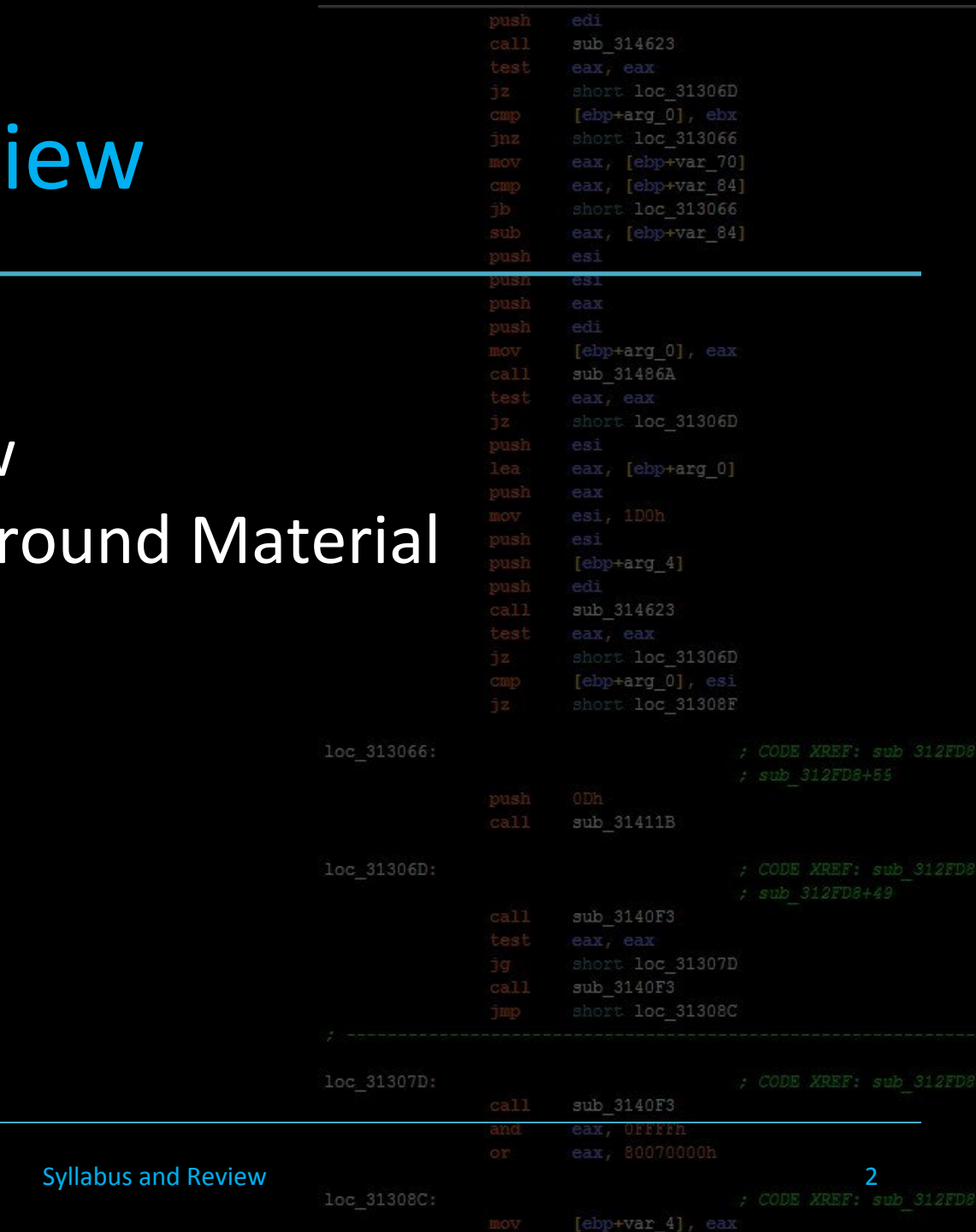
Syllabus and Review

Modern Binary Exploitation
CSCI 4968 - Spring 2015
Alex Bulazel

Lecture Overview

1. Syllabus
2. Course Overview
3. Review of Background Material
   a. Linux
   b. C
   c. x86 Assembly



Course Details

- Modern Binary Exploitation
  - Course Number: CSCI 4968
  - Credit Hours: 4
  - Semester / Year: Spring 2015
  - Meeting Days: Tuesday/Friday 2-4PM
  - Room Location: Walker 5113
  - Course Website:
    - http://security.cs.rpi.edu/courses/binexp-spring2015/
    - http://rpis.ec/binexp
  - Prereqs:
    - CSCI 2500 - Computer Organization
    - ECSE 2660 - Computer Architecture, Networks, and Operating Systems

Instructor

- Instructor: Dr. Bülent Yener
  - Office: Lally 310
  - Email: yener@cs.rpi.edu

Cyber Is A Team Sport

Markus

Branden

Sophia

Alex

Jeremy

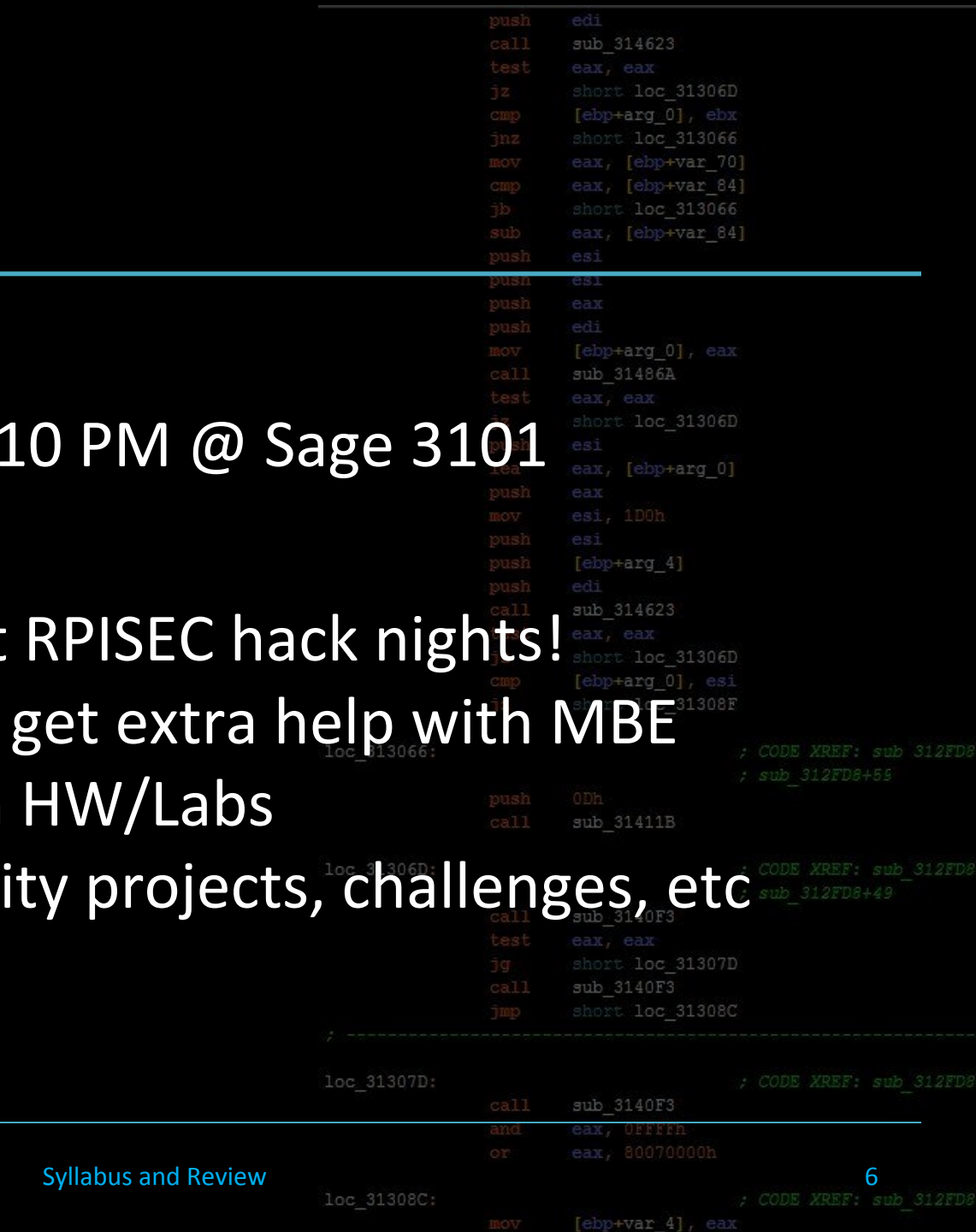
Patrick

Austin

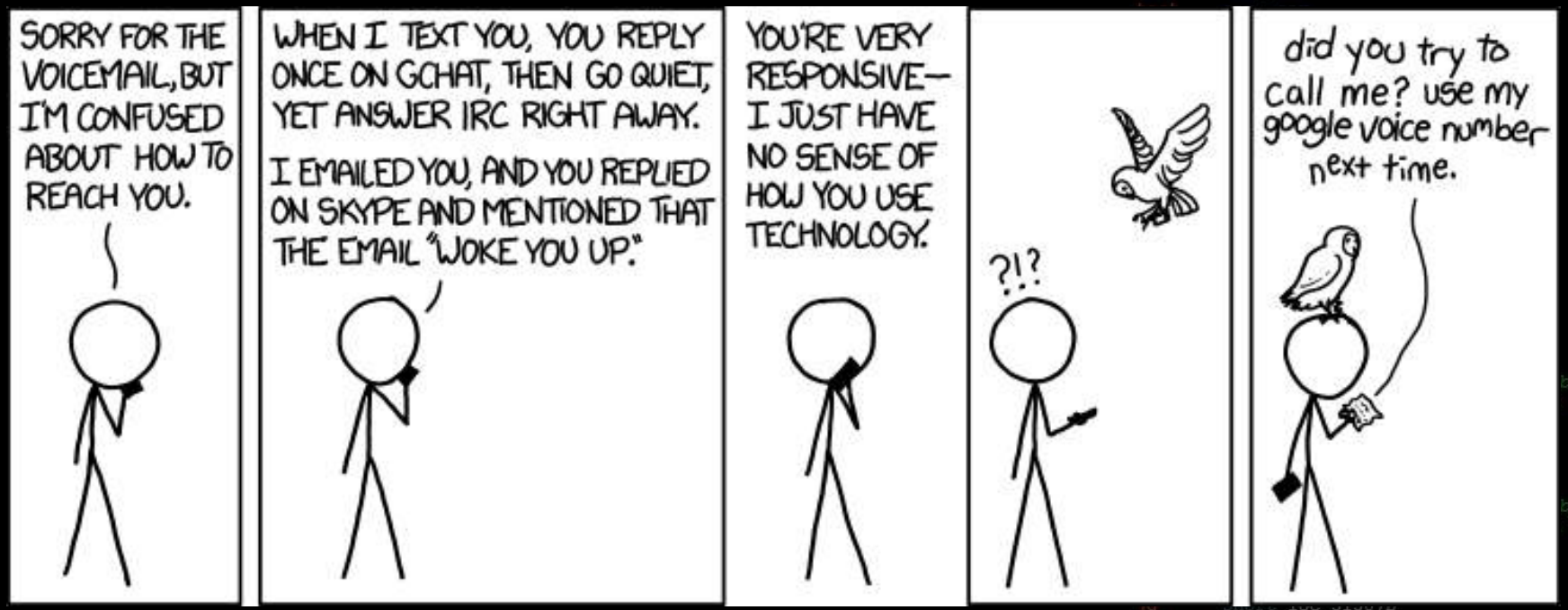
Office Hours

- Office hours:
  - Wednesday 7-10 PM @ Sage 3101

- Come hang out at RPISEC hack nights!
  - Ask questions, get extra help with MBE
  - Collaborate on HW/Labs
  - Work on security projects, challenges, etc



Other Options



Digital Office Hours (IRC)

- The RPISEC IRC - http://rpis.ec/irc
  - server: irc.rpis.ec
  - port: 6667 (6697 for SSL)
  - room: #rpisec

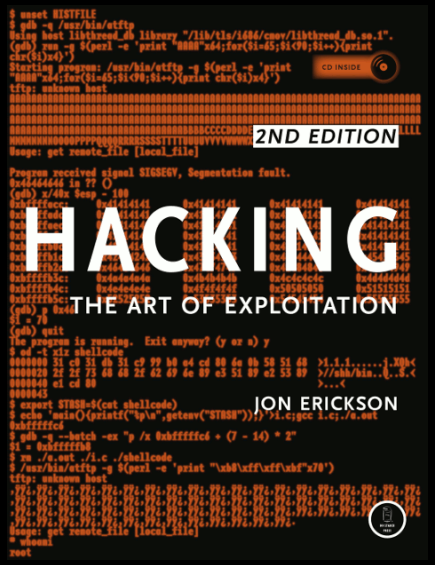
- Way faster than emailing back and forth
- Some of us are usually on at ridiculous hours
  - basically a 24/7 channel

