Software Exploitation Basics

Mike Ham

You’re here to learn software exploitation. What do YOU think it is, and what can successful exploitation result in?

Something More Relatable

- Halo 2 was a game that I played probably too much of
- There were many flaws in the software that allowed players to do some cool stuff, useful or not
- One particular flaw was called the “Super Bounce”
  - https://www.youtube.com/watch?v=6gyZ9Hfd5IE
- The flaw in the software was taken advantage of for some gain…exploits

Software Exploitation

- It takes on many forms and meanings so there isn't a *great* definition
- Taking advantage of a vulnerability in some software to get the system to execute code you provide
- Vulnerabilities are flaws or weaknesses in a system that can be *exploited*
  - Bad programmer, functions in software were designed poorly, etc.
- We exploit software for many purposes, it's up to you what the outcome should be!

Precursor

- If you're going to exploit something, you REALLY need to understand how it works or is designed to work in the first place
- Computers are super complex, different languages (C, Java, etc.) respond and operate differently, processors interpret commands differently, etc.
- Windows and other OS provide some security mechanisms to protect vulnerable programs from being exploited (DEP, ASLR, EMET)
- High value software is pretty big, may have millions of lines of code, well protected
- The list goes on….

Don’t panic!

Why write exploits when I can use tool X or Y?

Exploitation Tools/Frameworks

- There are tons of these out there, Metasploit is a really common one
  - Armitage/Cobalt Strike, CORE Impact, Nexpose
  - There absolutely is a time/place for using them, I'd actually encourage it
  - Someone had to write the exploits in those tools
- Anyone can fire up a tool and click "GO" - skiddies
  - If you want to make $$$ and be 1337, learn to write some on your own or at least understand what you're running

Machine & Compiler Setup

- If you are running a x64 Kali install, it will be easier to look at 32-bit programs
- Install the following software:
  - apt-get install libc6-dev-i386 -y
  - apt-get install hexedit -y
- Address Space Layout Randomization (ASLR), disable in terminal temporarily:
  - setarch `uname -m` -R /bin/bash

Software, Processors, RAM, and HDD

Software

- We write software in a high level language like C, C++, C#, Java, etc.
  - It's (pretty) easy for us to understand and read, but yet still efficient
  - Called **source code**
- Computers operate in **binary**, 1's and 0's to represent ALL information
  - They can't simply run a C program as is without compiling it
- Compiling takes the language we write in and translates it to a set of instructions that the processor can understand

C —> Binary

- When you compile a C program, it gets turned into processor instructions
- Processors read the instructions in binary
- Obviously, binary is really tough to read
  - Use xxd -b <file> to read in binary

hello_world.c

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

int main(void)
{
    printf("Hello, World!");
    exit(0);
}
```

```
0000f4e: 00000000 00000000 01010101 01001000 10001001 11100101  ..UH..
0000f54: 01001000 10000011 11101100 00010000 01001000 10001101  H...H.
0000f5a: 00111101 01001001 00000000 00000000 00000000 11000111  =I....
0000f60: 01000101 11111100 00000000 00000000 00000000 00000000  E.....
0000f66: 10110000 00000000 11101000 00010001 00000000 00000000  ......
0000f6c: 00000000 00110001 11111111 10001001 01000101 11111000  .1..E.
0000f72: 11101000 00000001 00000000 00000000 00000000 10010000  ......
0000f78: 11111111 00100101 10010010 00000000 00000000 00000000  .%....
0000f7e: 11111111 00100101 10010100 00000000 00000000 00000000  .%....
0000f84: 01001100 10001101 00011101 01111101 00000000 00000000  L..}..
0000f8a: 00000000 01000001 01010011 11111111 00100101 01101101  .AS.%m
0000f90: 00000000 00000000 00000000 10010000 01101000 00000000  ....h.
0000f96: 00000000 00000000 00000000 11101001 11100110 11111111  ......
0000f9c: 11111111 11111111 01101000 00001100 00000000 00000000  ..h...
0000fa2: 00000000 11101001 11011100 11111111 11111111 11111111  ......
0000fa8: 01001000 01100101 01101100 01101100 01101111 00101100  Hello,
0000fae: 00100000 01010111 01101111 01110010 01101100 01100100   World
0000fb4: 00100001 00000000 00000000 00000000 00000001 00000000  !.....
0000fba: 00000000 00000000 00011100 00000000 00000000 00000000  ......
0000fc0: 00000000 00000000 00000000 00000000 00011100 00000000  ......
0000fc6: 00000000 00000000 00000000 00000000 00000000 00000000  ......
0000fcc: 00011100 00000000 00000000 00000000 00000010 00000000  ......
0000fd2: 00000000 00000000 01010000 00001111 00000000 00000000  ..P...
0000fd8: 00110100 00000000 00000000 00000000 00110100 00000000  4...4.
0000fde: 00000000 00000000 01111000 00001111 00000000 00000000  ..x...
0000fe4: 00000000 00000000 00000000 00000000 00110100 00000000  ....4.
0000fea: 00000000 00000000 00000011 00000000 00000000 00000000  ......
```

Assembly

- We don’t always have source code available, and we can’t read binary
- There is a language in-between called **assembly**
  - It’s not super easy to read, but we can see the processor instructions and work from those - takes practice!
- There is a tool called **objdump** to view assembly in Kali

