C, x86 Assembly, and the Call Stack

January 23rd, 2019

Today's Goals

- What is the exploitation meeting?
- C, assembly, and where to learn them
- What topics will we cover next?

Announcements

- CTFs!
  - Fireshell
  - Codegate
- Engineering Expo

Reading Material

- Hacking: The Art of Exploitation
  - Available through USF Library online!
  - Live CD available on Starch Press website
  - Read the whole dang thing… but Section 0x200 for what's covered today
- Phrack Magazine: Smashing the Stack for Fun and Profit
  - Intro up to Buffer Overflows (next week ;))
- C Function Call Conventions and the Stack
- A description of the lea instruction
- Yale's x86 Assembly Guide

Exploitation Meeting: What is it?

- Exploitation meetings cover a wide range of computer security topics, focusing on examining the vulnerabilities that attackers use to exploit systems
- What's different this year?
  - Less CTF, deeper coverage, more topics
    - This club was built to play CTFs… Why Less?
    - Not enough hands-on experience can happen in a meeting time
    - Saturday CTF meetings! We can play then
  - So, does this mean no demos?
    - Demos will still be an important part of meetings, but not the center

- Papers, citations, and readings
  - You aren't expected to read them, but I hope you do
  - The important sections will be narrowed down
  - The majority, if not all, will be freely obtainable through USF resources
  - As a graduate student, my hope is to get you all interested in security research

C, Assembly, and the Stack

C - Better than B

- Low level, wide spread programming language. Many drivers, operating systems, and libraries are implemented using this language, and there are many exploits that exist in C that we will examine.
- We won't cover programming C in any more detail, but here is a sample program

```c
#include <stdio.h>

void printIt(int n)
{
    int i;
    for(i=0; i < n; i++) // Loop n times.
    {
       puts("Hello, exploitation meeting!\n"); // Print a string
    }
}

int main()
{
    printIt(10);
    return 0;
}
```

- Not familiar with C? See Hacking: The Art of Exploitation, Section 0x200

Assembly

A human readable language that is even lower than C. Assembly code can be directly assembled to machine code, and machine code can be disassembled to assembly. We will be looking at the intel syntax for x86 assembly.

1. Compile hello.c (an abbreviated version of the program above) with -m32 flag
2. Run the following:

```
objdump -M intel -D a.out | grep main -A10
```

