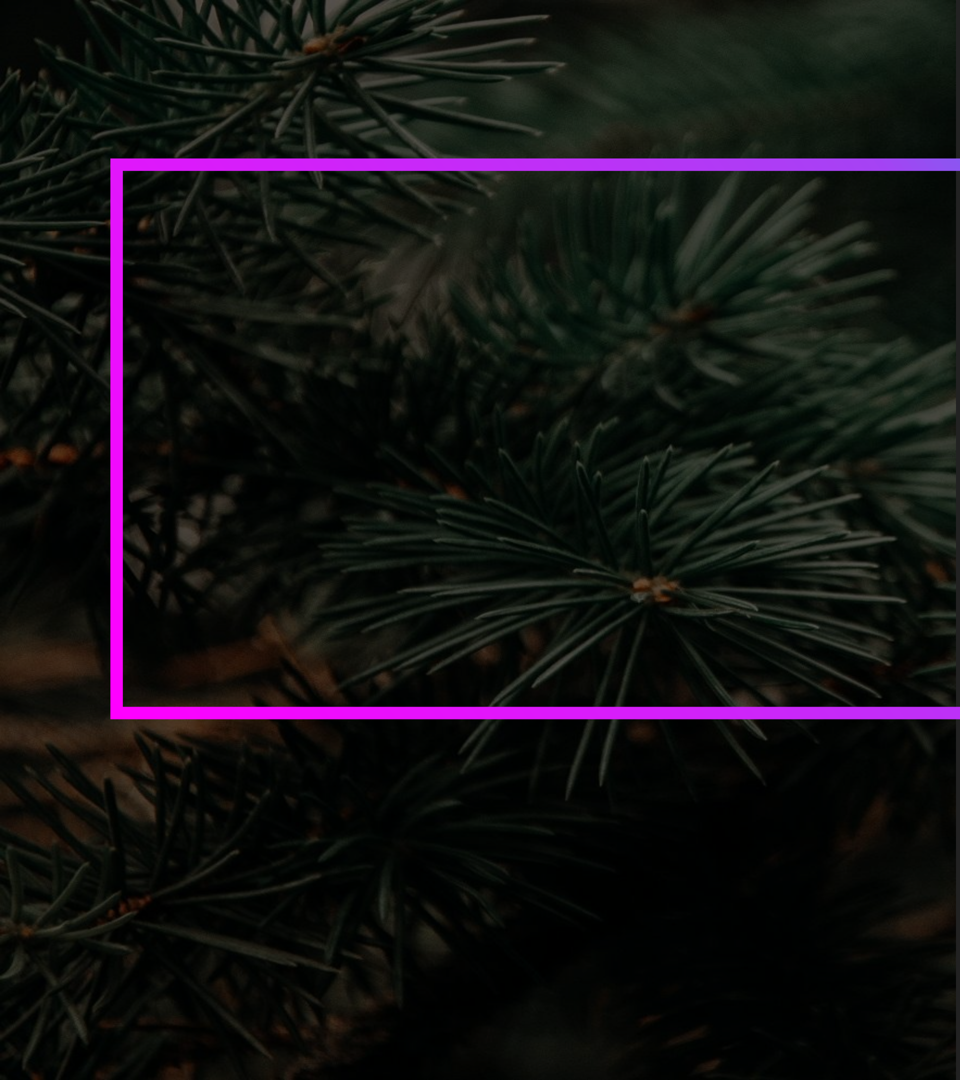
Dec 10, 2018

Cyber Offensive / Defensive Engineering

Columbus, OH

Hack the Holidays 2018

The Joys of Binary Reverse Engineering



Purpose & Practical Uses of Software RE

Software reverse engineering is a critical skill needed in many industries. You might only associate reverse engineering with malware analysis, video games hacking, or software piracy, but it’s much more common than you might think.

