



IntroToCTF:

Intro to Reverse Engineering



What is Reverse Engineering?

What is Reverse Engineering?

- **Reverse engineering** is the process of **analysing a complex system** to see **how it works**
- In a Cyber Security context, “Reverse Engineering” normally refers to analysing a compiled binary program
- The most important technique in reverse engineering is **static analysis**. This involves analysing **what** a program does **without needing to run it**
- We’ll be covering basic static analysis this week



Reverse Engineering CTF Challenges

- Reverse Engineering **CTF Challenges** require you to **understand a program's logic**, and figure out a way to **reverse what it's doing**
- “rev” challenges do **not** require you to **exploit the program**. That's a **separate category** called **binary exploitation**, which we'll cover in a future lesson
- A lot of rev challenges involve analysing a program which has **encrypted a flag**, and **building a tool to decrypt it**





Background: How Variables are Stored in Memory

Memory: Overview

- To reverse engineer a program, you need to understand **how it stores data** in your **computer's memory** as the program runs.
- Memory is just a **giant array of bytes**
- Each byte is **numbered**, and that number is called a **memory address**
- Bytes are 8-bit values from **00000000** to **11111111**
- To make them **easier to read**, bytes are usually **written in hexadecimal**
- We're going to cover how variables are stored in **programs written in the C programming language**. Programs written in **C++** or **Rust** are a bit different, that's **a topic for another time**



Memory: Hexadecimal

- **Hexadecimal** is **base 16**
- Each digit can take values from **0-F** (e.g. **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**)
- Hexadecimal numbers are normally **prefixed with 0x**
- **Bytes** are written as **hexadecimal numbers** from **0x00** to **0xFF** (0 to 255)



Memory: Integers

- Integers can be **different sizes**, including **32-bit** (4 bytes) and **64-bit** (8 bytes). This depends on your **platform** (**Operating System** and **architecture**)
- Integers are generally stored in **little endian format**. This can be **a bit confusing**
- Essentially, the **least significant byte** is stored **first**.
- It's like if we wrote "**one thousand and thirty four**" as **4301** instead of **1034**



Memory: Integers - Example

```
#include <stdio.h>

int main() {
    int example_int = 0x11223344;
    printf("%d", example_int);
}
```



00101149	f3 0f 1e	ENDBR64	
	fa		
0010114d	55	PUSH	EBP
0010114e	48 89 e5	MOV	EBP,ESP
00101151	48 83 ec	SUB	ESP,0x10
	10		
00101155	c7 45 fc	MOV	dword ptr [EBP + local_c],0x11223344
	44 33 22		
	11		
0010115c	8b 45 fc	MOV	EAX,dword ptr [EBP + local_c]
0010115f	89 c6	MOV	ESI,EAX
00101161	48 8d 05	LEA	EBX,[DAT_00102004]
	9c 0e 00		= 25h %
	00		

Memory: Strings

- Strings are a bit different in **C-based programs** compared to **Python**
- They have a **fixed length**, and **can't be extended** without **creating a new string**
- Strings in C are actually **arrays of characters**, with each **character** being a **1-byte integer** which represents a printable letter/symbol.
- Strings in C end in a null byte (a byte with the value 0, or 0x00)



Memory: Strings - Example

00402000	01 00 02 00	4e 65 78 74-20 63 68 61 72 61 63 74Next charact
00402010	65 72 3a 20	25 2e 31 73-0a 00 00 00 01 1b 03 3b	er: %.1s.....;
00402020	38 00 00 00	06 00 00 00-04 f0 ff ff 6c 00 00 00	8.....1...
00402030	34 f0 ff ff	94 00 00 00-44 f0 ff ff ac 00 00 00	4.....D.....
00402040	64 f0 ff ff	54 00 00 00-4d f1 ff ff c4 00 00 00	d...T...M.....
00402050	a5 f1 ff ff	e4 00 00 00-14 00 00 00 00 00 00 00
00402060	01 7a 52 00	01 78 10 01-1b 0c 07 08 90 01 00 00	.zR..x.....
00402070	14 00 00 00	1c 00 00 00-08 f0 ff ff 26 00 00 00&...
00402080	00 44 07 10	00 00 00 00-24 00 00 00 34 00 00 00	.D.....\$.4...
00402090	90 ef ff ff	30 00 00 00-00 0e 10 46 0e 18 4a 0f0.....F..J.
004020a0	0b 77 08 80	00 3f 1a 39-2a 33 24 22 00 00 00 00	.w...?.9*3\$"....
004020b0	14 00 00 00	5c 00 00 00-98 ef ff ff 10 00 00 00\.....
004020c0	00 00 00 00	00 00 00 00-14 00 00 00 74 00 00 00t...
004020d0	90 ef ff ff	20 00 00 00-00 00 00 00 00 00 00 00
004020e0	1c 00 00 00	8c 00 00 00-81 f0 ff ff 58 00 00 00X...
004020f0	00 45 0e 10	86 02 43 0d-06 02 4f 0c 07 08 00 00	.E....C...O.....
00402100	1c 00 00 00	ac 00 00 00-b9 f0 ff ff 51 00 00 00Q...
00402110	00 45 0e 10	86 02 43 0d-06 02 48 0c 07 08 00 00	.E....C...H.....
00402120	00 00 00 00	



Memory: Pointers

- **Pointers** are special integers which store a **memory address**. They're **64-bits** (8 bytes) long on modern computers
- Pointers are **very common in C**
- Most functions which interact with **strings** actually take **string pointers**, which contain the **memory address** containing the **first character of the string**
- The function will then **read characters** until it reaches a **null byte**, then stops





Interactive Decompilers

What is an Interactive Decompiler

- **Interactive decompilers** analyse a program and **try to determine how it's structured**
- It will:
 - **Disassemble** the program's **machine code** into **assembly code**
 - Try to **Decompile** the program **back into C code**
 - Look for **strings** and **functions** within the program
 - Try to guess **what data type** different variables are



Why do we need one?

