UMBC Cyber Defense Team First Meeting

and intro to reverse engineering!

Welcome to the UMBC Cyber Defense Club!

- Also called the UMBC Cyber Dawgs!
- A student run organization since 2009.
- We’re a group of students that share a passion for-and recognize the importance of-computer and network security

Some Introductions

- President: Bryan Vanek
  - Comp Sci & Math, Spring 2018
  - Firewalls, Secure Programming
- Vice President: Chris Gardner
  - Comp Sci & Math, Spring 2018
  - Offensive Security, RE
- Secretary: Kevin Bilzer
  - Computer Science, Spring 2020
  - Web Exploitation, Recon
- Treasurer: Christian Beam
  - Computer Science, Spring 2019
  - Computer Networking, SDN
- Historian: Alexander Spizler
  - Comp Sci Grad Student
  - History, Awesome Hype Man
- Technical Advisor: Zack Orndorff
  - Computer Science, Spring 2019
  - Linux Administration

What is Cyber Security?

- The protecting of computers, networks, programs and data from unintended or unauthorized access, change or destruction
- Basically keeping the bad guys out, while keeping everything up and running
- Several different types of cyber security
  - Application Security
  - Network Security
  - Information Security
  - Recovery/backup security
  - Offensive Security
  - And much, much more!!

How can the Cyber Dawgs help me learn Cyber Security?

- There are lots of ways to learn about cyber security…
- …but we believe in approaching it through multiple tracks.
- We have three main tracks that we follow throughout the year:
    - Education + Hands-on
    - Competition Participation
    - Industry Exposure

Track #1: Education and Hands-On

- Our regular club meetings: every Wednesday; 7-9 in ITE227 (maybe PUP 105 in the future)
- We go over lots of relevant and important topics pertaining to cyber security
- We have a schedule for this semester!
  - Check it out on our website: http://umbccd.umbc.edu/
- We'll mainly be covering Reverse Engineering and Binary Exploitation this semester, but we will also cover:
  - Metasploit
  - Cryptography

Track #2: Competition Participation

- This is what the club was founded on
- We have bunch of competitions for you to compete in this semester!
  - Our CTF - DawgCTF - March 11th in the UC Ballroom!
  - UMDCTF - April 14th at UMD
  - Some people are doing CCDC
  - MITRE CTF - April 20th
  - DEFCON CTF Qualifiers! (at some point…)
  - Other various online CTFs
- You should all try and compete in a competition - it’s really fun and a great way to learn.

Track #3: Industry Exposure

- The fields of IT, CS, IS, and cyber security is ever changing
- It’s important to know what industry is using today!
- We’ll have some industry folks come in and give talks, demos, and labs!
- Dates + Times will be released as they become available!

How to stay in contact with us

- Remember: our regular club meetings are Wednesdays from 7pm-9pm in ITE227.
- Check out our website to see our schedule, resources page, and link up with the group at http://umbccd.umbc.edu/
- Subscribe to our mailing list: send an email to umbccd-group+subscribe@umbc.edu to join (umbc email only)
- Get on our slack channel! https://umbccd.slack.com
    - This is really the best way to communicate with us, so join this ASAP!

Important Legal Disclaimer!

- While we are a cyber defense team, the tools and techniques we will be going over are used to break into a system/network
- So unless it is YOUR server, computer, virtual machine, or service, or unless you have EXPLICIT and COMPLETE permission to do so…
- DO NOT USE ANY OF THESE TOOLS OR TECHNIQUES ON A SYSTEM THAT YOU DON'T HAVE COMPLETE AUTHORIZATION TO DO SO
- If you don't adhere to this, some of the legal ramifications include-but are not limited to-the following:
  - Expulsion from UMBC
  - Being charged for either a misdemeanor or a felony, depending on the severity
    - Fines up to $100,000 for misdemeanors, up to $250,000 for felonies
    - Jail time up to 1 year for misdemeanors, up to 10 years for felonies
  - AND THIS IS JUST THE FEDERAL LEVEL.

Questions about the club?

