COMPSCI 390R

Reverse Engineering & Vulnerability Analysis

Welcome!

This class is brought to you by:

- UMass Cybersecurity Club
- The University of Massachusetts Cybersecurity Institute

UMASS CYBERSEC CLUB

Cybersecurity Institute
SECURITY FOR THE COMMON GOOD™

Introductions

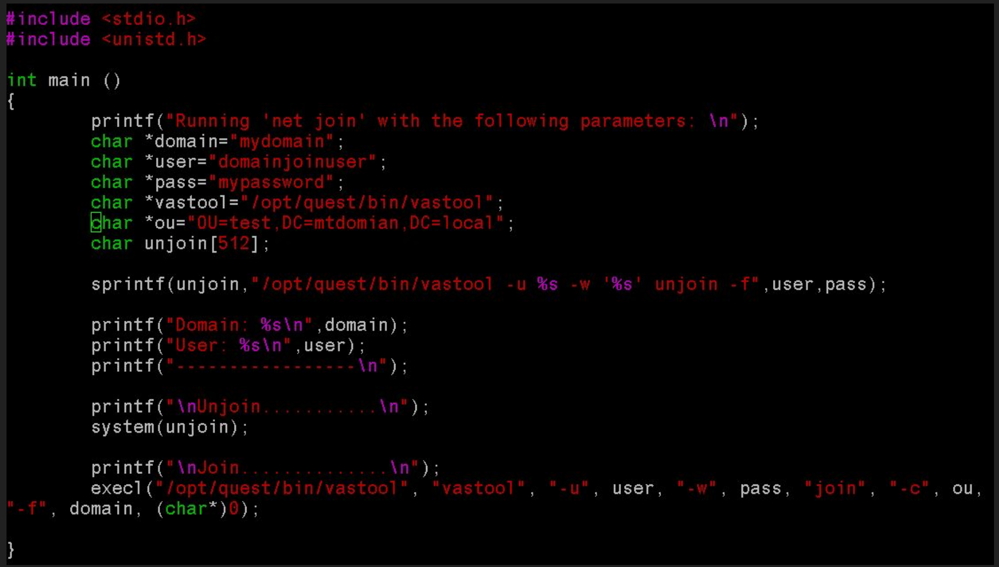
Disclaimer!

Everything taught in this class is for educational purposes only. Do not use anything taught in this class without permission or in illegal ways.

So what is RE & VA anyways?

Reverse Engineering

The process of figuring out how a device, process, system or any object works with little to no insight of the internal mechanisms.



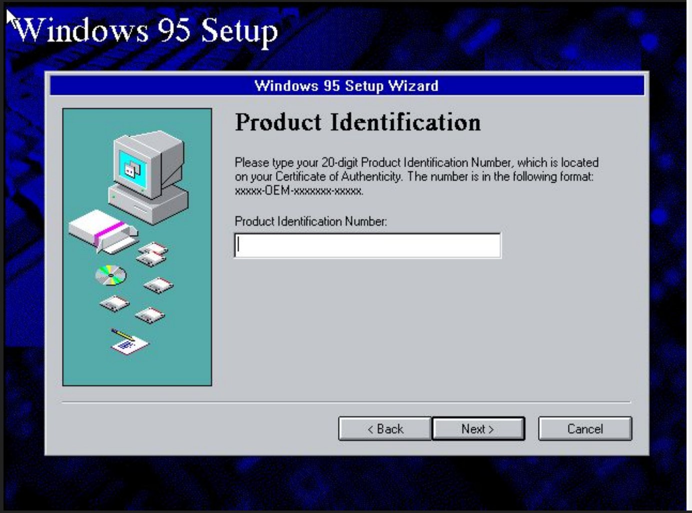
```c
#include <stdio.h>
#include <unistd.h>

int main ()
{
        printf("Running 'net join' with the following parameters: \n");
        char *domain="mydomain";
        char *user="domainjoinuser";
        char *pass="mypassword";
        char *vastool="/opt/quest/bin/vastool";
        char *ou="OU=test,DC=mtdomian,DC=local";
        char unjoin[512];

        sprintf(unjoin,"/opt/quest/bin/vastool -u %s -w '%s' unjoin -f",user,pass);

        printf("Domain: %s\n",domain);
        printf("User: %s\n",user);
        printf("-----------------\n");

        printf("\nUnjoin..........\n");
        system(unjoin);

        printf("\nJoin..............\n");
        execl("/opt/quest/bin/vastool", "vastool", "-u", user, "-w", pass, "join", "-c", ou,
"-f", domain, (char*)0);

}
```

Reverse Engineering Example 1:

If you can reverse an algorithm that verifies product license keys then you can generate unlimited licenses

Windows 95 Product Keys:

Product Key Check consists of checking starting values then seeing if the key is 0 mod 7

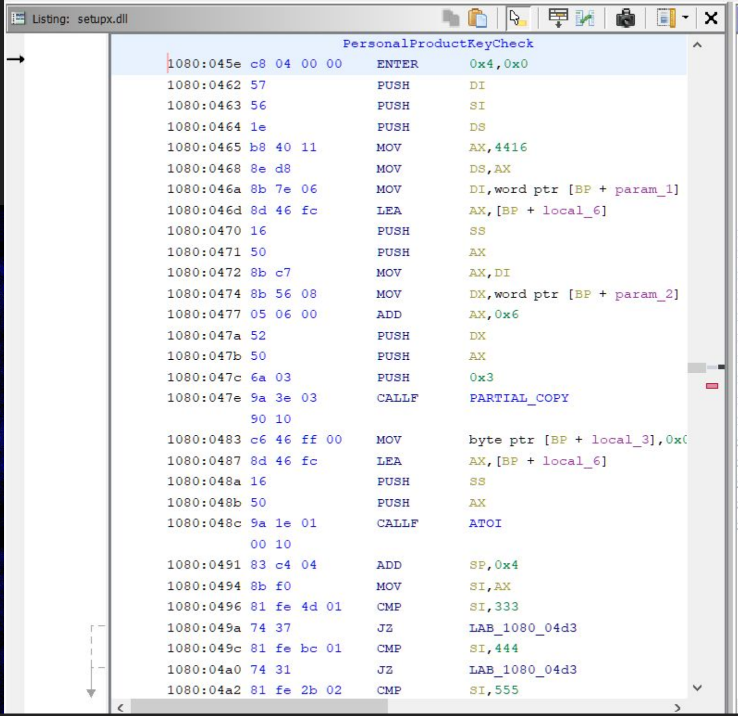

Windows 95 Setup

Windows 95 Setup Wizard

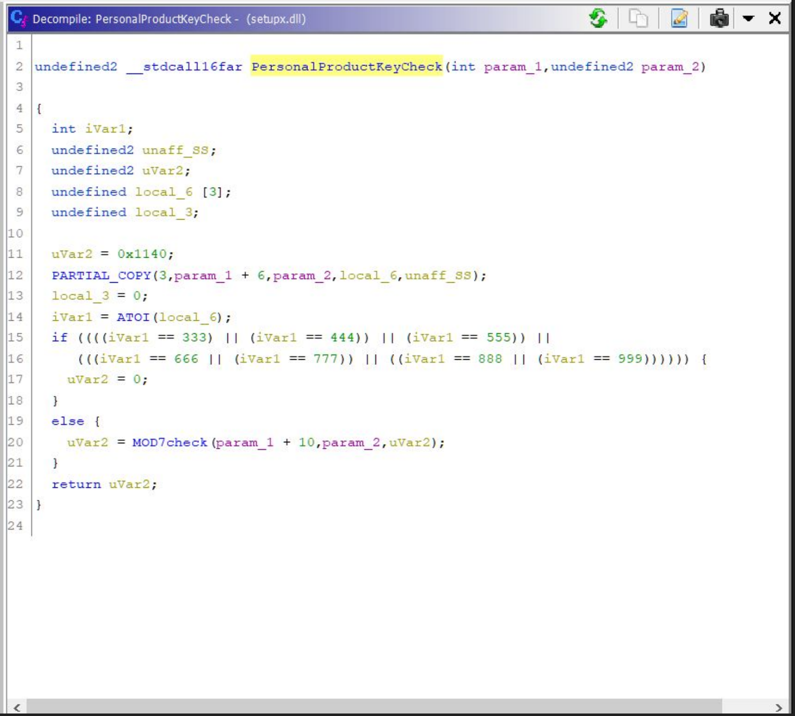
Product Identification

Please type your 20-digit Product Identification Number, which is located on your Certificate of Authenticity. The number is in the following format: xxxxx-OEM-xxxxxxx-xxxxx.

Product Identification Number:


Listing: setupx.dll

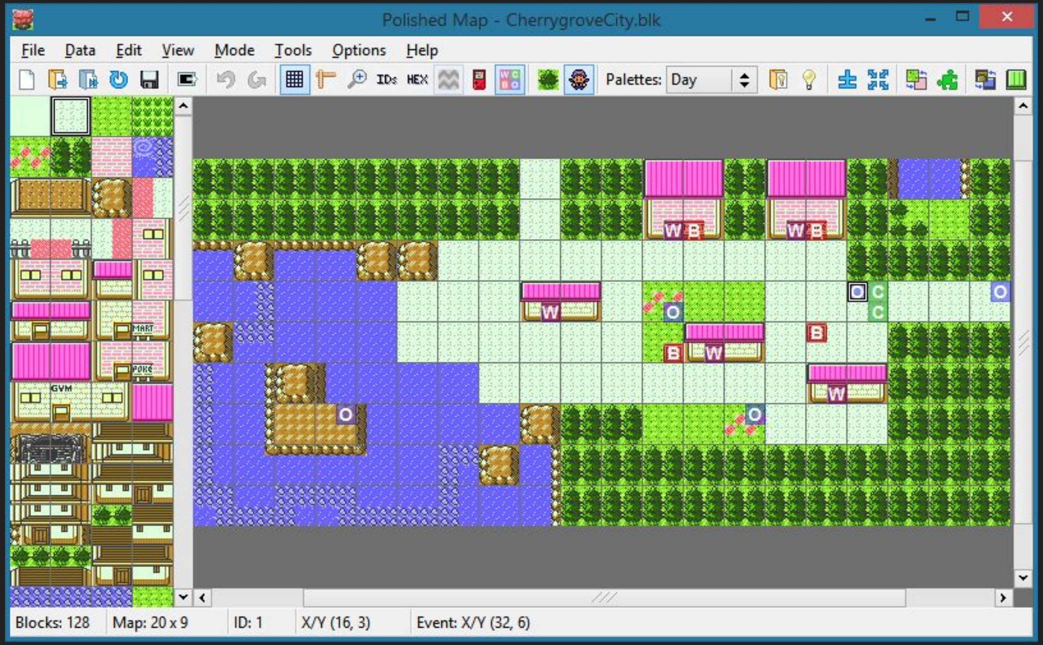
```
                        PersonalProductKeyCheck
1080:045e c8 04 00 00     ENTER      0x4,0x0
1080:0462 57              PUSH       DI
1080:0463 56              PUSH       SI
1080:0464 1e              PUSH       DS
1080:0465 b8 40 11        MOV        AX,4416
1080:0468 8e d8           MOV        DS,AX
1080:046a 8b 7e 06        MOV        DI,word ptr [BP + param_1]
1080:046d 8d 46 fc        LEA        AX,[BP + local_6]
1080:0470 16              PUSH       SS
1080:0471 50              PUSH       AX
1080:0472 8b c7           MOV        AX,DI
1080:0474 8b 56 08        MOV        DX,word ptr [BP + param_2]
1080:0477 05 06 00        ADD        AX,0x6
1080:047a 52              PUSH       DX
1080:047b 50              PUSH       AX
1080:047c 6a 03           PUSH       0x3
1080:047e 9a 3e 03        CALLF      PARTIAL_COPY
          90 10
1080:0483 c6 46 ff 00     MOV        byte ptr [BP + local_3],0x
1080:0487 8d 46 fc        LEA        AX,[BP + local_6]
1080:048a 16              PUSH       SS
1080:048b 50              PUSH       AX
1080:048c 9a 1e 01        CALLF      ATOI
          00 10
1080:0491 83 c4 04        ADD        SP,0x4
1080:0494 8b f0           MOV        SI,AX
1080:0496 81 fe 4d 01     CMP        SI,333
1080:049a 74 37           JZ         LAB_1080_04d3
1080:049c 81 fe bc 01     CMP        SI,444
1080:04a0 74 31           JZ         LAB_1080_04d3
1080:04a2 81 fe 2b 02     CMP        SI,555
```


Decompile: PersonalProductKeyCheck - (setupx.dll)

```c

undefined2 __stdcall16far PersonalProductKeyCheck(int param_1,undefined2 param_2)

{
  int iVar1;
  undefined2 unaff_SS;
  undefined2 uVar2;
  undefined local_6 [3];
  undefined local_3;

  uVar2 = 0x1140;
  PARTIAL_COPY(3,param_1 + 6,param_2,local_6,unaff_SS);
  local_3 = 0;
  iVar1 = ATOI(local_6);
  if ((((iVar1 == 333) || (iVar1 == 444)) || (iVar1 == 555)) ||
     (((iVar1 == 666 || (iVar1 == 777)) || ((iVar1 == 888 || (iVar1 == 999)))))) {
    uVar2 = 0;
  }
  else {
    uVar2 = MOD7check(param_1 + 10,param_2,uVar2);
  }
  return uVar2;
}

```

Reverse Engineering Example 2:

If you can reverse a game to learn how to modify it or create your own game based off the same engine


Polished Map - CherrygroveCity.blk
File Data Edit View Mode Tools Options Help
Palettes: Day
Blocks: 128 Map: 20 x 9 ID: 1 X/Y (16, 3) Event: X/Y (32, 6)

Polished Map: Rom map editor

Yuzu Emulator: Nintendo Switch Emulator

Reverse Engineering Example 3:

U.S. Missile AIM-9 Sidewinder hit and got lodged in a Soviet plane.