3. Let's take a look at the printIt function in assembly

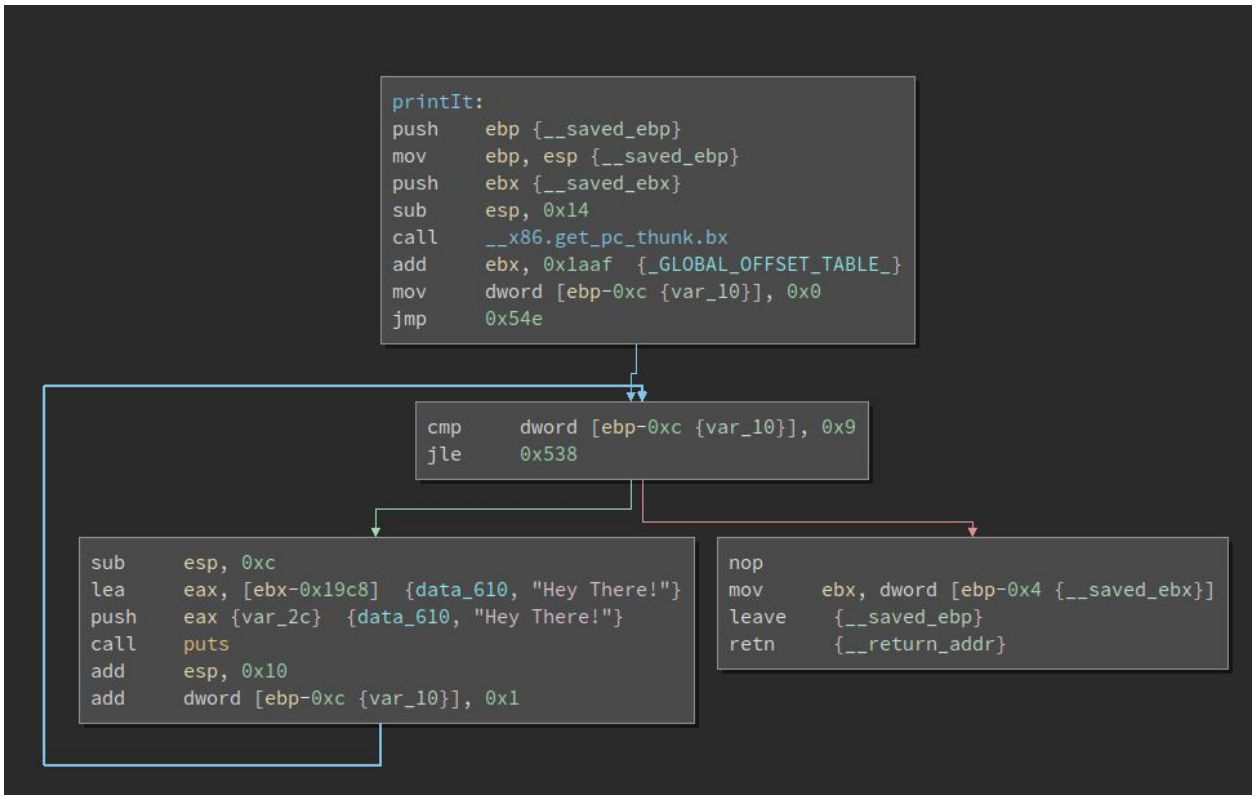
```
0000051d <printIt>:
 51d: 55                    push   ebp
 51e: 89 e5                 mov    ebp,esp
 520: 53                    push   ebx
 521: 83 ec 14              sub    esp,0x14
 524: e8 f7 fe ff ff        call   420 <__x86.get_pc_thunk.bx>
 529: 81 c3 af 1a 00 00     add    ebx,0x1aaf
 52f: c7 45 f4 00 00 00 00  mov    DWORD PTR [ebp-0xc],0x0
 536: eb 16                 jmp    54e <printIt+0x31>
 538: 83 ec 0c              sub    esp,0xc
 53b: 8d 83 38 e6 ff ff     lea    eax,[ebx-0x19c8]
 541: 50                    push   eax
 542: e8 69 fe ff ff        call   3b0 <puts@plt>
 547: 83 c4 10              add    esp,0x10
 54a: 83 45 f4 01           add    DWORD PTR [ebp-0xc],0x1
 54e: 83 7d f4 09           cmp    DWORD PTR [ebp-0xc],0x9
 552: 7e e4                 jle    538 <printIt+0x1b>
 554: 90                    nop
 555: 8b 5d fc              mov    ebx,DWORD PTR [ebp-0x4]
 558: c9                    leave
 559: c3                    ret

0000055a <main>:
 55a: 8d 4c 24 04           lea    ecx,[esp+0x4]
 55e: 83 e4 f0              and    esp,0xfffffff0
--
 575: e8 a3 ff ff ff        call   51d <printIt>
 57a: b8 00 00 00 00        mov    eax,0x0
 57f: 83 c4 04              add    esp,0x4
 582: 59                    pop    ecx
 583: 5d                    pop    ebp
 584: 8d 61 fc              lea    esp,[ecx-0x4]
 587: c3                    ret
```

Our C code in assembly

Let's point out things we recognize:

1. Call puts! That is recognizable
2. I see add [something], 1. We added 1 in our for loop!
3. Uh.... What else?

Now let's take a look at in Binary Ninja, a graphical disassembler with a control flow graph. In Binja, blue lines represent unconditional jumps, green lines are taken branches, and red lines are untaken branches.