CAN Bus Sensors

- Designed to plug into your car’s OBD-II port and passively monitor CAN traffic
- Much of this data must be available for emissions testing / regulations
- Messages can vary greatly between manufacturers and vehicles
- Companies providing these devices to auto insurers and data enthusiasts must:
  - Understand the arbitration IDs and payloads for various messages
  - Correlating specific messages to various sensors and features

Manufacturers in Highly Competitive Markets

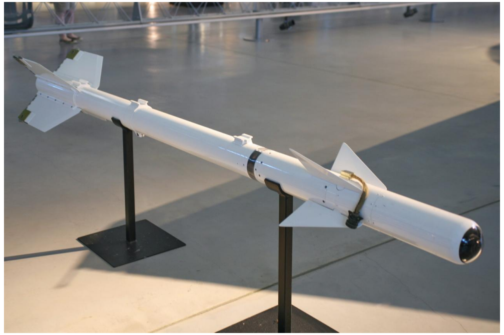
- Communications
- Transportation
- Home Automation
- Electronic Children's Toys
- Software, Mechanical, Electronic, Chemical, Biological

Software & System Security

- Security Researchers Evaluating Software
- Evaluations of Competitor & Vendor Claims
- Proprietary Communications Protocols
- Analysis of Competitor Intellectual Property
- Maintenance & Documentation of Legacy Systems

Military Applications

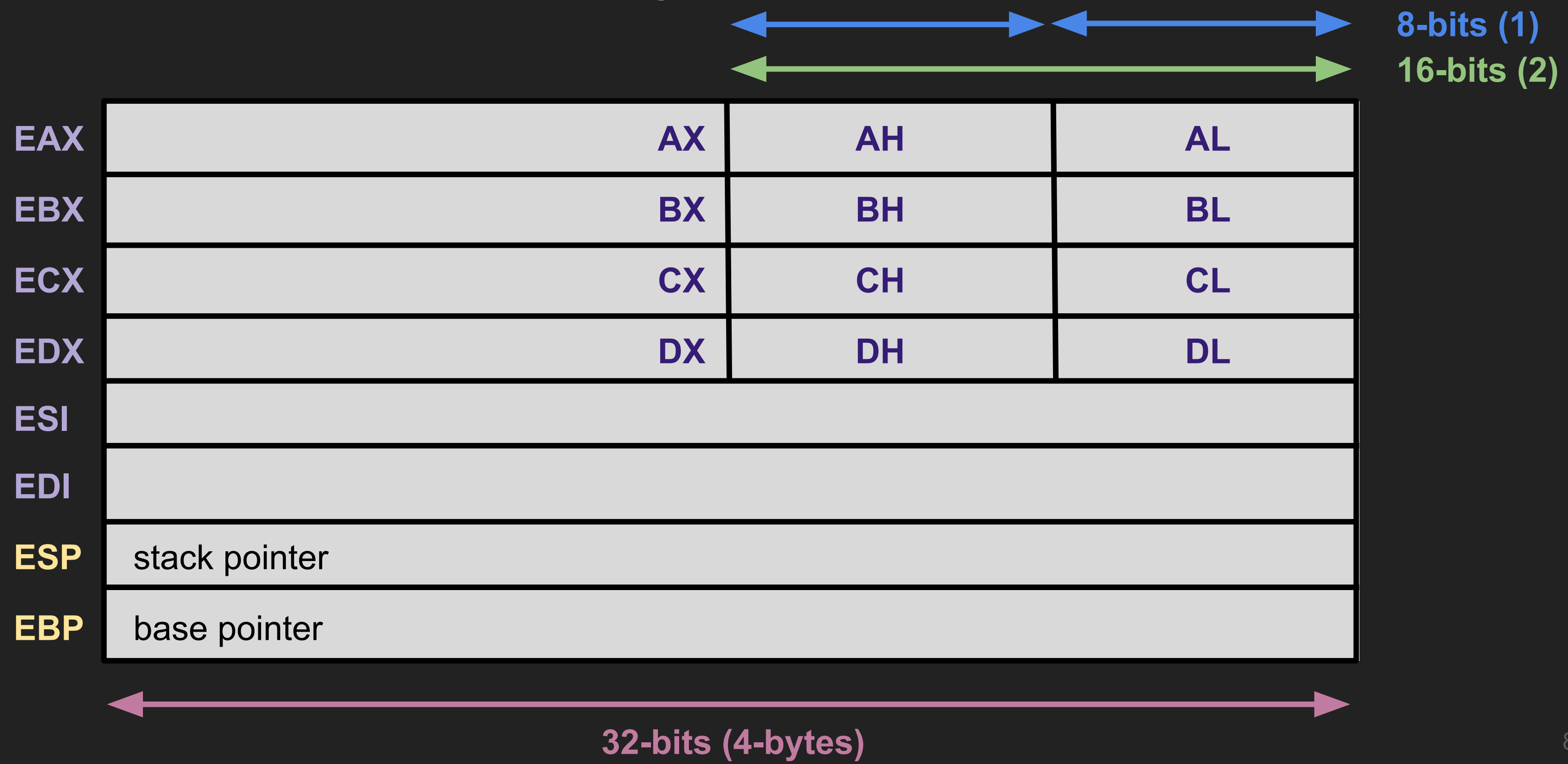
- WWII German Enigma Machine
- Soviet K-13/R-3S missile developed after RE of the AIM-9 Sidewinder
    - Taiwanese AIM-9B hit a Chinese MiG-17 without exploding in 1958. The missile became lodged in the airframe, Russian scientists / engineers copied the design