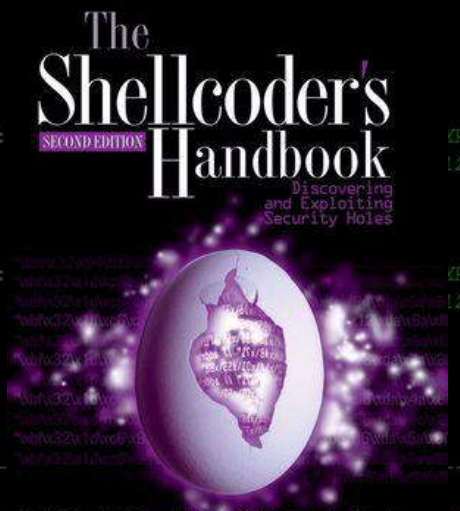
Options of Last Resort

- Email us
  - binexp_ta@cs.lists.rpi.edu

Suggested Textbooks

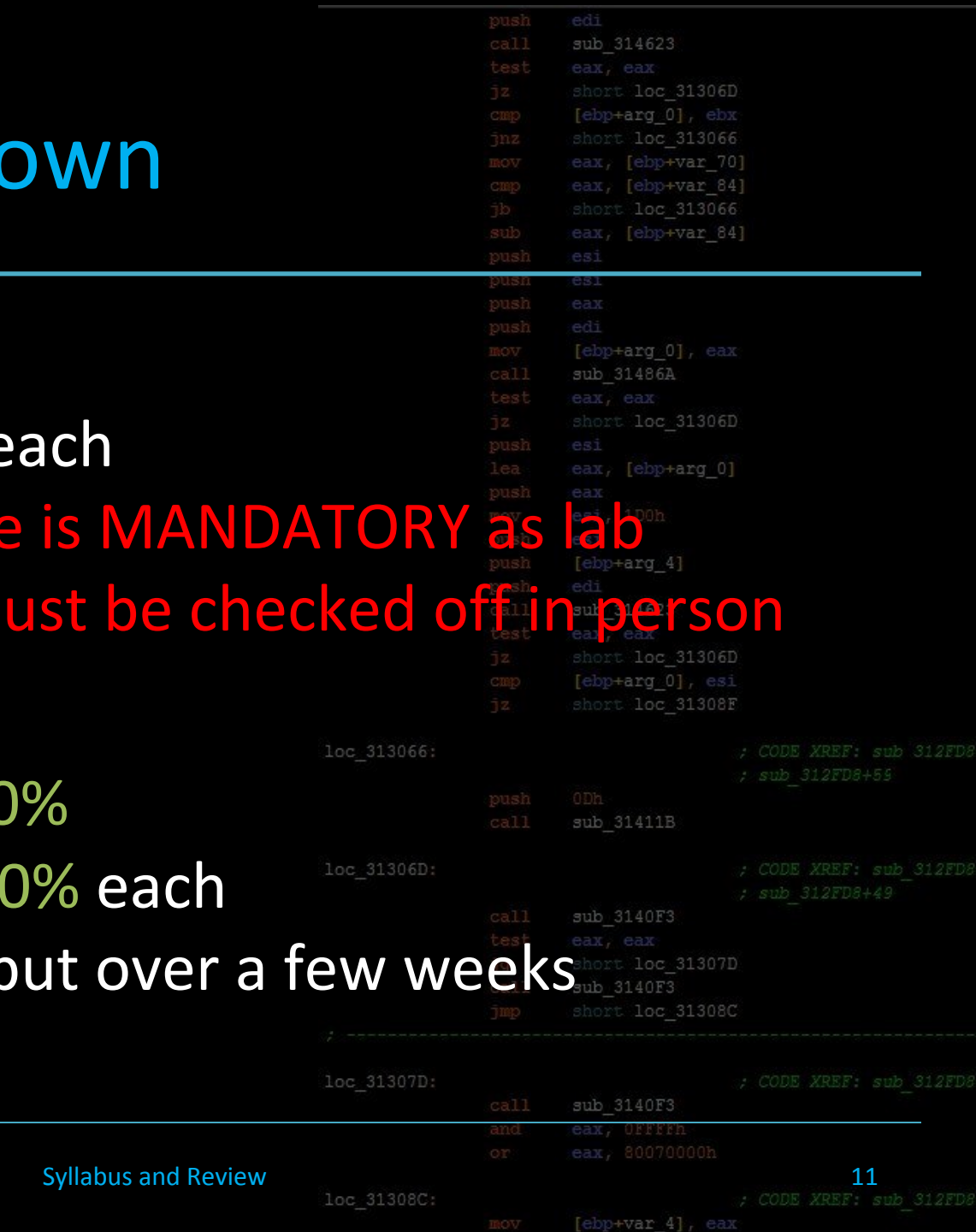
- Hacking: The Art of Exploitation, 2nd Edition by Jon Erickson
  - ISBN 978-1593271442
- The Shellcoder's Handbook: Discovering and Exploiting Security Holes, 2nd Edition by Chris Anley et al
  - ISBN 978-0470080238





Grade Breakdown

- Labs - 60%
  - 10 labs @ 6% each
  - Lab attendance is MANDATORY as lab submissions must be checked off in person
- Term Projects - 40%
  - 2 Projects @ 20% each
  - Like a big lab, but over a few weeks

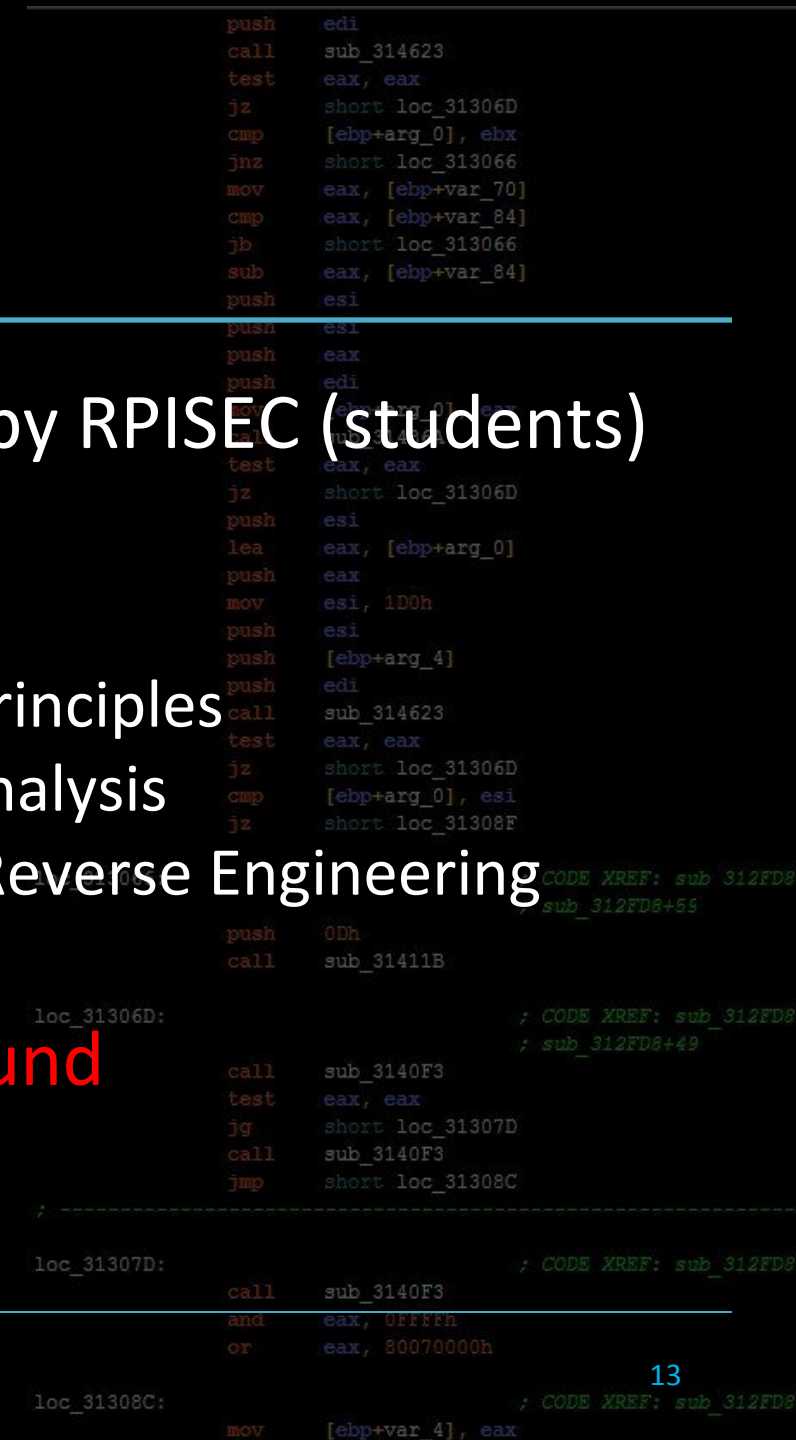


Syllabus

- READ THE SYLLABUS - Well written, full of details
  - It's on the course website - rpis.ec/binexp

An Atypical Class

- Designed and orchestrated by RPISEC (students)
- Biggest RPISEC class yet!
  - CSCI 4971 Secure Software Principles
  - CSCI 4972 / 6963 Malware Analysis
  - CSCI 4974 / 6974 Hardware Reverse Engineering
- We're not here to mess around

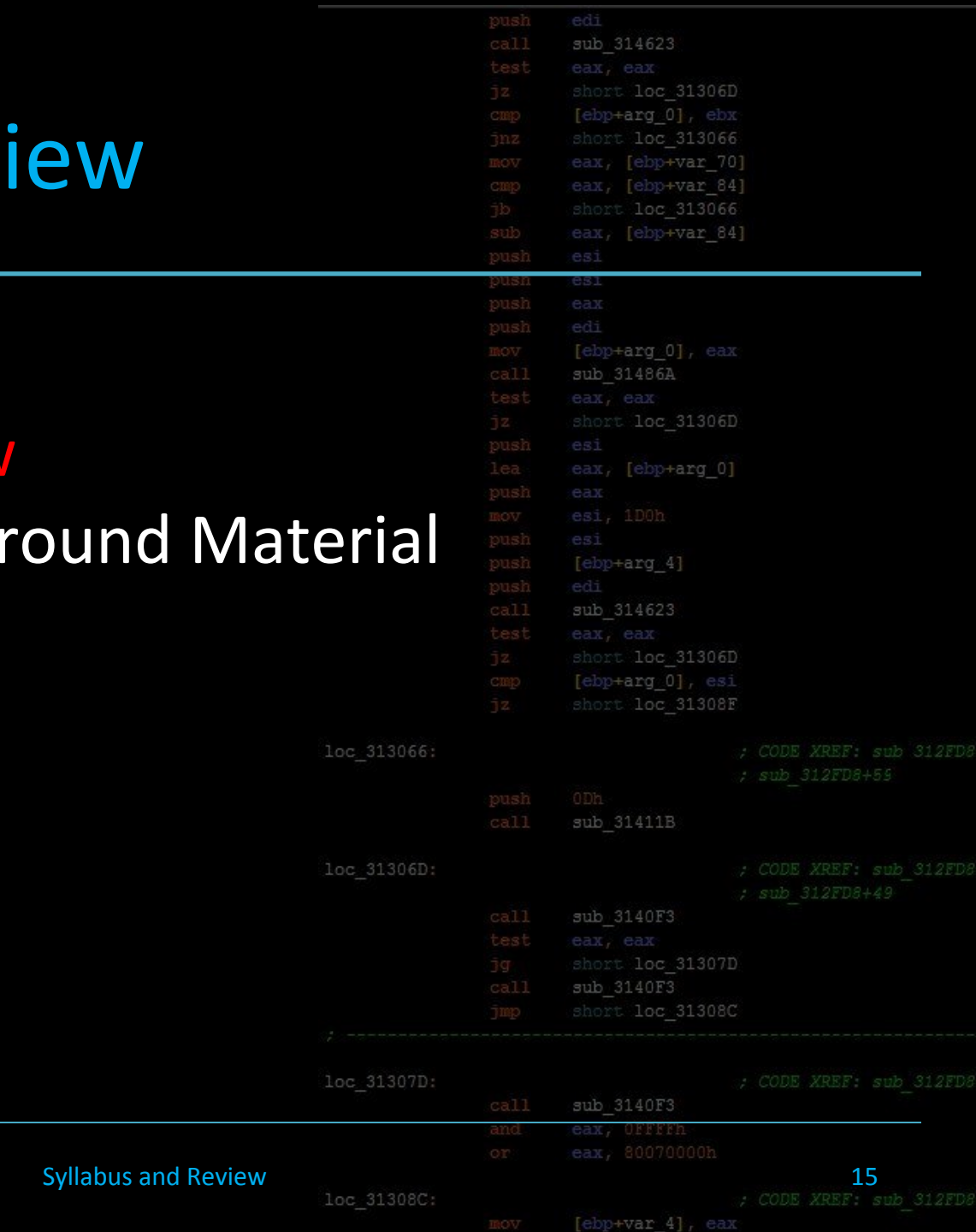


RPISEC

- Good to see lots of familiar faces!
- RPISEC meetings are Friday 5-7 PM in DCC 324
- Come learn other topics in computer security
  - Web hacking
  - Malware analysis
  - Reverse Engineering
  - Digital Forensics
  - So so much more
- Meet people from industry, get internships/jobs
- Read more - http://rpis.ec

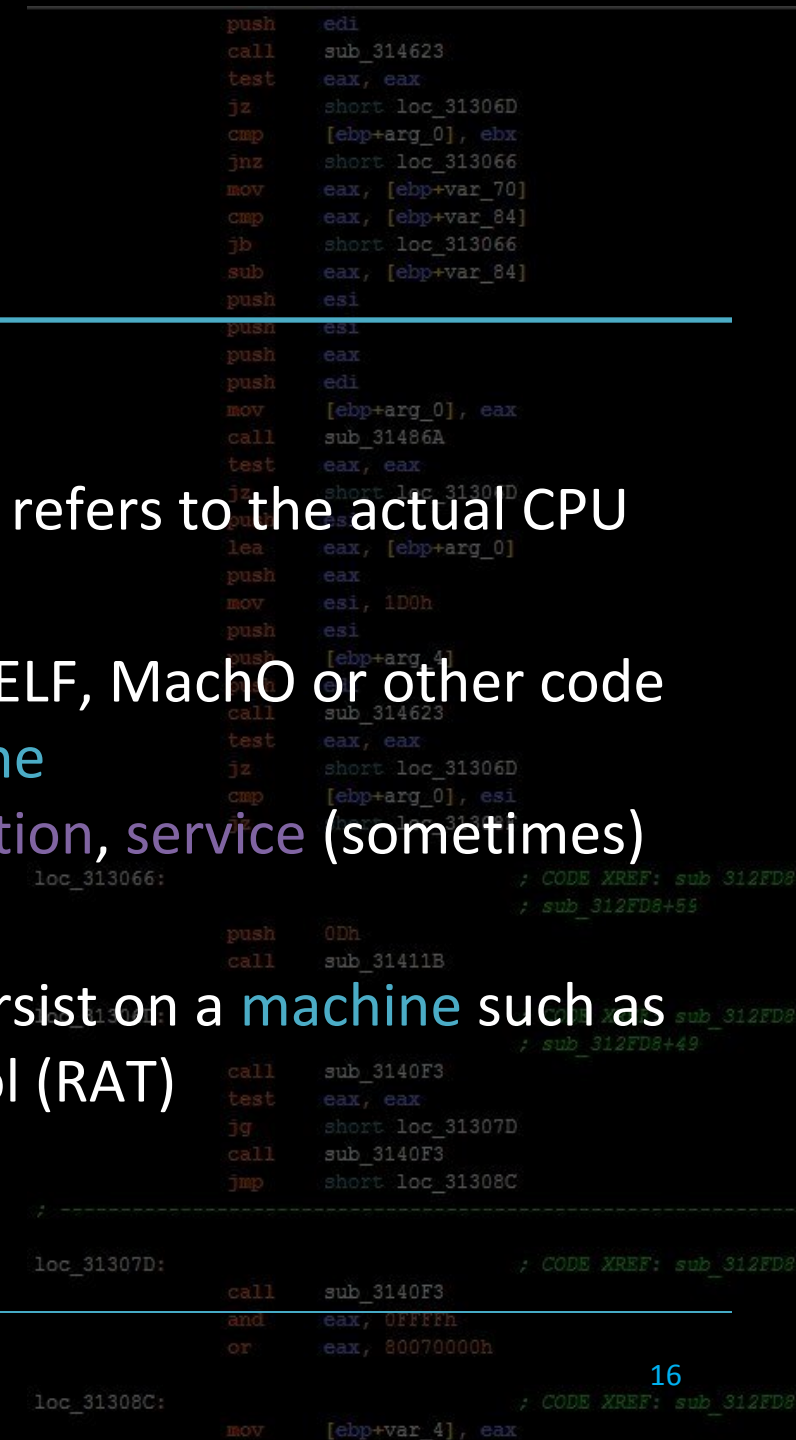
Lecture Overview

1. Syllabus
2. Course Overview
3. Review of Background Material
   a. Linux
   b. C
   c. x86 Assembly



Course Terminology

- Machine
  - A computer, server, sometimes refers to the actual CPU
- Binary
  - An executable such as an .EXE, ELF, MachO or other code containers that run on a machine
  - Other names: program, application, service (sometimes)
- Malware
  - A malicious binary meant to persist on a machine such as a Rootkit or Remote Access Tool (RAT)