Binary Ninja Disassembly for our program

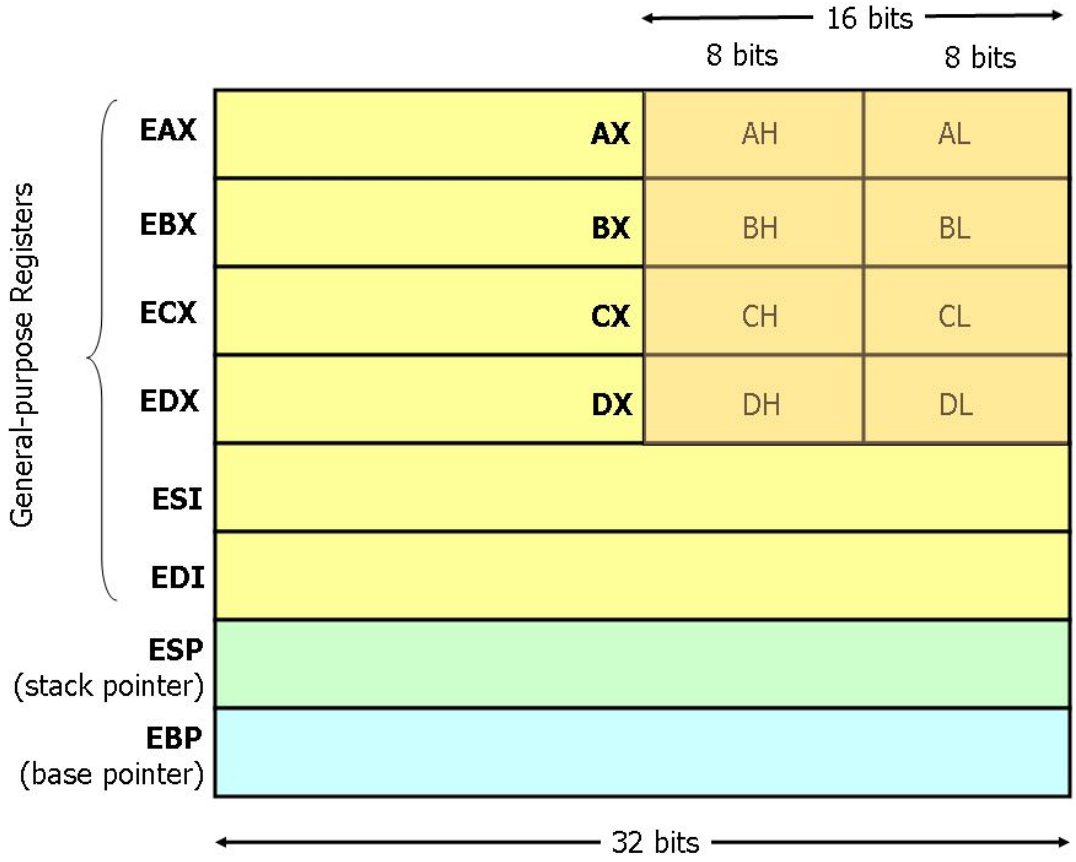
I feel better looking at this… Let's see what we recognize now

1. Our "puts" call
2. Our string… why does it push eax? What is in eax?
3. There is a loop where we print our string
4. The cmp operator must be part of our for loop
    a. We set it to loop 10 times though.... Why 0x9?
    b. The compiler is too smart for us, and knows we ONLY called it with an argument of 10

Neat! We now understand our code by looking at the assembly. At least, we understand it's high-level function… Let's begin looking at the rest of the assembly and break it down line by line.

Registers in x86

For our purposes, you can think of registers as variables in assembly. Registers are special memory that store values in the CPU. x86 offers a wide range of registers, some with special, conventional meaning. Instructions will typically act on registers, changing or using the values contained in them. The most important registers are listed below in an image from the x86 Assembly Guide from Yale. Not shown are EIP (the instruction pointer or program counter) which points to the next instruction to be executed, and the Flags register, which contains important state information for arithmetic operations.