x86 in a Nutshell

x86 is a beast of an architecture that could take a lifetime to master - thankfully there’s only a few concepts you need to understand to start working with this architecture.

The General Purpose Registers



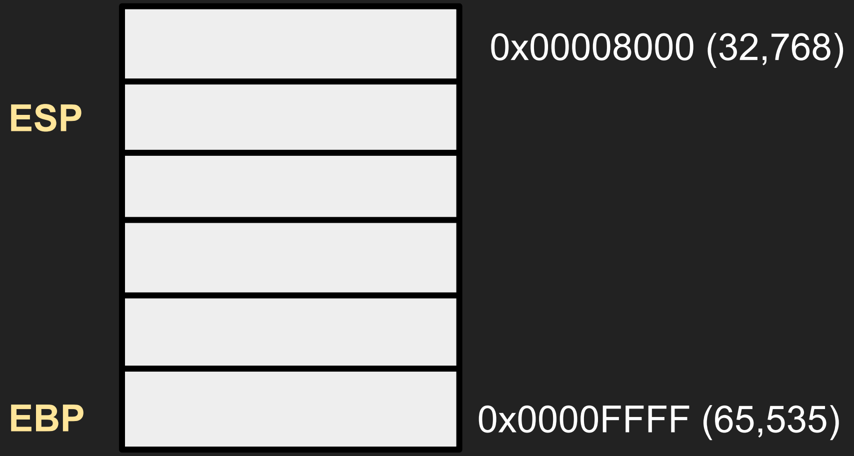
Intel Syntax: Destination First

```
ins dest, src

mov eax, 0x01              (eax = 1)

add ebx, ecx               (ebx = ebx + ecx)

lea eax, [ebp-0x68]        (eax = <stack addr>)
```