Course Terminology

- Vulnerability
  - A bug in a binary that can be leveraged by an exploit
- Exploit (as a noun)
  - Specially crafted data that utilizes vulnerabilities to force the binary into doing something unintended
  - By this definition, exploits are not explicitly malware
- 0day
  - A previously unknown or unpatched vulnerability that can be used by an exploit
  - An 0day can also be an exploit using the unpatched vuln

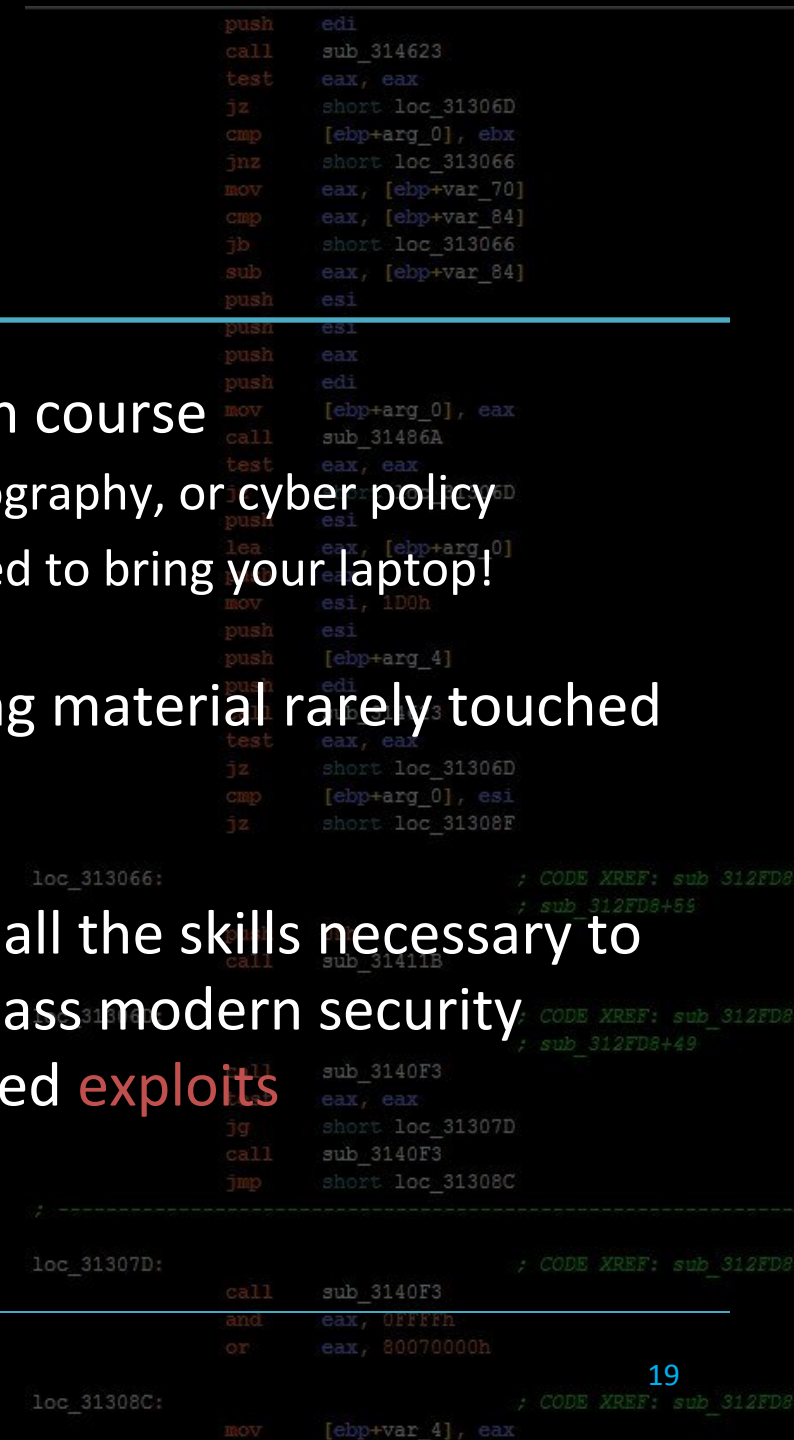
Premise For This Class

"Can we teach a bunch of programmers how to pwn?"

- Pwn/Pwning
  - In security, pwning commonly refers to vulnerability research and exploit development

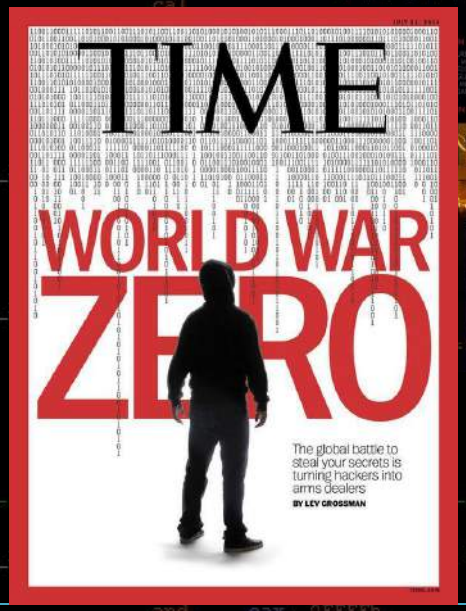
Goals for This Course

- This will be a very applied, hands on course
  - No data structures, algorithms, cryptography, or cyber policy
  - Every lecture after this you're expected to bring your laptop!
- We will cover technically challenging material rarely touched upon in other classes
- As an individual you will leave with all the skills necessary to perform vulnerability research, bypass modern security protections, and develop weaponized exploits



“Dark Arts” of Computer Science

- Almost non-existent in academia
  - Taboo around offensive security
  - Rapidly evolving, very technical
- Why learn binary exploitation?
  - Can’t defend against what you don’t understand
  - Gain an intimate understanding of how programs *really* work
  - Fun, intriguing, rewarding problems
  - So few people know how to pwn
  - Exploding job market



The Market for An 0day (2012)

ADOBE READER	$5,000–$30,000
MAC OSX	$20,000–$50,000
ANDROID	$30,000–$60,000
FLASH OR JAVA BROWSER PLUG-INS	$40,000–$100,000
MICROSOFT WORD	$50,000–$100,000
WINDOWS	$60,000–$120,000
FIREFOX OR SAFARI	$60,000–$150,000
CHROME OR INTERNET EXPLORER	$80,000–$200,000
IOS	$100,000–$250,000

2015? Double these numbers

Underappreciated Wisdom

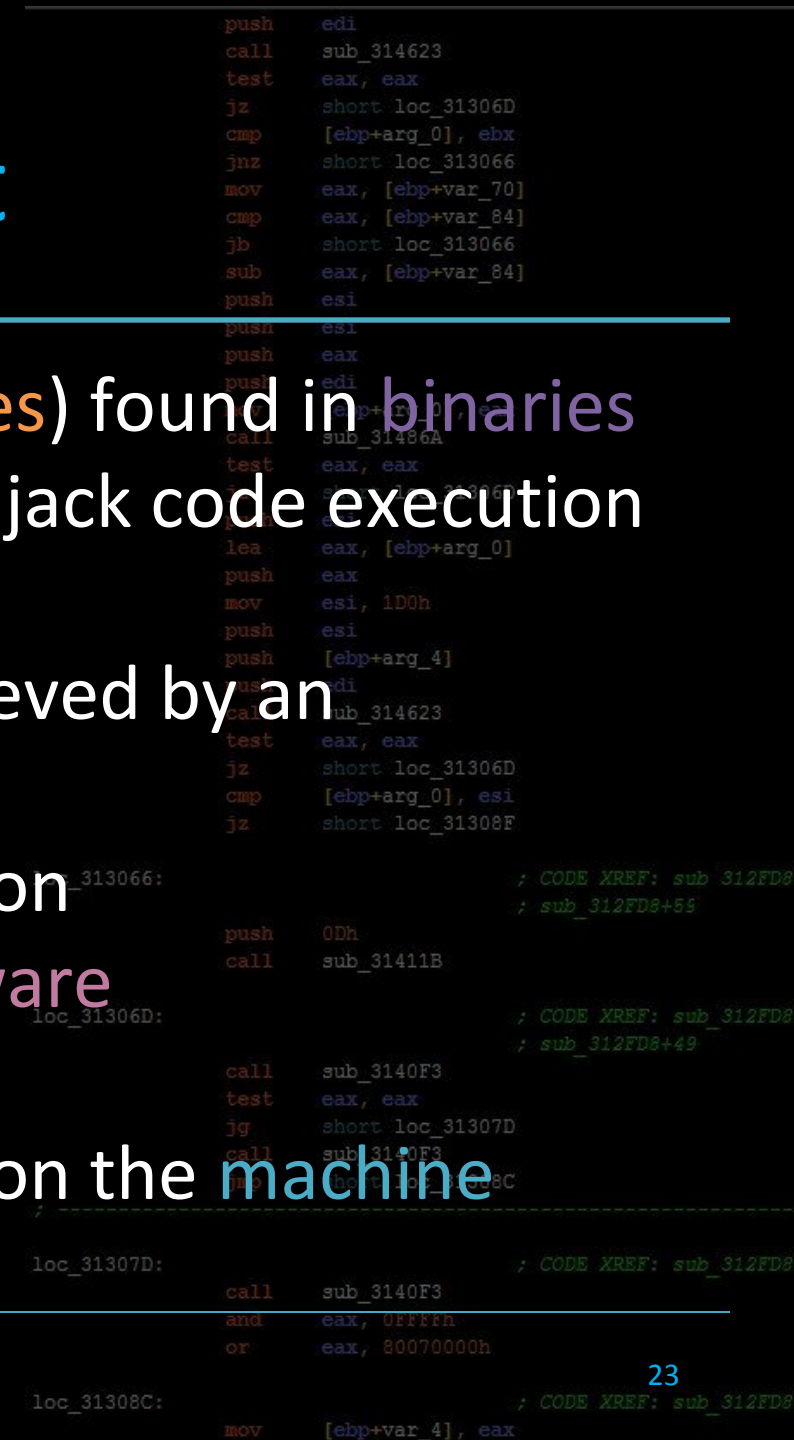
"If your program simply segfaulted, consider yourself lucky."

- Prof. Stewart

More Than a Segfault

- The right bugs (vulnerabilities) found in binaries can be used by exploits to hijack code execution

- Once code execution is achieved by an attacker...
  - Gain privileged information
  - Download or install malware
  - Steal data
  - Wreak any sort of havoc on the machine



Fun Example of Binary Exploitation

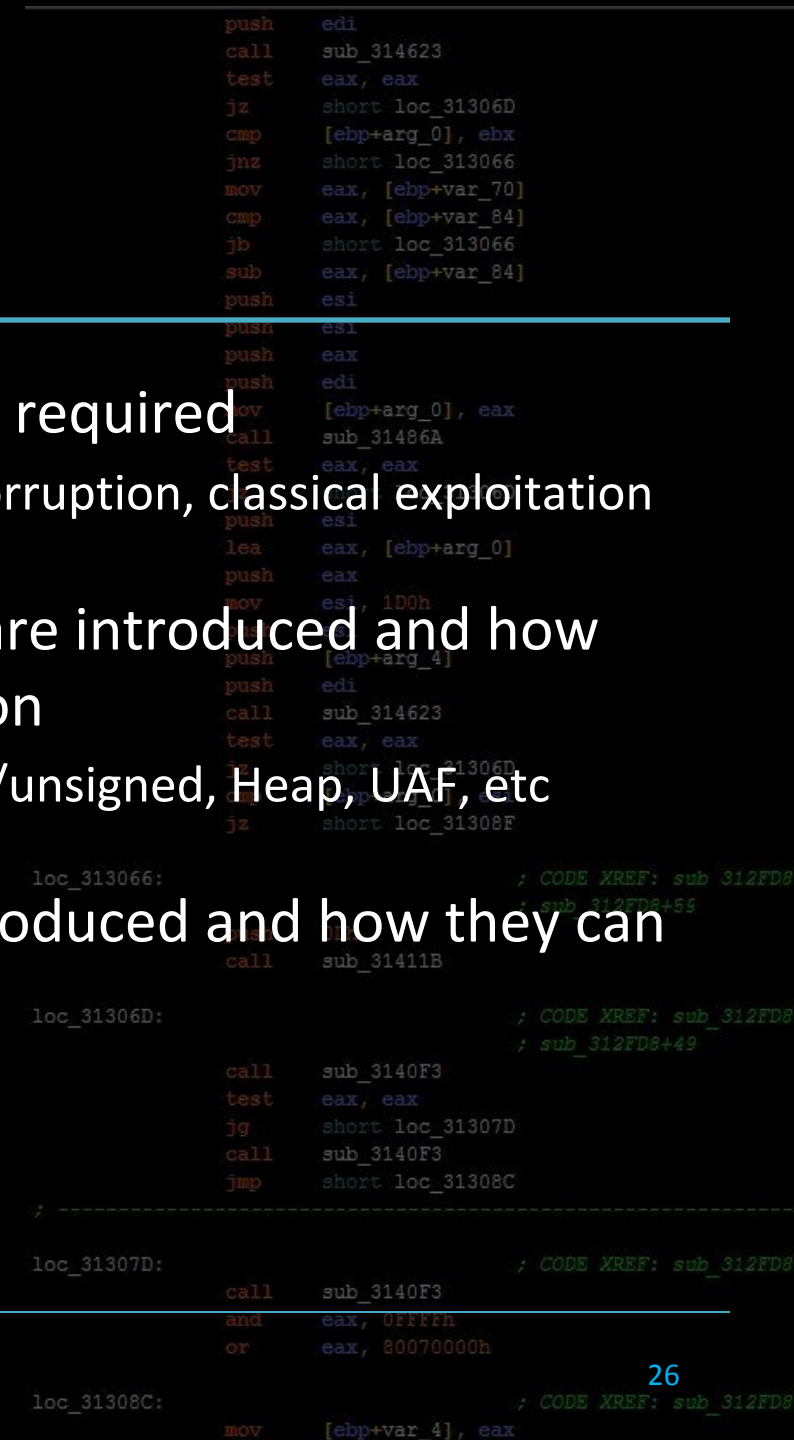
"[TAS] Super Mario World "**Arbitrary Code Execution**" in 02:25.19 by Masterjun"

