Reverse Engineering 101

Take a peek under the hood!

01

Introduction

What is reversing?

02

Compilers and Assembly

The compilation process and machine code

03

Reversing Basics

Disassembling machine code, tools, and analysis

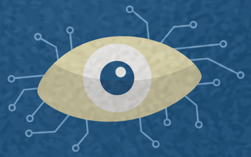
04

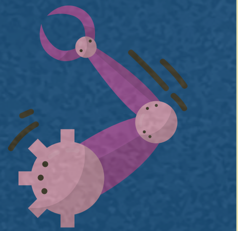
Live Demo

Reversing a compiled executable

INTRODUCTION

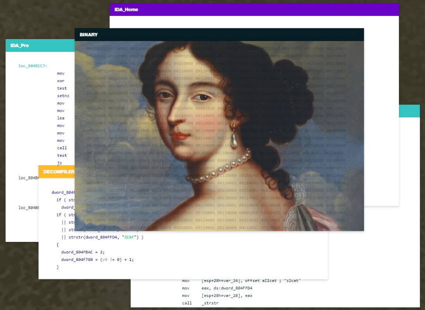
What is reverse engineering?





Reverse Engineering

- The process of analyzing the internals of a piece of software, to figure out how it does what it does
- Various processes and tools for doing so
  - Ghidra, IDA Pro, Radare, etc.
- Static and Dynamic Analysis



Compilers & ASM

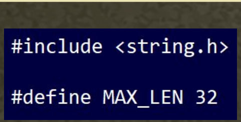
How do processors execute code? How do programming languages compile to executable code?

Compiled Languages

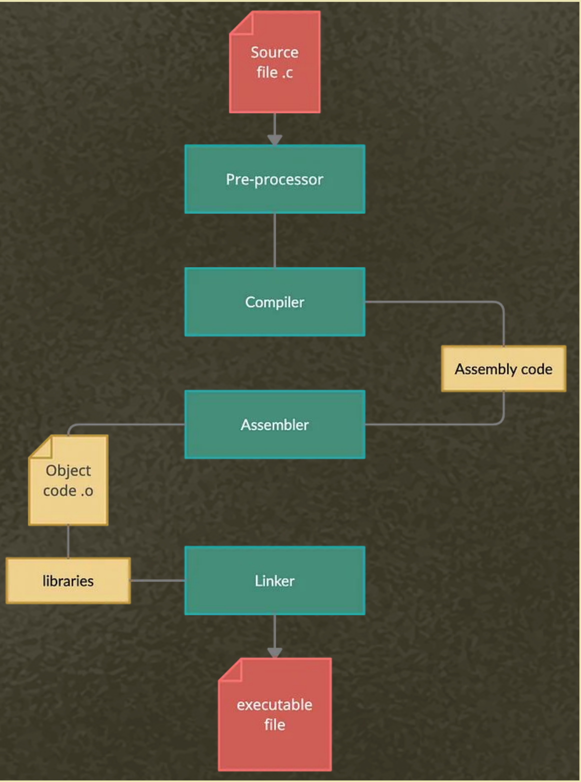
- Some high level languages are compiled into machine code
    - C, C++, Go, Rust
- Machine code is directly interpreted by the processor
    - EXE, DLL, OSX, ELF files contain machine code
- Machine code is composed of instructions that the processor executes
    - mul (multiply), add (add), mov (move), jmp (jump)
- The format and set of instructions is defined by the ISA
    - Instruction Set Architecture

How Does Compilation Work?

- Preprocessing
    - Stripping comments, preprocessor directives
- Compilation
    - AST construction, intermediate representation (IR)
- Assembly
    - From IR, to assembly, to machine code (object files)
- Linking
    - Stitching object files together, adding dynamic library entries