The Stack

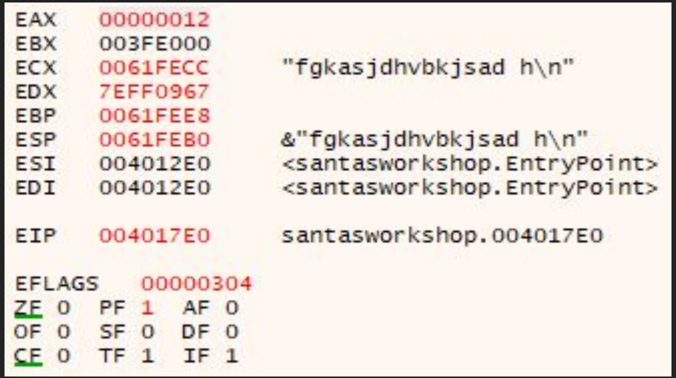
- Each function has its own “frame”, where EBP points to the base, and ESP points to the top
- The “top” (ESP) of the stack moves up and down as things are PUSHed and POPped
  - Think of a stack of plates at a buffet
- In a confusing turn of events, the stack “grows towards lower addresses”
  - So when we subtract from the stack, it means we’re actually adding space to it

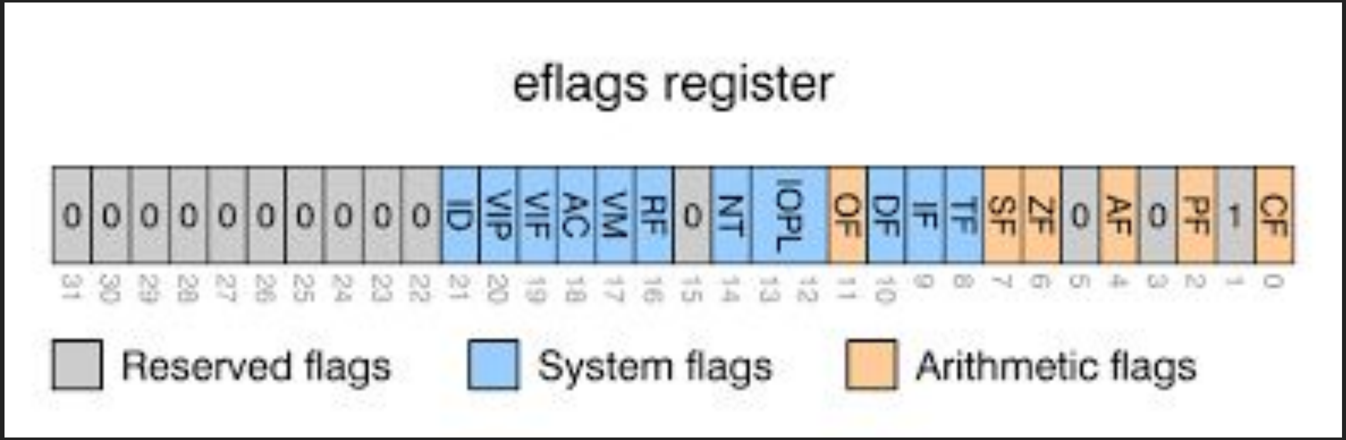
ESP “floats” up and down, always pointing to the current top of the stack, EBP is fixed at the bottom



Flags

CF - Carry Flag
OF - Overflow Flag
SF - Sign Flag
ZF - Zero Flag
TF - Trap Flag





Comparison Operations: test

```
1. test al, al
2. test eax, eax
```

test performs a bitwise AND between two operands.

1. Sets **ZF=1** if **al** zero, otherwise **ZF=0**
2. Sets **SF=1** if **eax** is negative, otherwise **SF=0**

```
op1: 0101
op2: 1110
---------------------------
     0100 (bitwise AND)
```

Comparison Operations: cmp

```
1. cmp eax, 0x0B
2. cmp [ebp-0x14], 0x00
```

cmp subtracts the **2nd** operand from the **1st** operand.
You can think of a **cmp** as a **sub** command that only sets flags.

1. eax - 0x0B (**SF=1** if **eax** < **0x0B**)
2. [ebp-0x14] - 0x00 (**SF=1** if value on stack is < 0)

cmp will also modify other flags, but SF is most interesting to us.