Some Assembly Instructions and What They Do

The following table gives a brief intro to all of the instructions seen in our assembly. This does not include all possible instructions or all the possible formats for these instructions, but these are, in my opinion, some of the most important and most commonly seen instructions.

Instruction	**Example**	**Purpose**
push {register|value}	push ebp	Pushes a value onto the stack, and decrements esp (stack grows down)
Pop {register}	Pop esp	Pops a value from the stack into a register, and increments esp (stack shrinks up)
mov {Destination} {Source|value}	mov ebp, esp	Moves a value from source to destination
add/sub {Destination} {Source|value}	sub esp, 0x14 add ebx, eax	Destination = Destination + Source
call {addr}	call puts	Calls the function at a given address
jmp {addr}	jmp 0x54e	Jumps to a given address
cmp {register|address} {register|value}	cmp eax, 0x9	Compares two values and sets the flags (EFL) register based on the value
jle {addr}	jle 0x538	Jumps to addr if flags say A <= B (Carry and Zero flags)
The [] operator	mov [eax+0x4], 0x9	Not an instruction, but important. Similar to a pointer dereference in C. This means add eax+0x4, then find the value pointed at by that value. In other words, *(eax+0x4)
lea	lea eax, [ecx +0x4]	An important instruction, this gives the address of a value. Somewhat similar to &p in C. In this case, eax = ecx+0x4, which is not dereferenced. See reading material
nop (0x90)	nop	Literally nothing - No operation
leave	leave	Set ESP to EBP, then pop EBP. This is the function prologue. We will cover this later.
retn	retn	Pop eip. Identical in purpose to a return in C. Returned values are in eax by convention

The Stack Data Structure

The stack is a common data structure seen in programming. The stack is a first-in last-out (or last-in first-out) structure, like a stack of plates.

Stack		Pop		Push F	
Top	D	Top	C	Top	F
	C		B		C
	B	Bottom	A		B
Bottom	A			Bottom	A

The x86 Call Stack

The x86 call stack is a normal stack, but values inside of the stack have special meaning. Values can be popped and pushed, changing the size of the stack, but instructions can access the values inside the stack and update them by referencing the bottom of the stack. This might seem strange, but it quickly becomes intuitive. The next page includes a diagram of the stack. Instead of thinking of the x86 stack as a stack with values, think of it as a stack of variables!

See the reading material for more information on the stack (all of the materials above mention the stack). The more you introduce yourself to the stack, the more familiar it becomes.

There are two special registers that monitor the stack:

EBP	Typically called the Base Pointer, since it is the base of the current function's stack
ESP	The Stack Pointer, ESP points to the top of the stack