3 years later Soviet Vympel-13 entered service (Vietnam War).

AIM-9 Sidewinder

Vympel-13

Both missiles had the same Infrared Tracking technology incorporated in the camera

Reverse Engineering Example 3 (cont.):

MIM-104 Patriot (Patriot Missiles) are currently expected to be used until at least 2040, have been used in Gulf war, Iraq, Israel, Ukraine…

MIM-104 Patriot

China’s HQ-9

The HQ-9 is believed to be based off of the Russian S-300, but with the radar and seeker systems of a Patriot Missile. Early HQ-9 are believed to use the guidance system developed by the U.S. from a missile stolen/lost by Israel or Germany during the Gulf War

Vulnerability Analysis:

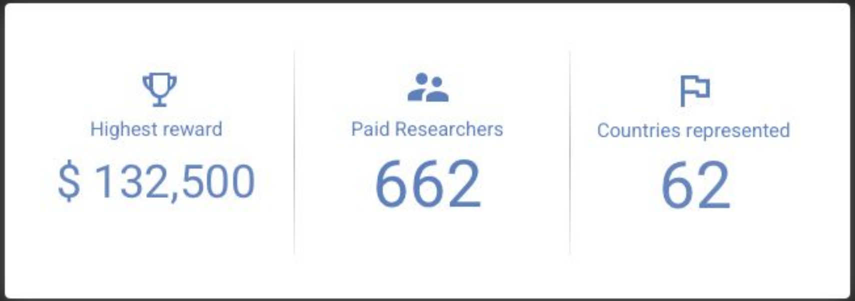
Inspecting a device, process or any object for vulnerabilities that can be exploited.

End goal is to understand how an attacker can abuse a system and then report/patch vulnerabilities

hackerone

Most popular bug bounty middleman

Google



Highest reward	Paid Researchers	Countries represented
$ 132,500	662	62

6.7 Million paid out in 2020

About CS 390R:

- This class will teach you introductory reverse engineering using modern tools, and vulnerability analysis/exploitation of ELF files.

- This will be a very fast paced class; the topics will be challenging but the material will be hands-on.

- Learning is the number one goal! Feel free to stop me and ask questions any time during class.

What will this class look like?

- Homework – 10%
- Labs – 10%
- Projects – 40%
- Midterm – 15%
- Presentations – 25%

Homework:

- Homework will be weekly assignments that are meant to test your knowledge on topics covered in class

- May be accompanied by a reading/video to help cover topics that would take too long in class or if we just feel like it’s a good resource

Labs:

- In person labs, done in place of a lecture. Come to class and walk through guided examples with your peers.
- Align more or less with the release of a new project, meant to be a practice of technical skills.
- Lab grade is based off of attendance and participation, as long as you come and try you should get full points.

Projects:

- Take home assignments, around two weeks to complete. Consist of challenges where you have to reverse engineer and exploit multiple programs

- Each project will have its own rubric of objectives to complete, with partial credit for how much you managed to do. Each project will have 3 main sections to complete for full credit.

Midterm:

- Midterm is take home and similar to projects in terms of workload. You are expected to complete these in 48/72 hours respectively

- All work and methodology must be shown in a write-up, no collaboration is allowed.

Presentations:

- Last week of lecture you will need to present to the class on a security topic of your choice (not covered in class but relevant to it), should start thinking about it soon

- Will be in groups, will talk about more later in the semester

So What Are We Learning?

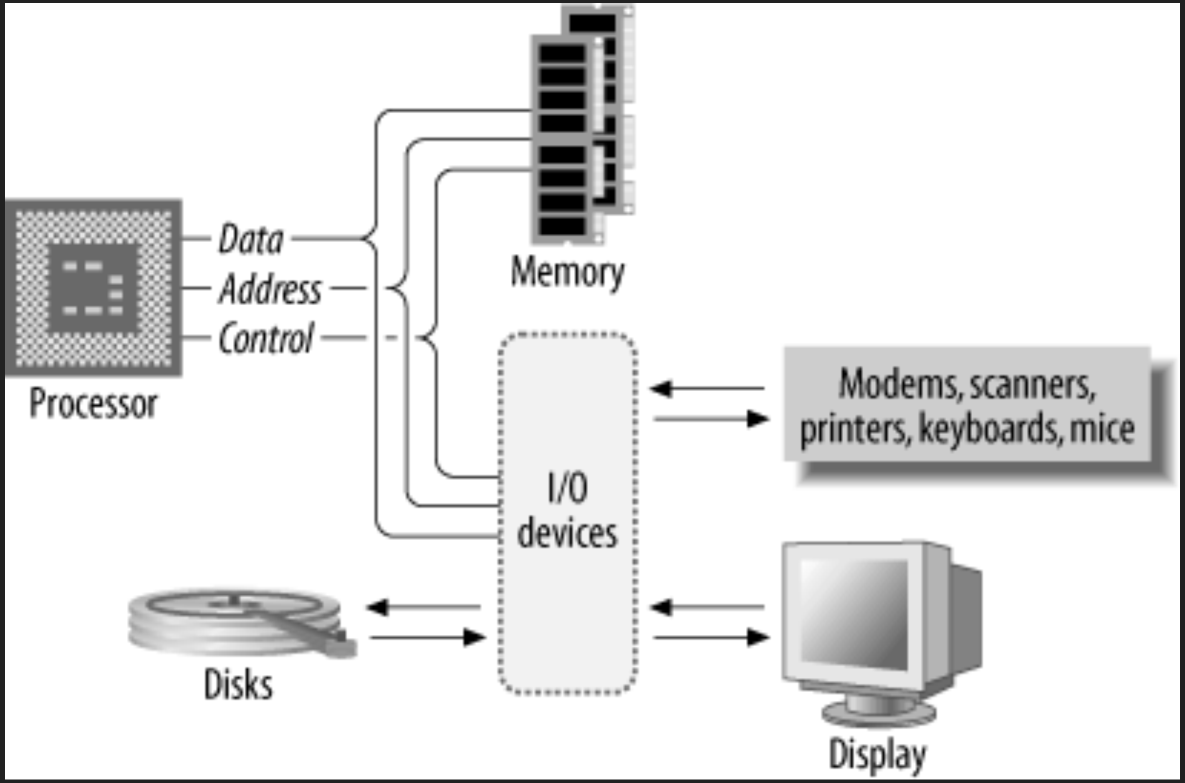
Plan for the semester:

We will be learning various techniques to find and exploit memory corruption vulnerabilities on ELF files

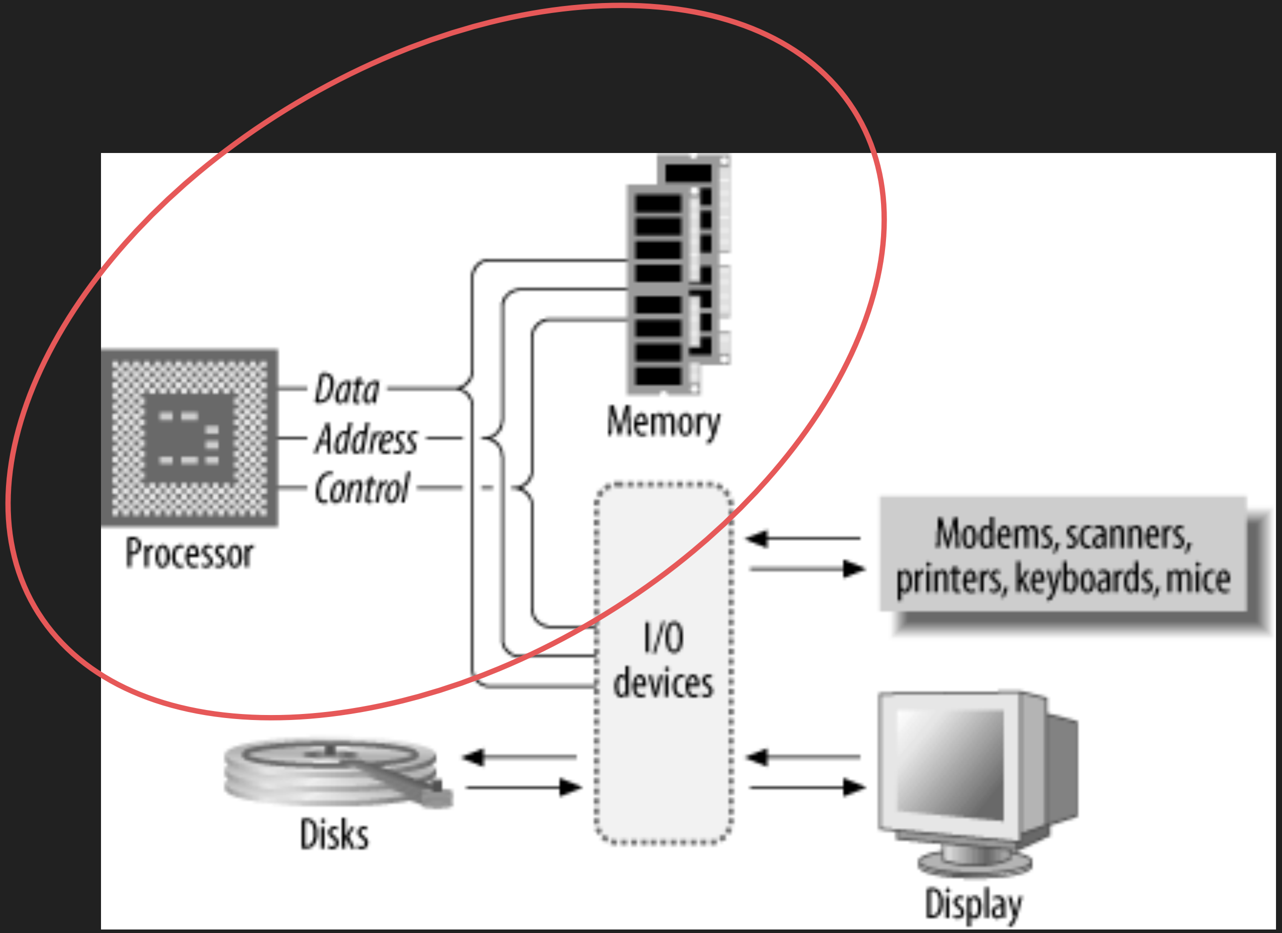
1. CS230 Concepts Review
2. Reverse Engineering w/ Ghidra
3. Basic Stack Based Exploitation
4. Advanced Stack Exploitation/Mitigation Bypass
5. Heap Exploitation
6. Intro to Software Analysis
7. Automated Program Analysis/Advanced Topics

Let’s Review Some Concepts

How do computers work?



How do computers work?