Intro to Reverse Engineering

With Chris!

WTF IS RE

Reverse engineering

verb (used with object)

to study or analyze (a device, as a microchip for computers) in order to learn details of design, construction, and operation, perhaps to produce a copy or an improved version.

In this class, we will be dealing with compiled programs that were written in C, and we will be trying to find some information hidden in the program (a flag)

We'll use several different names for programs in this course - binaries, assemblies, services, etc

What can we use RE for?

There are a lot of uses for reverse engineering:

- Vulnerability Research (focus of this course)
- Malware analysis
- Structural reverse engineering
  - For application interoperability
- Game hacking/mod creation
- Capture The Flag (CTF) competitions
- DRM cracking
- Pure fun
  - RE is the best puzzle game ever created

What you’re gonna learn this semester

You’ll learn to:

- Reverse engineer Linux binaries that have been written in C (and maybe C++)
- Use static analysis tools such as IDA Pro and Binary Ninja
- Find bugs in both x86 and x86_64 Linux binaries
- Write exploits for those bugs
- Learn a bit about Metasploit

Why the focus on Linux/CTFs?

We focus on Linux because Windows programs are typically harder to reverse engineer (although better tooling exists for Windows) and are more annoying to exploit

We focus on CTFs because I don’t want to be responsible for infecting your machines with malware, and ‘real’ vulnerability research takes too much time for a 2 hour lecture/lab

I know a negative amount about reversing/exploiting OSX stuff, so we won’t cover that

Tools of the trade

Disassemblers - Convert the machine code of a program to (somewhat) human readable assembly code

Debuggers - Run a program, get information about its state at a given instruction

Other static analysis tools - provide information about the compiled program, edit the binary, convert the binary to another format, etc

A web browser and a connection to Google - arguably as important as a disassembler

Emulators, constraint solvers, symbolic execution - Advanced tools that let you cheat at RE

RE == reading assembly??

You might think that Reverse Engineering is reading all the assembly in a binary, and then figuring out what is hidden.

This is wrong.

If you do this, you will never RE anything non-trivial in a reasonable amount of time.

RE is not about reading assembly code.

RE is the art of NOT reading assembly.

RE != reading assembly

~90% of the assembly in a given binary is irrelevant to our task

So our job is to figure out what parts of the binary are relevant, then read them.

We want to reduce the amount of assembly we need to look at (reduction of work). I'll tell you how to do this efficiently

Once you've found the assembly, reading it is easy and you can figure out what you need to do to get the flag

Myths about RE

- Myth: Reverse Engineering is hard
  - Wrong. Math is hard, crypto is hard, exploitation is hard. RE is just tedious.
- Myth: You need to know assembly really well to be able to RE
  - I can’t write assembly for the life of me, but I can read it despite forgetting what half the instructions do every time I open a disassembler
  - Reading assembly is usually pretty straightforward
- Myth: You need $1000s worth of software to RE effectively
  - This used to be true, but not anymore. Binary Ninja is pretty affordable, and radare2 is free (but it sucks). IDA Pro has a free version.

A quick x86 primer

- Instructions are of the form “opcode destination, source”
  - mov eax, ebx
  - add ebx, ecx
- Numbers are usually expressed in hexadecimal, in these two forms: 0x1A or 1Ah
- There are registers: eax, ebx, ecx, edx, esi, edi, esp, ebp, some others….
  - They are all 32 bit, however the word size is 16 bits (2 bytes). Each register is a doubleword
  - The first four registers can be referenced as 16 bit and 8 bit registers:
    - ax = last 16 bits of eax, al = last 8 bits of eax, ah = first 8 bits of ax (not often used)
- [something] is a pointer dereference treating something as a pointer
  - *some_ptr in C
  - You can put expressions in the something
- “lea eax, [some_expr]” computes the result of some_expr and stores it in eax
  - It doesn’t modify memory at all

