Capture_7he_F1ag

Legal and fun way to Cybersecurity...

Basics of CyberSecurity:

- **A shield :** Cybersecurity protects our digital world—devices, networks, and data—from cyber threats like hackers and malware, much like a multi-layered fortress.
- **What is done to protect?** Various fields such as **network security** (guarding data in transit), **application security** (securing software), **endpoint security** (protecting individual devices), and **incident response** (swiftly countering breaches) comes under the field of **CyberSec**.
- **Ever-Evolving Battle:** Cybersecurity is a dynamic field where constant innovation meets creative defense strategies as new threats emerge and defenses adapt in real time.

What is Hacking?

Hacking is the practice of exploring computer systems to understand their inner workings and uncover vulnerabilities.

It is about pushing technological boundaries and continuously evolving in the digital landscape.

Threats :

Injection Attacks:

SQL and XSS injection to manipulate databases or web apps.

In 2009, attackers exploited a SQL injection vulnerability at Heartland Payment Systems, gaining access to millions of credit card records by manipulating input fields.

Malware & Ransomware:

Viruses or Ransomware can cripple systems by corrupting data or locking files until a ransom is paid.

WannaCry (2017), Exploited a Windows vulnerability to lock up computers worldwide, notably disrupting the UK's NHS and costing billions in losses.

DDoS Attacks:

Overloading networks with excessive traffic can shut down services, resulting in lost revenue and tarnished reputations.

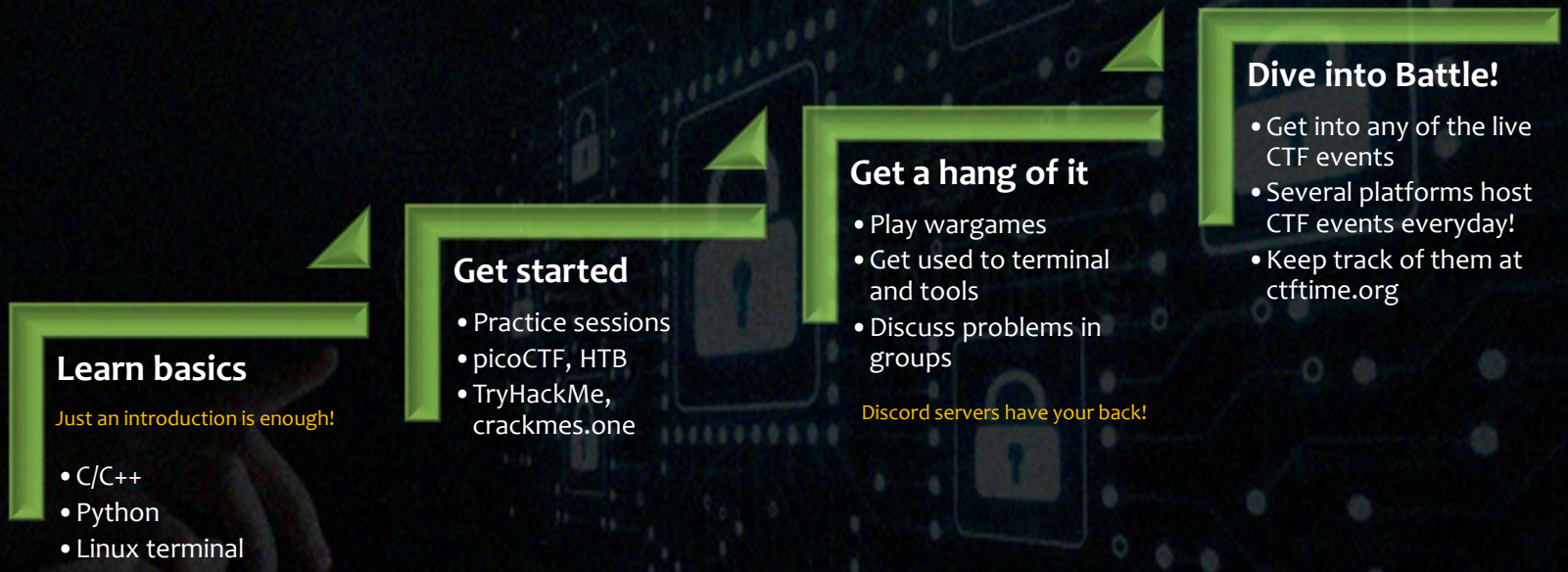
GitHub DDoS Attack (2018), In one of the largest DDoS events, GitHub was hit by an amplified attack peaking at 1.35 Tbps that led to a brief but critical service disruption, stalling code deployments and impacting developers worldwide.