Parts relevant for this course:

- Central Processing Unit (CPU)
  - Performs arithmetic, logic, process control and input/output operations
  - Basically the brains of the computer

- Memory/RAM
  - Location to store/read data from
  - There is no differentiation between instructions and data in memory (kind of, page tables can mark certain locations as executable or not)

Our Target CPU (Intel x86-64):

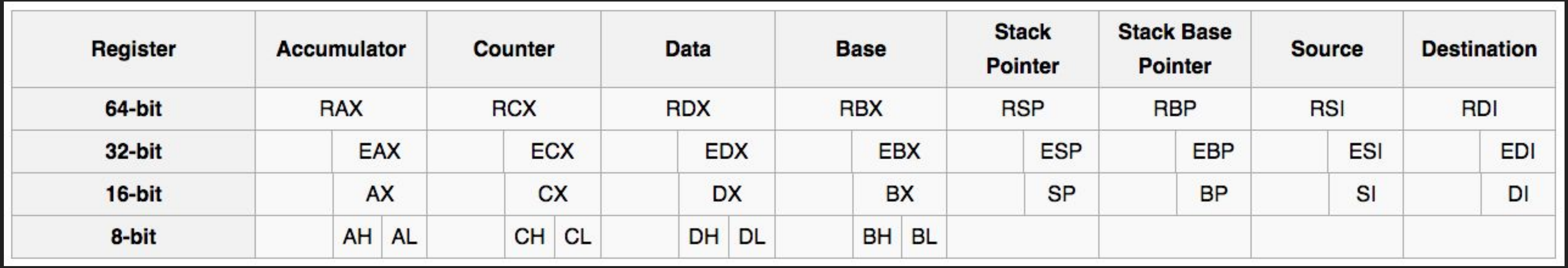
- Every chip manufacturer has to implement a standardized set of assembly instructions that code can compile to

- We will be working with programs and exploits for x86-64, if you have an ARM based laptop (M1 Macbook), come talk to us after class

- Intel x86-64 (also known as amd64) is a 64-bit version of Intel's 32-bit architecture x86

Intel x86-64 (cont.):

- 16 64-bit general registers
  - rax, rbx, rcx, rdx, rsp, rbp, rsi, rdi, r8-r15
  - Registers can be indexed by lower 32-bits, and then by 16 and 8 bit intervals too



Register	Accumulator	Counter	Data	Base	Stack Pointer	Stack Base Pointer	Source	Destination
64-bit	RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
32-bit	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
16-bit	AX	CX	DX	BX	SP	BP	SI	DI
8-bit	AH AL	CH CL	DH DL	BH BL				

- Able to index up to 48-bits of memory
  - Read about paging to understand what the other 16 bits are for
  - Abundant memory means that now processes can perform operations such as reading entire files in memory at once

ASM Review, moves:

Assembly operations are generally of the following form,

    Operation, Destination, Source 1, Source 2

```
Ex:  mov              eax, 1
     mov              eax, ebx
```

Sources can also be pointers, denoted with brackets []

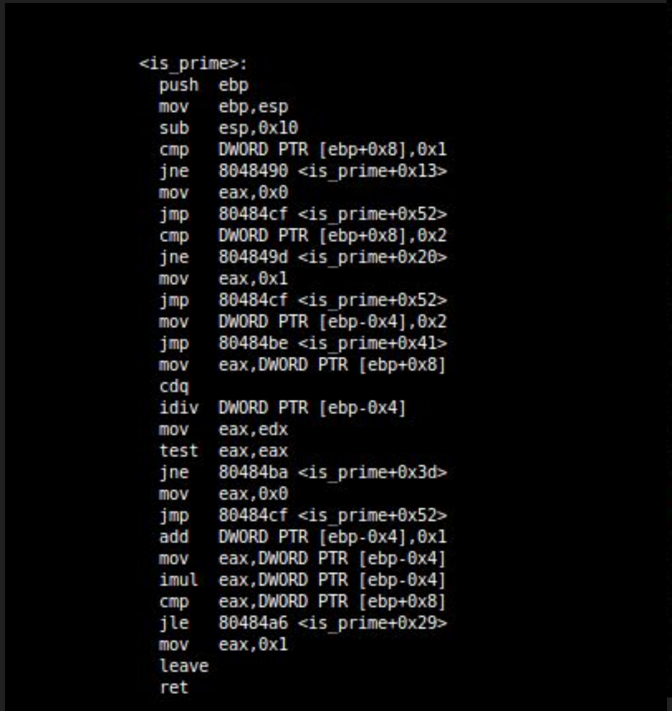
```
Ex: mov eax, [rbp]
```

Fun side note!

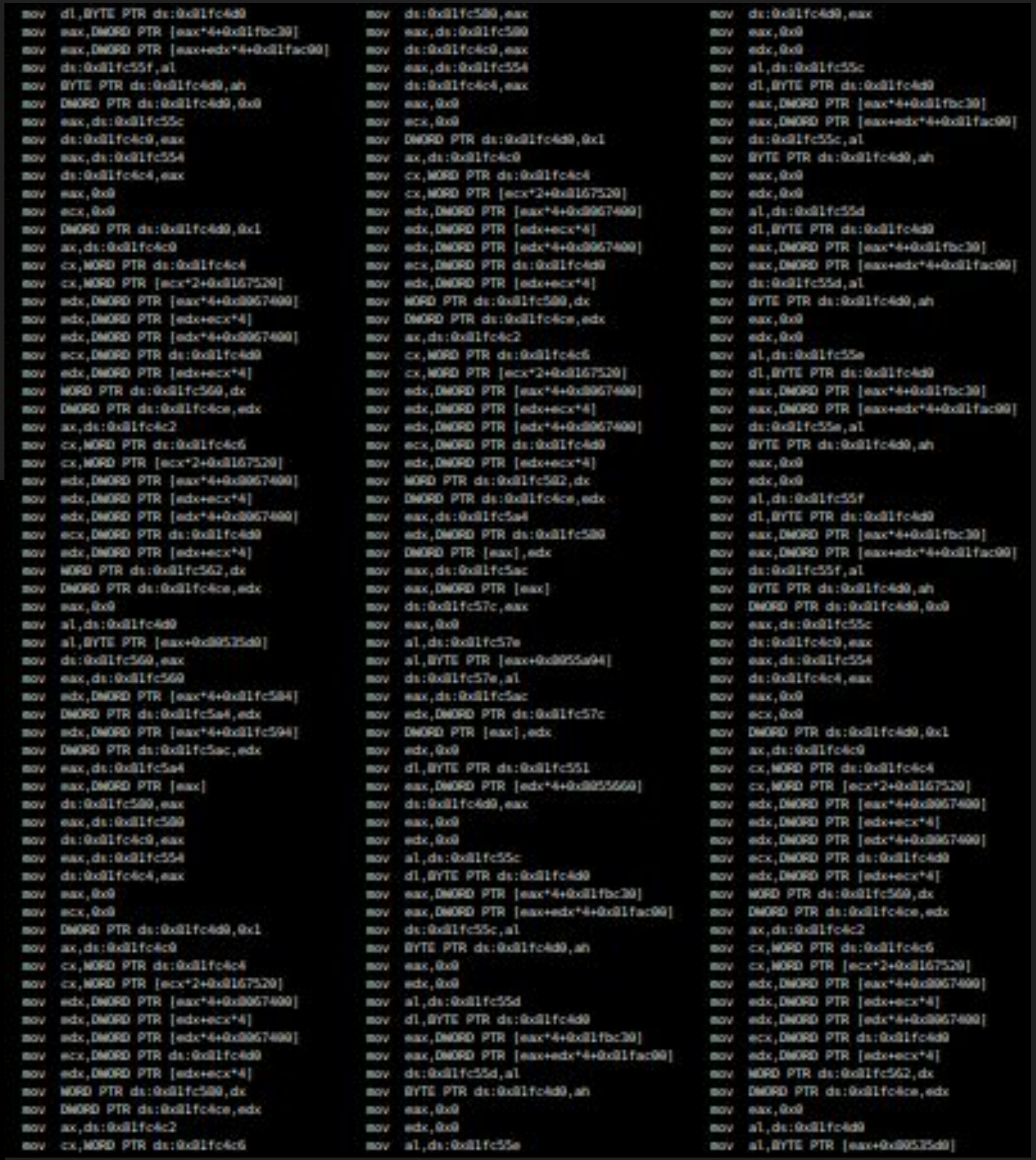
The mov operator is Turing complete!