Events in Security & Exploitation

- 1972 - USAF Computer Security Technology Planning Study describes buffer overflows
- 1988 - Morris Worm exploits use of gets() in finger daemon
- 1996 - Aleph1 publishes “Smashing the Stack for Fun and Profit” in Phrack
- 2001 - Code Red worm exploits a MS web server vulnerability to hit hundreds of thousands of computers
- 2004 - Windows XP SP2 released, exploit mitigation era begins
- 2007 - The first iPhone jailbreak is developed by GeoHot
- 2008-2010 - Stuxnet employs four Windows 0days to spread through Iranian nuclear refinery control system networks

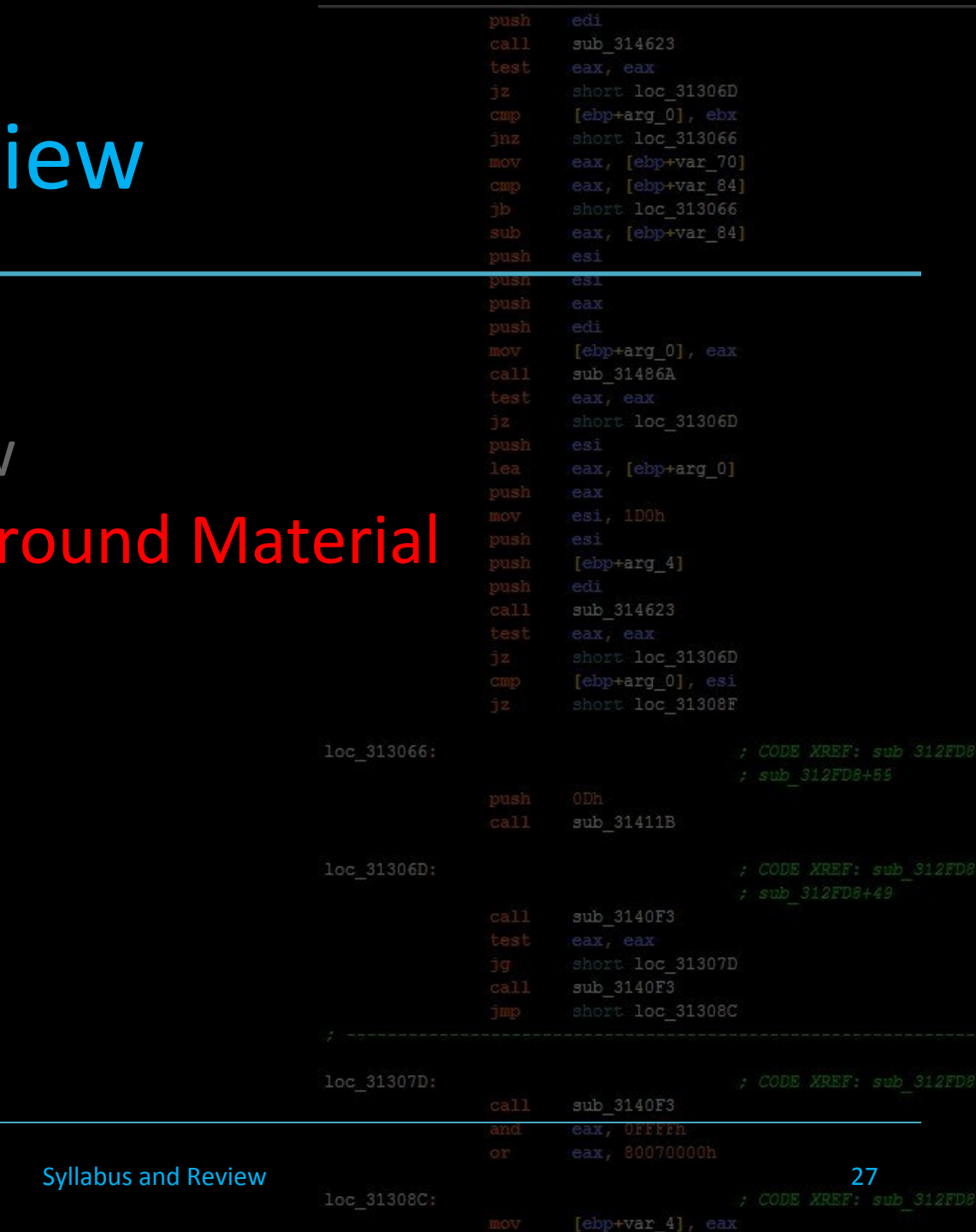
Course Roadmap

- We start off with the fundamentals required
  - Basic reverse engineering, memory corruption, classical exploitation
- Different classes of vulnerabilities are introduced and how they can be leveraged in exploitation
  - Stack smashes, format strings, signed/unsigned, Heap, UAF, etc
- Modern exploit mitigations are introduced and how they can be bypassed in exploitation
  - DEP, ASLR, GS/Cookies,



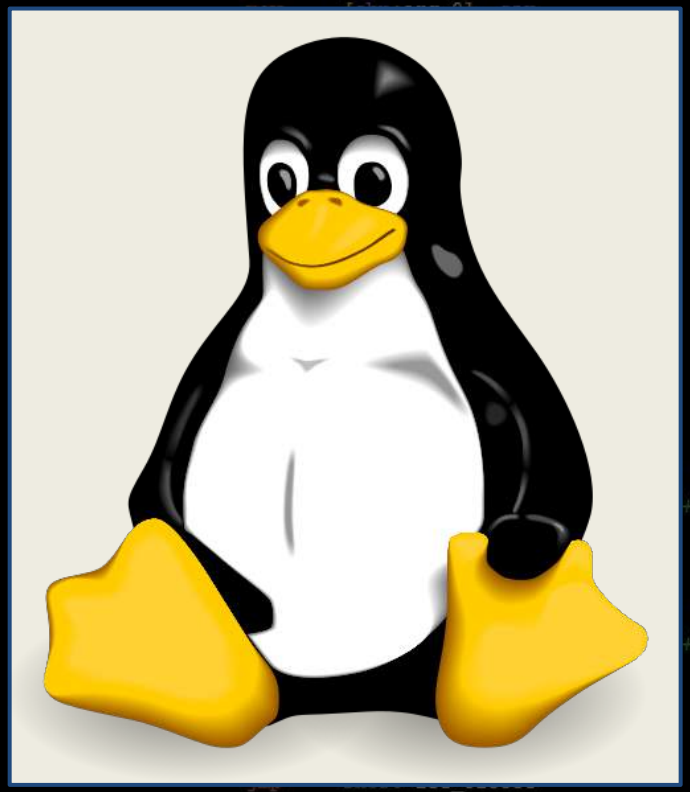
Lecture Overview

- Syllabus
- Course Overview
- Review of Background Material
  - Linux
  - C
  - x86 Assembly



Quick Linux Overview

- UNIX-like open source kernel used by many open source operating systems distros
- Written in C and assembly
- ELF (Executable and Linkable Format) files for binaries
- We'll be teaching on Ubuntu 14.04 systems, but exploitation techniques are pretty universal



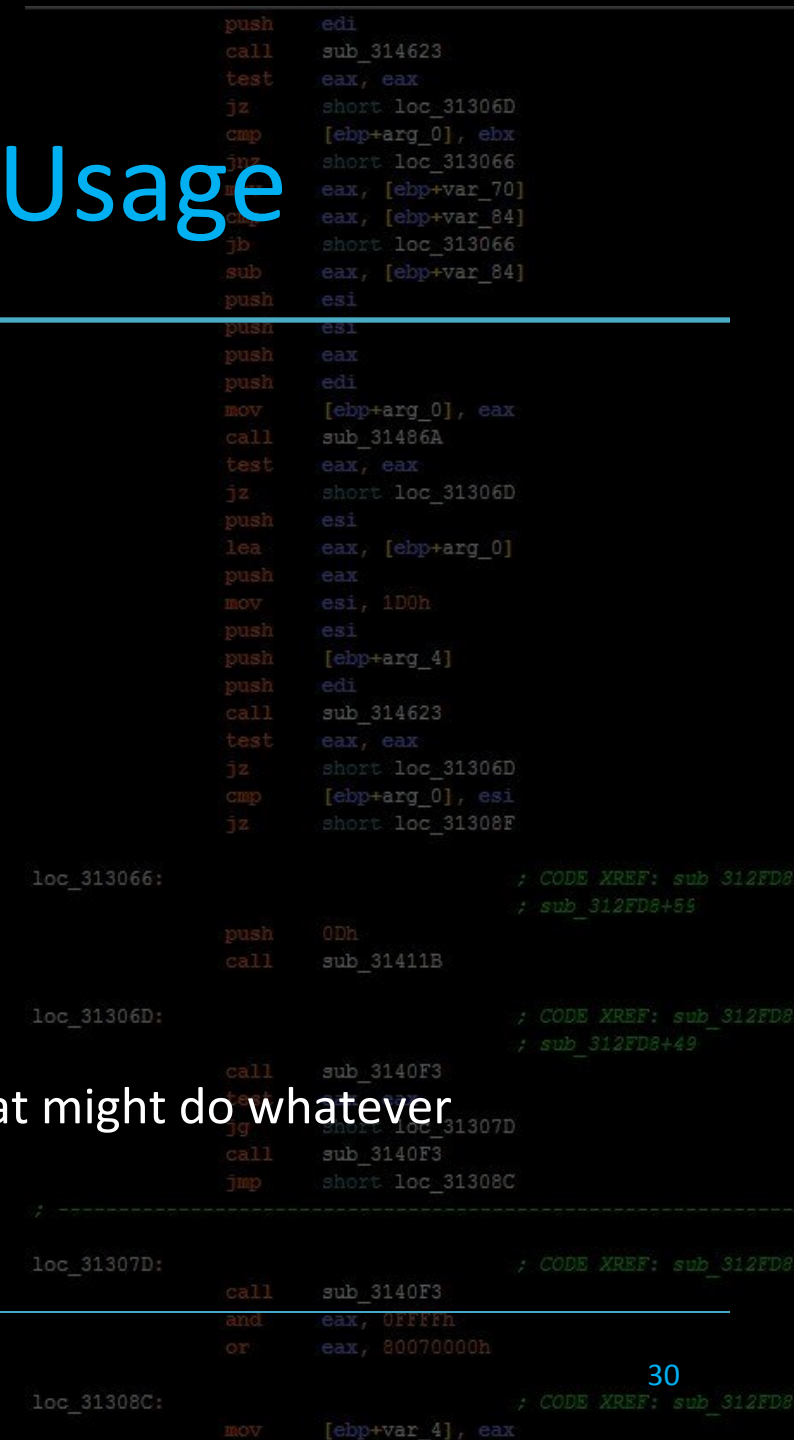
Learning Command Line Linux

- We'll be spending a lot of time using linux at the command line in this class, so you'll need to learn your way around

- Get familiar with the linux command line if you aren't already
  - http://overthewire.org/wargames/bandit/

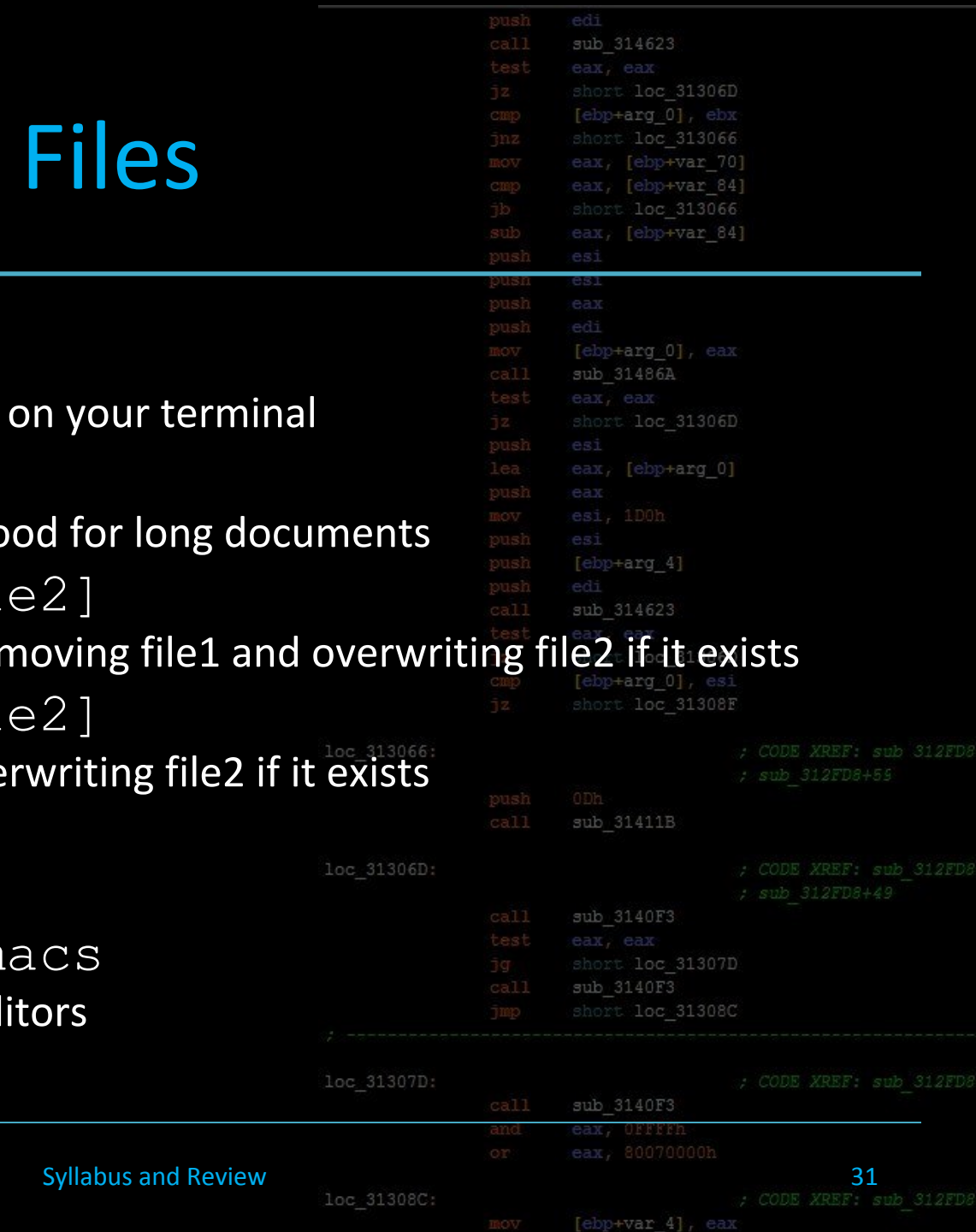
Basic Command Line Usage

- `ls`
  - List directory contents
- `cd [path]`
  - change directory
  - `".."` = previous
- `pwd`
  - Print working directory
- `man [command]`
  - Manual for command
- `apropos [whatever]`
  - Get info on commands/man pages that might do whatever