A quick x86 primer

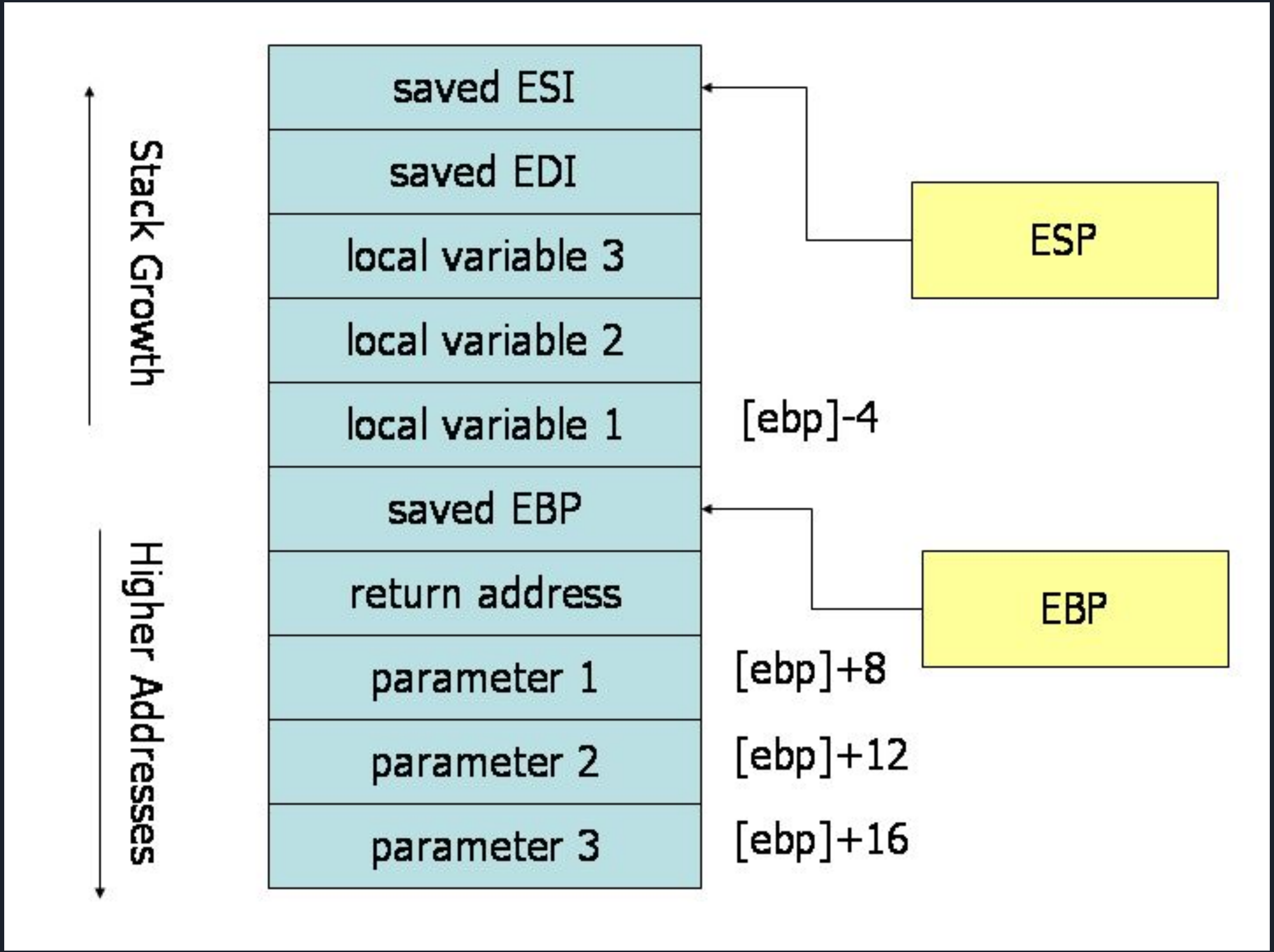
- Conditionals start with a cmp instruction, then a conditional jump
  - cmp subtracts the two operands and updates flags, doesn’t actually update the registers
  - test works similarly, but ANDs the operands together
- The conditional jumps are simple acronyms:
  - je = Jump Equal
  - jne = Jump Not Equal
  - jge = Jump Greater Than or Equal
  - jz = Jump Zero (same as je)

