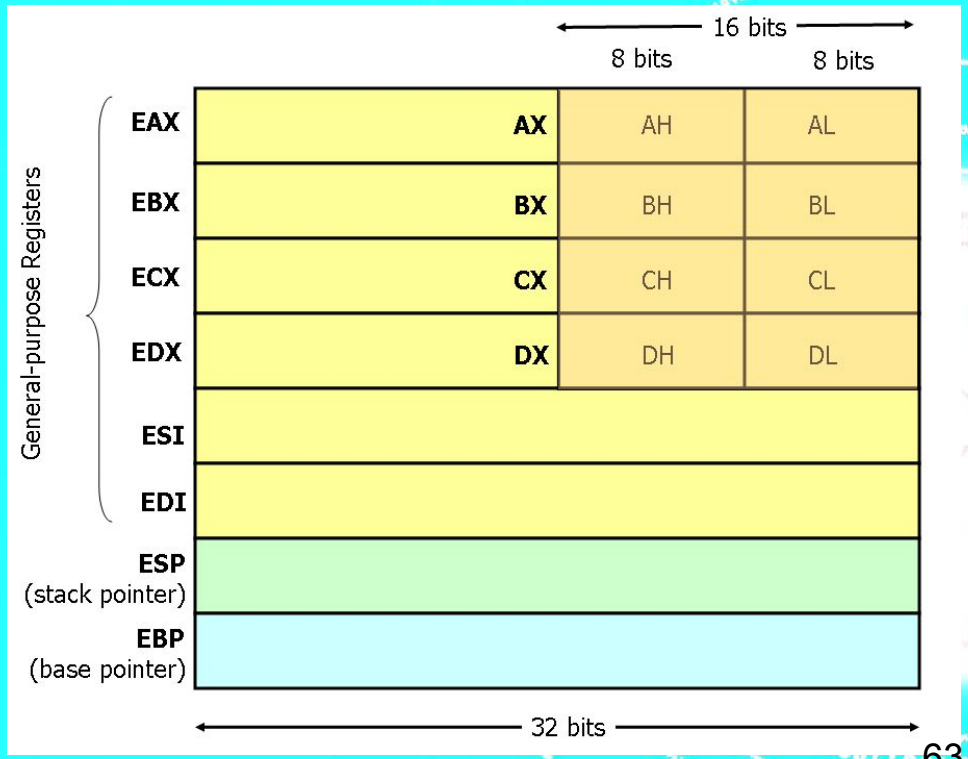
	RISC	CISC
Instruction Size	Fixed (2 or 4 bytes)	Variable (1-15 bytes)
Registers	16 General Purpose	8 General Purpose
Memory Reference	Load/Store	Embedded in instructions
Calculations	Fast Computations (single CPU cycle), Large amount of code	Slow Computations (multiple CPU cycles), Small amount of code
Emphasis	Hardware, Highly pipelined	Software, Not as pipelined or not pipeline at all

Endianness



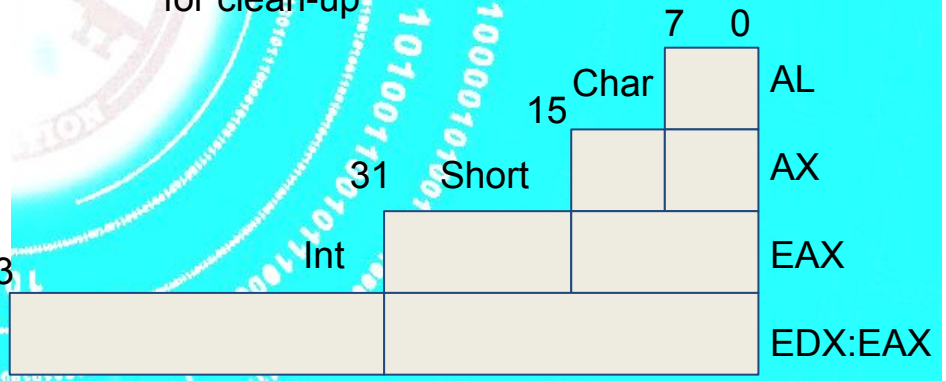
- Intel is Little Endian - Observe on Linux:
  - echo -n I | od -to2 | head -n1 | cut -f2 -d" " | cut -c6
  - 1 = little; 0 = big
- Networking (IP Protocol) = big-endian
  - Be careful raw sockets follow system architecture
- Some architectures can be configured as either (Bi-Endian)

Registers



Where do your variables go?