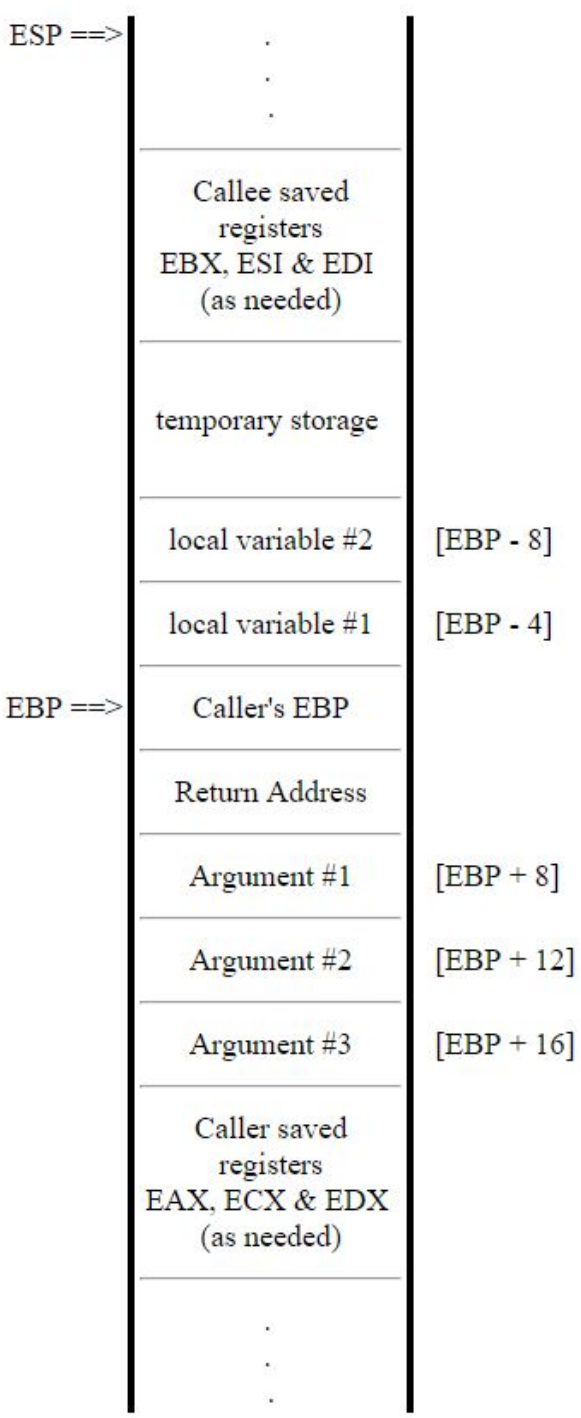


Fig. 1

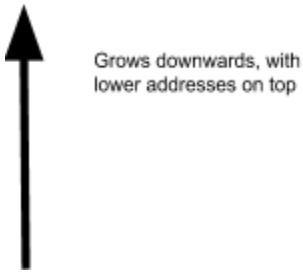
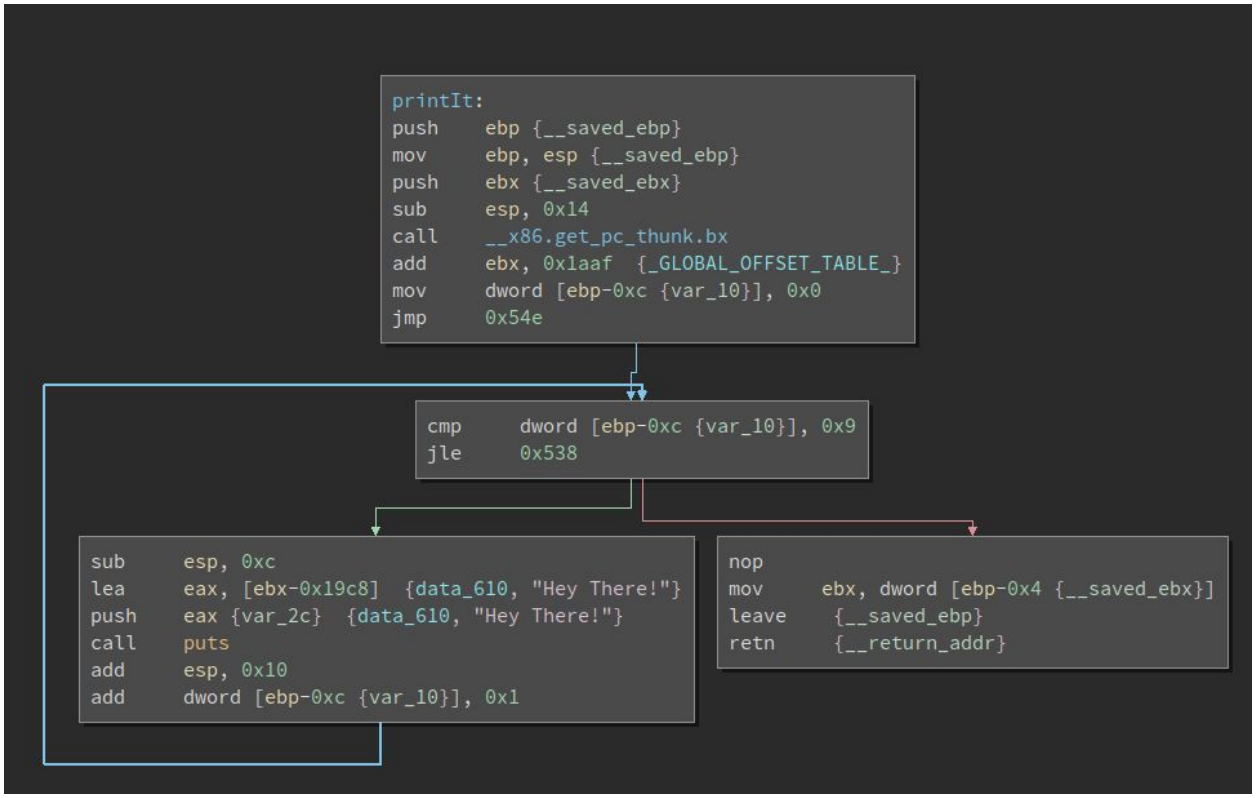