Movfuscator turns a normal program into only mov instructions

https://github.com/xoreaxeaxeax/movfuscator



```
<is_prime>:
  push   ebp
  mov    ebp,esp
  sub    esp,0x10
  cmp    DWORD PTR [ebp+0x8],0x1
  jne    8048490 <is_prime+0x13>
  mov    eax,0x0
  jmp    80484cf <is_prime+0x52>
  cmp    DWORD PTR [ebp+0x8],0x2
  jne    804849d <is_prime+0x20>
  mov    eax,0x1
  jmp    80484cf <is_prime+0x52>
  mov    DWORD PTR [ebp-0x4],0x2
  jmp    80484be <is_prime+0x41>
  mov    eax,DWORD PTR [ebp+0x8]
  cdq
  idiv   DWORD PTR [ebp-0x4]
  mov    eax,edx
  test   eax,eax
  jne    80484ba <is_prime+0x3d>
  mov    eax,0x0
  jmp    80484cf <is_prime+0x52>
  add    DWORD PTR [ebp-0x4],0x1
  mov    eax,DWORD PTR [ebp-0x4]
  imul   eax,DWORD PTR [ebp-0x4]
  cmp    eax,DWORD PTR [ebp+0x8]
  jle    80484a6 <is_prime+0x29>
  mov    eax,0x1
  leave
  ret
```



```
mov    dl,BYTE PTR ds:0x81fc4d0
mov    eax,DWORD PTR [eax*4+0x81fbc30]
mov    eax,DWORD PTR [eax+edx*4+0x81fac00]
mov    ds:0x81fc55f,al
mov    BYTE PTR ds:0x81fc4d0,ah
mov    DWORD PTR ds:0x81fc4d0,0x0
mov    eax,ds:0x81fc55c
mov    ds:0x81fc4c0,eax
mov    eax,ds:0x81fc554
mov    ds:0x81fc4c4,eax
mov    eax,0x0
mov    ecx,0x0
mov    DWORD PTR ds:0x81fc4d0,0x1
mov    ax,ds:0x81fc4c0
mov    cx,WORD PTR ds:0x81fc4c4
mov    cx,WORD PTR [ecx*2+0x8167520]
mov    edx,DWORD PTR [eax*4+0x8067400]
mov    edx,DWORD PTR [edx+ecx*4]
mov    edx,DWORD PTR [edx*4+0x8067400]
mov    ecx,DWORD PTR ds:0x81fc4d0
mov    edx,DWORD PTR [edx+ecx*4]
mov    WORD PTR ds:0x81fc560,dx
mov    DWORD PTR ds:0x81fc4ce,edx
mov    ax,ds:0x81fc4c2
mov    cx,WORD PTR ds:0x81fc4c6
mov    cx,WORD PTR [ecx*2+0x8167520]
mov    edx,DWORD PTR [eax*4+0x8067400]
mov    edx,DWORD PTR [edx+ecx*4]
mov    edx,DWORD PTR [edx*4+0x8067400]
mov    ecx,DWORD PTR ds:0x81fc4d0
mov    edx,DWORD PTR [edx+ecx*4]
mov    WORD PTR ds:0x81fc562,dx
mov    DWORD PTR ds:0x81fc4ce,edx
mov    eax,0x0
mov    al,ds:0x81fc4d0
mov    al,BYTE PTR [eax+0x80535d0]
mov    ds:0x81fc560,eax
mov    eax,ds:0x81fc560
mov    edx,DWORD PTR [eax*4+0x81fc584]
mov    DWORD PTR ds:0x81fc5a4,edx
mov    edx,DWORD PTR [eax*4+0x81fc594]
mov    DWORD PTR ds:0x81fc5ac,edx
mov    eax,ds:0x81fc5a4
mov    eax,DWORD PTR [eax]
mov    ds:0x81fc580,eax
mov    eax,ds:0x81fc580
mov    ds:0x81fc4c0,eax
mov    eax,ds:0x81fc554
mov    ds:0x81fc4c4,eax
mov    eax,0x0
mov    ecx,0x0
mov    DWORD PTR ds:0x81fc4d0,0x1
mov    ax,ds:0x81fc4c0
mov    cx,WORD PTR ds:0x81fc4c4
mov    cx,WORD PTR [ecx*2+0x8167520]
mov    edx,DWORD PTR [eax*4+0x8067400]
mov    edx,DWORD PTR [edx+ecx*4]
mov    edx,DWORD PTR [edx*4+0x8067400]
mov    ecx,DWORD PTR ds:0x81fc4d0
mov    edx,DWORD PTR [edx+ecx*4]
mov    WORD PTR ds:0x81fc580,dx
mov    DWORD PTR ds:0x81fc4ce,edx
mov    ax,ds:0x81fc4c2
mov    cx,WORD PTR ds:0x81fc4c6
```

```
mov    ds:0x81fc580,eax
mov    eax,ds:0x81fc580
mov    ds:0x81fc4c0,eax
mov    eax,ds:0x81fc554
mov    ds:0x81fc4c4,eax
mov    eax,0x0
mov    ecx,0x0
mov    DWORD PTR ds:0x81fc4d0,0x1
mov    ax,ds:0x81fc4c0
mov    cx,WORD PTR ds:0x81fc4c4
mov    cx,WORD PTR [ecx*2+0x8167520]
mov    edx,DWORD PTR [eax*4+0x8067400]
mov    edx,DWORD PTR [edx+ecx*4]
mov    edx,DWORD PTR [edx*4+0x8067400]
mov    ecx,DWORD PTR ds:0x81fc4d0
mov    edx,DWORD PTR [edx+ecx*4]
mov    WORD PTR ds:0x81fc580,dx
mov    DWORD PTR ds:0x81fc4ce,edx
mov    ax,ds:0x81fc4c2
mov    cx,WORD PTR ds:0x81fc4c6
mov    cx,WORD PTR [ecx*2+0x8167520]
mov    edx,DWORD PTR [eax*4+0x8067400]
mov    edx,DWORD PTR [edx+ecx*4]
mov    edx,DWORD PTR [edx*4+0x8067400]
mov    ecx,DWORD PTR ds:0x81fc4d0
mov    edx,DWORD PTR [edx+ecx*4]
mov    WORD PTR ds:0x81fc582,dx
mov    DWORD PTR ds:0x81fc4ce,edx
mov    eax,ds:0x81fc5a4
mov    edx,DWORD PTR ds:0x81fc580
mov    DWORD PTR [eax],edx
mov    eax,ds:0x81fc5ac
mov    eax,DWORD PTR [eax]
mov    ds:0x81fc57c,eax
mov    eax,0x0
mov    al,ds:0x81fc57e
mov    al,BYTE PTR [eax+0x8055a94]
mov    ds:0x81fc57e,al
mov    eax,ds:0x81fc5ac
mov    edx,DWORD PTR ds:0x81fc57c
mov    DWORD PTR [eax],edx
mov    edx,0x0
mov    dl,BYTE PTR ds:0x81fc551
mov    eax,DWORD PTR [edx*4+0x8055660]
mov    ds:0x81fc4d0,eax
mov    eax,0x0
mov    edx,0x0
mov    al,ds:0x81fc55c
mov    dl,BYTE PTR ds:0x81fc4d0
mov    eax,DWORD PTR [eax*4+0x81fbc30]
mov    eax,DWORD PTR [eax+edx*4+0x81fac00]
mov    ds:0x81fc55c,al
mov    BYTE PTR ds:0x81fc4d0,ah
mov    eax,0x0
mov    edx,0x0
mov    al,ds:0x81fc55d
mov    dl,BYTE PTR ds:0x81fc4d0
mov    eax,DWORD PTR [eax*4+0x81fbc30]
mov    eax,DWORD PTR [eax+edx*4+0x81fac00]
mov    ds:0x81fc55d,al
mov    BYTE PTR ds:0x81fc4d0,ah
mov    eax,0x0
mov    edx,0x0
mov    al,ds:0x81fc55e
```

```
mov    ds:0x81fc4d0,eax
mov    eax,0x0
mov    edx,0x0
mov    al,ds:0x81fc55c
mov    dl,BYTE PTR ds:0x81fc4d0
mov    eax,DWORD PTR [eax*4+0x81fbc30]
mov    eax,DWORD PTR [eax+edx*4+0x81fac00]
mov    ds:0x81fc55c,al
mov    BYTE PTR ds:0x81fc4d0,ah
mov    eax,0x0
mov    edx,0x0
mov    al,ds:0x81fc55d
mov    dl,BYTE PTR ds:0x81fc4d0
mov    eax,DWORD PTR [eax*4+0x81fbc30]
mov    eax,DWORD PTR [eax+edx*4+0x81fac00]
mov    ds:0x81fc55d,al
mov    BYTE PTR ds:0x81fc4d0,ah
mov    eax,0x0
mov    edx,0x0
mov    al,ds:0x81fc55e
mov    dl,BYTE PTR ds:0x81fc4d0
mov    eax,DWORD PTR [eax*4+0x81fbc30]
mov    eax,DWORD PTR [eax+edx*4+0x81fac00]
mov    ds:0x81fc55e,al
mov    BYTE PTR ds:0x81fc4d0,ah
mov    eax,0x0
mov    edx,0x0
mov    al,ds:0x81fc55f
mov    dl,BYTE PTR ds:0x81fc4d0
mov    eax,DWORD PTR [eax*4+0x81fbc30]
mov    eax,DWORD PTR [eax+edx*4+0x81fac00]
mov    ds:0x81fc55f,al
mov    BYTE PTR ds:0x81fc4d0,ah
mov    DWORD PTR ds:0x81fc4d0,0x0
mov    eax,ds:0x81fc55c
mov    ds:0x81fc4c0,eax
mov    eax,ds:0x81fc554
mov    ds:0x81fc4c4,eax
mov    eax,0x0
mov    ecx,0x0
mov    DWORD PTR ds:0x81fc4d0,0x1
mov    ax,ds:0x81fc4c0
mov    cx,WORD PTR ds:0x81fc4c4
mov    cx,WORD PTR [ecx*2+0x8167520]
mov    edx,DWORD PTR [eax*4+0x8067400]
mov    edx,DWORD PTR [edx+ecx*4]
mov    edx,DWORD PTR [edx*4+0x8067400]
mov    ecx,DWORD PTR ds:0x81fc4d0
mov    edx,DWORD PTR [edx+ecx*4]
mov    WORD PTR ds:0x81fc560,dx
mov    DWORD PTR ds:0x81fc4ce,edx
mov    ax,ds:0x81fc4c2
mov    cx,WORD PTR ds:0x81fc4c6
mov    cx,WORD PTR [ecx*2+0x8167520]
mov    edx,DWORD PTR [eax*4+0x8067400]
mov    edx,DWORD PTR [edx+ecx*4]
mov    edx,DWORD PTR [edx*4+0x8067400]
mov    ecx,DWORD PTR ds:0x81fc4d0
mov    edx,DWORD PTR [edx+ecx*4]
mov    WORD PTR ds:0x81fc562,dx
mov    DWORD PTR ds:0x81fc4ce,edx
mov    eax,0x0
mov    al,ds:0x81fc4d0
mov    al,BYTE PTR [eax+0x80535d0]
```

ASM Review, arithmetic:

Operations like add and subtract only take one source and apply the function onto the destination

Ex:
```
add        eax, 0x1
add        eax, ebx
```

There are other instructions that do bitwise math such as and, or, xor etc.

ASM Review, jumps:

ASM has jump instructions that can ‘jump’ to other blocks of code depending on conditionals

```
Ex.   cmp        rax, rbx
      jne        main+39
      jmp        main+43
```

The “cmp” opcode and addition opcodes set flags which the jump opcodes check
Ex: Zero flag, signed flag, carry flag

Quick Detour, Virtualization:

- Modern computers have multiple cores and lots of memory

- Programs need to have isolated memory regions else they run the risk of overwriting shared locations

- Direct Memory Access Remapping (DMAR sometimes called IOMMU) allows us to remap memory requests of processes to physical hardware addresses, and the process is none the wiser

Today’s Recap:

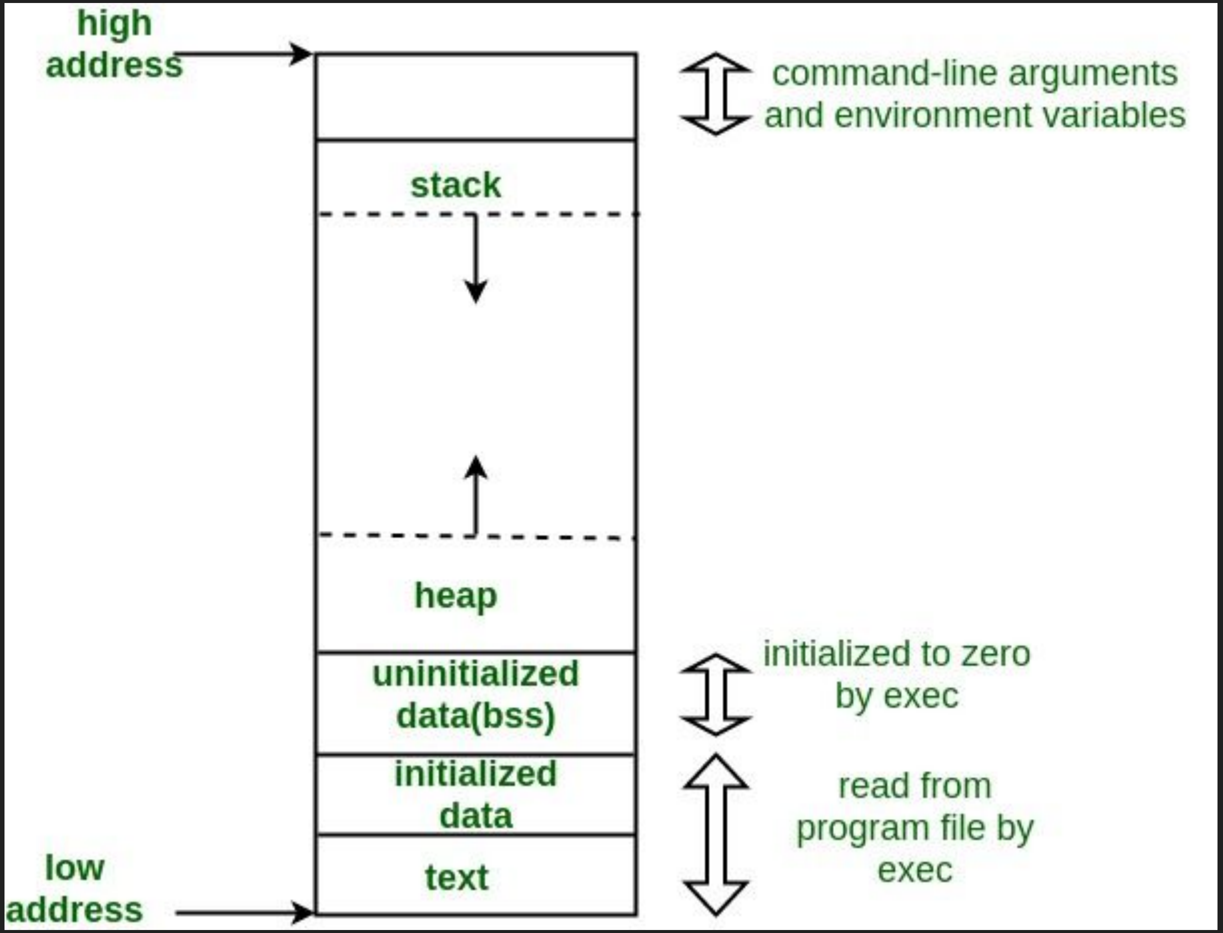
1. Project 0 is out and due Tuesday at midnight