- Your computer **doesn't care about**:
 - The **names of variables**
 - The **names of functions** within your program
 - The **type** of each variable
- This means that when we **compile a program**, that **information is lost**
- It's our job to use an **interactive decompiler** to try and figure that information out from context





Key Static Analysis Concepts

Symbols

- **Symbols** are **names for functions** stored within the program
- Programs can **import** and export **functions/symbols**
 - **Imported functions** are functions from **outside libraries/programs**
 - **Exported functions** can be **called by other programs**
- All programs contain an “**entry**” or “**start**” function
- Programs written in C generally have a “**main**” function, which is called from a special mini function called **__libc_start_main**
- **Most programs** are compiled **without symbols**, meaning **we’re not given names** for most of their functions



The Static Analysis Process

- Static Analysis boils down to a **fairly simple loop**:
 1. Look at the **program's code**
 2. **Rename/retype** variables, which will **improve your decompiled C code**
 3. See if you can **understand the program better** with the **improved decompiled code**
 4. **Repeat**





Guided Example

Guided Example

1. Create a new project in Ghidra
2. Import “basic-function-with-symbols” into the project
3. Double click it to open the analysis window
4. Follow along with the guided analysis

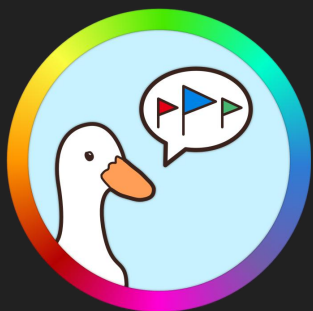




Challenges

Challenges (until end of lesson)

1. “shuffle” (in the google drive)
2. “Simple Encryptor” – Free rev challenge on HackTheBox



Interrupt Labs Internship

- Interrupt Labs have an internship out
- It's in Vulnerability Research, which involves a lot of reverse engineering
- Feel free to ask me about it, we have pamphlets about working in VR

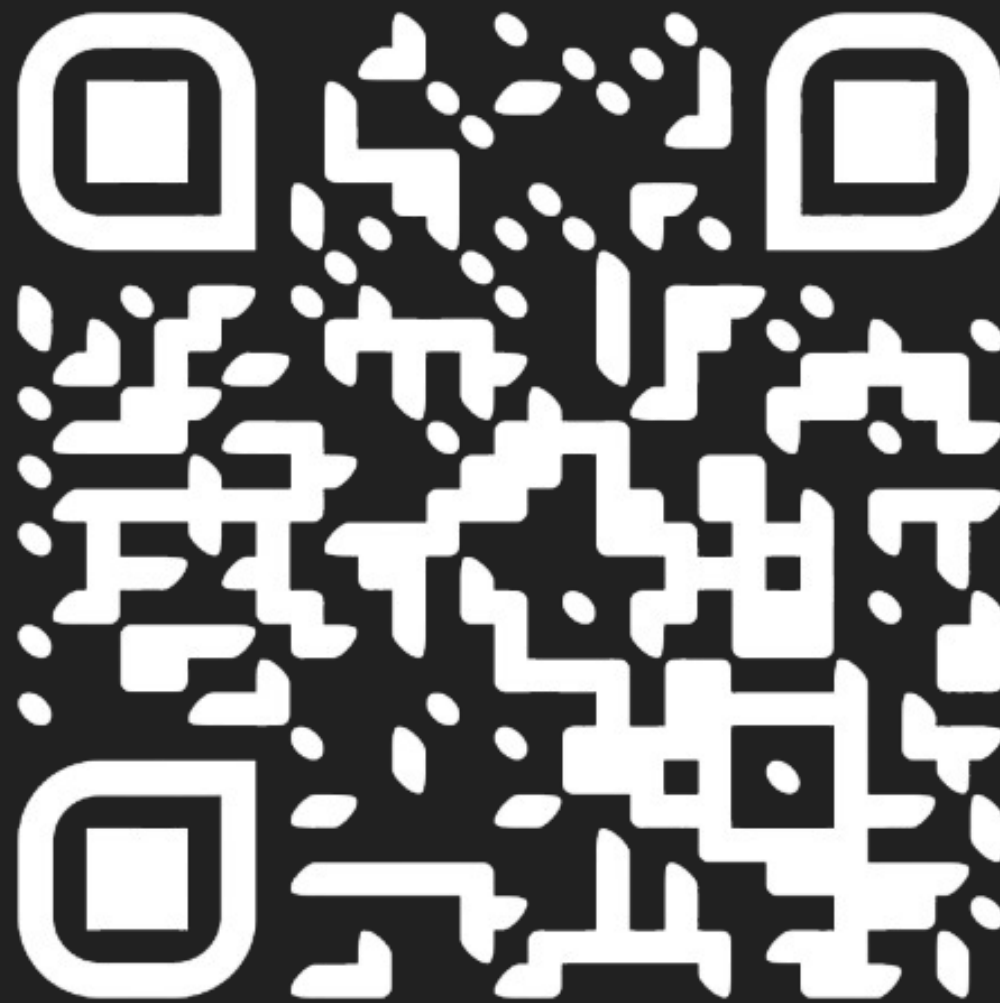


Aston x Warwick CTF



- beginner
- in-person
- pizza
- prizes





<https://forms.gle/oocdnikysoudbHDD7>