```
0804842b <main>:
 804842b:       8d 4c 24 04             lea    0x4(%esp),%ecx
 804842f:       83 e4 f0                and    $0xfffffff0,%esp
 8048432:       ff 71 fc                pushl  -0x4(%ecx)
 8048435:       55                      push   %ebp
 8048436:       89 e5                   mov    %esp,%ebp
 8048438:       51                      push   %ecx
 8048439:       83 ec 04                sub    $0x4,%esp
 804843c:       83 ec 0c                sub    $0xc,%esp
 804843f:       68 f0 84 04 08          push   $0x80484f0
 8048444:       e8 a7 fe ff ff          call   80482f0 <printf@plt>
 8048449:       83 c4 10                add    $0x10,%esp
 804844c:       83 ec 0c                sub    $0xc,%esp
 804844f:       6a 00                   push   $0x0
 8048451:       e8 ba fe ff ff          call   8048310 <exit@plt>
 8048456:       66 90                   xchg   %ax,%ax
 8048458:       66 90                   xchg   %ax,%ax
 804845a:       66 90                   xchg   %ax,%ax
 804845c:       66 90                   xchg   %ax,%ax
 804845e:       66 90                   xchg   %ax,%ax
```

How Software is Loaded

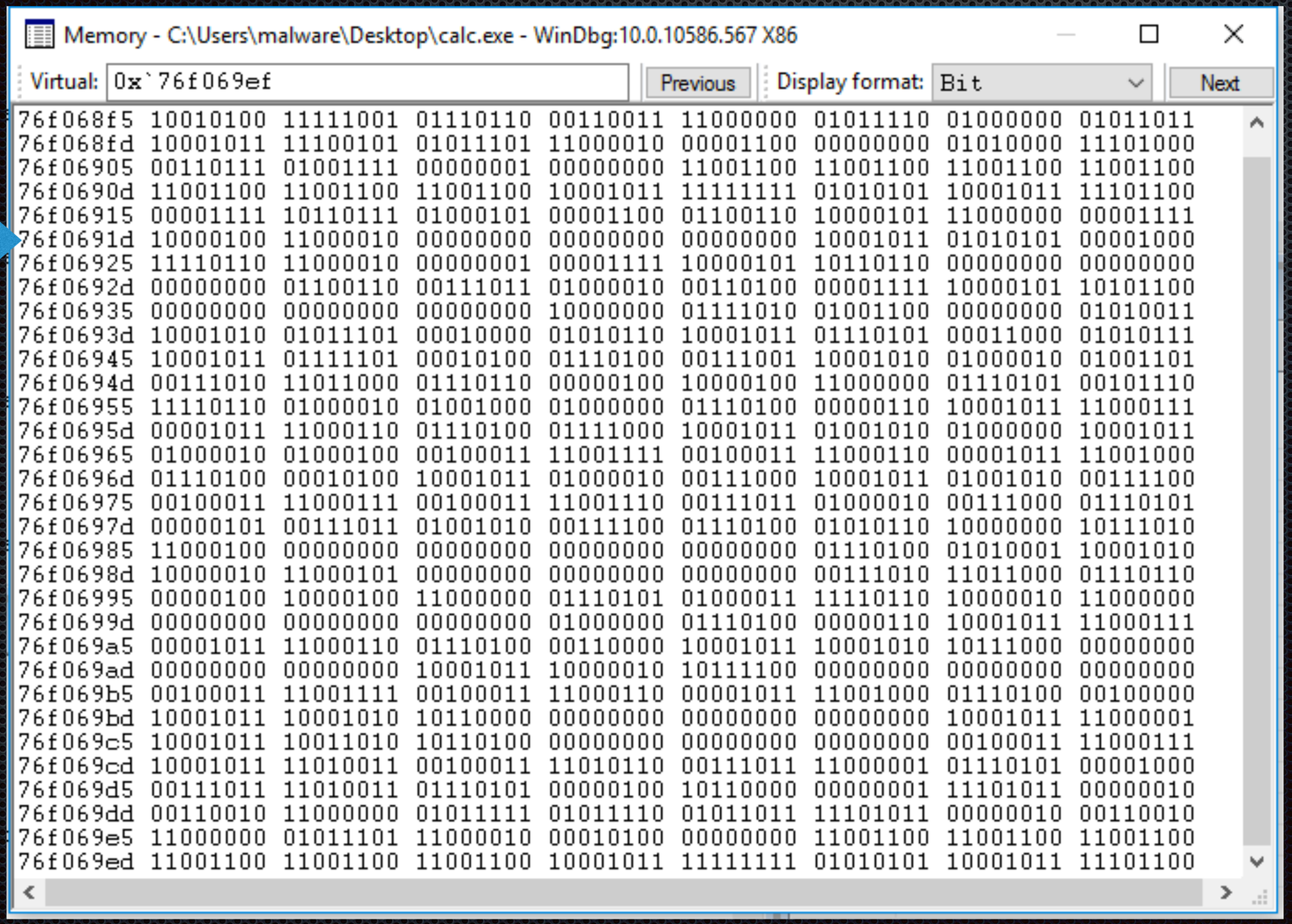
- Software is stored on your HDD in binary and exists permanently (Word, Steam)
- When you want to run software, a copy of it gets moved from the HDD over into your computers RAM
  - Why?
- The processor then can read the instructions within the software and run the program