1. Homework 1 will come out later tonight or tomorrow and due Thursday

1. Just got gradescope access, will be up soon, and remember to join Campuswire

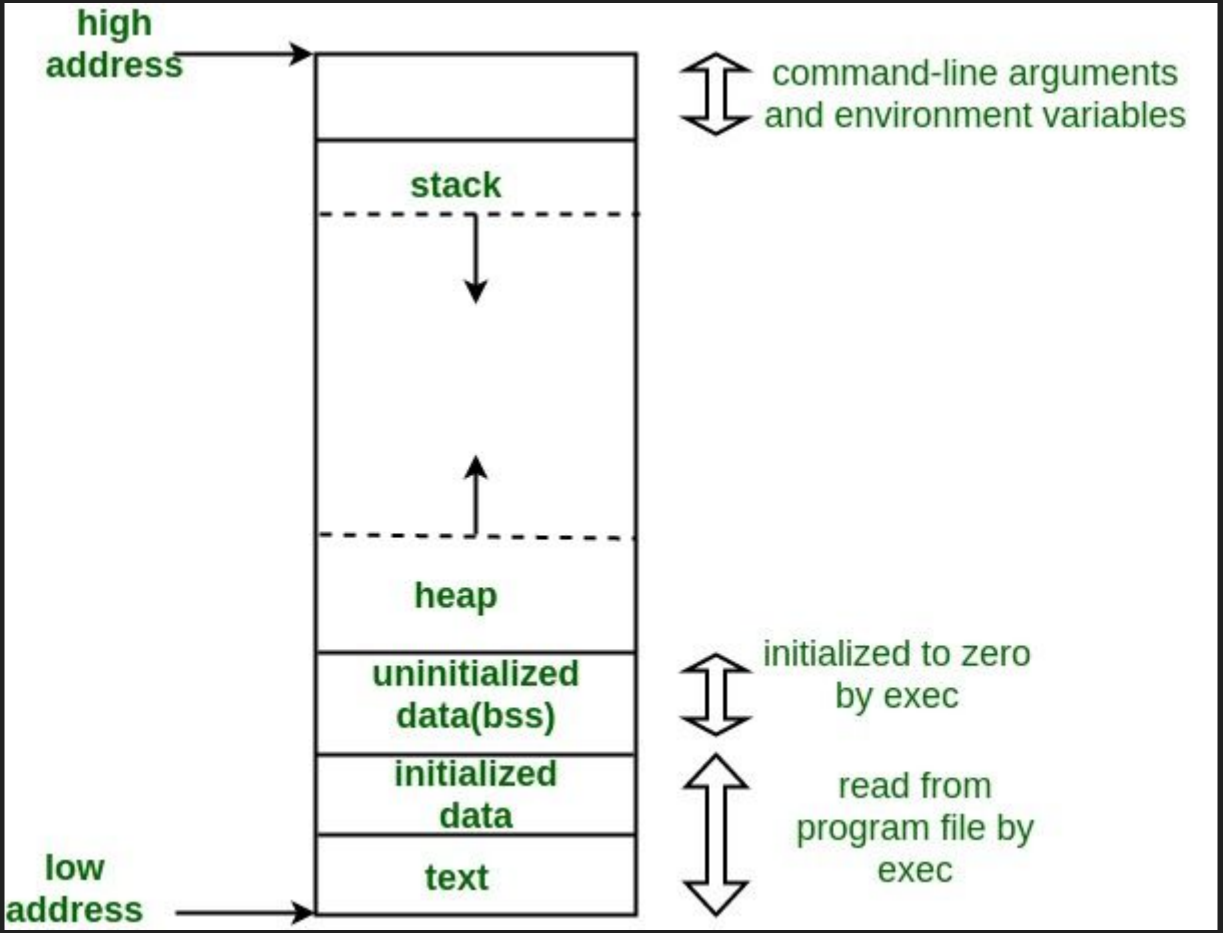
1. Office hours have been listed on Campuswire and start today right after class

Virtualized Memory Layout


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

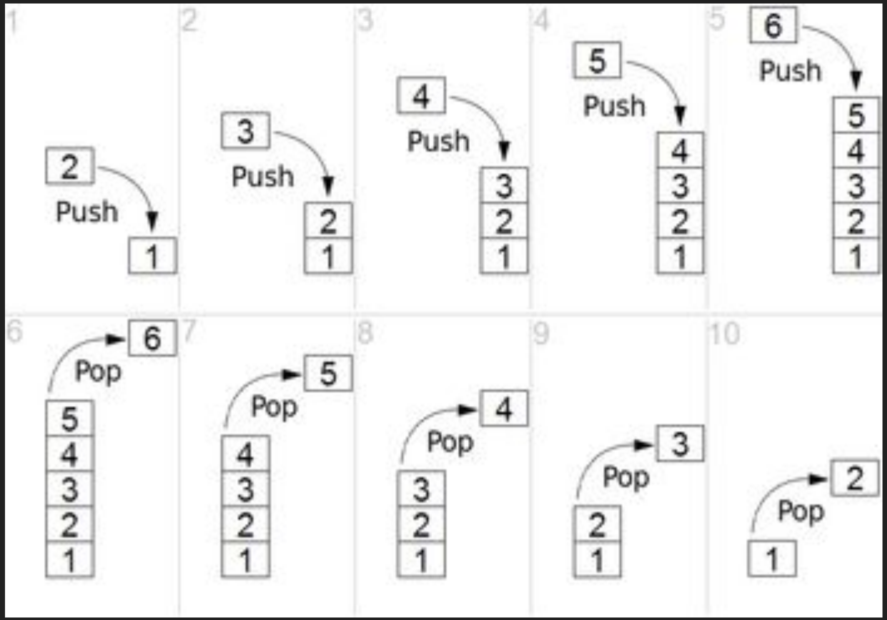
- Programs are loaded into consistent memory locations allowing for generalized assembly programing
    - Ex: All program entries should be loaded to the same virtualized memory location (kind of, we don’t do this anymore for security reasons)
- Stack and Heap are originally uninitialized data, and the stack grows *down* while the heap grows up
- The sections mentioned previously in the ELF file are loaded in at the bottom of the allocated memory

Virtualized Memory Layout (cont.)


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

- Kernel memory is located above user space memory, inaccessible to the normal userland program execution
- Memory ranges have permissions locking what can be done.
  - Falls under Read\Write\eXecute permissions, can have multiple at once
- Heap data doesn’t exist unless the program specifically calls for it.

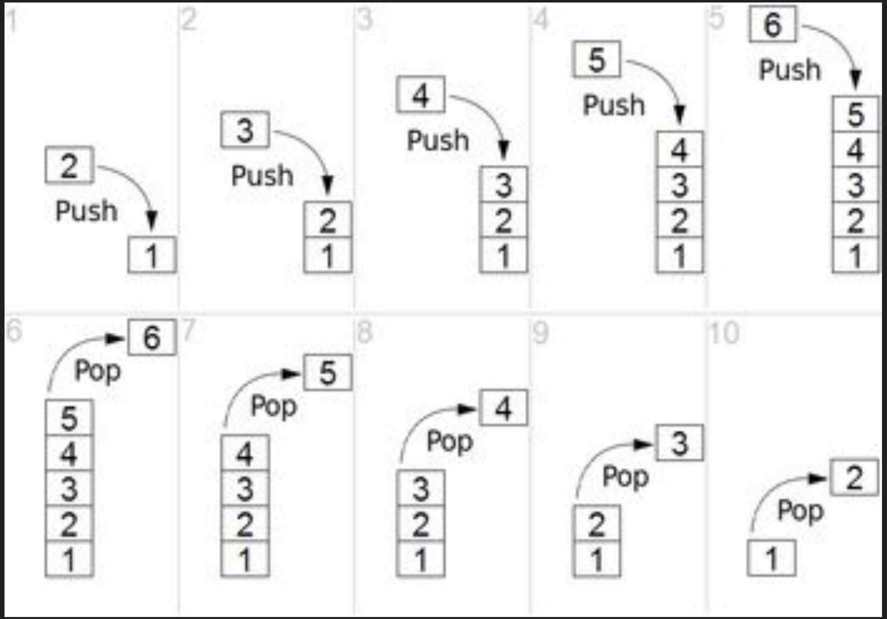
Call Stack and x86-64 Calling Convention



What is a stack again?

- A Stack is a data structure where data must be read in order of top bottom
- Can “push” data to the top of the stack
- To read a value in the center, we must “pop” all the data above it

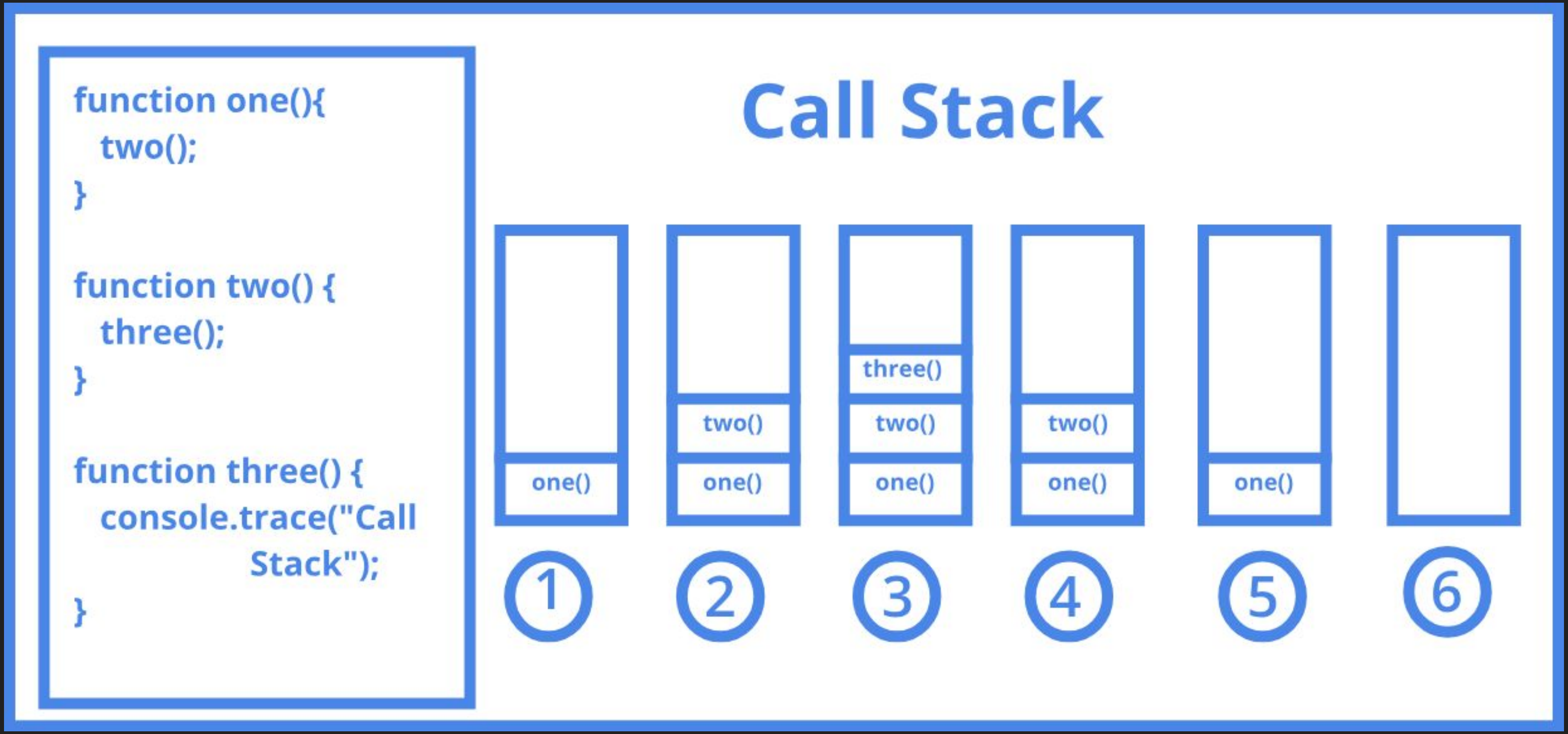
Call Stack and x86-64 Calling Convention



How do computers use stack?

- All local variables and execution context for a function are stored on a stack!
- Not only does each function have a stack, but there is a “call stack” of functions

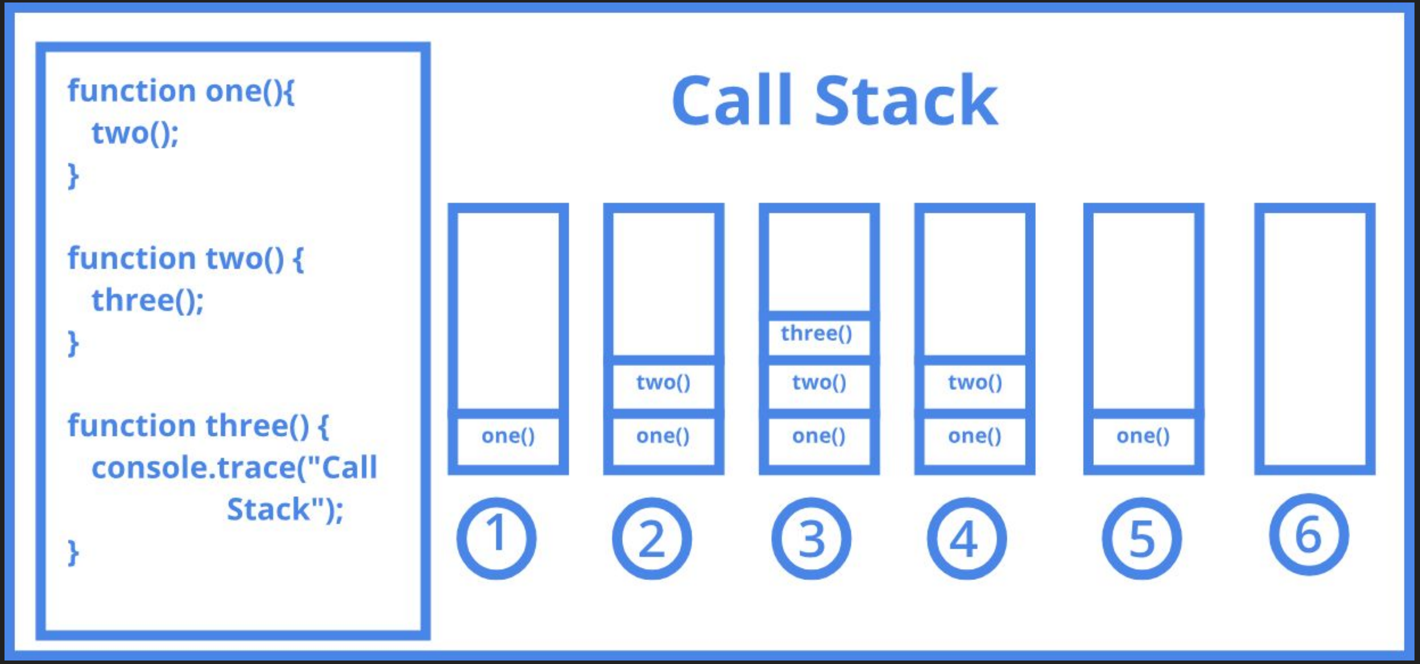
Call Stack and x86-64 Calling Convention



```
function one(){
  two();
}

function two() {
  three();
}

function three() {
  console.trace("Call
            Stack");
}
```