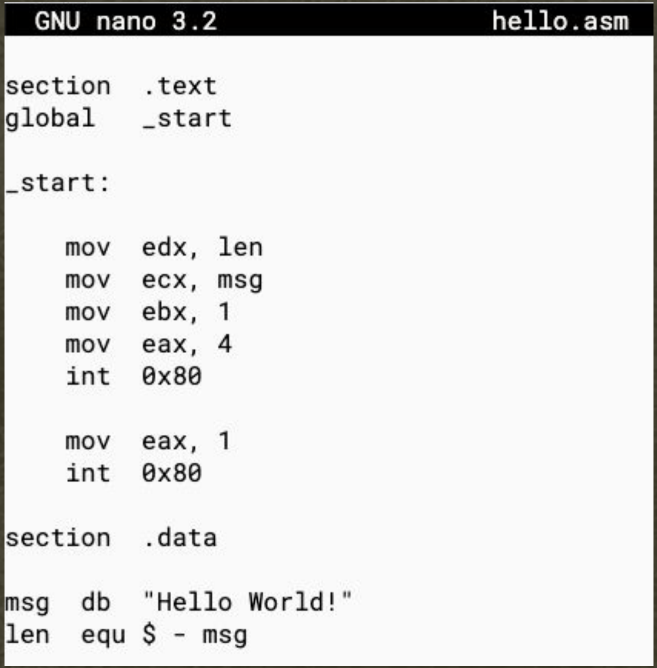


```
#include <string.h>

#define MAX_LEN 32
```



Assembly

- Machine code consists of non-human readable instructions
- Assembly is essentially human-readable machine code
  - An architecture-specific programming language
- x86, ARM, MIPS, RISC-V, etc.



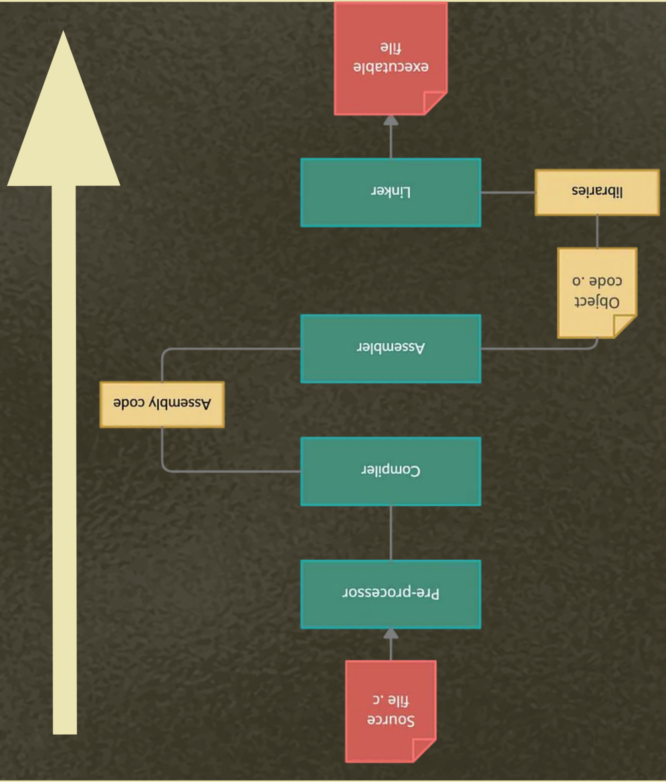
```
section   .text
global    _start

_start:

    mov   edx, len
    mov   ecx, msg
    mov   ebx, 1
    mov   eax, 4
    int   0x80

    mov   eax, 1
    int   0x80

section   .data

msg   db   "Hello World!"
len   equ $ - msg
```

Reversing Basics

How do we disassemble executables? Can we derive the original source code from a compiled executable?

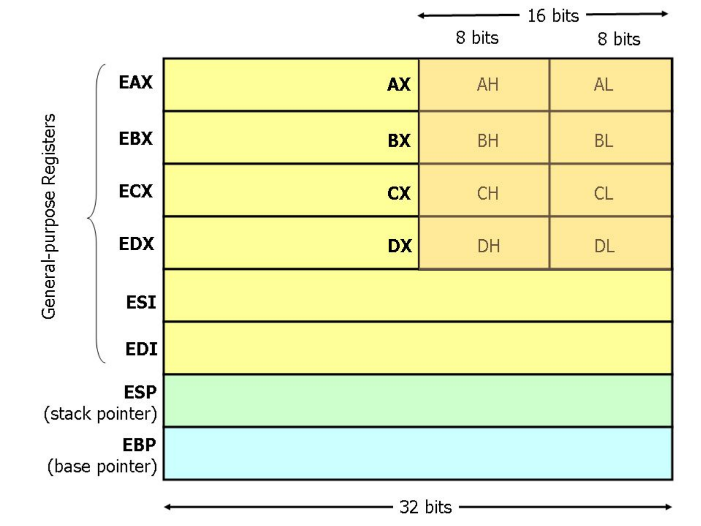
A 30,000 foot view

- Static Analysis
  - Disassembly
  - Decompilation
- Dynamic Analysis
  - Debugging (GDB)
  - System call tracing
  - Network activity tracing