Working With Files

- `cat [file]`
  - Print the file contents on your terminal
- `less [file]`
  - Like cat, but paged, good for long documents
- `mv [file1] [file2]`
  - Move file1 to file2, removing file1 and overwriting file2 if it exists
- `cp [file1] [file2]`
  - Copy file1 to file2, overwriting file2 if it exists
- `rm [file]`
  - Deletes file
- `nano / vim / emacs`
  - Command line text editors



Piping Program Input / Output

- Pipes - "|"
  - Take output of one program, send it as input to another
  - `$ echo "hello"  | cowsay`

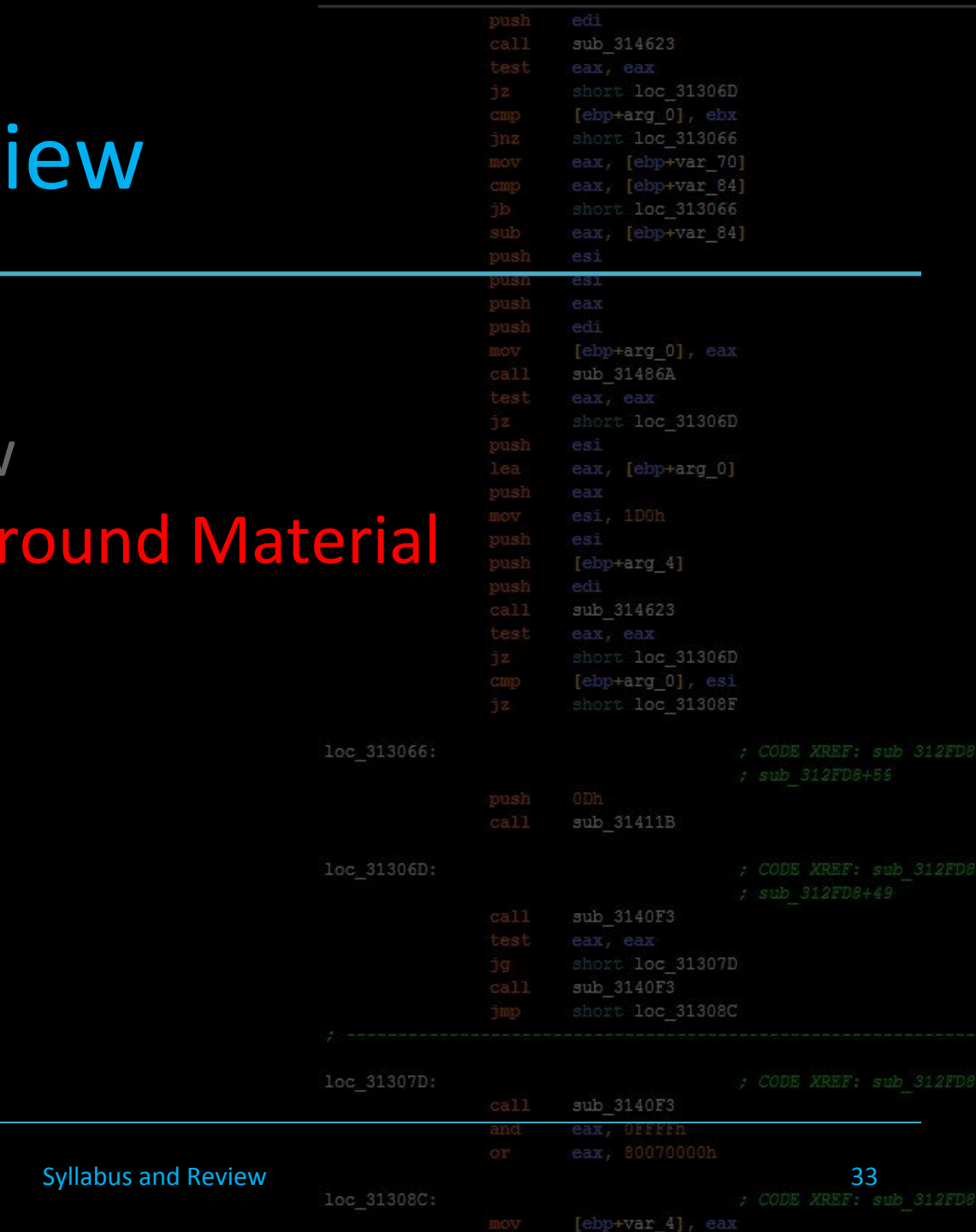
```
 _______
< hello >
 -------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

- `$ echo "hello" | cowsay | grep "hello"`

```
< hello >
```

Lecture Overview

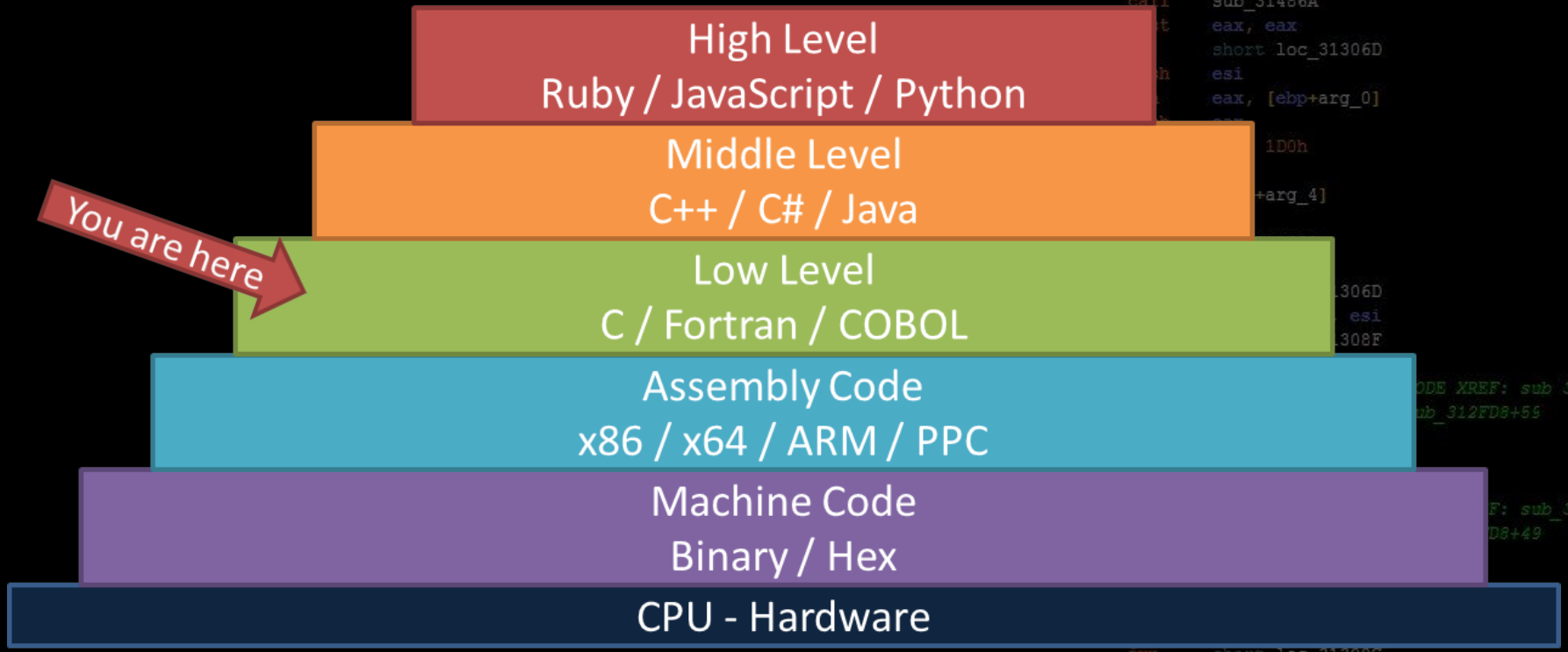
- Syllabus
- Course Overview
- Review of Background Material
  - Linux
  - C
  - x86 Assembly



The C Programming Language

- Designed in 1969-1972 for writing UNIX operating system
- Imperative systems programming language
  - Very fast, compiled language
  - Extremely fine control over memory and the machine
- Compared to modern languages, C is considered a ‘low level’ language

Language Depth



Hello World! - C Source

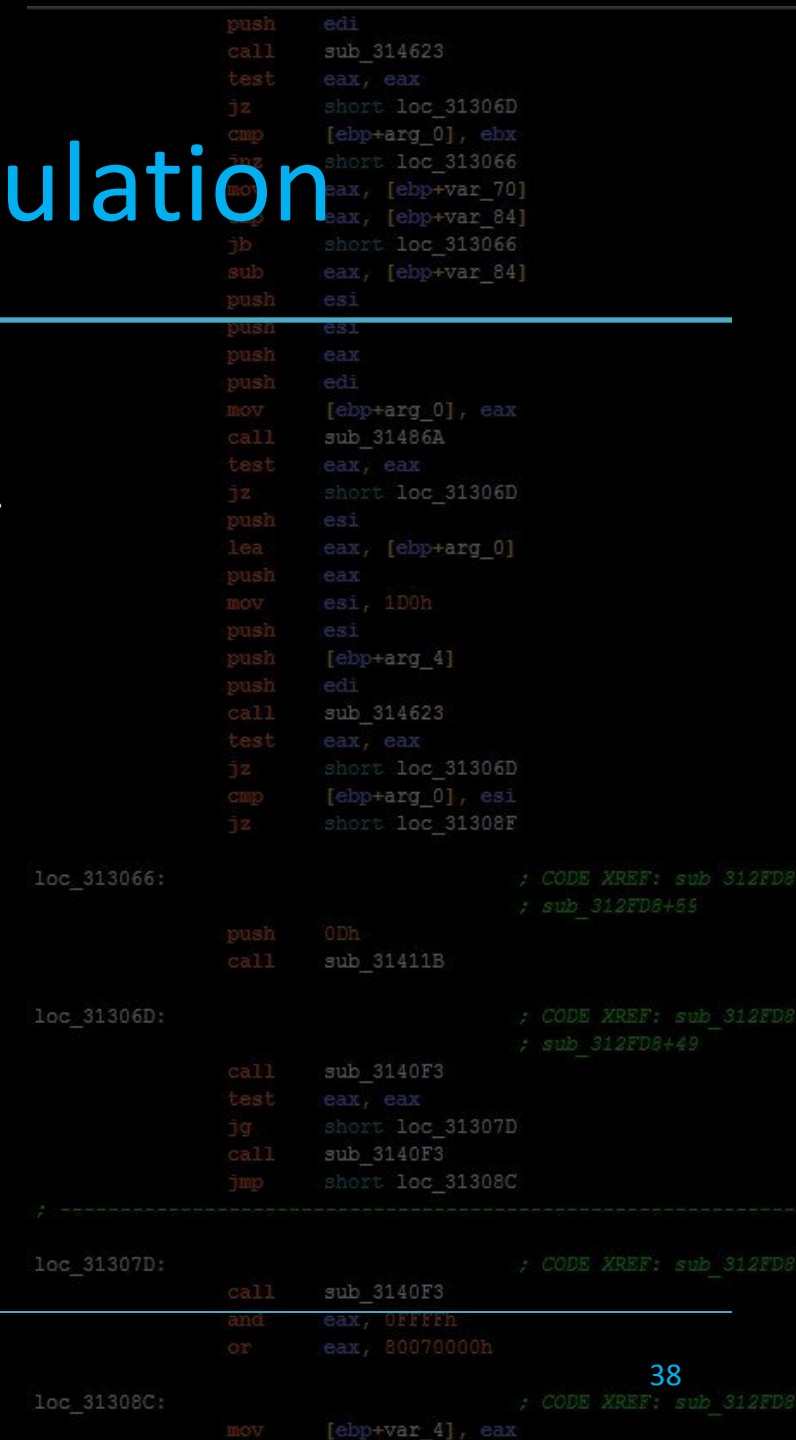
```c
#include <stdio.h>

int main(int argc, char * argv[])
{
  printf("Hello World!\n");
  return 0;
}
```

Hello World! - Compiling/Running

```
$ gcc helloworld.c -o helloworld
$ ./helloworld
Hello World!
```

Basic Memory Manipulation