This stack should really be upside down but the same idea holds

Call Stack and x86-64 Calling Convention



```
function one(){
  two();
}

function two() {
  three();
}

function three() {
  console.trace("Call
            Stack");
}
```

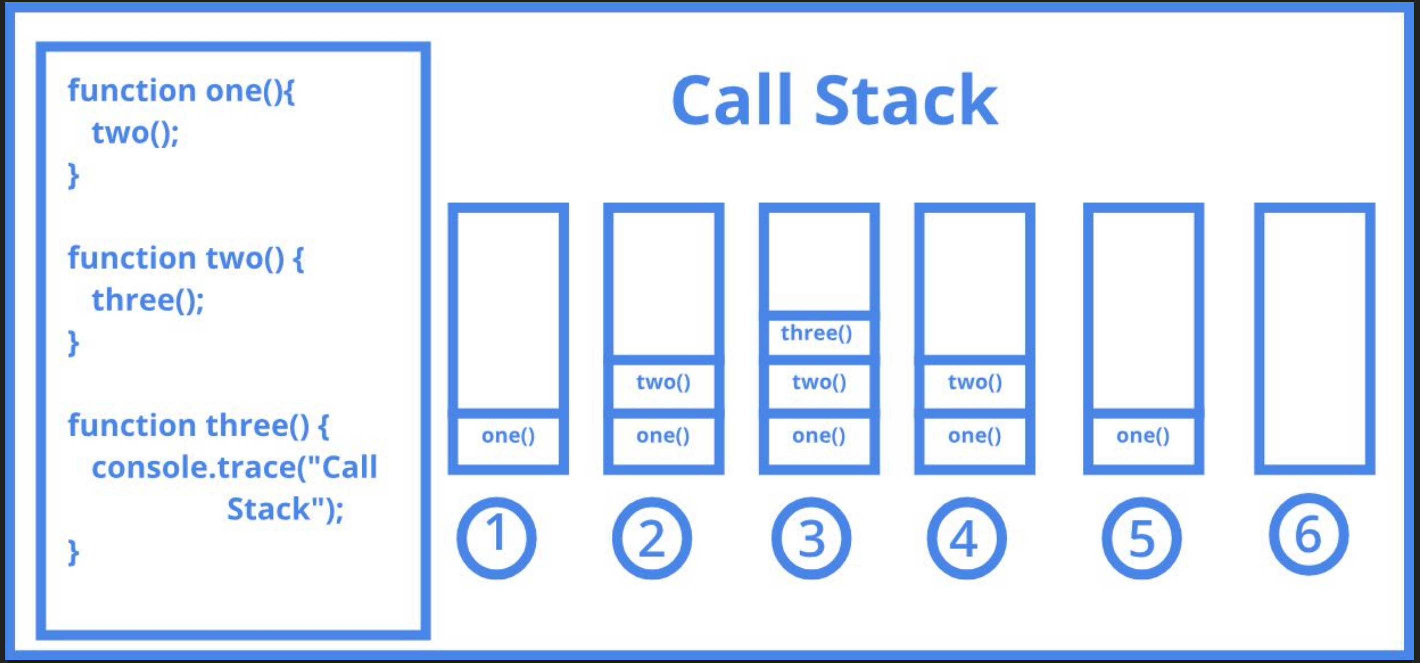
This stack should really be upside down but the same idea holds

When a new function is called, we create a new function stack at the top, and pop it off when we are done

All programs are linear like this, you must pop off the top function before revisiting the data contained in the previous functions.

If you want data from other functions, it must be a global and thus not on the stack (.bss or .data section)

Call Stack and x86-64 Calling Convention



```
function one(){
  two();
}

function two() {
  three();
}

function three() {
  console.trace("Call
            Stack");
}
```

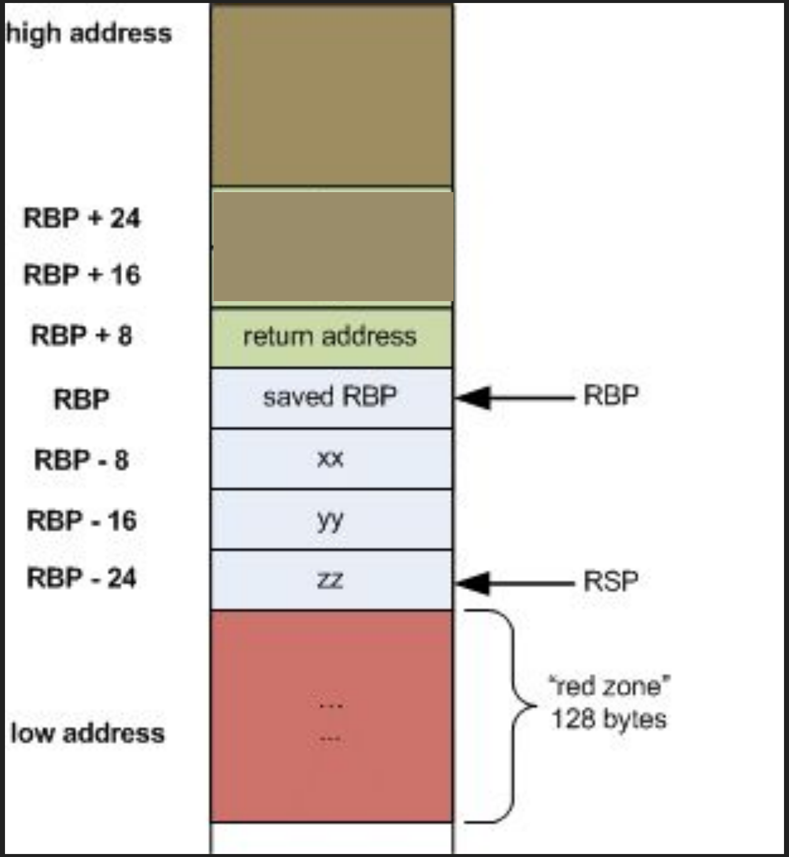
This stack should really be upside down but the same idea holds

So how do we accomplish this?

- RBP - The Base Pointer
  - Keeps track of the "base" of the stack
  - Points to a value which contains the base pointer for the previous function
- RSP - The Stack Pointer
  - Keeps track of the top of the stack
  - Used by the "push" and "pop" opcodes to know what data we want to read/write
- Call Opcode
  - Pushes the return address of function to we previously were in

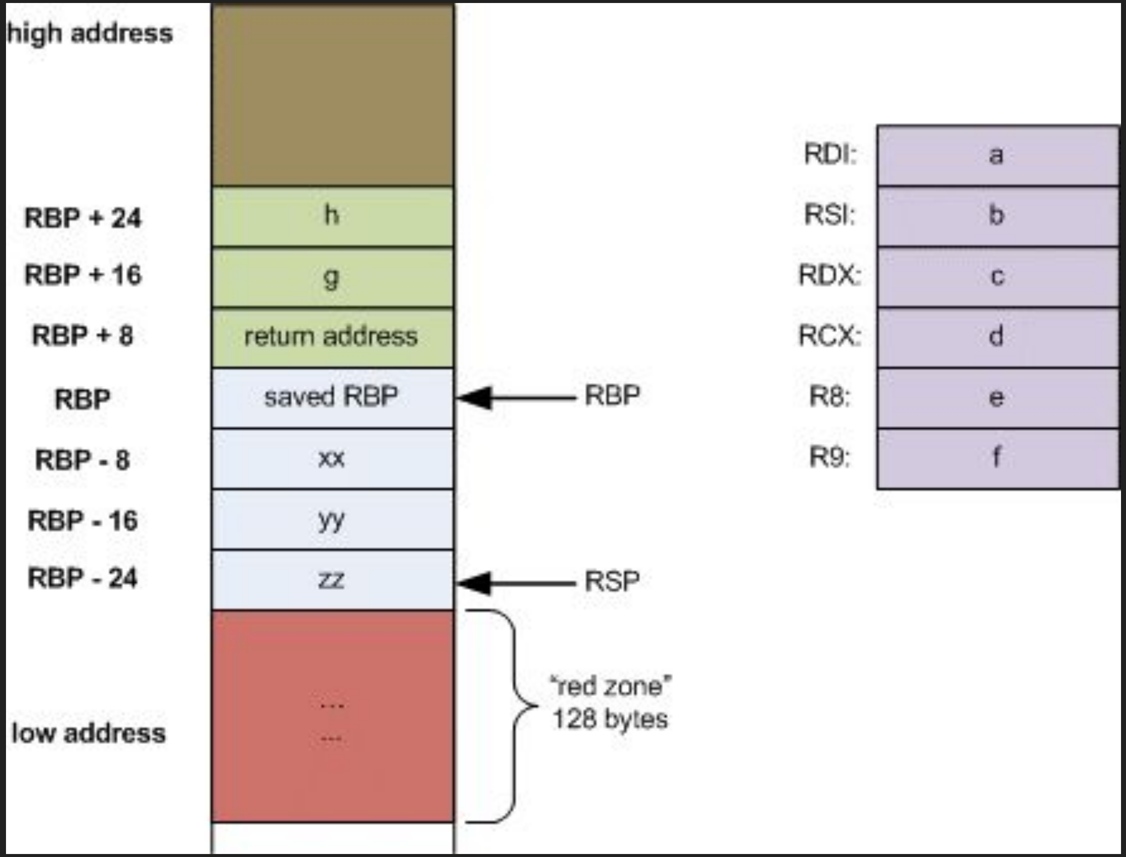
Function Stacks

Let’s look setting up a Function Stack



- Move parameters to appropriate registers
- First call the function with the “call” opcode
  - This pushed the return address to continue the code of the previous function when we are done with the new function
  - Sets RIP register (instruction register) to code of new function
- Push the base pointer on the stack
  - This lets us know where the old base pointer is after finishing the functions execution, this happens in the new functions opcodes
- Set stack pointer to the base pointer
  - Sets the stack pointer to the base of the current function stack
- Subtract from the stack pointer
  - This makes space on the stack for local variables, (needed if we are to call another function before returning else we don’t bother as we assume the memory below RBP is trash)

Call Stack and x86-64 Calling Convention



Individual Function Stack Frame:

- Function arguments are stored in registers,
  - RDI, RSI, RDX, RCX, R8, R9, additional arguments are passed at the top of the stack frame

- Return Address is the first value pushed onto the stack, it is the address of the code to jump to after returning in this function

- RBP (Base Pointer) points to the base of the current stack frame, and the data at the base is the previous functions base

- RSP (Stack Pointer) points to the top of the current stack frame (used to index pops and pushes)

Call Stack and x86-64 Calling Convention

What happens when we call a function?

- Opcode “call” is used, pushes the return address onto the stack
- Opcode “push” is used, pushing RBP onto the stack
- Opcode “mov” is used, moving RSP to RBP
- This sets up the new stack frame for our next function

When we want to exit a function

- Set RAX register to the desired return value
- Opcode “mov” or “leave” is used, sets RSP to RBP
- Opcode “pop” is used, pops saved base pointer into RBP
- Opcode “ret” is used, pops the return address into RIP, redirects code execution

Call Stack and x86-64 Calling Convention

What happens when we call a function?