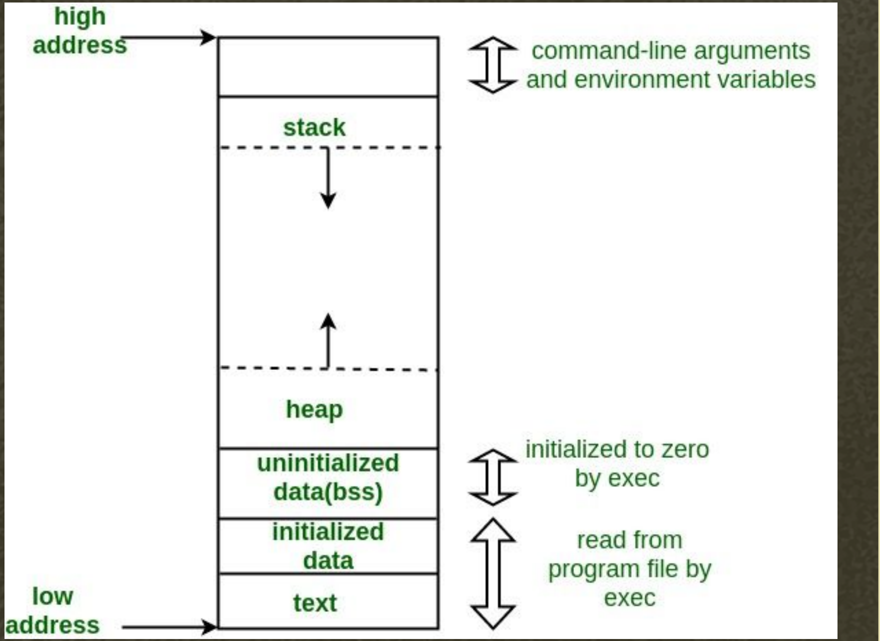
How to Read Assembly

- Registers
  - eax, ebx, ebp, esp (x86)
- Basic instructions and their operands
  - e.g. mul eax, ebx
- The C Calling Convention (cdecl)
  - How function calls are implemented in C
  - How accessing variables work
- Executable File Sections
  - What each section does and its properties
  - (for ELF) .text, .data, .bss, .rodata



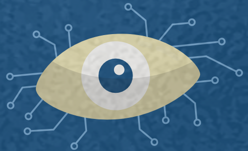
1 More Thing - The Stack

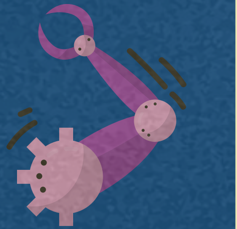
- Some memory space used primarily for:
  - Local variables
  - Passing function arguments
- Behaves like a stack
  - Push & Pop operations
- Grows into lower address space
  - RBP is higher than RSP