- 8 General Purpose Registers, lower half can be addressed, and a few can also be addressed by nibbles
- 6 Segment Registers
- 2 Special Purpose Registers (EIP, EFLAGS)
- Length is also equivalent to data types
- ESP can be used to grow or shrink the stack, but the programmer is responsible for clean-up



EDI -> Destination Register
ESI -> Source Register
EBP -> Frame Pointer
ESP -> Stack Pointer
EIP -> Instruction Pointer

Initialized Data (in bits):

DB – 8
DW – 16
DD – 32
DQ – 64

Can use quoted string w/ all except DQ
Ex. db 'cat', 0

No Clobber Registers:
EBX, EBP, ESI, EDI, ESP

System Call expect parameters in specific registers
32 Bit - EAX -> EBX -> ECX -> EDX -> ESI -> EDI -> EBP

10 Most Common Instructions

1. **MOV** dest, src *#moves src into dest*
2. **PUSH** src | **POP** dest *#manipulate the top of the stack*
3. **CALL** target | **RET** *#typically used as a pair for functions*
4. **CMP** op1, op2 *#computes op1-op2*
5. **ADD** dest, src / **SUB** dest, src *#src+-dest=dest*
6. **LEA** dest, src *#load whatever src points at and move value*
7. **JMP** target | **Jcc** target *#jump to a specific place in memory*
8. **INC** dest | **DEC** dest *#change the value by 1*
9. **TEST** op1, op2 *#computes op1 AND op2*
10. **XOR** dest, src *#XOR the two values*

*** Intel syntax shown; AT&T syntax is reverse ***

Calling Conventions

- There are two typical calling conventions called “cdecl” and “stdcall”. Both adhere to the following:
  - Function parameters pushed onto stack right to left
  - Saves old stack frame pointer and sets up new stack frame
  - EAX or EDX:EAX returns the result for all primitive data types
- There is one difference between the two calling conventions when handling entrance and exit from a function:
  - cdecl requires the caller to clean up the stack before returning
  - stdcall requires the callee to clean up any stack parameters before returning

The Stack

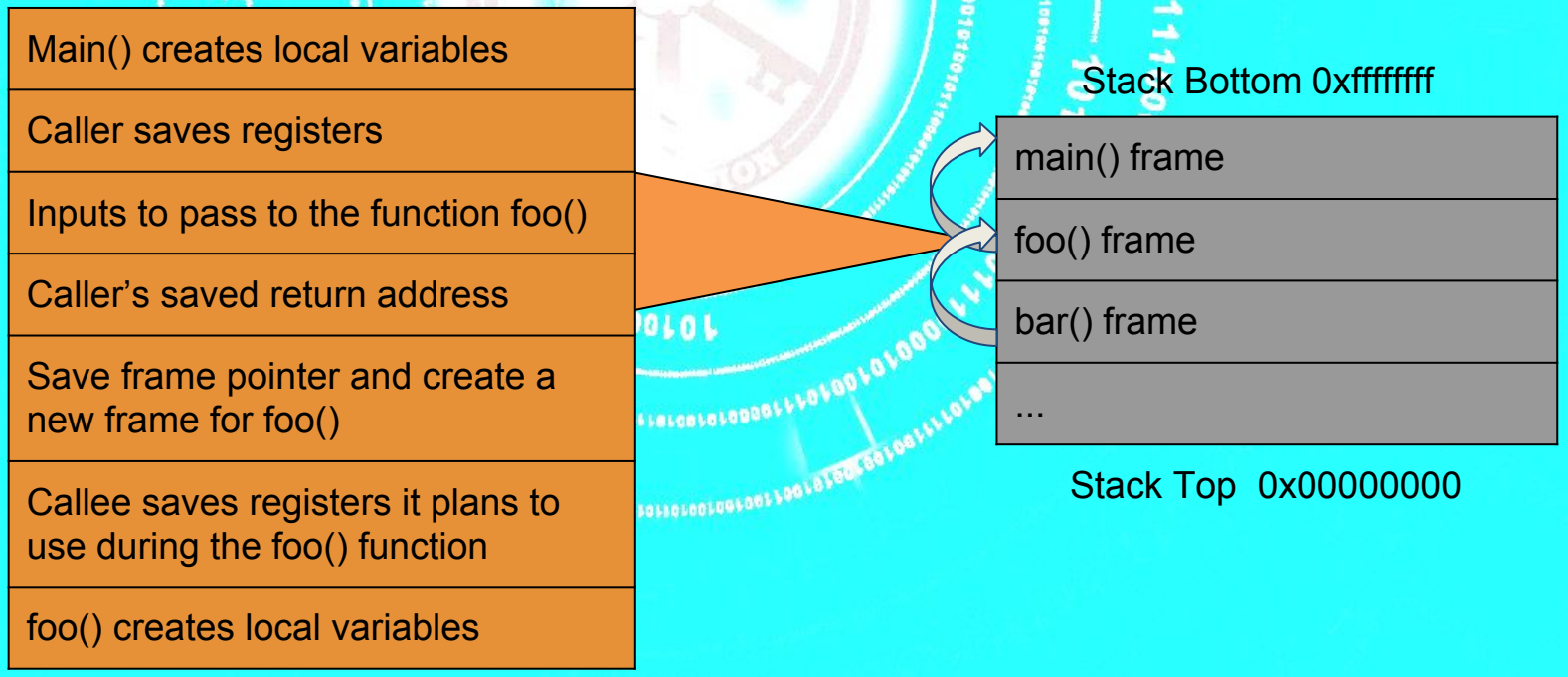
- The stack refers to a region of memory that is assigned to a program at execution.
- The stack is a Last-In-First-Out (LIFO/FILO) data structure where data is "pushed" on to the top of the stack and "popped" off the top.
- Memory is divided into “high” and “low” memory.
  - Low memory starts at 0xffffffff and high memory starts at 0x00000000
  - The stack normally starts at 0xbfffffff in a Linux environment and continues to “grow” towards the higher part of memory
- The stack keeps track of which functions were called before the current one, it holds local variables and is frequently used to pass arguments to the next function to be called.