- Opcode “call” is used, pushes the return address onto the stack
- Opcode “push” is used, pushing RBP onto the stack
- Opcode “mov” is used, moving RSP to RBP
- This sets up the new stack frame for our next function

When we want to exit a function

- Set RAX register to the desired return value
- Opcode “mov” or “leave” is used, sets RSP to RBP
- Opcode “pop” is used, pops saved base pointer into RBP
- Opcode “ret” is used, pops the return address into RIP, redirects code execution

ELF File Format

File Types:

Each file type starts with what is called a “magic header”

The first few bytes in a file denote what type of file it is and how the OS should parse the following data

“7F 45 4C 46” corresponds to an ELF file

“89 50 4E 47 0D 0A 1A 0A” is the start of a PNG

“FF D8 FF E0” is a JPEG

Programs often use the name of a file (“.txt”) to determine file type, but this is bad practice and can lead to parsing issues

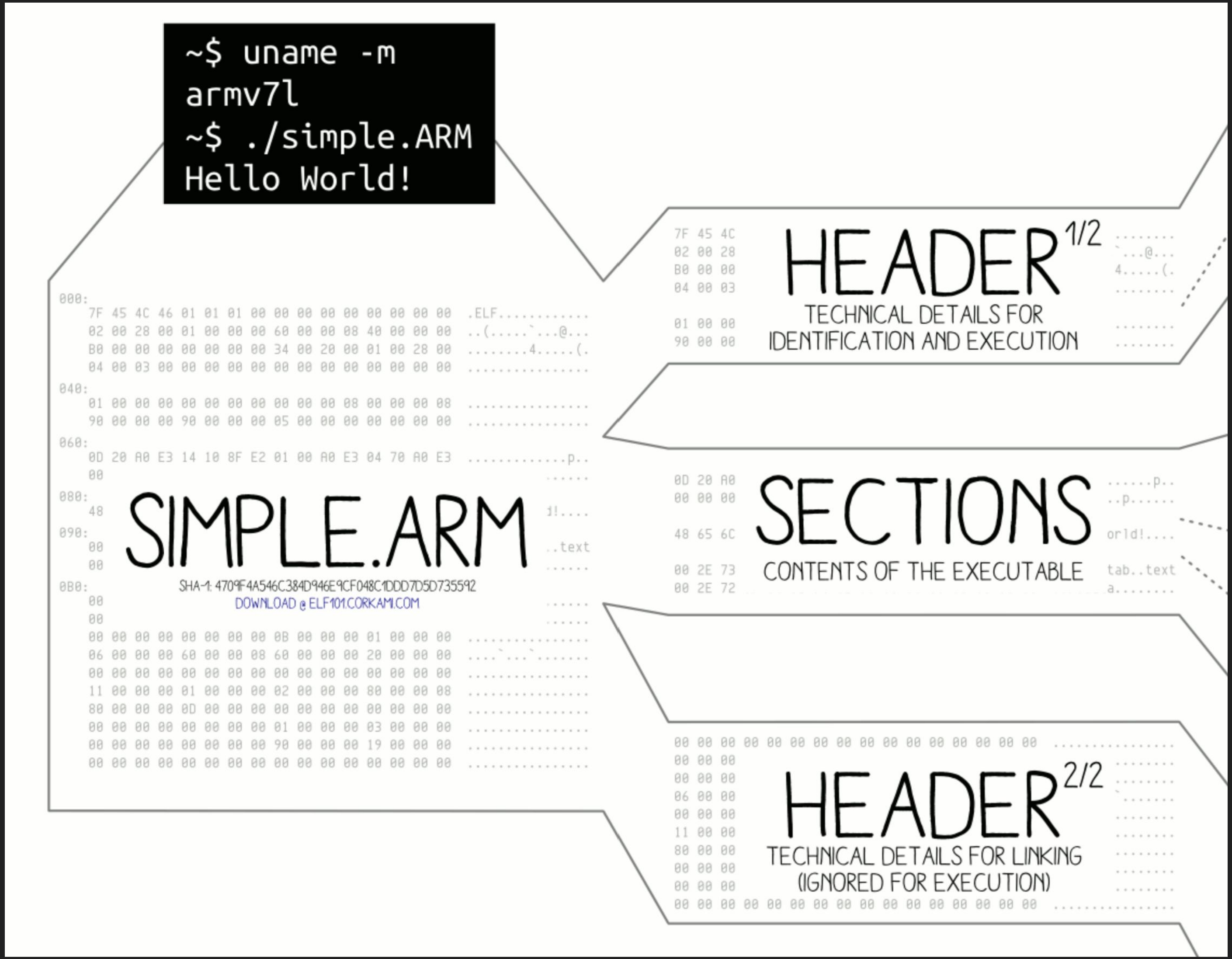
Our Target Program (ELF):

We will be attacking Executable and Linkable Format (ELF) files, they are the executable files for Unix operating systems

What is the purpose of compiling our code

Computers can’t understand our code, so we must compile to an intermediate executable file that the computer can parse

ELF Structure:



```
~$ uname -m
armv7l
~$ ./simple.ARM
Hello World!
```

```
000:
    7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF............
    02 00 28 00 01 00 00 00 60 00 00 08 40 00 00 00  ..(.....`...@...
    B0 00 00 00 00 00 00 00 34 00 20 00 01 00 28 00  ........4.....(.
    04 00 03 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
040:
    01 00 00 00 00 00 00 00 00 00 00 08 00 00 00 08  ................
    90 00 00 00 90 00 00 00 05 00 00 00 00 00 00 00  ................
060:
    0D 20 A0 E3 14 10 8F E2 01 00 A0 E3 04 70 A0 E3  .............p..
    00                                                          ......
080:
    48                                                          d!....
090:
    00                                                          ..text
    00                                                          ......
0B0:
    00                                                          ......
    00                                                          ......
    00 00 00 00 00 00 00 00 0B 00 00 00 01 00 00 00  ................
    06 00 00 00 60 00 00 08 60 00 00 00 20 00 00 00  ....`...`.......
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
    11 00 00 00 01 00 00 00 02 00 00 00 80 00 00 08  ................
    80 00 00 00 0D 00 00 00 00 00 00 00 00 00 00 00  ................
    00 00 00 00 00 00 00 00 01 00 00 00 03 00 00 00  ................
    00 00 00 00 00 00 00 00 90 00 00 00 19 00 00 00  ................
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

SIMPLE.ARM
SHA-1: 4709F4A546C384D946E9CF048C1DDD7D5D735592
DOWNLOAD @ ELF101.CORKAMI.COM
HEADER 1/2
TECHNICAL DETAILS FOR IDENTIFICATION AND EXECUTION
SECTIONS
CONTENTS OF THE EXECUTABLE
HEADER 2/2
TECHNICAL DETAILS FOR LINKING (IGNORED FOR EXECUTION)

ELF File Header:

- Contains information such as 32/64-bit, endianness, target system, and other compilation details

- Contains pointers to other objects in the ELF file such as;
  - Code entry point
  - Offsets to other headers in the file
  - Number of sections in the section header table

- Can use command: readelf --file---header, to view the file header

ELF Sections:

- There are multiple types of sections, each corresponding to a different type of information
  - .text section is the actual code of the executable
  - .bss is uninitialized read/write data (global/static variables)
    - This is zeroed out on program start
  - .data is initialized read/write data (global/static variables)
- These sections are loaded into memory during execution
- Can use the command: readelf --sections, to view ELF sections

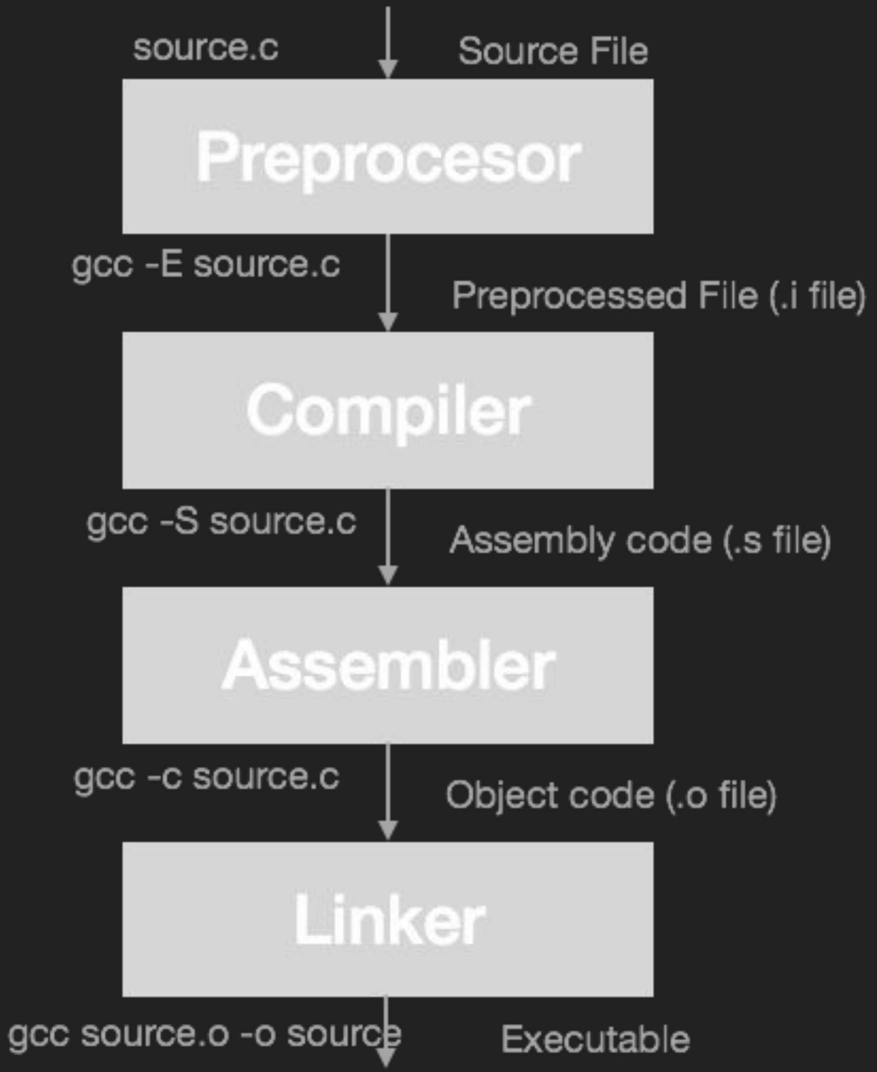
ELF Execution:

When you run an ELF file, the computer first parses the headers for the preprocessing data, then loads all sections into memory

Then you jump to the entry point of the file, which is found in the header information, and then you run the assembly

If memory can be corrupted through a vulnerability, control flow and data in memory can be manipulated

Compilation:



1) We start by taking our written C code

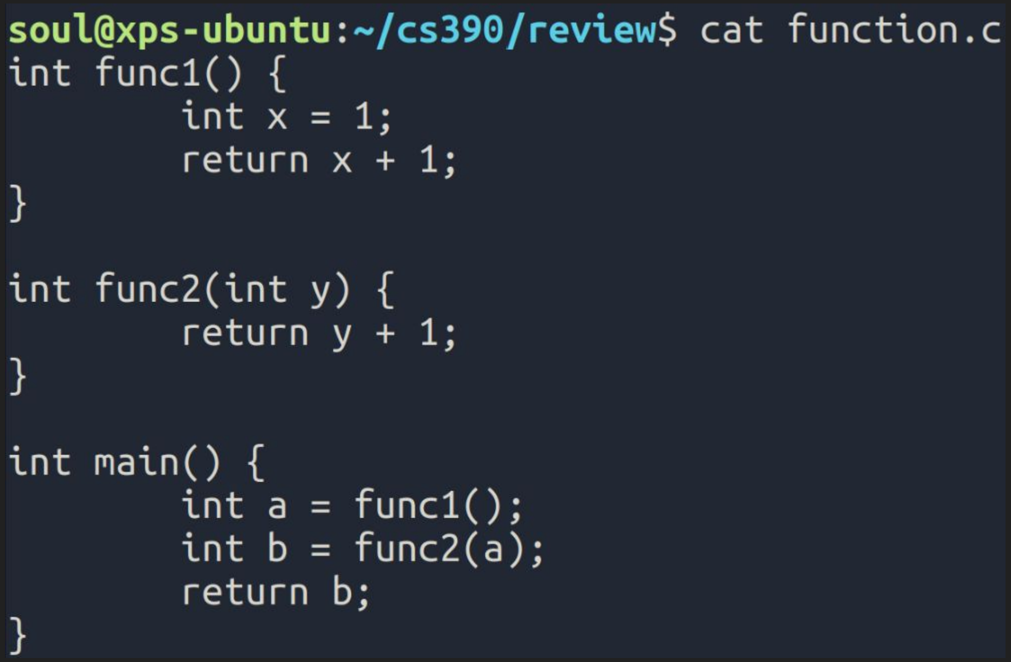
2) The preprocessor analyzes for preprocessing such as macros, merging code and header files, etc.

3) Convert the preprocessed C code into assembly instructions

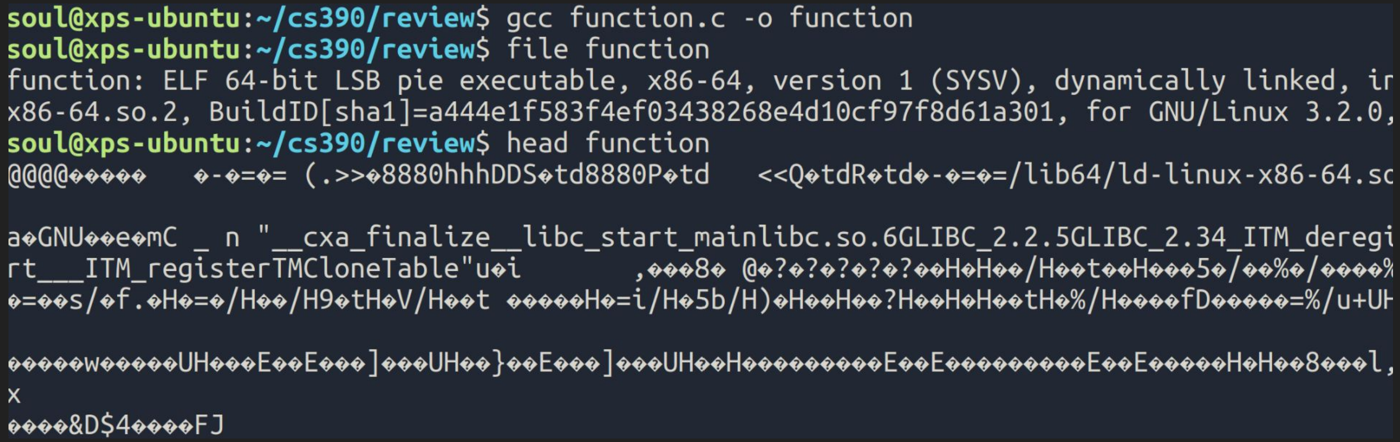
4) Convert assembly into object code, in this case an ELF file

5) Link all the objects to create an executable file

Compilation:



