Intro to Reverse Engineering

Adi

Housekeeping

Friday: History of Anonymous Talk presented Krishna

Saturday: GlacierCTF

Next Wednesday: Dead Week Boba

What is Reverse Engineering

If **engineering** is turning human ideas into a complete product...

Then **reverse engineering** is turning that product back into the original ideas

You can reverse engineer anything!

- Toys
- Languages
- EBooks

In cybersecurity/CTF contexts you will be usually reversing applications

Why Reverse Engineer?

- It's applicable to many areas of security.
- You become a better programmer.
- Everything becomes open-source!

Computer Understandable Instructions

How does your computer understand what this means?

```python
myStr = "abcde"
for char in myStr:
    print(char)
```

Or this...

```java
String myStr = "abcde";
for (int i = 0; i < myStr.length(); i++) {
    System.out.println(myStr.charAt(i));
}
```

Orrrrr this.

```rust
let myStr = String::from("abcde");
for c in myStr.chars() {
    println!("{}", c);
}
```

Computer Understandable Instructions

All programming languages are converted to a common 'assembly' language by your compiler

```python
myStr = "abcde"
for char in myStr:
    print(char)
```

```java
String myStr = "abcde";
for (int i = 0; i < myStr.length(); i++) {
    System.out.println(myStr.charAt(i));
}
```

```rust
let myStr = String::from("abcde");
for c in myStr.chars() {
    println!("{}", c);
}
```

```
.data
myStr DB 0x61, 0x62, 0x63, 0x64,
0x65, 0x00
.text
MOV RCX, 0x00
FOR_LOOP:
CMP RCX, 0x05
JGE END
MOV RAX, 0x01
MOV RDI, 0x01
LEA RSI, [myStr + RCX]
SYSCALL
INC RCX
JMP FOR_LOOP
END:
```

Computer Understandable Instructions

This assembly is then directly converted to and stored as bytes

```
.data
myStr DB 0x61, 0x62, 0x63, 0x64, 0x65,
0x00
.text
MOV RCX, 0x00
FOR_LOOP:
CMP RCX, 0x05
JGE END
MOV RAX, 0x01
MOV RDI, 0x01
LEA RSI, [myStr + RCX]
SYSCALL
INC RCX
JMP FOR_LOOP
END:
```

```
616263646548C7C1000000004883F9057D1C48C7C0
0100000048C7C701000000488DB1000000000F0548
FFC1EBDE
```

A Quick Bit on Bytes

Computers do everything in 1s and 0s (bits)

`0b0001 = 1, 0b0010 = 2, 0b0100 = 4, 0b1000 = 8, 0b1111 = 15`

You can do your typical add and subtract operations on them

`0b0001 + 0b1000 = 0b1001 = 9`

`0b0001 + 0b0101 = 0b0110 = 6`

But there are also unique bitwise operations to binary:

Bitwise And: `0b00000101 & 0b00101011 = 0b00000001 = 1`

Bitwise Or: `0b00000101 | 0b00101011 = 0b00101111 = 47`

Bitwise Xor: `0b00000101 ^ 0b00101011 = 0b00101110 = 46`

A Quick Bit on Bytes

Computers like to deal with data in bundles of 8 bits

`0b00000000`

These bundles are called bytes

These are displayed as two hexadecimal digits represented by characters 0-9 and A-F.

A-F corresponds to values 10-15

`0b00111011 = 0011 1011 = 3 11 = 3 b = 0x3b`

`0x9f = 9 f = 9 15 = 1001 1111 = 0b10011111`

Lower Languages

Some languages are closer to assembly than others

Python and Java are very abstracted away (High level)

Rust and C are more directly converted (Low level)

It is possible to turn assembly code back into approximate C code!

C Crash Course

C is very similar to Java, but there are no classes/objects and **you have to manage memory yourself**

Java

```java
String myStr = "abcde";
for (int i = 0; i < myStr.length(); i++) {
    System.out.println(myStr.charAt(i));
}
```

C

```c
char *myStr = "abcde";
for (int i = 0; i < strlen(myStr); i++) {
    puts(myStr[i]);
}
```

C Crash Course

C has the a lot of the same data types as Java

```
int, float, bool, long, char
```

but they are all just bytes!