```
int i = 0;
char * message = "hello world";
char * buffer = (char *)malloc(7);

if(buffer == NULL)
    return 1;

strncpy(buffer, message, 5);
buffer[5] = '\n';
buffer[6] = '\0';

for(i = 0; i < 10; i++)
    printf("%s", buffer);

free(buffer);
```

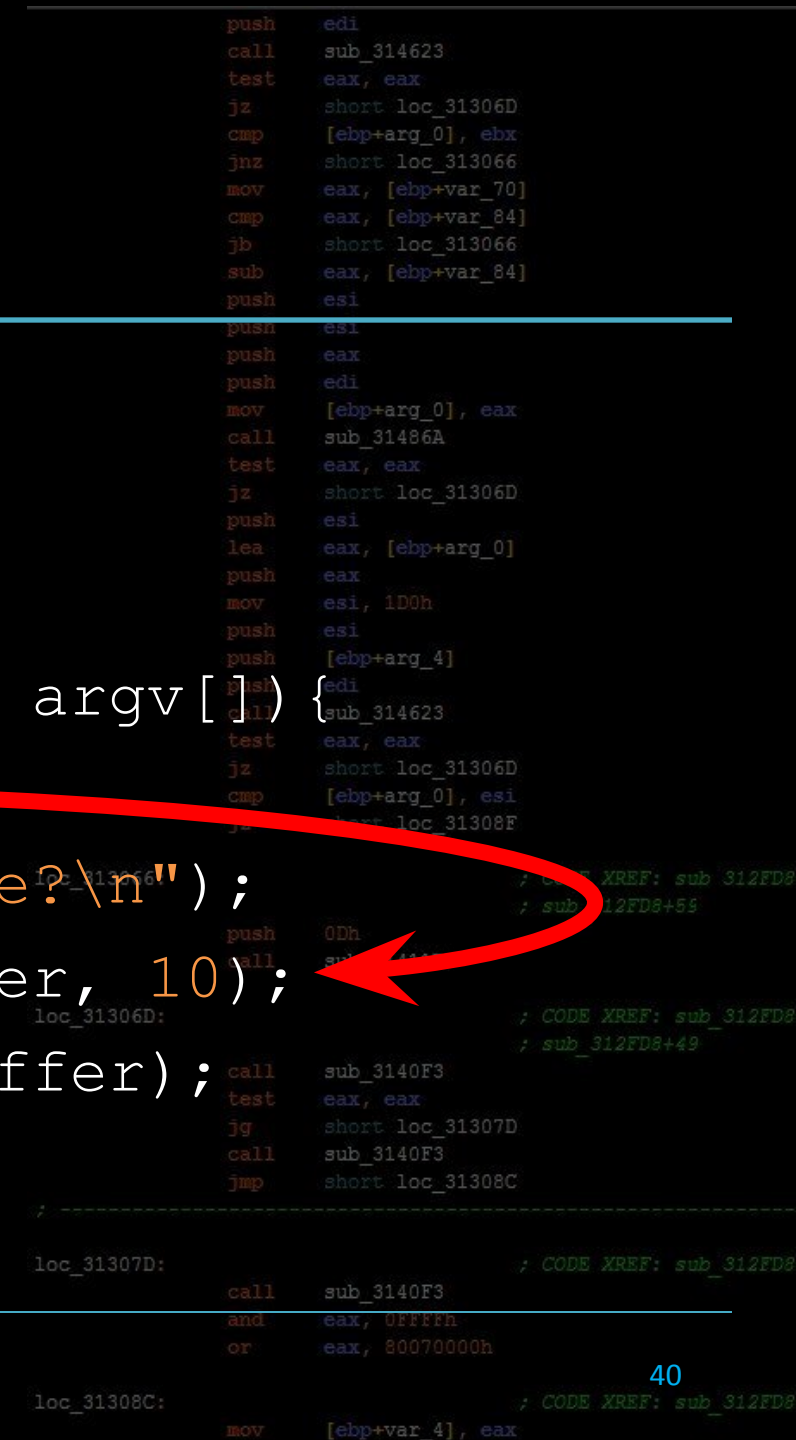


Running It

```
$ gcc basic.c  -o basic -std=gnu99
$ ./basic
hello
hello
hello
hello
hello
...
```

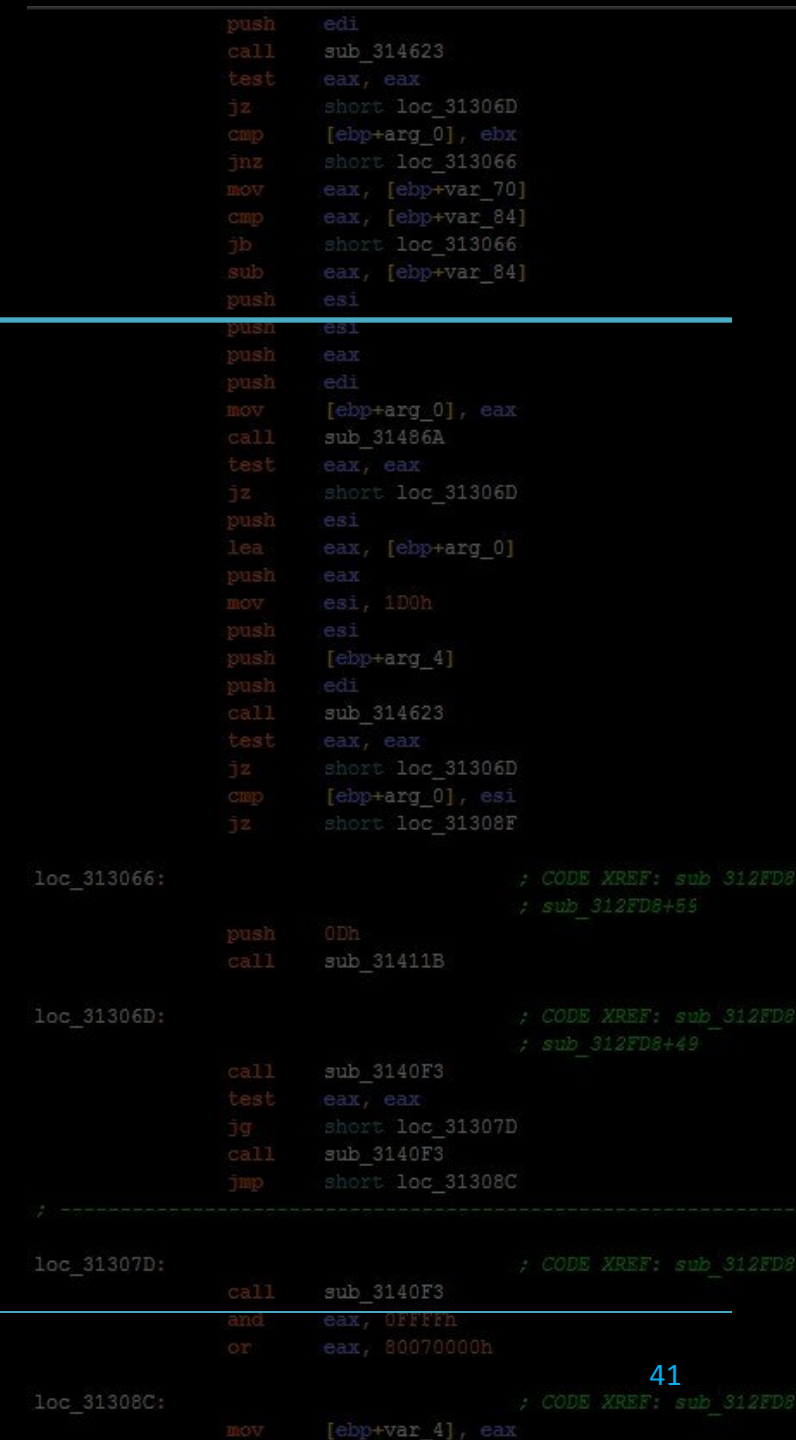
What's your name?

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[]){
  char buffer[10] = {0};
  printf("What's your name?\n");
  read(STDIN_FILENO, buffer, 10);
  printf("Hello %s\n", buffer);
  return 0;
}
```



Hello ALEX 1234 ??

```
$ gcc name.c -o name
$ ./name
What's your name?
ALEX 1234 ABCD
Hello ALEX 1234 ??
```



What's your name? - 2.0

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

int main(int argc, char * argv[]){
  char buffer[10] = {0};
  printf("What's your name?\n");
  read(STDIN_FILENO, buffer, 100);
  printf("Hello %s\n", buffer);
  return 0;
}
```

Crash!

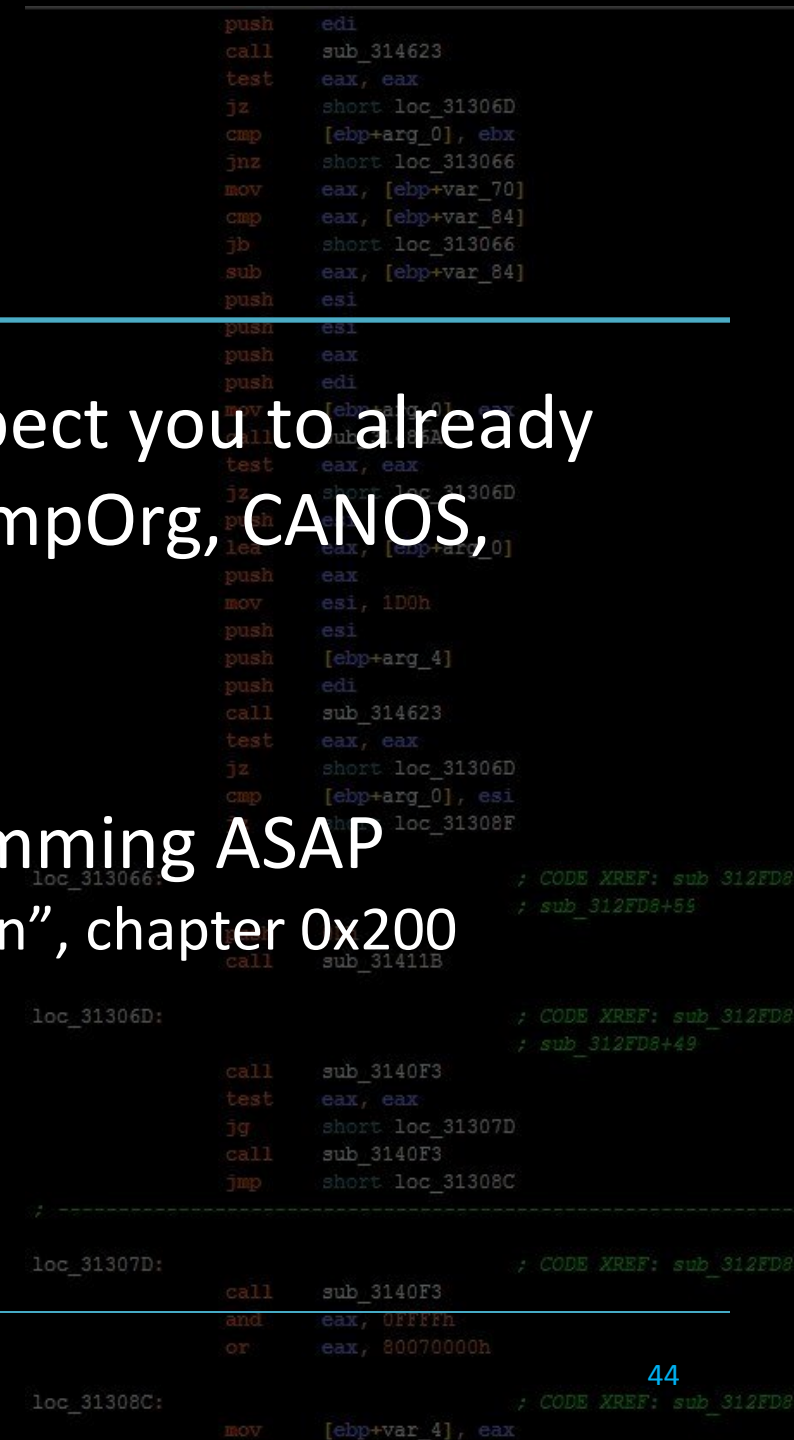
```
$ gcc name2.c -o name2
$ ./name2
What's your name?
ALEX 1234 ABCD EFGH IJKL
Hello ALEX 1234 ABCD EFGH IJKL