Call and Ret

- The Call instruction is used to transfer control to a function or sub-function and allows the program to “remember” where to return.
- It pushes the address of the next instruction onto the stack and then changes the value in EIP to the address of the given instruction.
- The Ret function completes the inverse of the call instruction by returning control to the parent of the current function it is located.
- It will pop the top address off where the current value ESP is pointing to, change the value in EIP, and then attempt to execute the next instruction at that address.

General Stack Frame Operations

- Stack frames are generally a linked list.
- When *main()* calls a function it becomes the caller and prepares to pass input arguments to the function *foo()* and then creates a “frame”.



Demo Time!!!

Summary

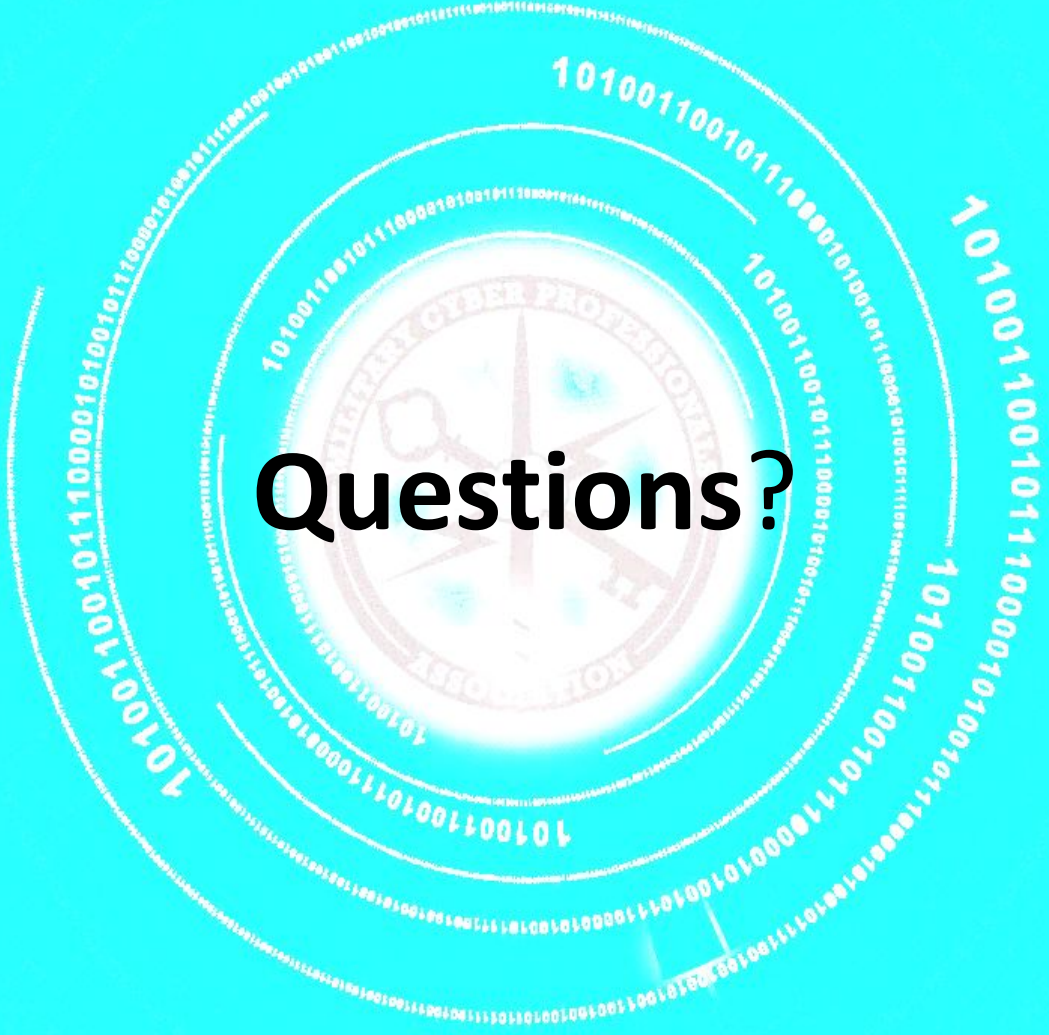
- Provided a basic understanding of 32-bit x86 assembly language and covered the 10 most common instructions.
- Explained how variables are handled and the most common calling conventions used by compilers.
- The key to reading and programming in assembly language is understanding how it affects the stack.