Coming to the point now ...

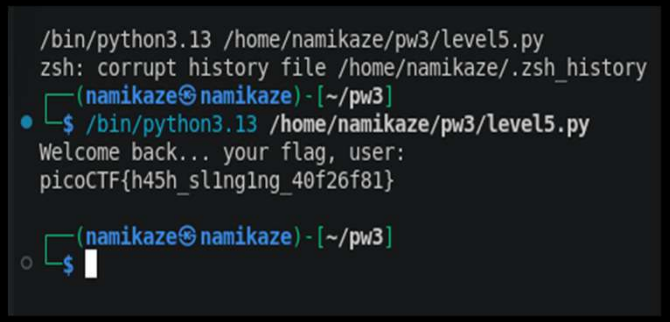
Capture the Flag!

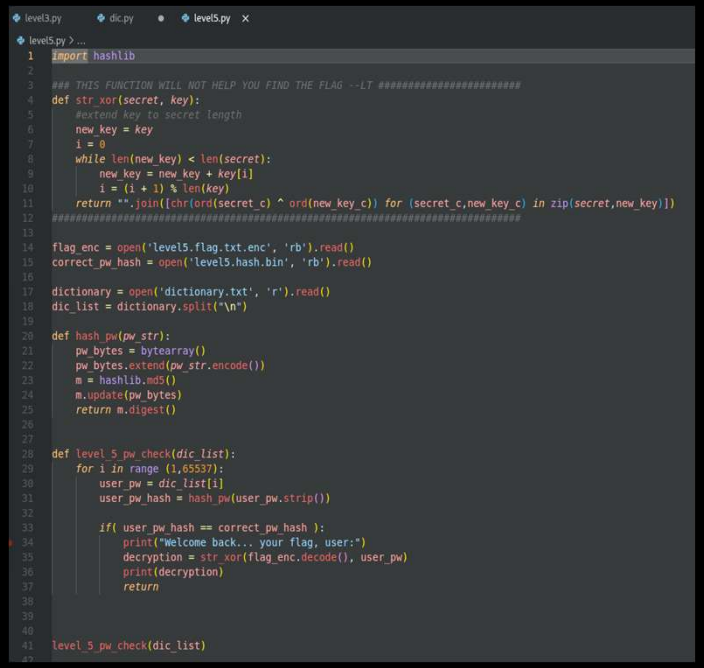
- **Interactive Learning Environment:** CTFs are hands-on competitions where participants solve cybersecurity puzzles by finding hidden "flags" within challenges.
- **Diverse Competition Formats:** They include Jeopardy-style contests (solving independent puzzles in areas like web, crypto, and binary) and attack-defense rounds (simulating real-world system breaches and defenses).
- **Skill Integration:** These events test a broad range of talents—from vulnerability discovery and exploit creation to toolkit development and operational security—merging theoretical concepts with practical application.
- **Real- World Relevance:** Success in CTFs mirrors professional cybersecurity skills, providing valuable experience and acting as an indicator for potential job performance in the field.

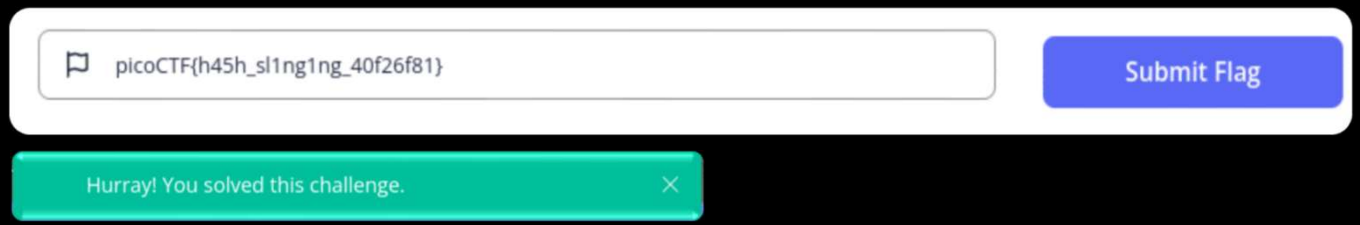
How to get into Capturing the Flags?