Each character is represented by a number `'a' = 0x61`

Strings are just an array of characters ended by a zero byte `"abcde" = {0x61, 0x62, 0x63, 0x64, 0x65, 0x00}`

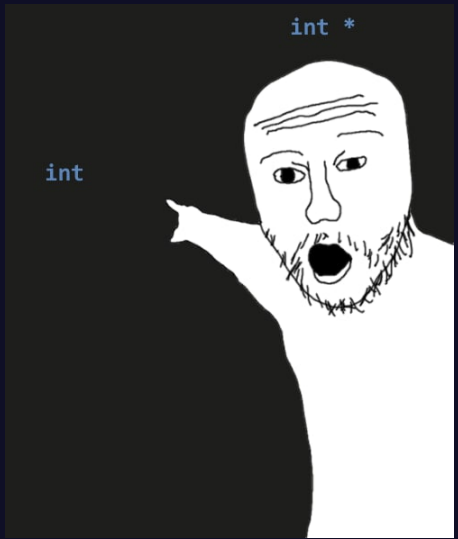
This means you can do all the same bitwise and arithmetic operations on chars as you could numbers

`'a' ^ '8' = 0x61 ^ 0x38 = 0x59 = 'Y'`

C Crash Course

C does have one important type that Java doesn't...

Pointers!



C Crash Course

Pointers are the address to data in memory

`int*, char*, long*`

If you think of memory as a mailroom, and data as mail.

Then pointers are mailbox numbers

C Crash Course

You get the pointer to a variable with `&` and modify the variable from a pointer with `*`

```c
void inc(int n) {
    n += 1;
}

void incp(int* n) {
    *n += 1;
}

int main() {
    int n = 1;
    printf("n is %d", n); //n is 1
    inc(n);
    printf("n is %d", n); //n is 1
    incp(&n);
    printf("n is %d", n); //n is 2
    return 0;
}
```

C Crash Course

Common C Functions:

`puts(const char *str)`: Print a string as a new line

`printf(const char *format, ...)`: Prints an arbitrary amount of variables as a string on the current line

`fgets(char *str, int size, FILE *stream)`: Read up to **size** bytes from file stream, most commonly used to read user input

`strlen(const char *str)`: Returns the length of a string

`strcmp(const char *str1, const char *str2)`: Compare two strings, and returns 0 if they are equal

`strcpy(char *dst, const char *src)`: Copies a string from src to dst

`memcpy(void *dst, const void *src, int size)`: Copies size bytes from src to dst, regardless of type

Reverse Engineering Tools

There are tools called decompilers that can do this process for you

IDA Pro: Industry one created by Hex-Rays

Ghidra: Open source one created by the NSA

Binary Ninja: Modern one created by Vector35

All of these are free or have free versions

Binary Ninja has a free web browser version that we will use

https://cloud.binary.ninja/

Challenges

https://binary.ninja/

Download the challenges in the discord

it won't hack your computer i pinky promise ;)

Decompilers are great, but...

Decompilers are a great tool to speed up understanding a program, but they aren't perfect

- It's a lossy conversion inheriently
- They don't work as well on programs made in different languages
- Sometimes it's just plain wrong

It's still important to be able to read and understand assembly.

Assembly Anatomy

There are multiple types of assembly instruction sets

x86-64 (We will be focusing on this one)

```
myStr DB 0x61, 0x62, 0x63,
0x64, 0x65
MOV RCX, 0x00
FOR_LOOP:
CMP RCX, 0x05
JGE END
MOV RAX, 0x01
MOV RDI, 0x01
LEA RSI, [myStr + RCX]
SYSCALL
INC RCX
JMP FOR_LOOP
END:
```

AArch64

```
myStr: .byte 0x61, 0x62,
0x63, 0x64, 0x65
MOV X0, 0x00
FOR_LOOP:
CMP X0, 0x05
BGE END
MOV X8, 0x40
MOV X1, 0x01
MOV X3, 0x01
ADR X2, myStr
ADD X2, X2, X0
SVC 0x00
ADD X0, X0, 0x01
B FOR_LOOP
END:
```