- When you close the software, RAM is cleared out and freed for something else

RAM/Memory

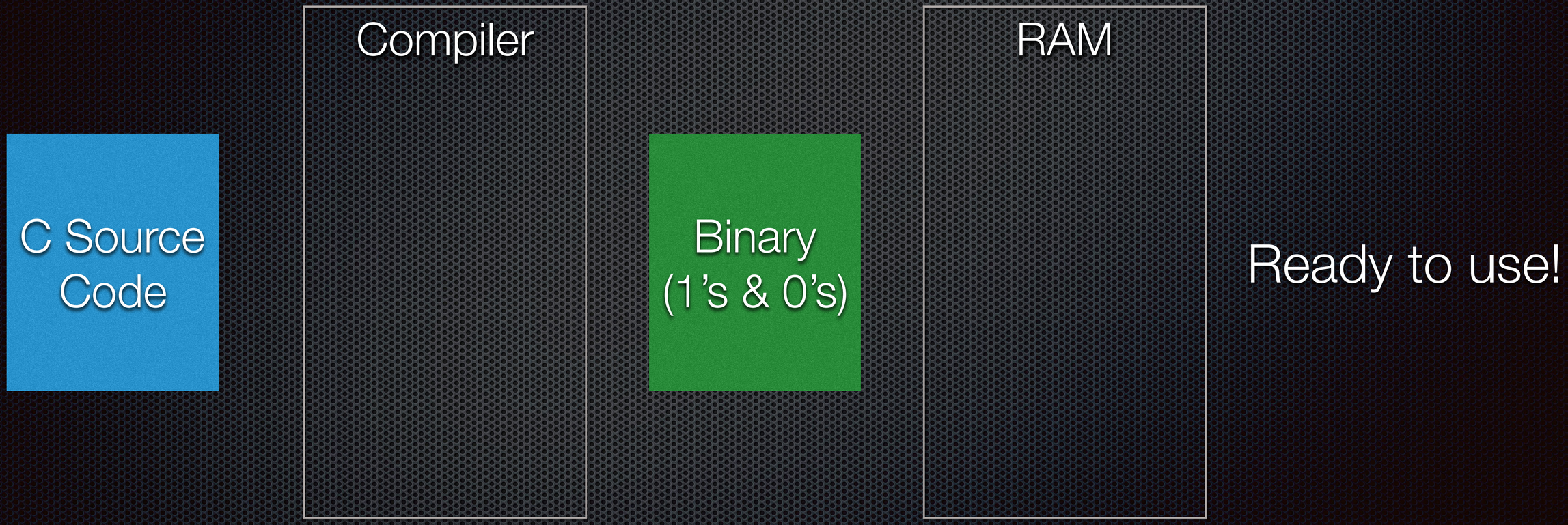
- Memory doesn’t just store programs
  - It also is capable of storing data (numbers, words, etc.) - variables
- CPU can jump to any line or portion of RAM in its operations
- CPU can access one row of RAM just as easily as any other
  - (Random Access Memory)

Ram is Like a Spreadsheet

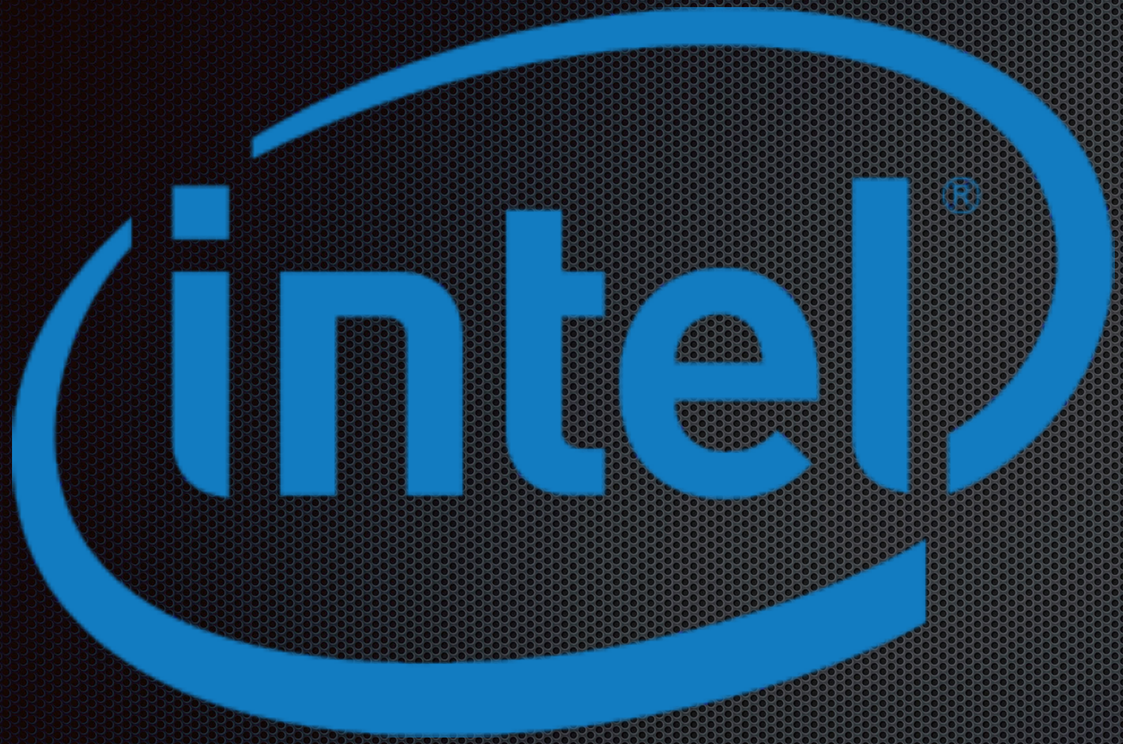
Address



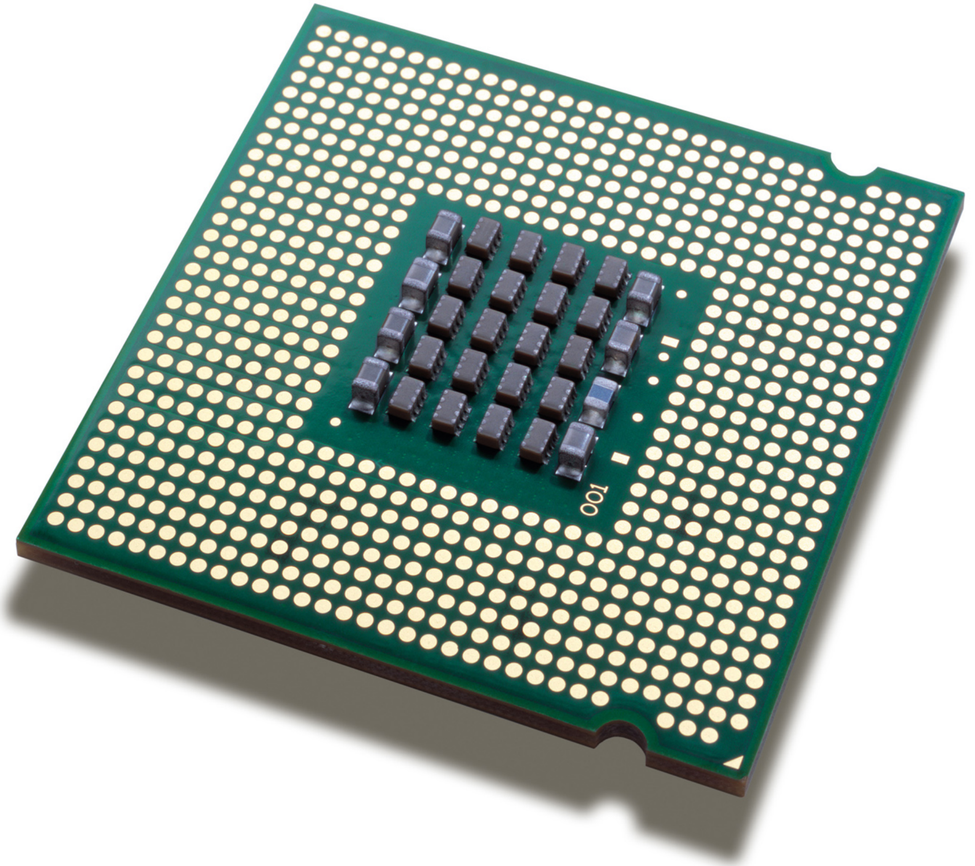
Very Quick Recap



The Processor



Physical Processor



More on Processors

- Processors come in different capacities and speeds
- Really common to find 32-bit and 64-bit in a computer
  - Effectively, how large of a number can the processor work with
- 32-bit binary number:
  - 11111111 11111111 11111111 11111111
- Kind of hard to read, isn’t it?

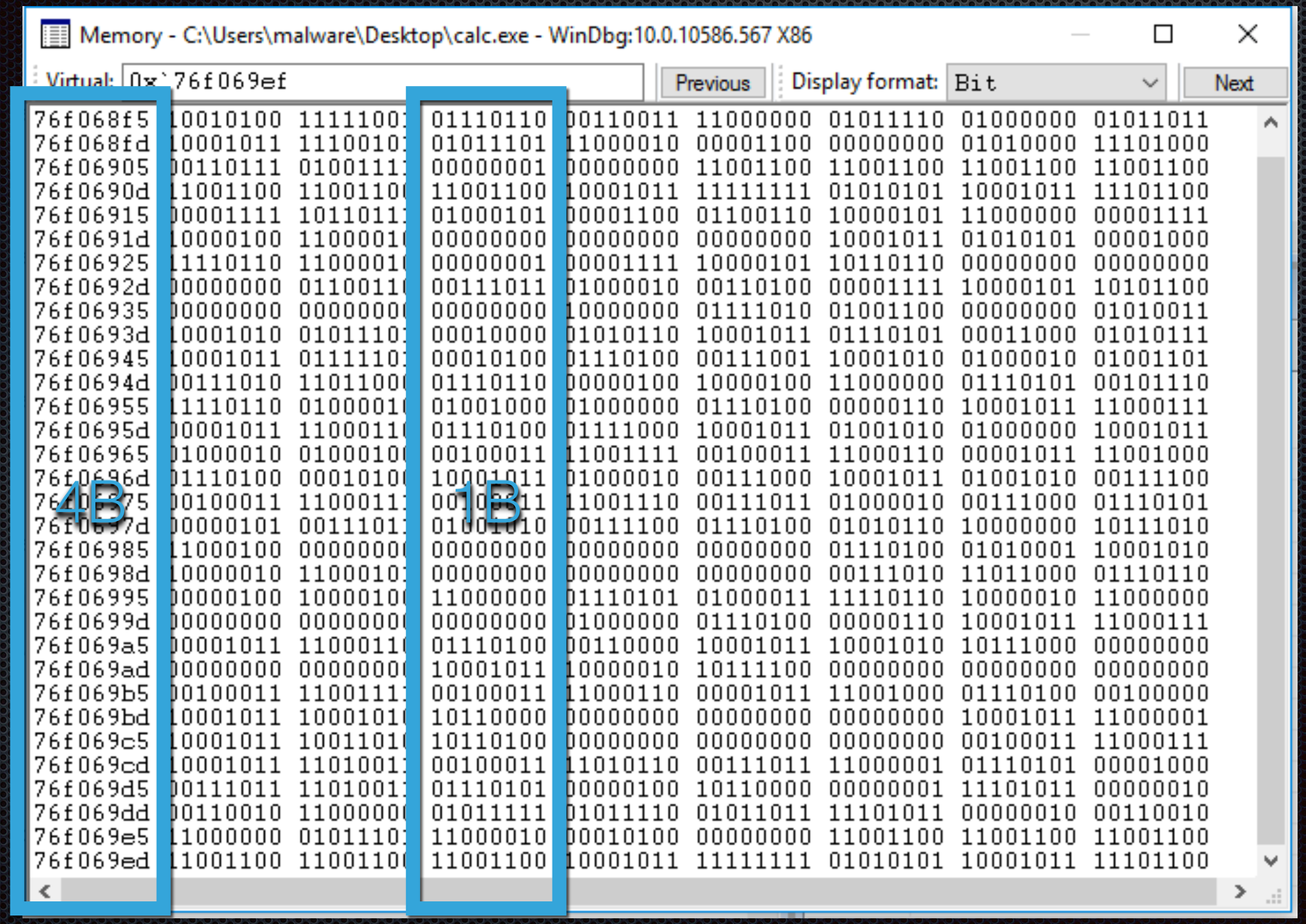
Data Sizes

- Everything in a computer is really based off of binary
- 1 digit (1 or 0) is called a **bit**
- 8 bits put together is a **byte** (11010111)
  - Single letters (characters) or numbers are typically a byte in size
- 1024 bytes is a **kilobyte**
- 1024 kilobytes is a **megabyte**

32-Bit Processor

- They prefer to deal in 32-bit numbers, how many bytes are those numbers?
  - 4 bytes! (32bits/8bits = 4 bytes)
- Often times, you will see your memory addresses in RAM represented by a 4 byte address
  - 0xaf12d34b <- random address I made up