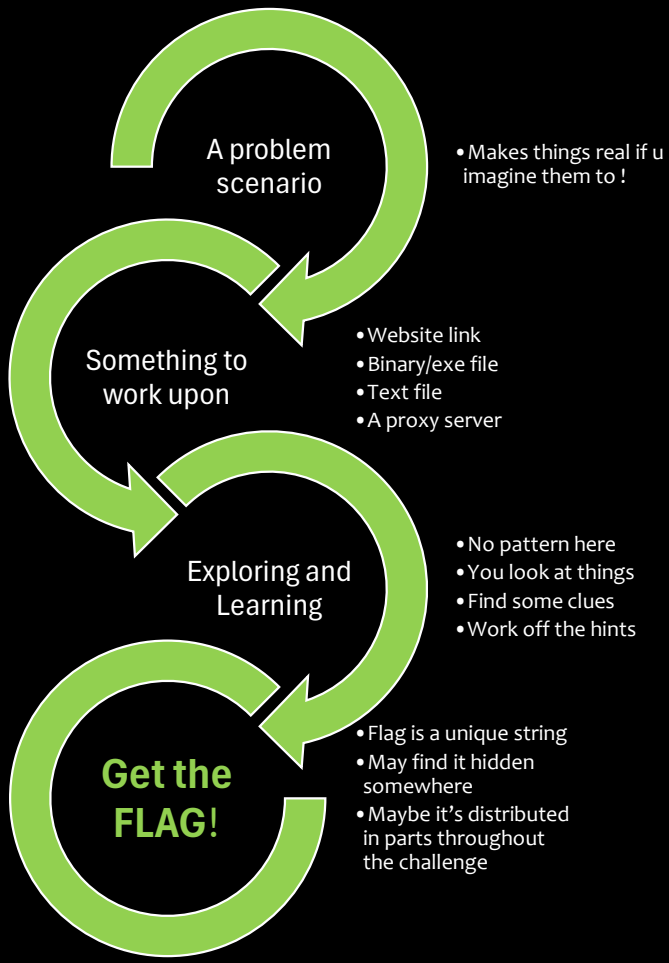


Almost all the CTF challenges have the same structure:









Challenges we face

```
Almost there!! Crack this hash: 916e8c4f79b25028c9e467f1eb8eee6d6bbdff965f9928310ad30a8d88697745
Enter the password for the identified hash: qwerty098
Correct! You've cracked the SHA-256 hash with a secret found.
The flag is: picoCTF{UseStr0nG_h@shEs_&PaSswDs!_869e658e}

namikaze2022-picoctf@webshell:~$
```

Hashes to crack and get the correct passwords...

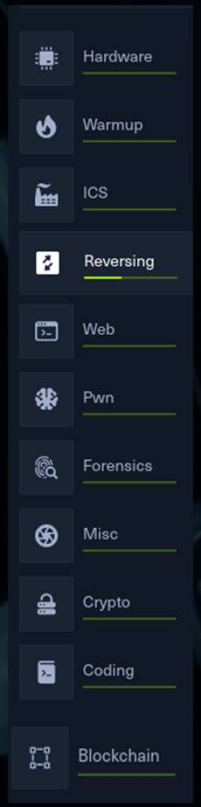
```
004022f8 51              PUSH    ECX
004022f9 8b 55 fc        MOV     EDX,dword ptr [EBP + local_8]
004022fc 52              PUSH    EDX
004022fd 6a 00           PUSH    0x0
004022ff ff 15 0c        CALL    dword ptr [->USER32.DLL::LoadStringA]      = u"FLAG{RESOURCES-ARE-POPULAR-F...
         30 40 00                                                           = 0000312a
00402305 8d 85 60        LEA     EAX=>local_4a4,[EBP + 0xfffffb60]
         fb ff ff
0040230b 50              PUSH    EAX
0040230c 8d 8d 68        LEA     ECX=>local_9c,[EBP + 0xffffff68]
         ff ff ff
00402312 e8 19 ff        CALL    MD5::digestString                          char * digestString(MD5 * this, ...
         ff ff
00402317 89 85 64        MOV     dword ptr [EBP + local_a0],EAX
         ff ff ff
0040231d 6a 30           PUSH    0x30
0040231f 68 30 30        PUSH    s_We've_been_compromised!_00403030         = "We've been compromised!"
         40 00
```