MIPS64

```
myStr: .byte 0x61, 0x62,
0x63, 0x64, 0x65
li $t0, 0x00
li $t1, 0x05
FOR_LOOP:
beq $t0, $t1, END
la $a0, myStr
add $a0, $a0, $t0
lb $a0, 0($a0)
li $v0, 0x0b
syscall
add $t0, $t0, 0x01
j FOR_LOOP
END:
```

Assembly Anatomy

Assembly instructions are made up of two parts,

Opcode: The type of instruction that it's executing

Operands: Arguments for those instructions, these can be either literal, address, or register

```
<OPCODE> <OPERANDS>
```

Assembly Anatomy

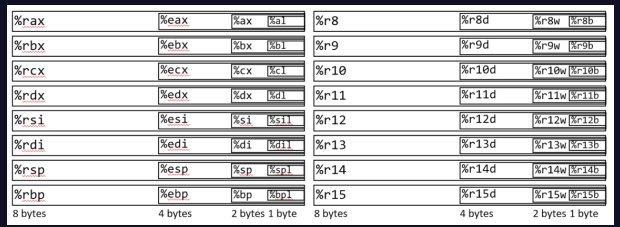
Registers: Think of these as short term global variables that are very fast for your computer to access

General Purpose:

RAX, RBX, RCX, RDX, RDI, RSI, R8-15

Special:

RSP (Stack Pointer), RBP (Base Pointer), RIP (Instruction Pointer)



You cannot create more registers as these are physical things on your CPU

Common Assembly Instructions

Moving:

`MOV <dst>, <src>`: Copies the value from src to dst

`LEA <dst>, <exp>`: Load evalutated value from exp to dst

Common Assembly Instructions

Arithemetic:

`ADD <dst>, <src>`: Adds src and dst, and saves it in dst

`SUB <dst>, <src>`: Subtracts src from dst, and saves it in dst

`MUL <dst>, <src>`: Multiplies src and dst, and saves it in dst

`AND <dst>, <src>`: Does bitwise and on src and dst, and saves it in dst

`OR <dst>, <src>`: Does bitwise or on src and dst, and saves it in dst

`XOR <dst>, <src>`: Does bitwise xor on src and dst, and saves it in dst

`INC <dst>`: Increments value in dst by 1

`DEC <dst>`: Decrements value in dst by 1

Common Assembly Instructions

Control Flow:

`CMP <arg1>, <arg2>`: Compares two values that will affect a jump instruction

`JE <lab>`: Jumps to label if arg1 == arg2, (Also called `JZ`)

`JNE <lab>`: Jumps to label if arg1 != arg2, (Also called `JNZ`)

`JL <lab>`: Jumps to label if arg1 < arg2

`JG <lab>`: Jumps to label if arg1 > arg2

`JLE <lab>`: Jumps to label if arg1 <= arg2

`JGE <lab>`: Jumps to label if arg1 >= arg2

Labels are just a named area in your assembly

Challenge 2-1

What value is in RAX at the end of this program?

```
.data
myBytes DB 0x01, 0x02, 0x03, 0x04
.text
MOV RAX, 0x0d
MOV RBX, 0x10
SUB RBX, RAX
MOV RCX, 0x02
MUL RBX, [myBytes + RCX]
ADD RAX, RBX
```

Challenge 2-2

What value is in RAX at the end of this program?

```
.data
myBytes DB 0x01, 0x02, 0x03, 0x04
.text
MOV RAX, 0x00
MOV RCX, 0x00
LABEL_1:
CMP RCX, 0x04
JGE LABEL_2
MOV RBX, [myBytes + RCX]
ADD RAX, RBX
INC RCX
JMP LABEL_1
LABEL_2:
ADD RAX, RCX
```

Reversing Methodology

What we have been doing with Binary Ninja is static analysis.

Gaining information by looking at the program's code without running it

There is also dynamic analysis, where you get info by running the program (usually with a debugger)

Static is great for understanding finding what **can** happen and **how**

Dynamic is great for understanding finding what **will** happen and **why**

Alternate between static and dynamic anaylsis if possible