??????????
Segmentation fault (core dumped)
```

- Bottom line: it's easy to make grievous errors in C

So If C Scared You...

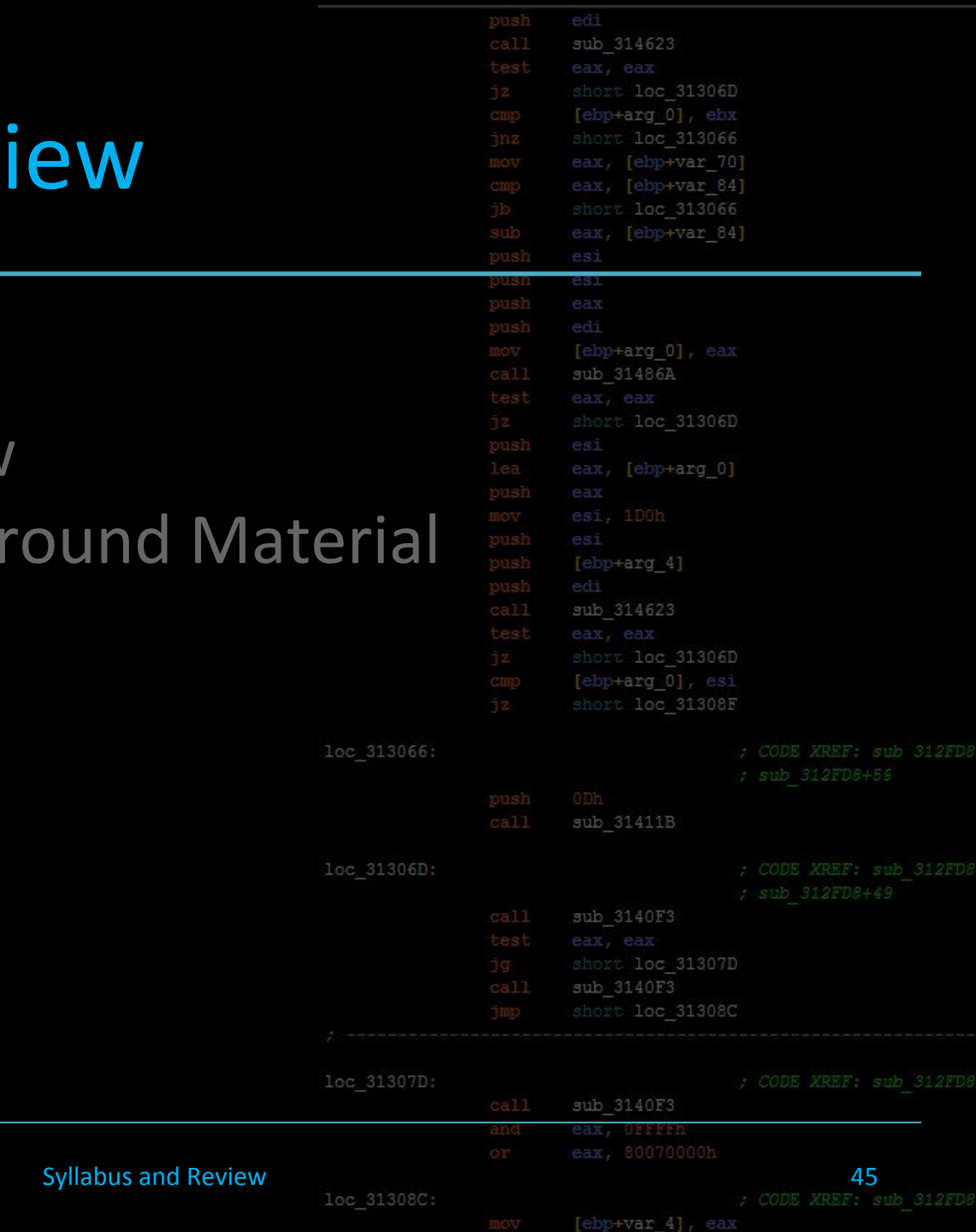
- If you're in this class, we expect you to already know some basic C from CompOrg, CANOS, OpSys, or NetProg

- Otherwise, review C programming ASAP
  - "Hacking: The Art of Exploitation", chapter 0x200



Lecture Overview

- Syllabus
- Course Overview
- Review of Background Material
  - Linux
  - C
  - x86 Assembly

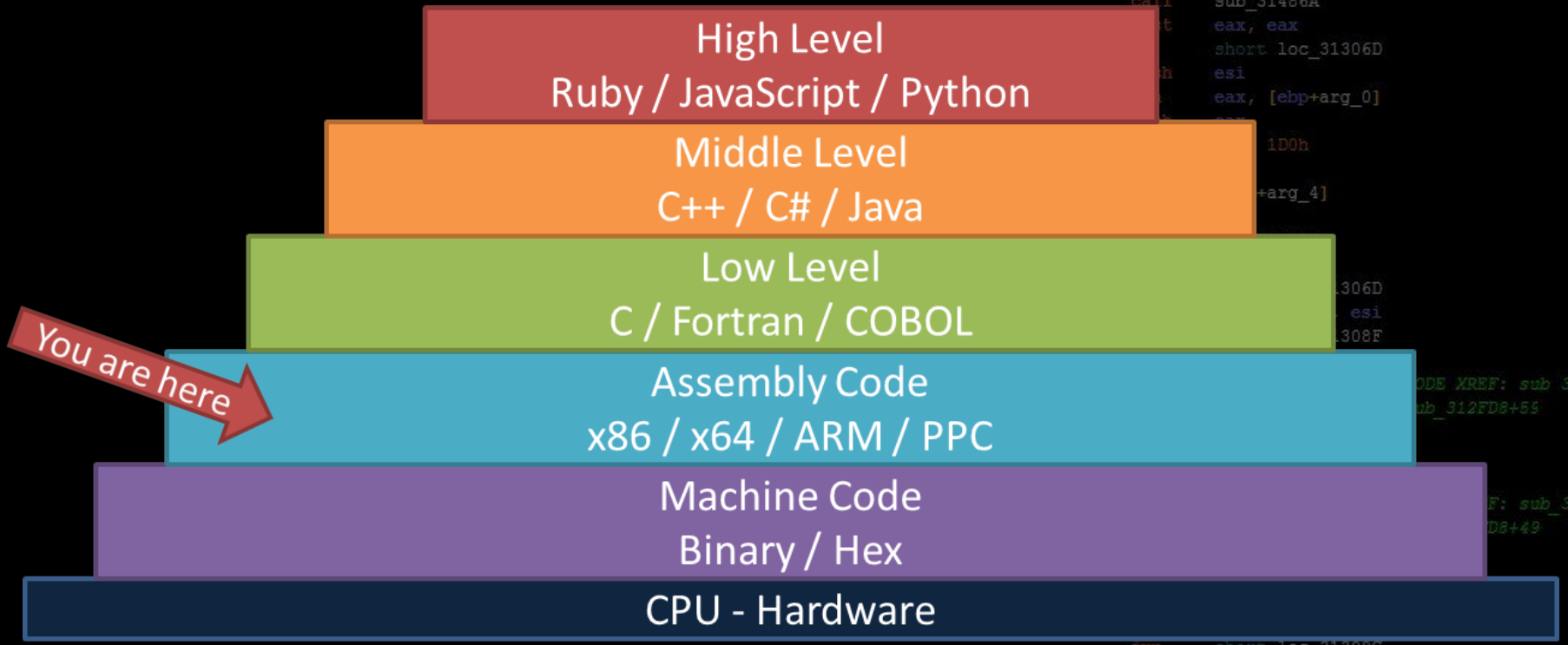


x86 Assembly

- An assembly instruction set introduced in 1978 by Intel
  - 1978 - 16bit
  - 1985 - 32bit
  - 2001 - 64bit (Itanium)
  - 2003 - 64bit (AMD64)
- Overwrought CISC, a total playground for exploitation
- As low level as we'll go

```
jmp     loc_8056CB8

                                ; CODE XREF
cmp     [esp+4BCh+var_494], 45h ;
jz      loc_8056CA0
mov     eax, [esp+4BCh+var_478]
mov     [esp+4BCh+var_470], 1
mov     esi, [eax+18h]
jmp     loc_8056DD0

                                ; CODE XREF
cmp     [esp+4BCh+var_494], 45h ;
jz      loc_8056CA0
mov     esi, [esp+4BCh+var_478]
mov     ebx, 92492493h
mov     eax, [esi+18h]
lea     ecx, [eax+6]
mov     [esp+4BCh+var_48C], eax
mov     eax, ecx
imul    ebx
lea     eax, [edx+ecx]
mov     edx, ecx
sar     edx, 1Fh
sar     eax, 2
```

Language Depth

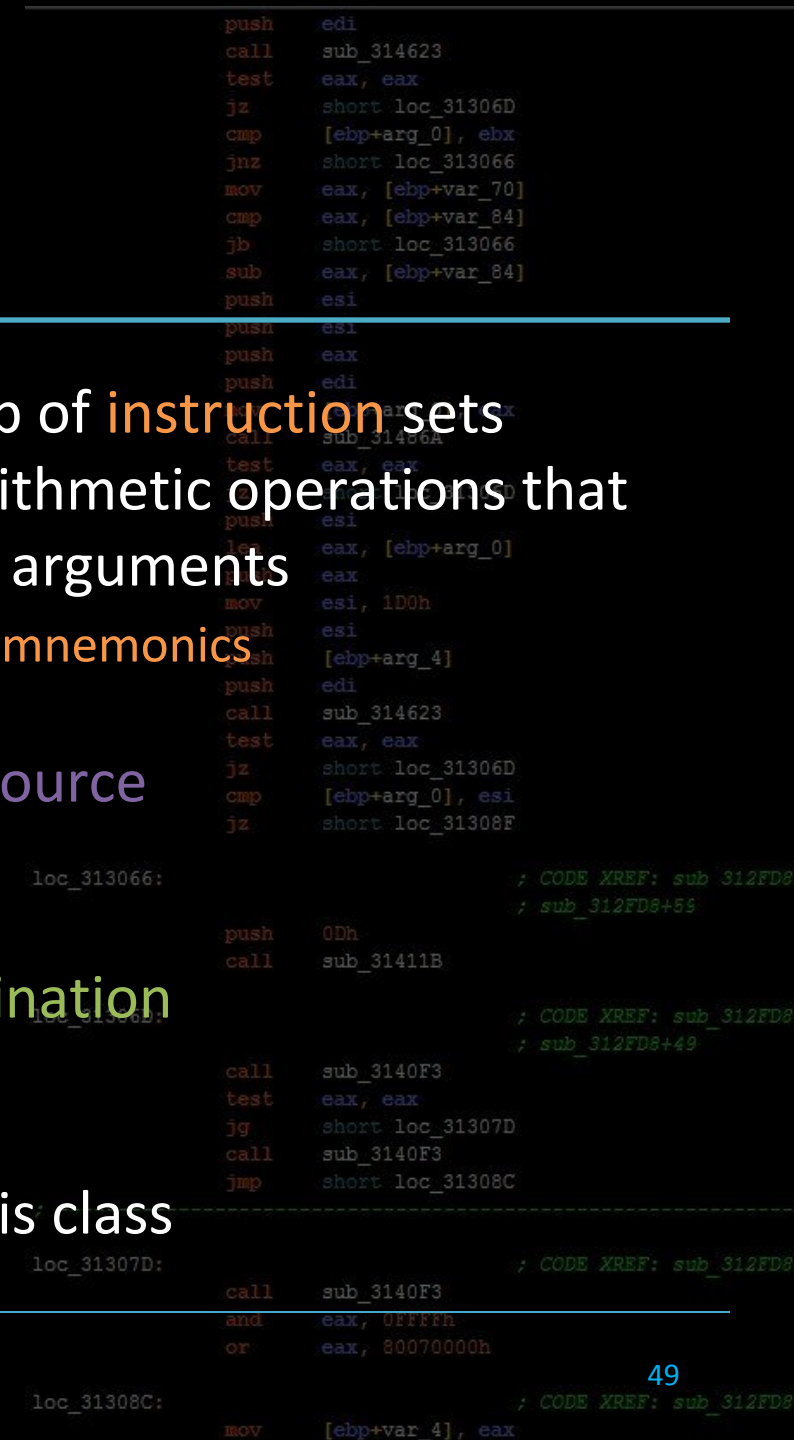


Pulling Back the Curtain

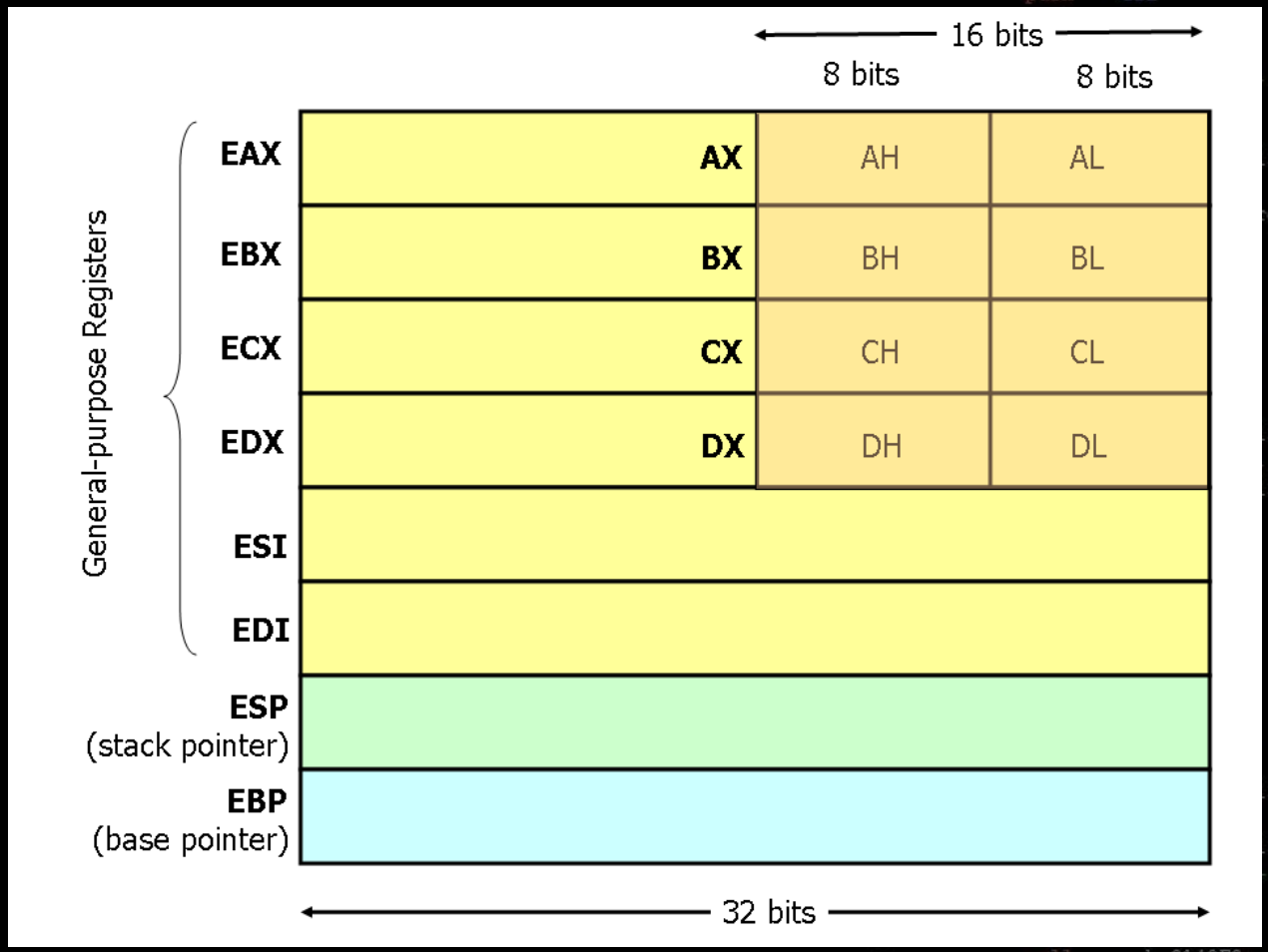
"... there's way too much information to decode the Matrix. You get used to it, though. Your brain does the translating. I don't even see the code. All I see is blonde, brunette, redhead." -Cypher, The Matrix

x86 Assembly Syntax

- All assembly languages are made up of instruction sets
- Instructions are generally simple arithmetic operations that take registers or constant values as arguments
  - Also called Operands, OpCode, Op(s), mnemonics
- Intel syntax: operand destination, source
  - `mov eax, 5`
- AT&T syntax: operand source, destination
  - `mov $5, eax`
- We'll be using the Intel syntax in this class



x86 Register Diagram



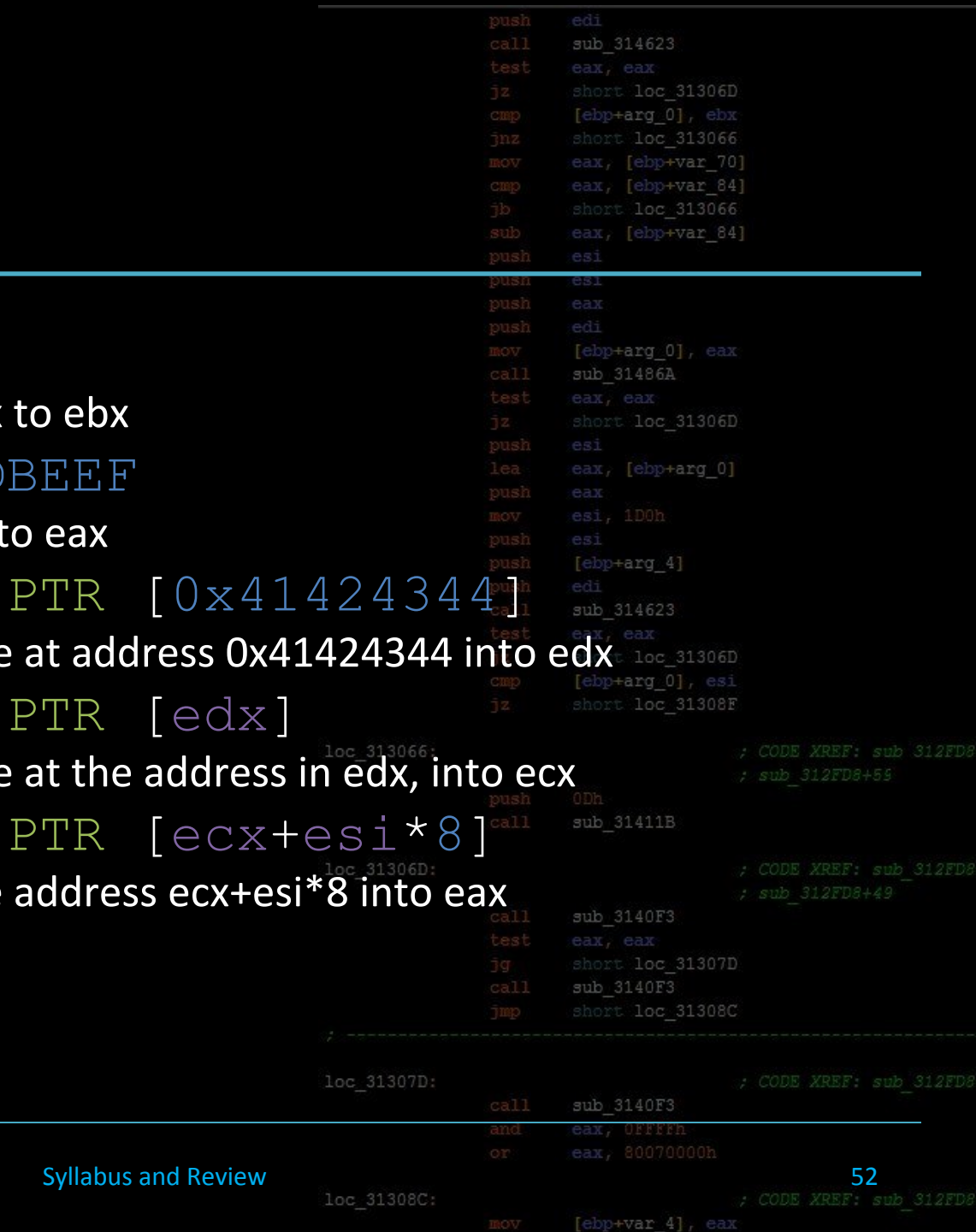
Important Registers

- EAX EBX ECX EDX - General purpose registers
- ESP - Stack pointer, “top” of the current stack frame (lower memory)
- EBP - Base pointer, “bottom” of the current stack frame (higher memory)
- EIP - Instruction pointer, pointer to the *next* instruction to be executed by the CPU

- EFLAGS - stores flag bits
  - ZF - zero flag, set when result of an operation equals zero
  - CF - carry flag, set when the result of an operation is too large/small
  - SF - sign flag, set when the result of an operation is negative

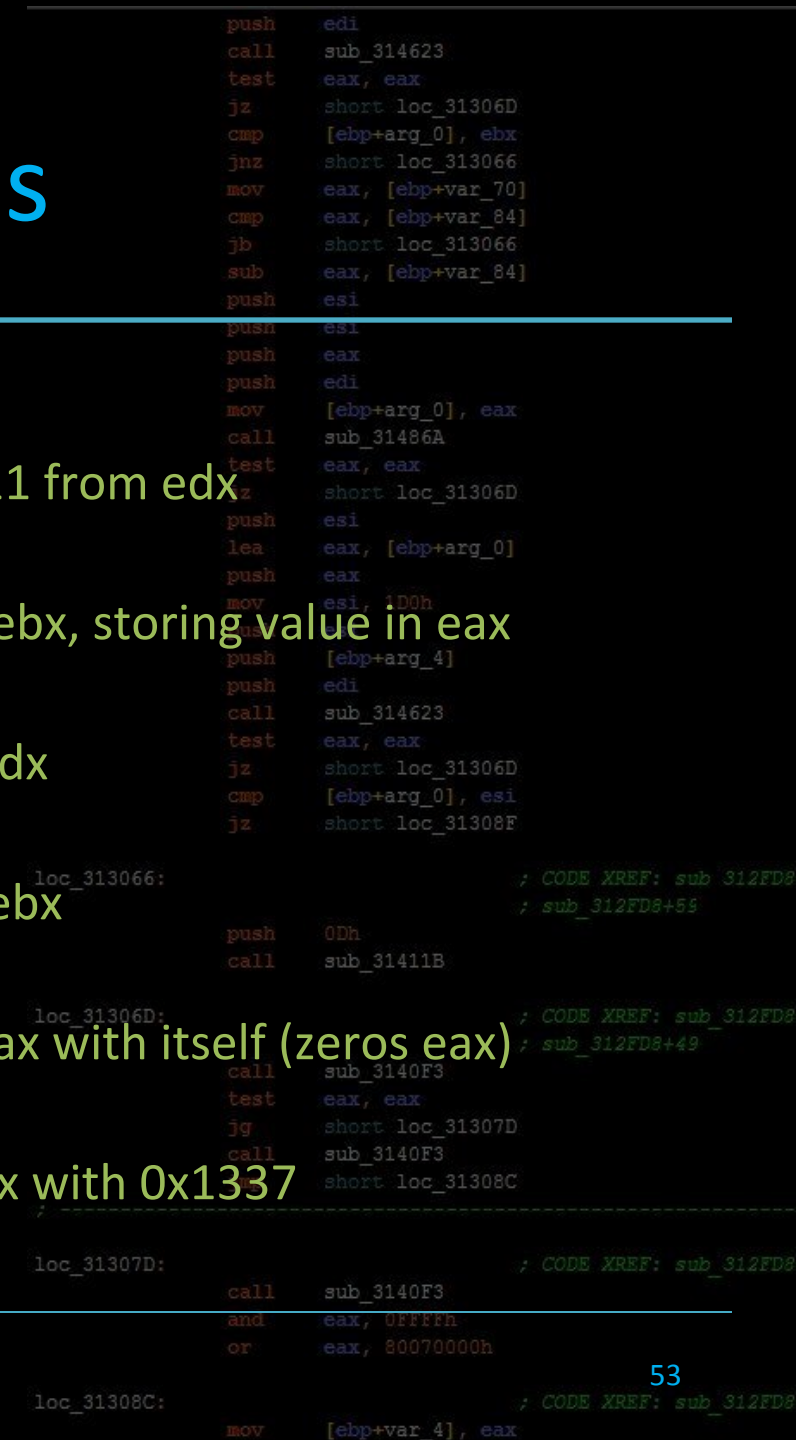
Moving Data

- `mov ebx, eax`
  - Move the value in eax to ebx
- `mov eax, 0xDEADBEEF`
  - Move 0xDEADBEEF into eax
- `mov edx, DWORD PTR [0x41424344]`
  - Move the 4-byte value at address 0x41424344 into edx
- `mov ecx, DWORD PTR [edx]`
  - Move the 4-byte value at the address in edx, into ecx
- `mov eax, DWORD PTR [ecx+esi*8]`
  - Move the value at the address ecx+esi*8 into eax



Arithmetic Operations

- `sub edx, 0x11`
  - edx = edx - 0x11; // subtracts 0x11 from edx
- `add eax, ebx`
  - eax = eax + ebx; // add eax and ebx, storing value in eax
- `inc edx`
  - edx++; // increments edx
- `dec ebx`
  - ebx--; // decrements ebx
- `xor eax, eax`
  - eax = eax ^ eax; // bitwise xor eax with itself (zeros eax)
- `or edx, 0x1337`
  - edx = edx | 0x1337; // bitwise or edx with 0x1337



Some Conditional Jumps

- `jz $LOC`
  - Jump to $LOC if ZF = 1
- `jnz $LOC`
  - Jump to $LOC if ZF = 0
- `jg $LOC`
  - Jump to $LOC if the result of a comparison is the destination is greater than the source

Stack Manipulation

- `push ebx`
  - Subtract 4 from the stack pointer to move it towards lower memory (zero,) and copy the value in EBX on top of the stack