Tools

- Netwide Assembler - http://www.nasm.us
- GNU Debugger - https://www.gnu.org/s/gdb
- objdump - https://goo.gl/n9uYVL

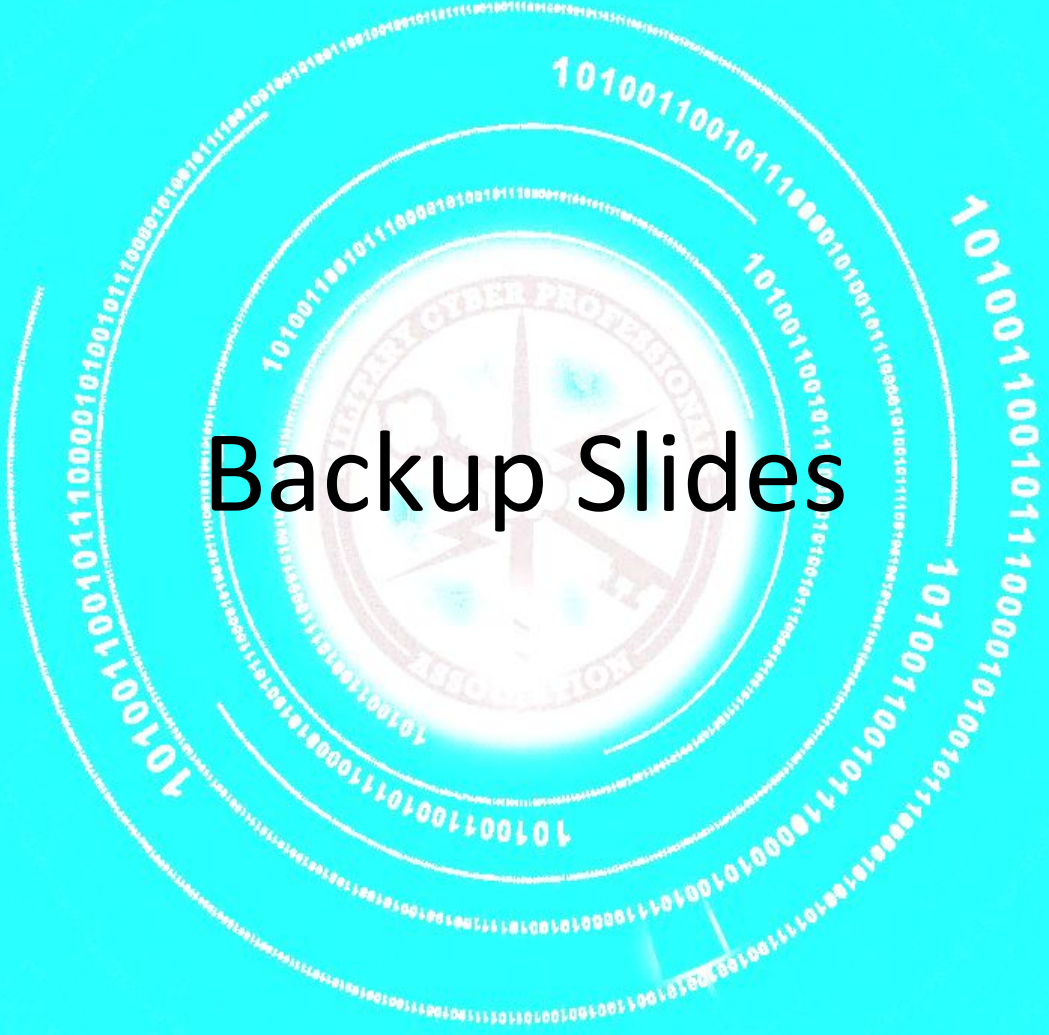
References

- Intel Reference Manuals Vol. 1, 2, & 3
- ARMv7-A Reference Manual



Questions?


MILITARY CYBER PROFESSIONALS
ASSOCIATION



Backup Slides