Conditional Operations: Jumps

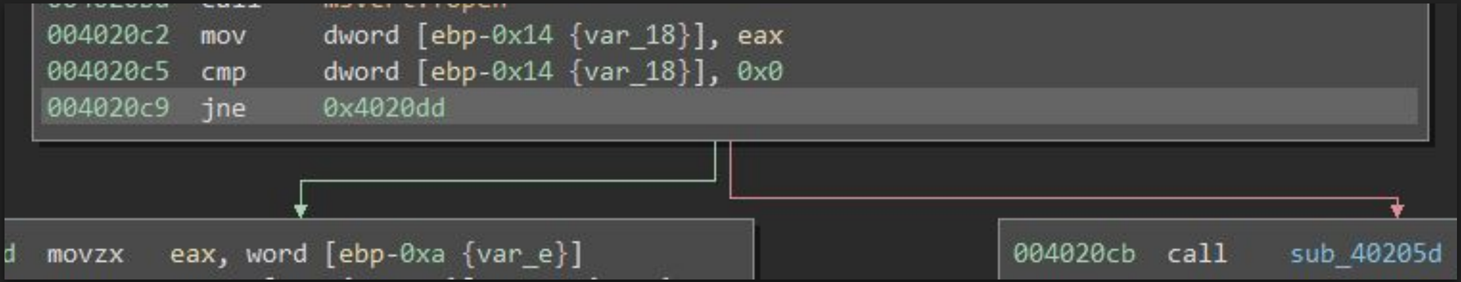
```
je  - Jump if Equal
jne - Jump if Not Equal
jg  - Jump if Greater
jge - Jump if Greater or Equal
ja  - Jump if Above (unsigned)
jae - Jump if Above or Equal (unsigned)
jl  - Jump if Less
jle - Jump if Less than or Equal
jb  - Jump if Below (unsigned)
jbe - Jump if Below or Equal (unsigned)
jz  - Jump if Zero
jnz - Jump if Not Zero
```

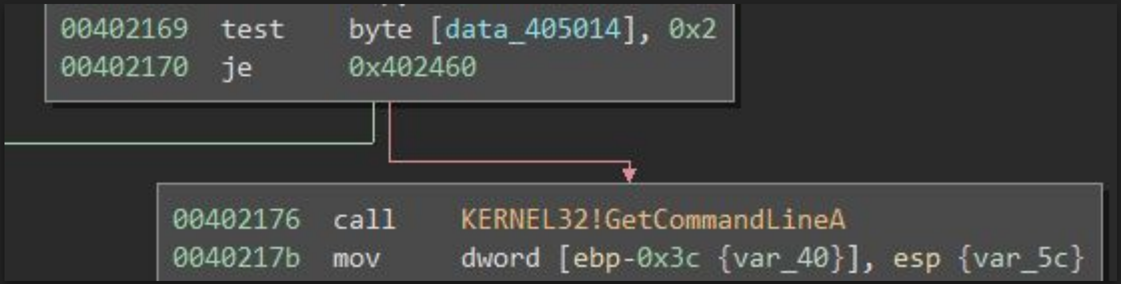
Conditional operations function purely on the flags registers.

These are generally preceded by a **test** or **cmp**.

Conditional Operations: Jumps

```
je  - Jump if Equal
jne - Jump if Not Equal
```





Word Up, What's the Bitness?

```
4 bits  = 1 nibble (0000 0000, think ax, bx, cx, etc.)
8 bits  = 1 byte
16 bits = 2 bytes = 1 WORD
32 bits = 4 bytes = 2 WORDS = 1 DWORD (double word)
64 bits = 8 bytes = 4 WORDS = 2 DWORD = 1 QWORD (quad word)
```