```
soul@xps-ubuntu:~/cs390/review$ cat function.c
int func1() {
        int x = 1;
        return x + 1;
}

int func2(int y) {
        return y + 1;
}

int main() {
        int a = func1();
        int b = func2(a);
        return b;
}
```



```
soul@xps-ubuntu:~/cs390/review$ gcc function.c -o function
soul@xps-ubuntu:~/cs390/review$ file function
function: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, ir
x86-64.so.2, BuildID[sha1]=a444e1f583f4ef03438268e4d10cf97f8d61a301, for GNU/Linux 3.2.0,
soul@xps-ubuntu:~/cs390/review$ head function
@@@@�����   �-�=�= (.>>�8880hhhDDS�td8880P�td   <<Q�tdR�td�-�=�=/lib64/ld-linux-x86-64.sc

a�GNU��e�mC _ n "__cxa_finalize__libc_start_mainlibc.so.6GLIBC_2.2.5GLIBC_2.34_ITM_deregi
rt___ITM_registerTMCloneTable"u�i        ,���8� @�?�?�?�?�?��H�H��/H��t��H���5�/��%�/����%
�=��s/�f.�H�=�/H��/H9�tH�V/H��t �����H�=i/H�5b/H)�H��H��?H��H�H��tH�%/H����fD�����=%/u+UH

�����w�����UH���E��E���]���UH��}��E���]���UH��H���������E��E���������E��E�����H�H��8���l,
x
����&D$4����FJ
```

Some notes on compilation:

- Compilation is a lossy process
  - During each stage of compilation, information from the previous stage is lost (usually, you can set flags during compilation to save some data)
  - This makes reverse engineering harder than just looking at ASM
  - Optimizations can make reverse engineering a simple program unintuitive
- Lots of functions are shared between programs, think puts and printf
  - We can link those functions from shared libraries and not thus not need to store them in every executable (more on this later)

Linking:

Often programs will want to use functions from external libraries (think printf, puts, fgets, etc). There are two ways of doing this

1. Static Linking: During compilation just take the code from the libraries into the file

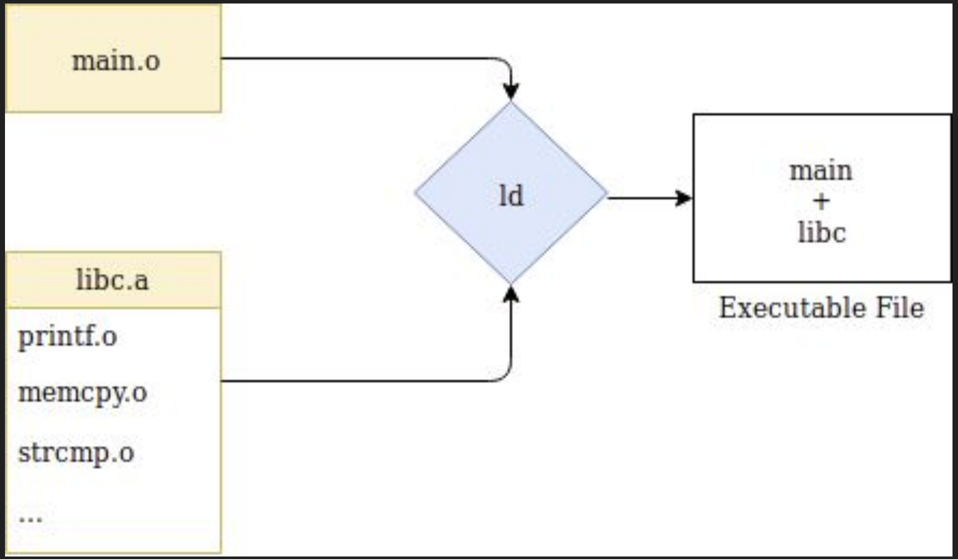
1. Dynamic Linking: During runtime, load the library in memory and just jump to it

Static Linking:

Static linking makes use of the compile time linker “ld” to collect all the necessary code and data from the library and put it in the final executable

Upsides: Binary can be put on any machine with the same architecture and run without the libraries being installed

Downsides: Binaries can become very bloated and end up with reused code in memory when many programs are running

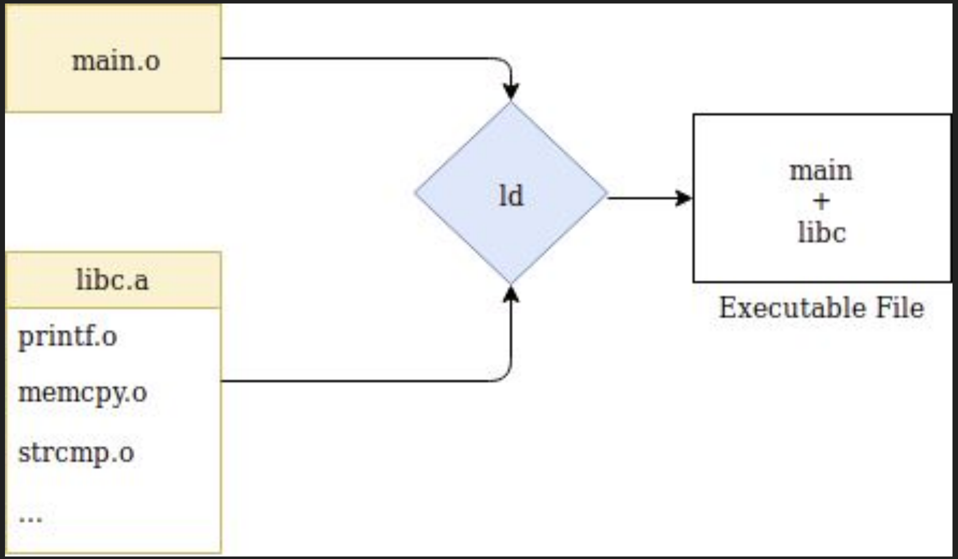


Dynamic Linking:

Dynamic linking makes use of the runtime linker to link to libraries already loaded in memory when program execution starts. Sections such as the .GOT and .PLT in the ELF file are populated with offsets of desired functions

Upsides: Binary is much smaller, and memory is saved as every program doesn’t need repeated printf code

Downsides: The entire library must be on the host machine even if you use one function



How ELFs Start:

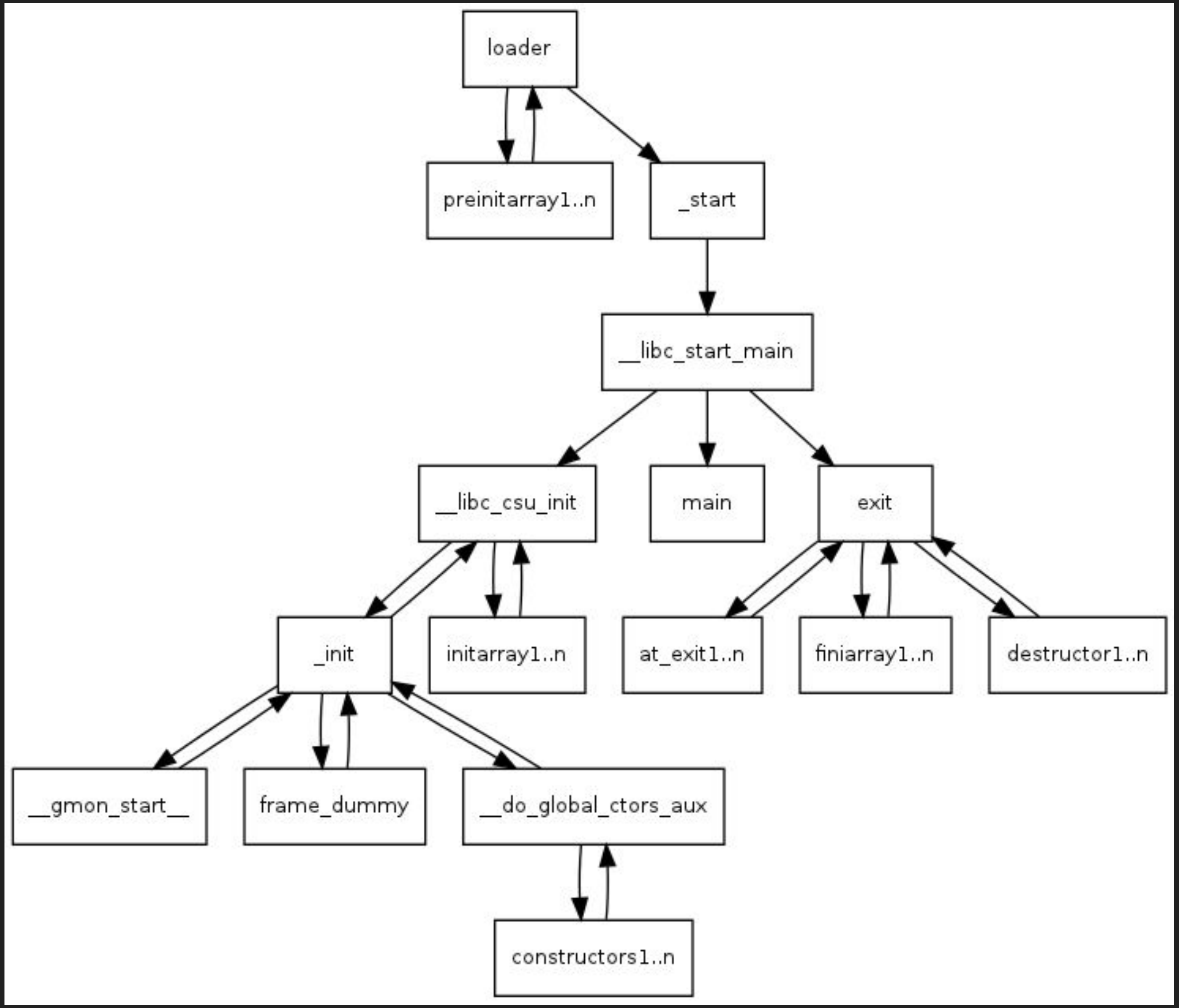
Main isn’t actually the first functions called!

_start is the entry point for the program, and its location is defined in the headers.

_start calls __libs_start_main, with parameters of the main function, the location of the initialization functions, and deconstruction functions (these are segments in the ELF called .init and .fini)

Also does some other stuff like setting up and exiting the dynamic linker, send parameters and environment variables to main, etc.
(all this is very complicated but for good reason)

How ELFs Start:


loader
preinitarray1..n
_start
__libc_start_main
__libc_csu_init
main
exit
_init
initarray1..n
at_exit1..n
finiarray1..n
destructor1..n
__gmon_start__
frame_dummy
__do_global_ctors_aux
constructors1..n