  - It is represented in hexadecimal, just a little shorter/easier to read
- 2 hex digits (AF) = 1 byte

RAM - A Closer Look

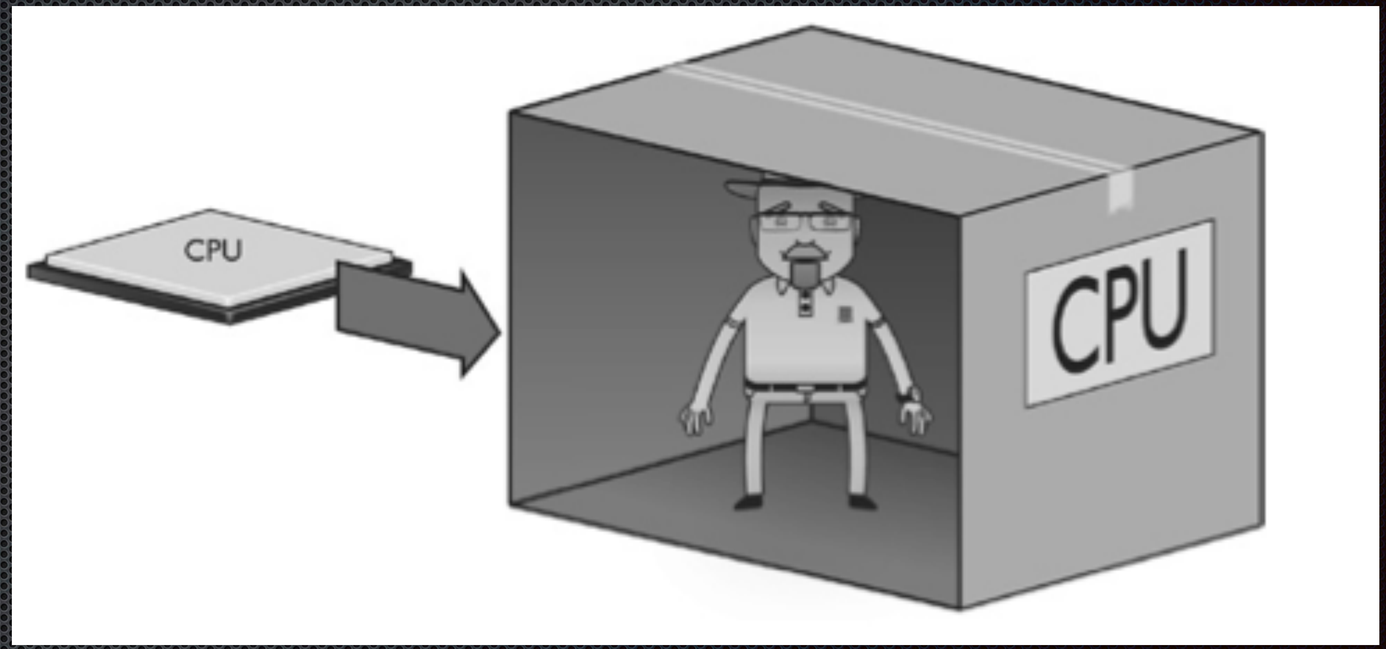


From RAM to the Processor

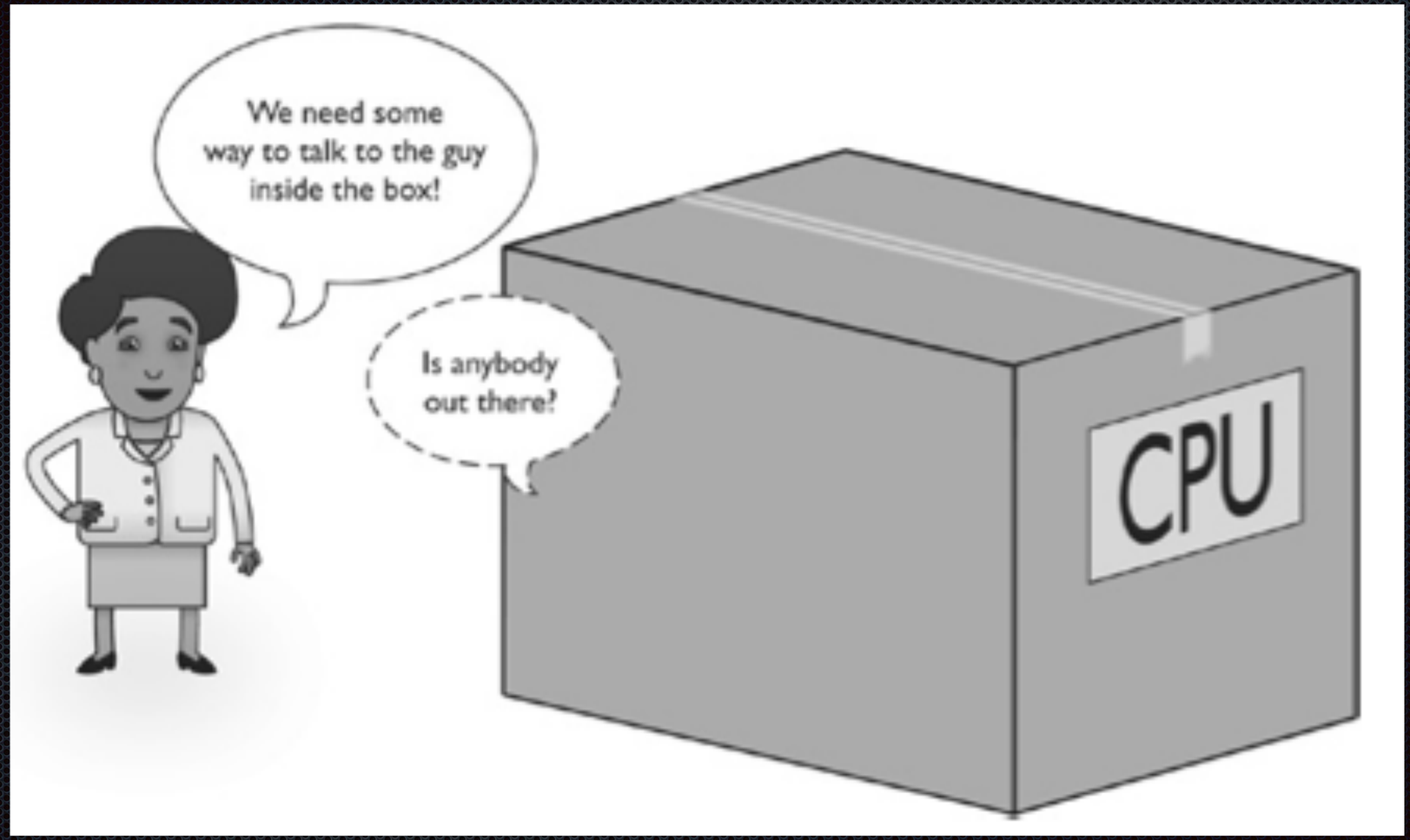
- Your program is currently sitting in RAM
- Processor needs to interpret the binary instructions (run the program)
- If we want to understand exploits, we need to understand a little more about the CPU/RAM and how they interact
- You can get really into the gritty details, but we'll keep it high level

A Man in a Box

- Think of him as a math genius
  - Can do nearly any equation
  - Provides answers really quick

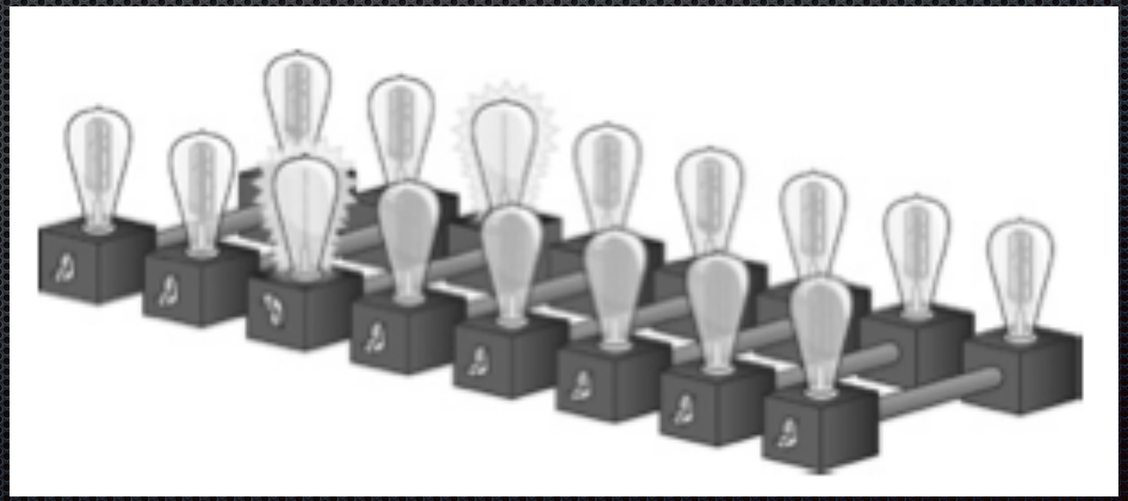


Talking to the man in a box...



Let’s Assume It’s an 8-bit Processor

- Think of a set of 16 lightbulbs
  - 8 of them are inside the box
  - 8 of them are outside of the box
  - Each bulb has a switch that will turn on or off a pair

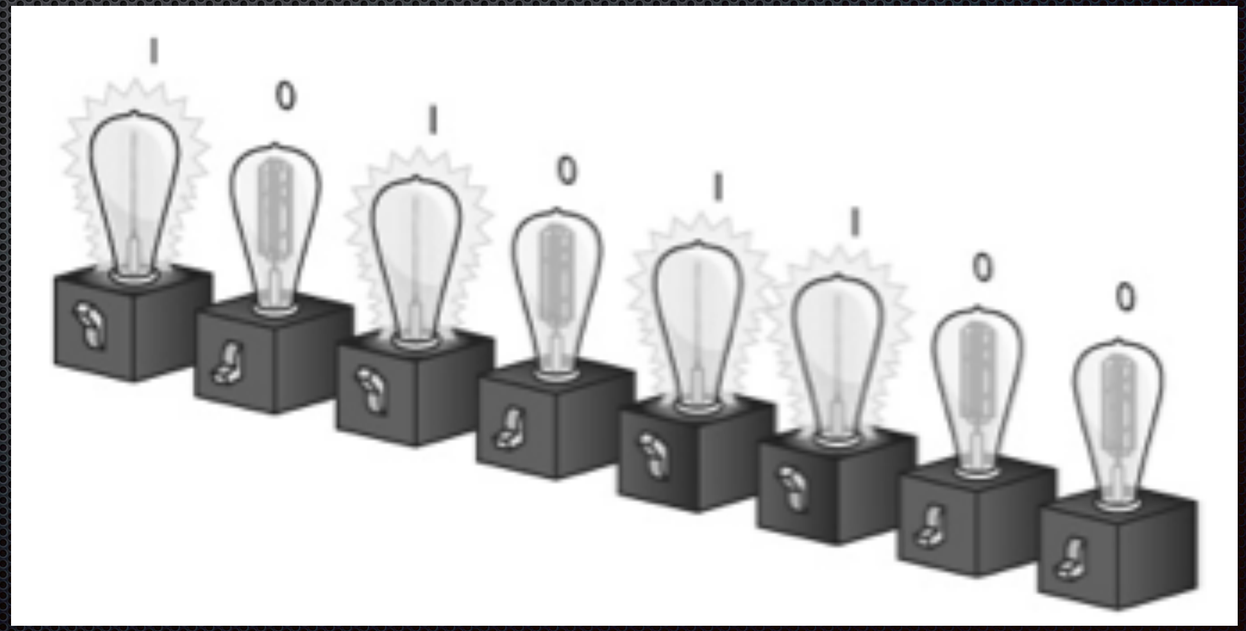


Data Bus

- Our light bulbs work as a data bus
- By turning bulbs on or off, different data sets can be represented
- Each bulb is visible to the guy in the box, and the people outside trying to talk to him
- Guy inside and people outside need to agree on what the patterns mean

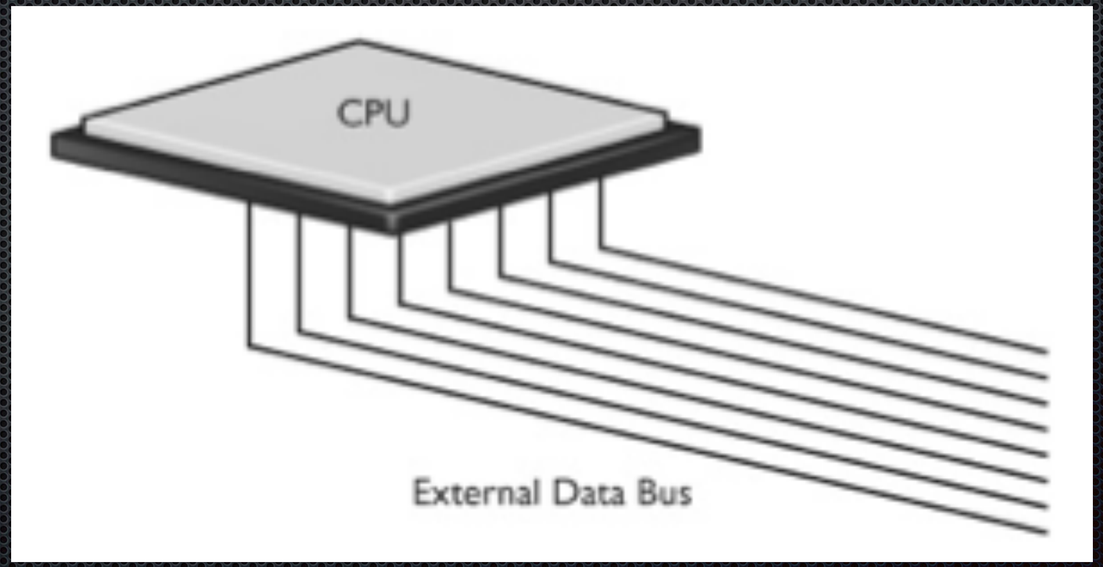
Binary Doesn’t Go Away

- Back to the light bulbs, it’s a pain to say on-off-off-on-etc.
- Binary will represent on and off
  - 1 = on