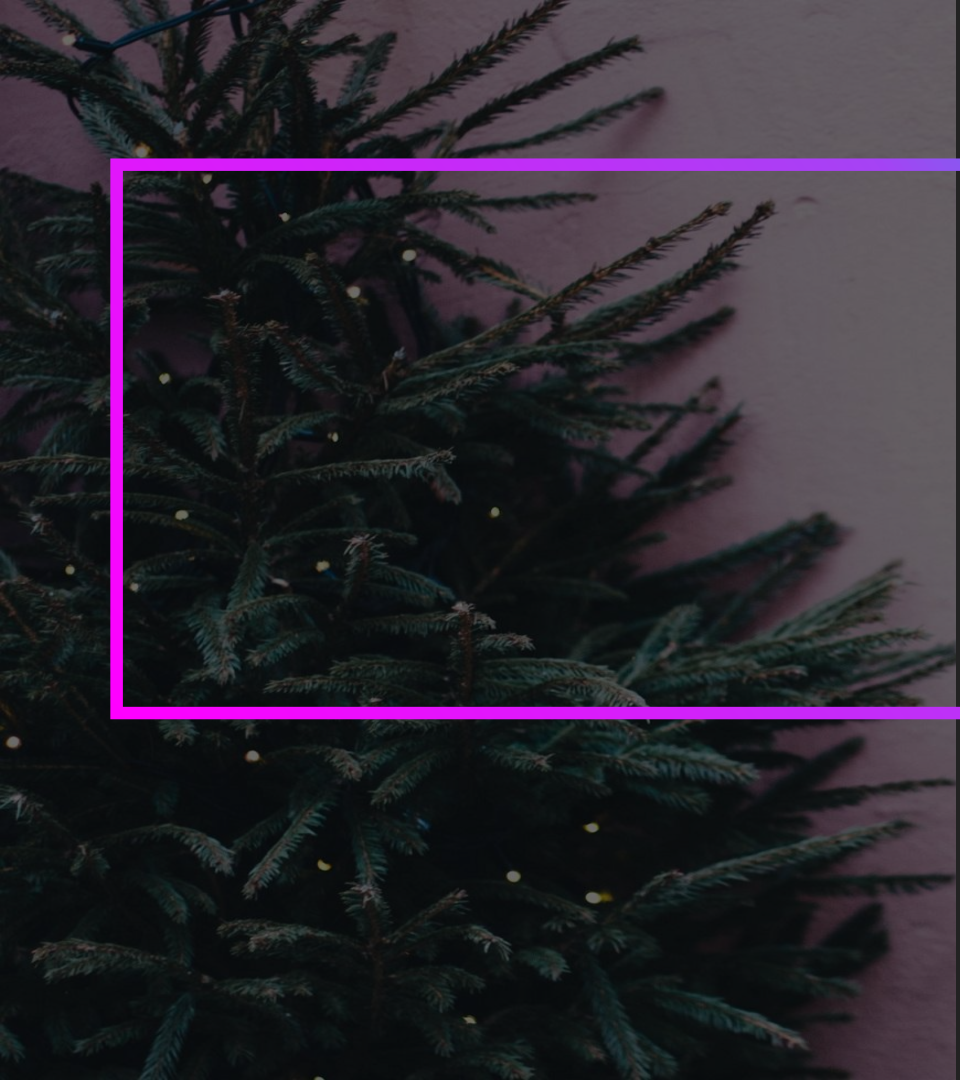
32-bit systems have a 32-bit address space
max val is 4,294,967,295 (4 billion)

64-bit systems have a 64-bit address space
Max val is 18,446,744,073,709,551,615 (18 quintillion)

Santa’s Workshop Main Lobby: Switch Statements

Navigating our way through Santa’s Workshop is easy with the right equipment. We’ll be using a disassembler called Binary Ninja.

In the main lobby, we’ll learn to identify switch statements.