A quick x86 primer: The Stack

- There is a stack, which is modified with the push and pop instructions
  - Unlike conventional stacks, the stack grows downwards. So push eax decrements the stack pointer (esp)
- Local variables are stored on the stack and are accessed like this: [ebp-some_offset]
  - Or like this: [esp+some_offset-some_other_offset]
  - Function arguments are accessed like [ebp+some_offset] (note the addition instead of subtraction)
- The call instruction pushes the current address onto the stack, then jumps to the operand
  - The ret instruction pops an address off the stack and jumps to it

A quick x86 primer: The Stack

- Each time a function is called it gets its own stack frame: ebp points to the bottom of the stack frame, esp points to the top of the stack frame
  - Note that esp < ebp, so the ‘bottom’ of the stack frame is actually at a higher address than the top
  - We use this terminology because the stack grows downwards
- Functions store their local variables and control flow data in their stack frame. Function arguments are stored just outside the stack frame (in the previous functions stack frame)
  - See diagram


Stack Growth
Higher Addresses
saved ESI
saved EDI
local variable 3
local variable 2
local variable 1
saved EBP
return address
parameter 1
parameter 2
parameter 3
ESP
EBP
[ebp]-4
[ebp]+8
[ebp]+12
[ebp]+16

A quick x86 primer: Calling conventions

- Function calls look like this:
- The return value is placed in eax
- Equivalent to function(arg0, arg1…) in C

```
push arg3
push arg2
push arg1
push arg0
call function
```

```
mov [esp+8], arg2
mov [esp+4], arg1
mov [esp], arg0
call function
```

- Most assembly instructions are straightforward, you can Google them if you don't understand what an instruction does
- No one knows what "repne scasb" does off to the top of their head, they just google it every time.
- 99% of the instructions that directly modify esp or ebp aren't worth looking at

Vocab lesson: Static Analysis

Static analysis involves using some tools to discover things about the binary, without actually running it.

Basic disassemblers just dump the assembly code into a text file, but the ones used in reverse engineering do some more analysis and allow you to browse the disassembly visually. They do this by splitting up code into “basic blocks”

Vocab Lesson: Dynamic Analysis

Dynamic analysis is when you run a program in a debugger (or similar tool) and get information about the program when it runs (such as contents of registers or memory)

This generally lets you save a lot of time when reverse engineering, as instead of writing a script to decrypt something you can let the binary decrypt it for you

We'll talk about dynamic analysis more in depth in a few weeks

Other vocab

So what’s in an ELF, anyway?

Elves? I thought we were talking about binaries....

ELF = Executable and Linkable Format - the standard executable file for Linux

Like an EXE, but for Linux

Has a bunch of different sections and segments, which contain different parts of the program

It’s got a header that describes everything

Sections and Segments

Sections contain all the stuff that comes with the program - the assembly code, the default data, the program header etc

Segments describe all of the memory that will be mapped to the binary - this includes all the sections, and stuff like the stack and the heap

Some common sections:

.text - All the executable code in a binary

.data - Global variables

.bss - Global variables that are set to 0 at program start

.got/.plt - Relocation data for the linker

Link her? I hardly know her!

The linker (ld.so) loads the segments and sections from the ELF file into memory, sets up relocations, and starts the file

Two kinds of binaries:

Statically linked - no relocations, all code executed is contained in the ELF

Dynamically linked - calls code from other libraries (this is what most programs are)

Demo/Lab!

Static Analysis tools

There are a lot of cool disassemblers out there that do a lot of work for you, like Binary Ninja and IDA Pro

We won’t be using them today

Instead we are going to statically analyze a binary using two of the most basic tools in the book: readelf and the venerable objdump

We’ll write and compile a test program together, then reverse engineer it and explain how the assembly maps to the source code

(if you want to follow along: apt-get install binutils gcc-multilib)