  - 0 = off



Address Bus

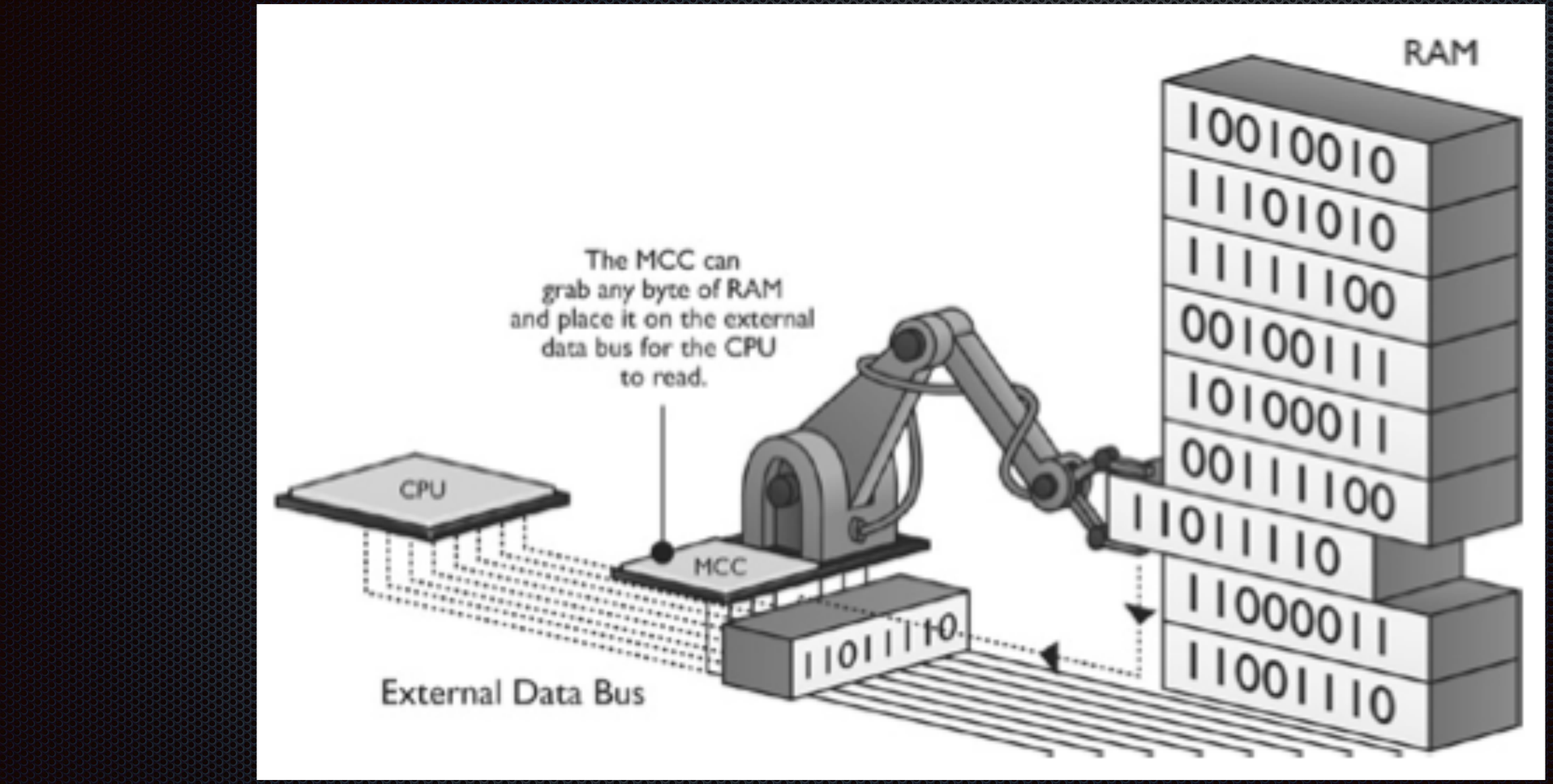
- So far, our computer is only a CPU and RAM
- Extend the EDB to support RAM



Address Bus

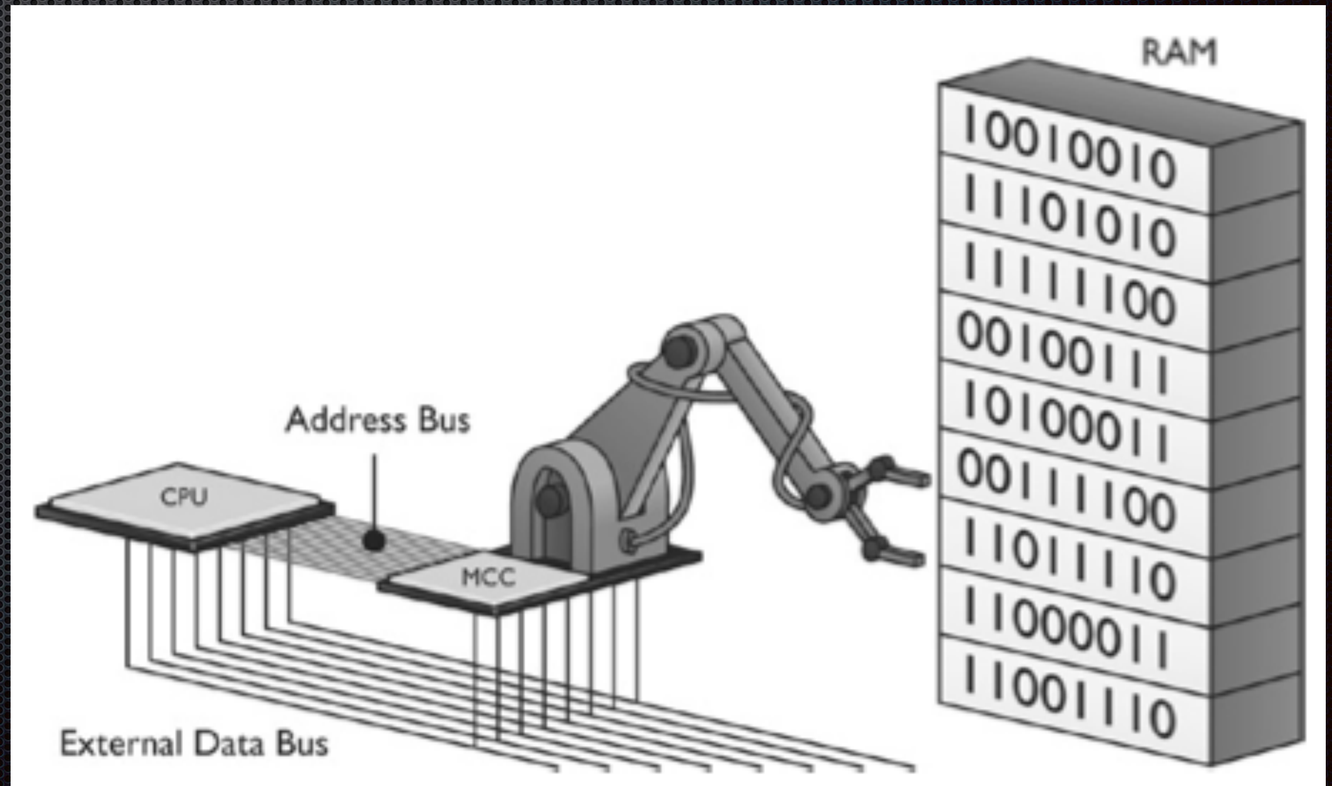
- It’s not quite as simple as plugging RAM into the EDB
- There needs to be something in between for the processor to be able to select which row of RAM it needs
- Memory Controller Chip (MCC)

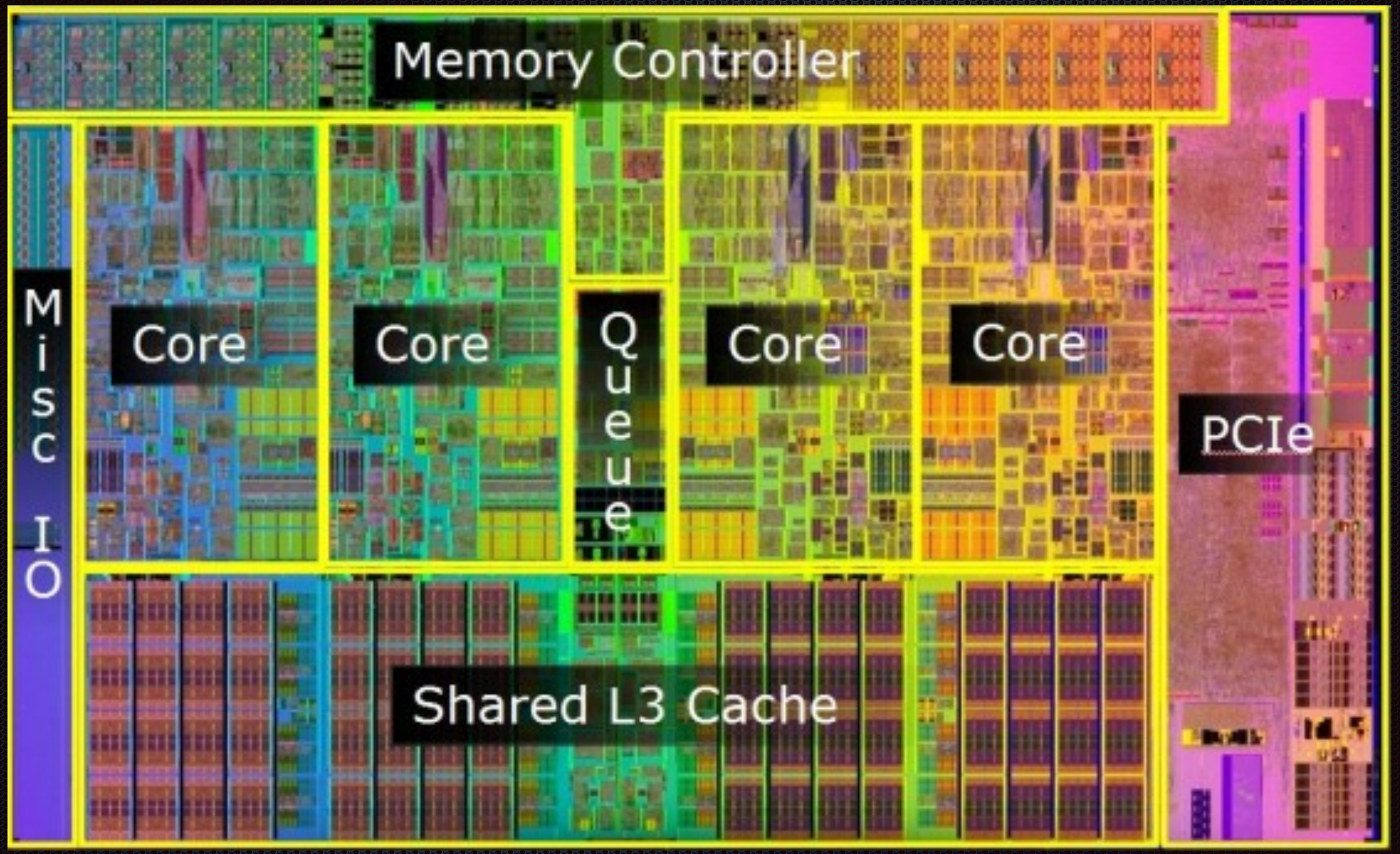
MCC


RAM
10010010
11101010
11111100
00100111
10100011
00111100
11011110
11000011
11001110
The MCC can grab any byte of RAM and place it on the external data bus for the CPU to read.
CPU
MCC
11011110
External Data Bus

Address Bus

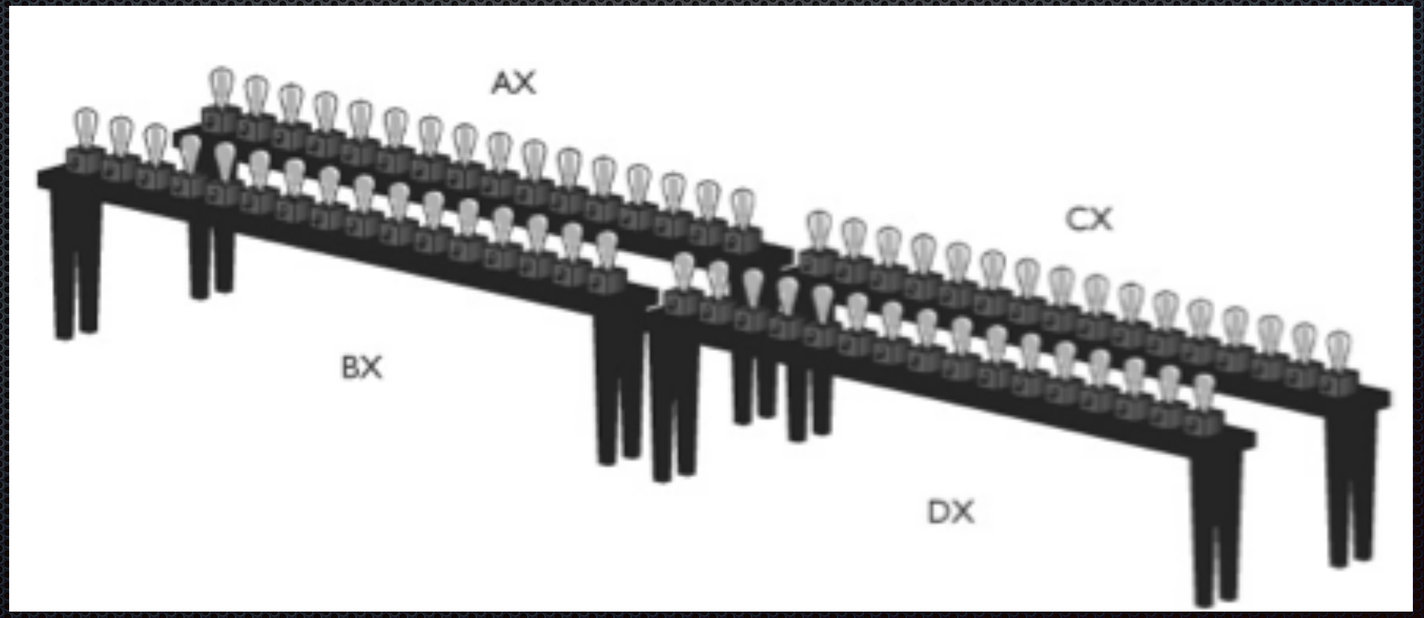
- MCC interfaces with EDB
- Address bus tells MCC which piece of RAM to grab




Memory Controller
Misc IO
Core
Core
Queue
Core
Core
PCIe
Shared L3 Cache

Going Inside the Box

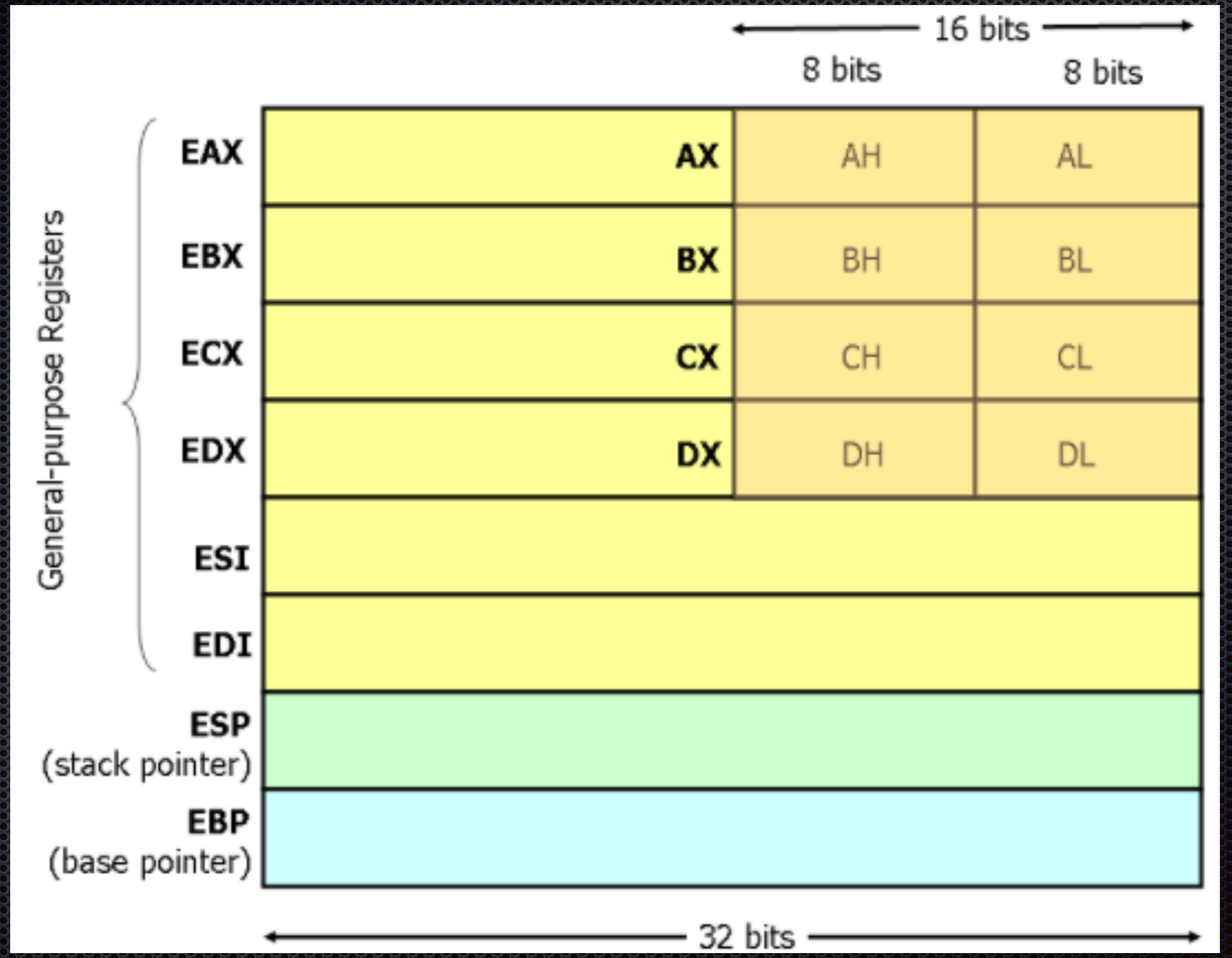
- The guy in the box needs work tables to handle the math and operations/ instructions he’s assigned
- These work tables are called registers



Registers

- All CPUs have them, some have different numbers of them
- 64-bit processors have more than registers than 32-bit processors
- They serve different functions, but let’s focus on some common, general-purpose registers
- When RAM passes the processor an instruction or some information, it is stored on a workbench, or the register

32-Bit Registers



Remember our assembly?