```
sub esp, 4
mov DWORD PTR [esp], ebx
```

- `pop ebx`
  - Copy the value off the top of the stack and into EBX, the add 4 to the stack pointer to move it towards higher memory (0xFFFFFFFF)

```
mov ebx, DWORD PTR [esp]
add esp, 4
```

Calling / Returning

- `call some_function`
  - Calls the code at some_function. We need to push the return address onto the stack, then branch to some_function

```
push eip
mov  eip, some_function  ; not actually valid
```

- `ret`
  - Used to return from a function call. Pops the top of the stack to eip

```
pop eip      ; not actually valid
```

- `nop`
  - 'no operation' - does nothing

Basic x86

```
0x08048624: "YOLOSWAG\0"
    mov ebx, 0x08048624
    mov eax, 0
LOOPY:
    mov cl, BYTE PTR [ebx]
    cmp cl, 0
    jz  end
    inc eax
    inc ebx
    jmp LOOPY
end:
    ret
```

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

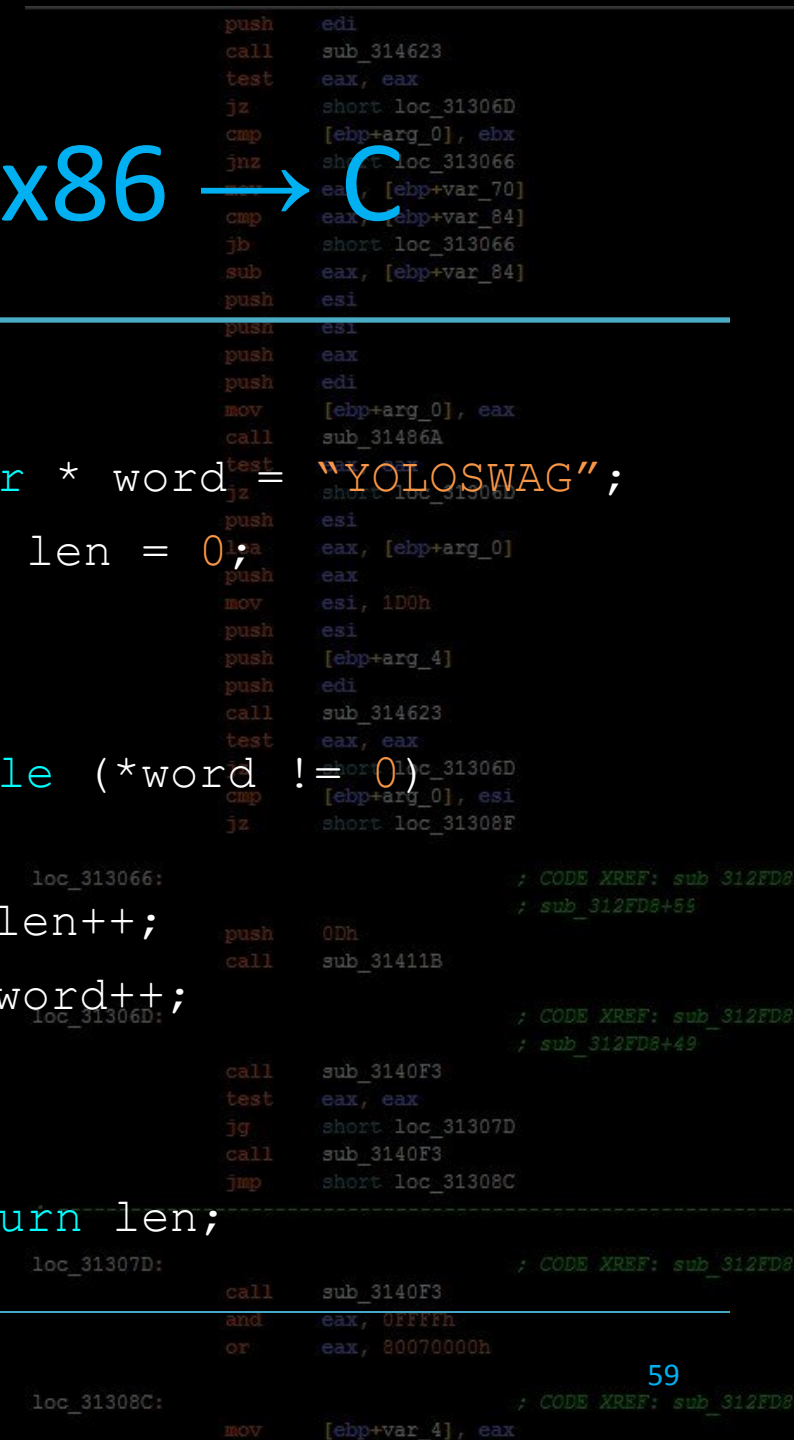
Basic x86

```
0x08048624: "YOLOSWAG\0"   ; 9 bytes of string data
    mov ebx, 0x08048624    ; char * ebx = "YOLOSWAG\0";
    mov eax, 0             ; set eax to 0
LOOPY:                     ; label, top of loop
    mov cl, BYTE PTR [ebx] ; char cl = *ebx;
    cmp cl, 0              ; is cl 0? (eg "\0")
    jz  end                ; if cl was 0, go to end
    inc eax                ; eax++; (counter for length)
    inc ebx                ; ebx++; ([ebx] = "Y", "O"... "\0")
    jmp LOOPY              ; go to LOOPY
end:                       ; label, end of loop/function
    ret                    ; return (len of str in eax)
```

Human Decompiler - x86 → C

```
0x08048624: "YOLOSWAG\0"
    mov ebx, 0x08048624
    mov eax, 0
LOOPY:
    mov cl, BYTE PTR [ebx]
    cmp cl, 0
    jz  end
    inc eax
    inc ebx
    jmp LOOPY
end:
    ret
```

```
...
char * word = "YOLOSWAG";
int len = 0;


while (*word != 0)
{
    len++;
    word++;
}

return len;
```



Additional Material

- Related Readings:
  - Hacking: The Art of Exploitation
    - Chapter 0x200: Programming - C programming and GDB
  - Practical Reverse Engineering (Dang et al)
    - Chapter 1 (x86)

- Get familiar with the linux command line if you aren't already
  - http://overthewire.org/wargames/bandit/