WELCOME!!

RevEng Will be Ours!!!

An intro to assembly language, reverse engineering, and buffer overflows

What is Assembly language?

- Assembly Language is the interface between higher level languages (C++, Java, etc) and machine code (binary). For a compiled language, the compiler transforms higher level code into assembly language code.
- Assemblers decompose Assembly instructions into their respective binary representations and replace the generic addresses of assembly code with explicit register and memory addresses of your computer.

What are registers?

- Your CPU uses a set of registers in order to manipulate data in your running program. These are storage holders, just like the RAM in your computer. However they’re located on the CPU itself very close to the parts of the CPU that need them. So these parts of the CPU can access these registers incredibly quickly.
- Most instructions involve one or more registers and perform operations such as writing the contents of a register to memory, reading the contents of memory to a register or performing arithmetic operations (add, subtract, etc.) on two registers.
- We are working with x86_64. There are **16 general purpose registers** used by the machine to manipulate data.
- The RSP register is decremented when items are pushed onto the stack, and incremented when they are popped off the stack. The RBP register points to the lowest address of the data structure that is passed from one function to another.

Reserved Pointers

- The registers store data elements for processing without having to access the memory
- **Instruction Pointer (RIP)** − The 16-bit RIP register stores the offset address of the next instruction to be executed.
- **Stack Pointer (RSP)** − The 16-bit RSP register provides the offset value within the program stack.
- **Base Pointer (RBP)** − The 16-bit RBP register mainly helps in referencing the parameter variables passed to a subroutine

Stacks on Stacks

- The stack is a memory area that can hold temporary data (functions parameters, variables, etc.) and is designed to behave in a “Last In, First Out” context, which means the first value stored in the stack (or pile) (pushed) will be the last entry out (popped)

hello.asm

```
; ------------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls. Runs on 64-bit Linux only.
; To assemble and run:
;
;     nasm -felf64 hello.asm && ld hello.o && ./a.out
; ------------------------------------------------------------------------------

          global    _start

          section   .text
_start:   mov       rax, 1                  ; system call for write
          mov       rdi, 1                  ; file handle 1 is stdout
          mov       rsi, message            ; address of string to output
          mov       rdx, 13                 ; number of bytes
          syscall                           ; invoke operating system to do the write
          mov       rax, 60                 ; system call for exit
          xor       rdi, rdi                ; exit code 0
          syscall                           ; invoke operating system to exit

          section   .data
message:  db        "Hello, World", 10      ; note the newline at the end
```

https://cs.lmu.edu/~ray/notes/x86assembly/

Terms

- Reverse engineering: process of taking a piece of software or hardware and analyzing its functions and information flow so that its functionality and behavior can be understood. Malware is commonly reverse-engineered in cyber defense.
- Disassembler: converting machine language to assembly language
- Debugger: Step through a program one line at a time to see its “state”
- Decompiler: converting executable into source code
- Buffer overflow: an untrusted input is transferred into a *buffer*, such as an array of characters on the stack, without checking to see whether the buffer is big enough for the untrusted input. Buffer overflows cause undefined behavior—it’s illegal to access memory outside any object.

Lab is Broken into 4 parts

```
#include
00:02:76:4D:6C:D2

int main(int argc, char
*argv[]) {

char buf[256];
memcpy(buf,
argv[1],strlen(argv[1]));
printf(buf);

}
```

- Preliminary Analysis of File
- Ghidra
- GDB
- Basic Buffer Overflow with GDB
- I’m basically learning at the same time (if you couldn’t tell lol) so don’t be worried if stuff is confusing and please correct me if I use the wrong term. Let's learn together :)

Sources

Sources for PowerPoint:

- https://www.tutorialspoint.com/assembly_programming/assembly_registers.htm
- https://medium.com/reverse-engineering-for-dummies/a-crash-course-in-assembly-language-695b07995b4d
- https://resources.infosecinstitute.com/topic/x86-assembly-language-applicable-to-reverse-engineering-the-basics-part-1/
- https://thecyberwire.com/glossary/reverse-engineering
- https://www.cs.utah.edu/~germain/PPS/Topics/debugging_programs.html
- https://cs61.seas.harvard.edu/site/2018/Asm4/
- https://www.freecodecamp.org/news/what-are-assembly-languages/

Sources for lab (check out in google doc!):

- https://www.exploit-db.com/docs/english/28475-linux-stack-based-buffer-overflows.pdf
- https://infosecwriteups.com/tryhackme-reversing-elf-writeup-6fd006704148
- https://infosecwriteups.com/tryhackme-reversing-elf-writeup-6fd006704148