```
0804842b <main>:
 804842b:       8d 4c 24 04                   lea    0x4(%esp),%ecx
 804842f:       83 e4 f0                      and    $0xfffffff0,%esp
 8048432:       ff 71 fc                      pushl  -0x4(%ecx)
 8048435:       55                            push   %ebp
```

Another Quick Recap

- We compile C code into binary instructions for the processor to understand
- The program is loaded from the HDD to a location in memory
  - Each “row” of memory is addressed by a hexadecimal value (0xFFFFDE73)
- The CPU retrieves instructions/data from memory and stores them in registers (workbenches)
  - The registers may hold variables, be used to do math/operations

Stack Buffer Overflow

- Open up the **stack_overflow_16.c** program and let's take a look at what it does
- The compilation instructions and execution instructions are in the code
- Run the program to see what it does, can you get an error?

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

void echo(void)
{
    printf("Enter some text:\n");
    char buffer[16]; // 16 byte buffer stored on the stack
    gets(buffer); //retrieve user input and store in the buffer
    printf("%s\n", buffer); //print out the user input as a string
    return;
}


void main(void)
{
    echo();
    exit(0);
}
```

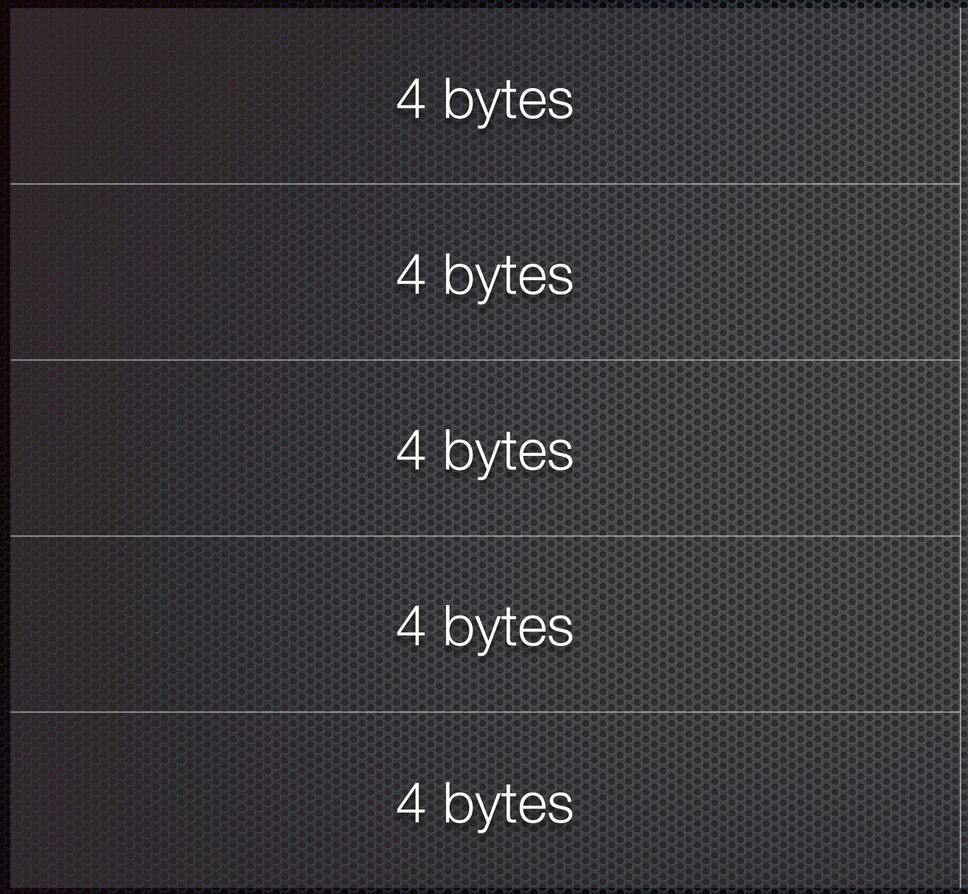
Manual Fuzzing (Testing Input)