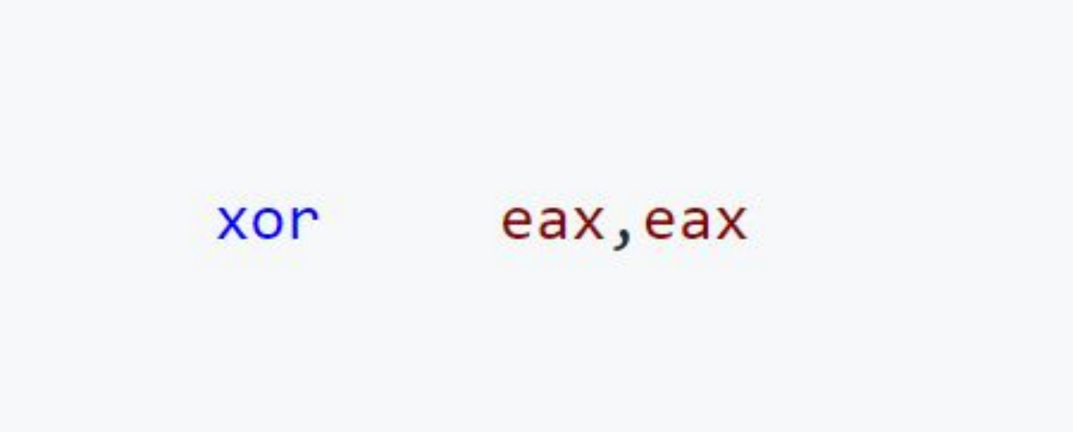

high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

Memory layout of a program

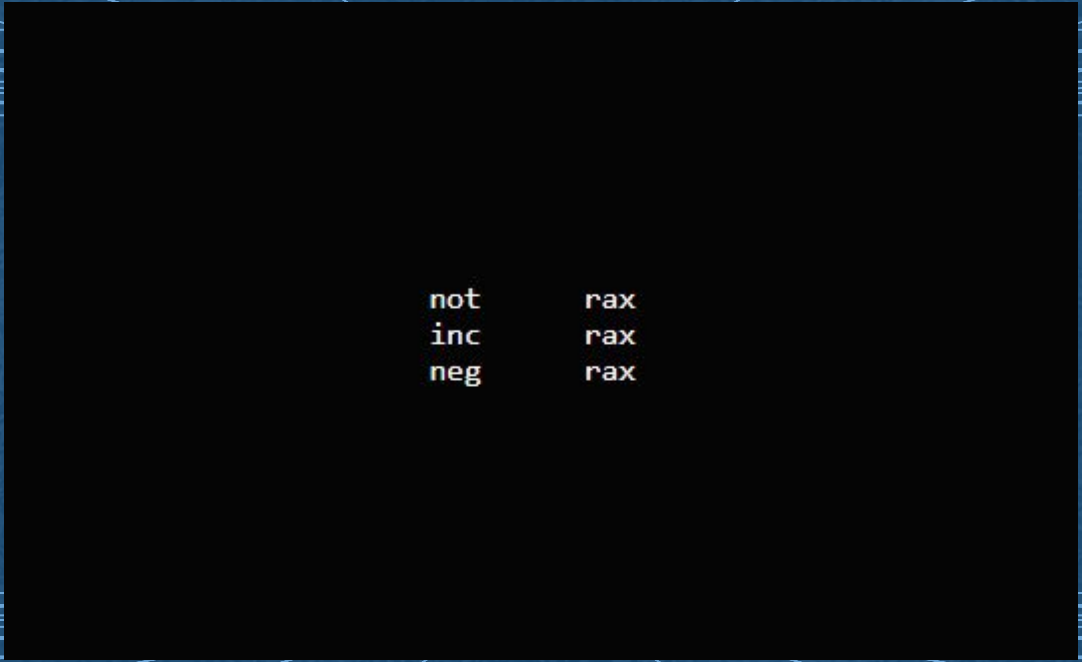
Reading ASM







```
xor     eax,eax
```



```
not     rax
inc     rax
neg     rax
```