Figure taken from C Function Call Conventions and the Stack (See Reading Material)

Back to Our Assembly…

Now that we know all of our instructions and know what the stack is, let's take a look at our code again.



1. First, let's set up our stack: See "The caller's actions before the function call" at https://www.csee.umbc.edu/~chang/cs313.s02/stack.shtml
2. Ok, so we push ebp, then move esp into ebp. See "The callee's actions after function call" part 1 and 2, respectively.
3. Awesome, out stack is setup! Next we call __x86.get_pc_thunk.bx
    a. Don't worry about this. It simply moves the current value of the program counter into ebx as part of C's Position Independent Execution (PIE).
4. Next, we add to ebx.
    a. Again, part of C's PIE. Don't worry about this either. We will cover the Global Offset Table in later detail.
5. Our first real tricky instruction. This mov moves 0 into the address at ebp-0xc, or 12 bytes up the stack.
    a. That section of the stack is local variables. This must be our i=0 instruction.
6. We jmp into the guard in our loop (the if comparison)
7. We compare the variable i with 0x9
8. We jump to our printing, since i < 0x9

9. Then we subtract 0xc from esp
    a. This is likely space for alignment. Here are some people smarter than I addressing this issue, but who knows why the compiler does what it does.
    b. Note that the next instruction, push, brings the size to 0x10; a perfect 16 byte alignment.
10. We move the address of our string into eax
11. We push eax onto the stack as an argument for puts
    a. Eax is the address of our string
12. We call puts
13. We add 0x10 to our stack
    a. We are undoing our arguments or whatever it actually is from our call
    b. Note we subtract 4 more bytes, since we also pushed the address of our string
14. Then we add 1 to i
15. …
16. Eventually (9 iterations, actually) we get to where we don't jump into our print block
17. We perform a nop
    a. Not sure why, but this is likely done for branch prediction reasons
18. We move our saved ebx back into ebx
19. Then we leave
    a. ESP=EBP
    b. Pop EBP
20. Retn
    a. Jump to value pointed at by ESP by pop EIP

Next topics?

Suggested topics from Members

1. Wi-Fi Vulnerabilities
2. BlueTooth Vulnerabilities
3. QRcode Vulnerabilities

My topics

1. Buffer overflows, stack guard, and other mechanisms
    a. Buffer overflows
    b. Stack Guard (the Canary)
    c. ASLR (Address Space Layout Randomization)
    d. Breaking these mechanisms
2. Format Strings and FormatGuard
3. Code-reuse Attacks
    a. ROP - Return-Oriented Programming
    b. RILC - Return Into Lib C
4. SQL Injection and other injection attacks

5. Anything on this top 25 list
6. SetUID, linux permissions, and other linux security features
    a. Not really an exploit per say… but important
7. Rootkits
8. Tools, tools, tools!
    a. Pwntools
    b. Angr
9. Cybersecurity in Public Transportation and other cyberphysical systems
    a. Again, not typical, but any security topics are game! Plus, I'm biased on this topic
    b. Jamming Autonomous Vehicles and the Cherokee jeep hack and Stuxnet, oh my!