- As you can see, if we enter a string larger than the buffer (16-bytes) we get a segmentation fault
- To see what is faulting, we need to open the program in a debugger:
  - gdb ./stack_overflow_16.c
  - list 1,50
  - run
- Cause the crash to happen again after running and look at the registers:
  - info reg

The Stack

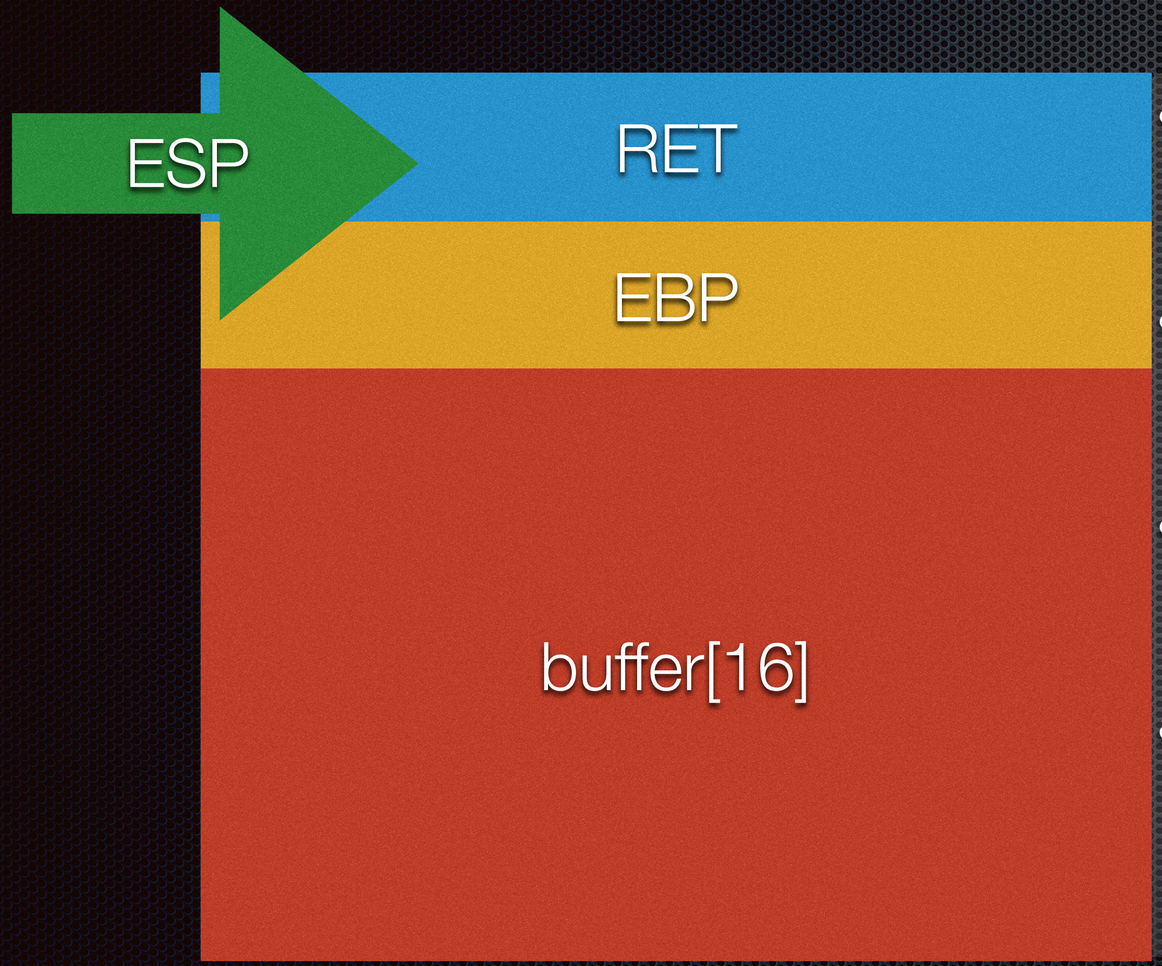
- This is a section of your computers RAM where exploits can occur
- It is an arbitrary location in RAM that can grow/shrink depending on what your code needs
  - If you have a lot of variables in a function, the space in the stack will be larger
- When you load a program, one register (ESP) is used to control the stack

The Stack



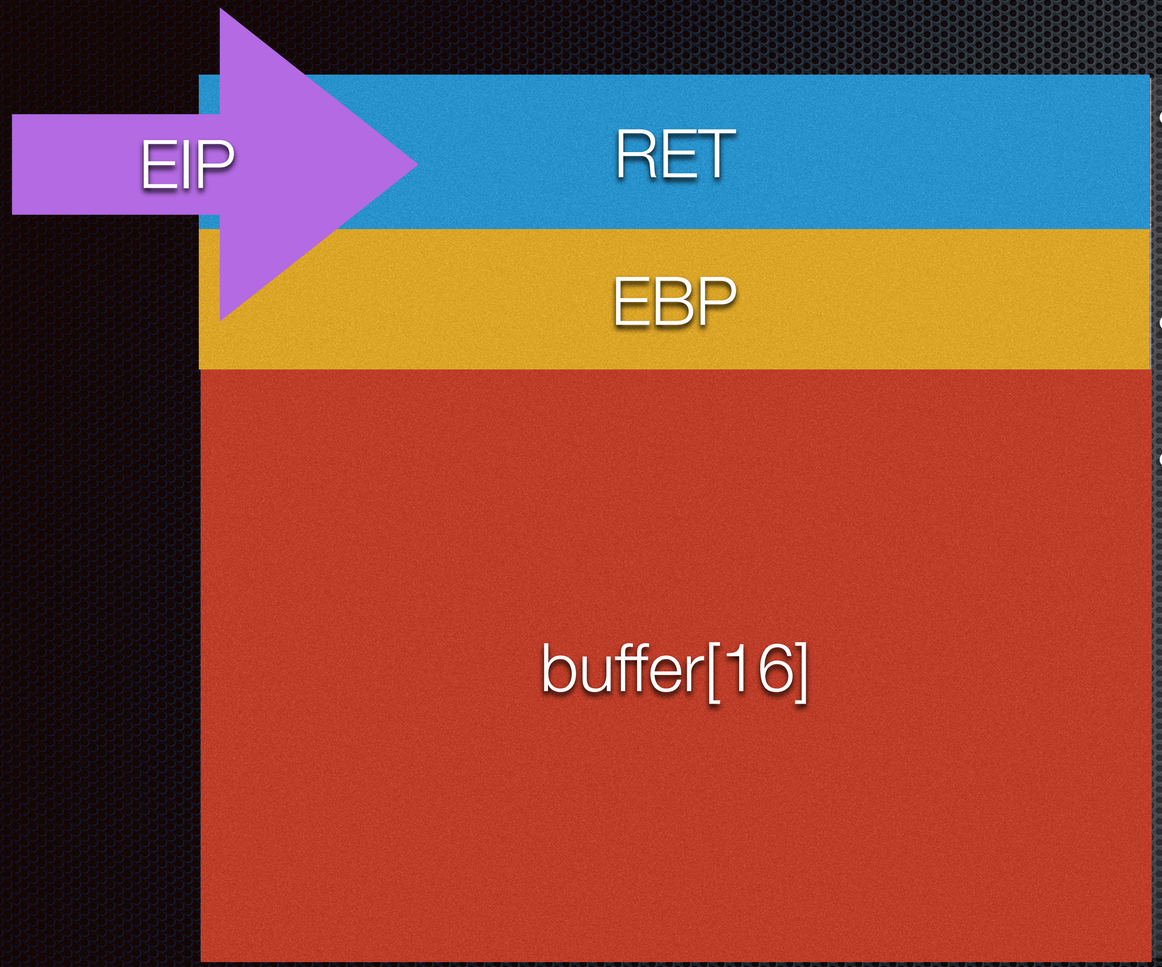
- Our program is loaded, some hunk of memory is allocated for it to run
- In here we have an empty stack
- When a function is called, a stack frame is created to hold the variables/contents of the function

Calling a Function



- A register, ESP, keeps track of the location of the stack
- When the function is called, a return address is saved (RET)
- The base address of the stack is also saved (EBP)
- Then variables are added into the stack as well

Exiting a Function



- When the function exits, it's going to use a register, EIP
- EIP will get set to RET
- If the RET address is valid, the function exits

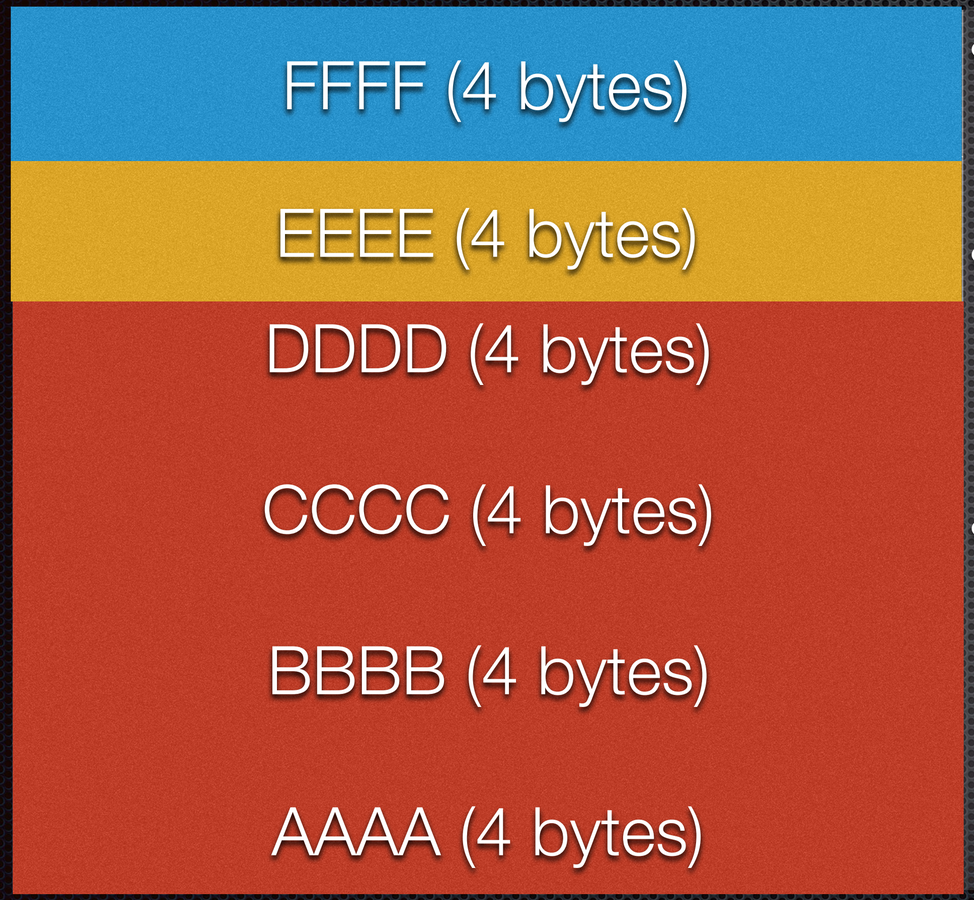
Stack Overflow

- Remember, EIP is the instruction pointer
  - Points to the address location in memory of the next instruction
  - In this case, its's trying to execute 0x48484848, an invalid address

```
Starting program: /root/Desktop/basic_exploitation/stack_overflow.out
Enter some text:
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH

Program received signal SIGSEGV, Segmentation fault.
0x48484848 in ?? ()
(gdb) INFO REG
eax            0x21     33
ecx            0xffffffff       -1
edx            0xf7fae878       -134551432
ebx            0xf7fad000       -134557696
esp            0xffffd3f0       0xffffd3f0
ebp            0x47474747       0x47474747
esi            0x0      0
edi            0x0      0
eip            0x48484848       0x48484848
```

What happened?