xchng rax, rax

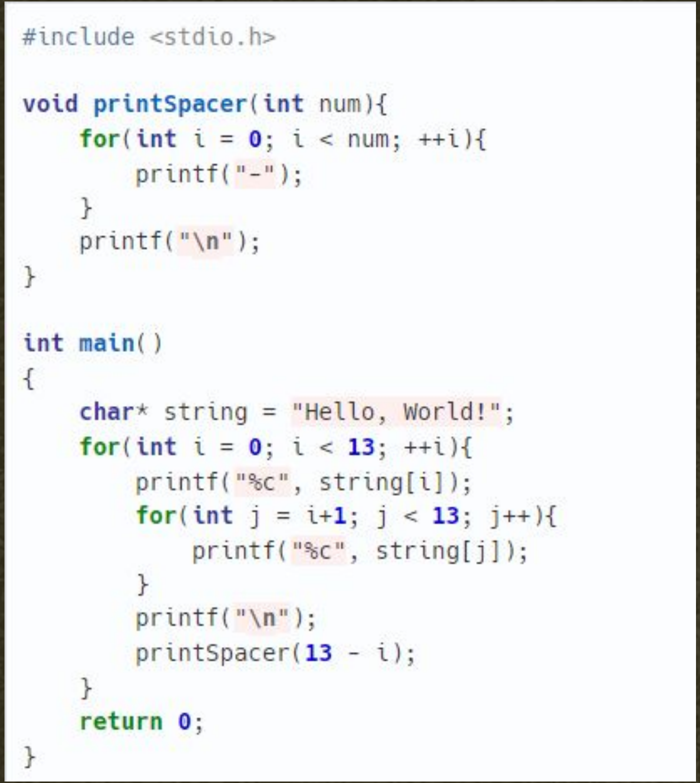
Translating C to ASM

- While loops, For loops
- Conditions
- Function Calls

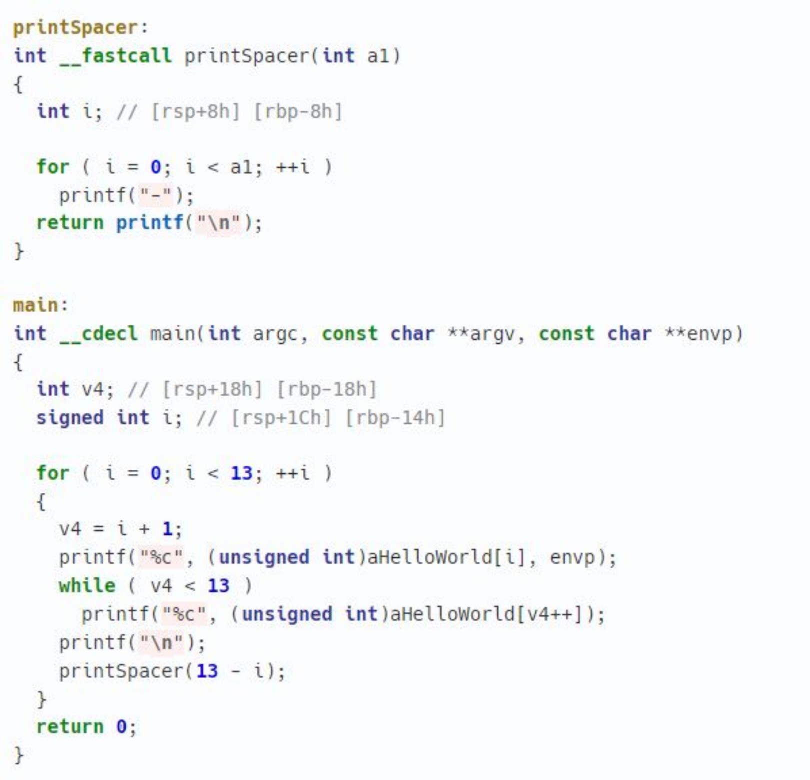
https://godbolt.org/

Decompilation

- Inverse operation of compilation - generating high level source code from a compiled binary
- Tools:
    - IDA Hex Rays
    - Ghidra
- Translation to high level pseudocode may not be 1-to-1
    - We’ll be taking a look at this



```c
#include <stdio.h>

void printSpacer(int num){
    for(int i = 0; i < num; ++i){
        printf("-");
    }
    printf("\n");
}

int main()
{
    char* string = "Hello, World!";
    for(int i = 0; i < 13; ++i){
        printf("%c", string[i]);
        for(int j = i+1; j < 13; j++){
            printf("%c", string[j]);
        }
        printf("\n");
        printSpacer(13 - i);
    }
    return 0;
}
```



```c
printSpacer:
int __fastcall printSpacer(int a1)
{
  int i; // [rsp+8h] [rbp-8h]

  for ( i = 0; i < a1; ++i )
    printf("-");
  return printf("\n");
}

main:
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [rsp+18h] [rbp-18h]
  signed int i; // [rsp+1Ch] [rbp-14h]

  for ( i = 0; i < 13; ++i )
  {
    v4 = i + 1;
    printf("%c", (unsigned int)aHelloWorld[i], envp);
    while ( v4 < 13 )
      printf("%c", (unsigned int)aHelloWorld[v4++]);
    printf("\n");
    printSpacer(13 - i);
  }
  return 0;
}
```

ctf101.org

What's The Point?

- Malware analysis
- Become a better developer
  - Understanding how programs may be vulnerable
- Embedded programming
- CTFs!
  - https://ctf.gdscutm.com/

Cool Applications

THANKS!

@gdscutm

CREDITS: This presentation template was created by **Slidesgo,** including icons by **Flaticon,** and infographics & images by **Freepik**