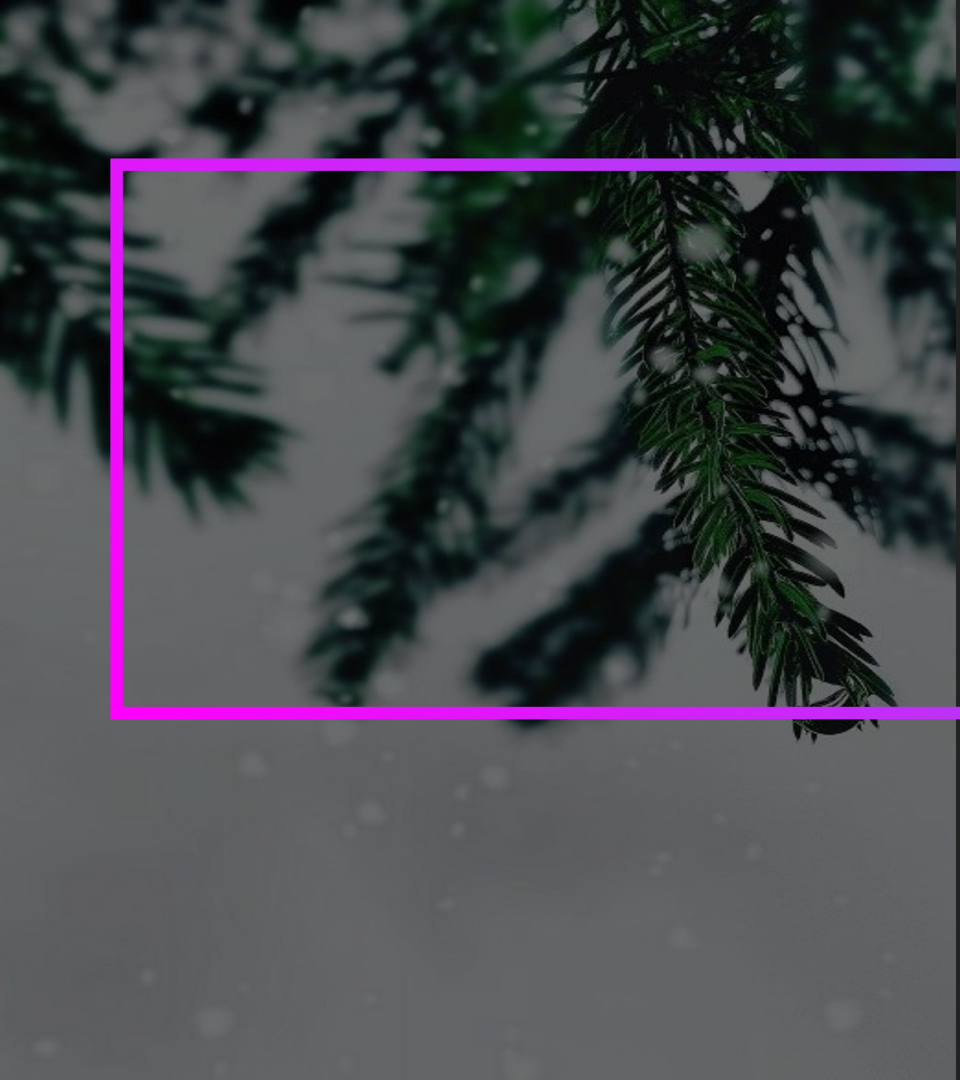


The North Wing: if / else Patterns

Reverse engineers make heavy use of patterns to sift through countless instructions looking for the functionality they are after. Let’s focus on two of the most commonly seen patterns: if/else stairs and ladders.

The South Wing: for & while Loops

Looping is extremely important and useful in numerous situations in application development. As a reverse engineer, you’re particularly interested in understanding the conditions required to exit a loop, and the number of times that loop may execute.



The East Wing: Calling Conventions

Calling Conventions provide a contract between the caller and the callee. They dictate how parameters are passed into functions and who is responsible for cleaning up the stack. There are many of these in x86, we’re going to focus on three common ones.

The West Wing: Dynamic Memory Allocs

Functions use locally scoped stack frames to store local variables and small amounts of data. But what if you need that data to stick around for a while, or if it’s really large? Welcome to the realm of the heap!

Santa’s Lair: The Not-so-random PRNG

Computers need random numbers for a variety of reasons, but it turns out generating a good random number is harder than you might think.