- Our one variable was only supposed to hold 16 bytes
- We wrote much more than that and ended up eventually overwriting the RET address
- EIP tries to execute FFFF and fails when the function exits

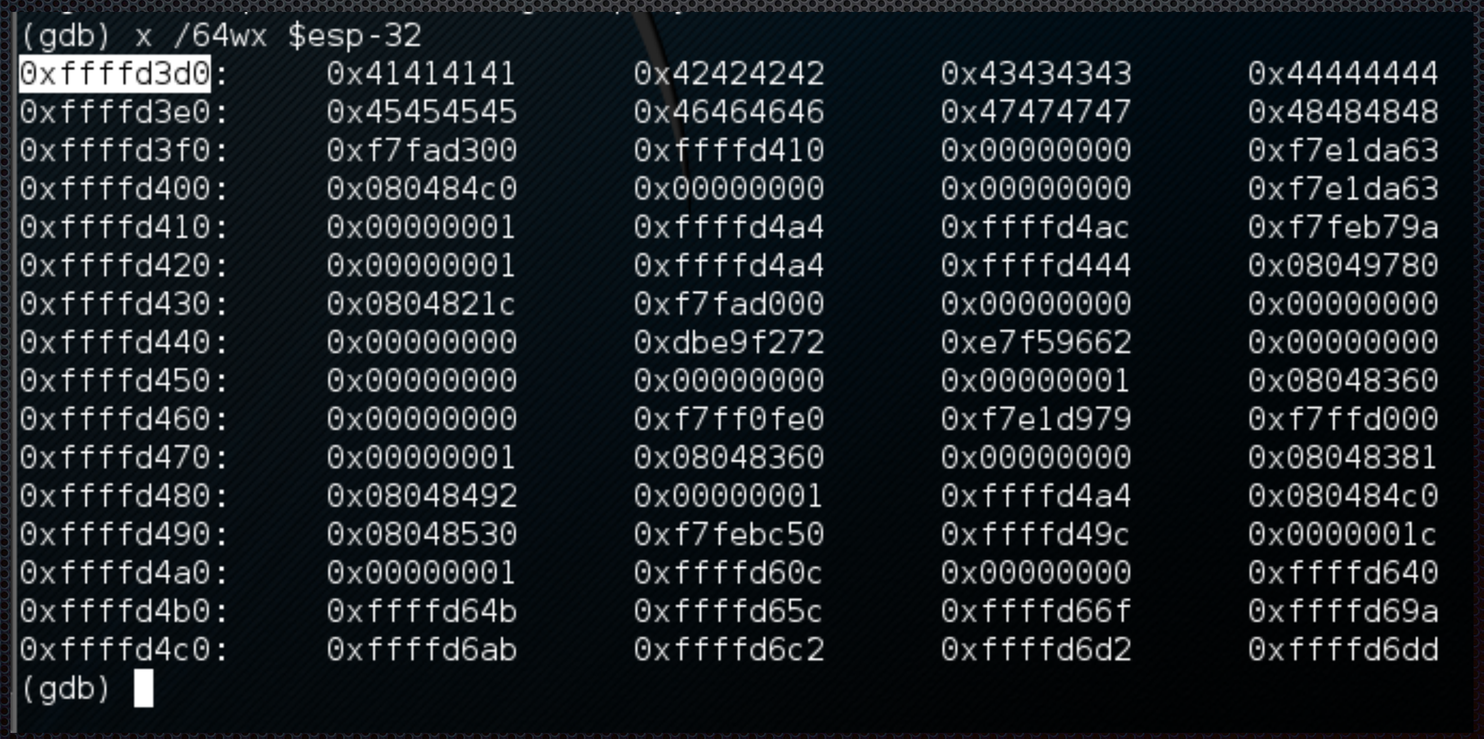
The Real Crash Happened at HHHH

- What's the deal, we crashed at HHHH, your slide showed FFFF?!?
- The way memory is managed and the compiler works, it's not always 100% picture perfect
- There are optimizations that occur and memory is organized in what should be the most efficient manner
- Sometimes segments of memory get aligned differently
- Exploitation is sometimes an art, not a science :)

Take a moment…we have manipulated what is in EIP
We choose what the program does next :)
This is a proof-of-concept (POC) exploit at this point!

What’s on the stack?

- We’ll look at the hex words leading up to ESP, the stack pointer
- The highlighted text is the start of our input (0xffffd3d0)



Making a Useful Payload

- Right now, our “payload” is a series of letters (AAAABBBB[…]HHHH)
- We can replace those letters in our payload with instructions to perform some other action on the system
  - HHHH is the instruction that should be executed next, but we could change that to the start of our payload (0xffffd3d0) so the payload gets executed
- Take a look at **msfvenom** a bit later
  - Built-in tool to help us generate a payload to be used

Let’s Modify the Program a Bit

- 16 bytes isn’t a whole lot of space to have to exploit or introduce code
- We’re going to switch to **stack_overflow_256.c**
- Enter in a large number of characters again to see if we can get a crash and control EIP again
- perl -e 'print “A"x256,"BBBBCCCCDDDDEEEEFFFFGGGGHHHH"'

```
18  void echo(void)
19  {
20      printf("Enter some text:\n");
21      char buffer[256]; // 256 byte buffer stored on the stack
22      gets(buffer); //retrieve user input and store in the buffer
23      printf("%s\n", buffer); //print out the user input as a string
24      return;
25  }
```

Fuzzing Again

- We did 256 A (256 bytes)
- BBBB (4 bytes)
- CCCC (4 bytes)
- DDDD (4 bytes)
- EEEE (4 bytes) <- what overwrote EIP
- Therefore we have 268 bytes of shellcode (256+4+4+4), 4 bytes for EIP
- Total payload length = 272 bytes

Shellcode

- Shellcode is code represented in hex that can be interpreted by the CPU
  - This is where we get to tell the system what to do
- Here is some simple shellcode to print out “y0u sp34k 1337 ?” (50 bytes):
  - eb 19 31 c0 b0 04 31 db b3 01 59 31 d2 b2 12 cd 80 31 c0 b0 01 31 db b3 01 cd 80 e8 e2 ff ff ff 20 79 30 75 20 73 70 33 34 6b 20 31 33 33 37 20 3f 20
- We’ll look at generating our own at some point

New Shellcode

- The buffer is 256 bytes that can hold real data
- The shellcode to print “y0u sp34k 1337 ?” is 50 bytes
- So, 256-50 leaves us with 206 bytes of free space
  - How do we fill up 206 bytes of free space?
- Add on 12 bytes of nothing and 4 bytes for EIP
- We should have 272 bytes in the end

NOP Sled

- When you have free space to fill up in some shellcode, you can use a NOP sled
- It's just a series of 1 byte instructions that basically tell the program to continue on
- On a normal 32-bit processor, the instruction for NOP is 0x90
- perl -e 'print "\x90"x206' will give you 90 bytes of NOP

MSFVENOM

- List the available payloads:
  - msfvenom ——help
  - msfvenom ——list payloads
- Choose a payload of your liking and we can use it:
  - msfvenom -p linux/x86/exec CMD=ls -f c -b ‘\x00’

Redirection

- Sometimes, we don’t need to execute our own shellcode
- We might just want the program to do something it is intended to
  - Give us administrative privileges
  - Say our serial number/product key is genuine
  - Move money from one bank account to another
- If the functionality is already built in, let’s just use it!

stack_overflow_redirection.c

- Let's start out like we've done before
- Run the program, see what it does
- Can you get it to crash?
- What functions are being called?

stack_overflow_redirection.c

```c
/*
    Compile instructions (x32):
    gcc -g -fno-stack-protector -z execstack stack_overflow_redirection.c -o stack_overflow_redire

    Compile instructions (x64):
    gcc -g -m32 -fno-stack-protector -z execstack stack_overflow_redirection.c -o stack_overflow_r

    Run instructions:
    ./stack_overflow_redirection.out

    Debug instructions:
    gdb ./stack_overflow_redirection.out
*/

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

void lockOutUser(void)
{
    // If the user enters in the wrong password, they must be a hacker. Lock the account.
    printf("You entered the wrong password. Account locked out!\n");
}

void giveUserRoot(void)
{
    // If we're here, the user must be legit.  Full admin privliges.
    printf("Root access granted!\n");
}

int passCheck(void)
{
    char buff[16]; // Passwords can be up to 15 charactesr
    int validPass = 0;

    // Prompt the user for a password and store it in "buff"
    printf("Enter the password : \n");
    gets(buff);

    // Password checking routine
    if(strcmp(buff, "gen_cyber=1337"))
    {
        // Incorrect
        printf ("Wrong Password. \n");
        validPass = 0;
    }
    else
    {
        // Correct
        printf ("Correct Password! \n");
        validPass = 1;
    }

    return validPass;
}

int main (void)
{
    if (passCheck() == 1)
        giveUserRoot();
    else
        lockOutUser();

    exit(0);
}
```

So you can control EIP, huh?

- Bringing this full circle
- The type of exploit (stack buffer overflow) isn't really different than the last
- Rather than redirecting input to our shellcode that we chose, this time we are going to redirect execution (EIP) to another spot in the real program
  - We have a really small buffer this time!
- Starting to build out your swiss army knife of exploitation
- All we need to do next is to find the memory address of a valid function…but how?

Helpful GDB Commands

- List all of the functions within a program
  - info func
- Set a breakpoint (stop the program) when a function is called
  - break <func name>
- List breakpoints
  - info break
- Disassemble Function
  - disas <func name>

Let’s take a look…

- Generate another long fuzzing string (maybe twice the length of the buffer)
- Find out where it is crashing like we’ve done in the past
- Once you know how many bytes you need to overwrite EIP, look for a call to the desired function
  - Replace EIP with the call address

```
(gdb) disass main
Dump of assembler code for function main:
   0x08048534 <+0>:     lea    0x4(%esp),%ecx
   0x08048538 <+4>:     and    $0xfffffff0,%esp
   0x0804853b <+7>:     pushl  -0x4(%ecx)
   0x0804853e <+10>:    push   %ebp
   0x0804853f <+11>:    mov    %esp,%ebp
   0x08048541 <+13>:    push   %ecx
   0x08048542 <+14>:    sub    $0x4,%esp
   0x08048545 <+17>:    call   0x80484bb <passCheck>
   0x0804854a <+22>:    cmp    $0x1,%eax
   0x0804854d <+25>:    jne    0x8048556 <main+34>
   0x0804854f <+27>:    call   0x80484a3 <giveUserRoot>
   0x08048554 <+32>:    jmp    0x804855b <main+39>
   0x08048556 <+34>:    call   0x804848b <lockOutUser>
   0x0804855b <+39>:    sub    $0xc,%esp
   0x0804855e <+42>:    push   $0x0
   0x08048560 <+44>:    call   0x8048370 <exit@plt>
End of assembler dump.
```

CTF Link

- http://goo.gl/forms/mivhCv3UELEgmDTM2