Analyzing disassembled code from binaries/executables

```
void entry(void)

{
  CHAR local_4a4;
  undefined1 local_4a3 [1027];
  char *local_a0;
  MD5 local_9c [144];
  HRSRC local_c;
  undefined4 local_8;

  MD5::MD5(local_9c);
  local_4a4 = '\0';
  memset(local_4a3,0,0x3ff);
  local_8 = 0;
  local_c = FindResourceA((HMODULE)0x0,"rc.rc",(LPCSTR)0x6);
  local_8 = 0x110;
  LoadStringA((HINSTANCE)0x0,0x110,&local_4a4,0x3ff);
  local_a0 = MD5::digestString(local_9c,&local_4a4);
  MessageBoxA((HWND)0x0,local_a0,"We\'ve been compromised!",0x30);
                    /* WARNING: Subroutine does not return */
  ExitProcess(0);
}
```

Understanding decompiled code to reverse engineer the algorithm

#Add one more screenshot from some new challenge type.
Maybe crypto or, Web?



Question types

The problems in CTF events are broadly divided into the following categories:

- Hardware
- ICS
- Reversing
- Web
- Python Wrangling
- Forensics
- Crypto
- Coding
- Blockchain
- Miscellaneous

Reverse Engineering (RE): A useful skill

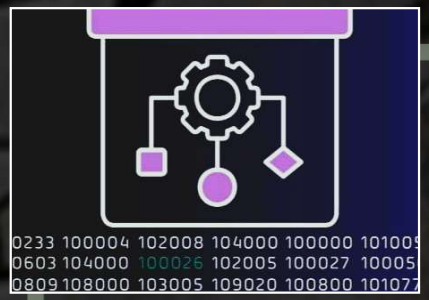
One of the more interesting and difficult type of challenge in RE :

- Provided a piece of code or any binary/exe file, the task would be to find out how it works modify or reverse the process to accomplish certain things (like obtaining a flag).

Analysing binaries

- Participants are provided with a compiled program (e.g., .exe , .bin , or .elf files).
- The goal is to understand the program's functionality without access to its source code.

- Challenges may require reverse-engineering algorithms used in the program.
- This could involve deciphering encryption methods, compression techniques, or custom logic.



Understanding Algorithms

Bypassing Protection

- Some challenges include anti-debugging mechanisms or obfuscation techniques.
- We may be required to bypass these protections to analyze the program effectively.

Common tools used for RE:

Disassembler

Tools like **IDA Pro, Ghidra** or **Radare2** are used to convert binaries into assembly code.

- Converts binary code into assembly language.
- Visualizes control flow and functions.
- Allows annotations for easier analysis.

Debuggers

Tools like **GDB, OLLY DBG** help in through the program's execution.

- Executes programs step-by-step in real time.
- Sets breakpoints and inspects memory and registers.
- Helps bypass anti-debugging techniques.

Hex Editors

Tools like **HxD** allow us to view and edit the raw binary data.

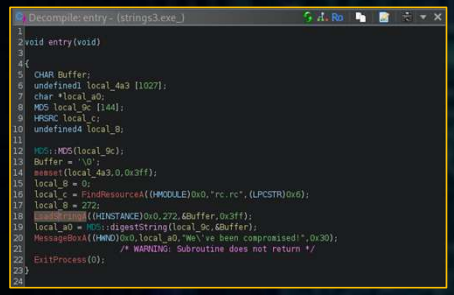
- Displays file data in hex and ASCII.
- Permits direct byte-level modifications.
- Detects Patterns and embedded signature.

Decompiler

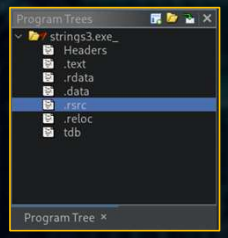
Tools like **Binary** Ninja or **Hopper** attempt to reconstruct source code from binaries.

- Converts binaries into high-level pseudocode.
- Highlights loops, conditionals and structures.
- Accelerates overall code comprehension.

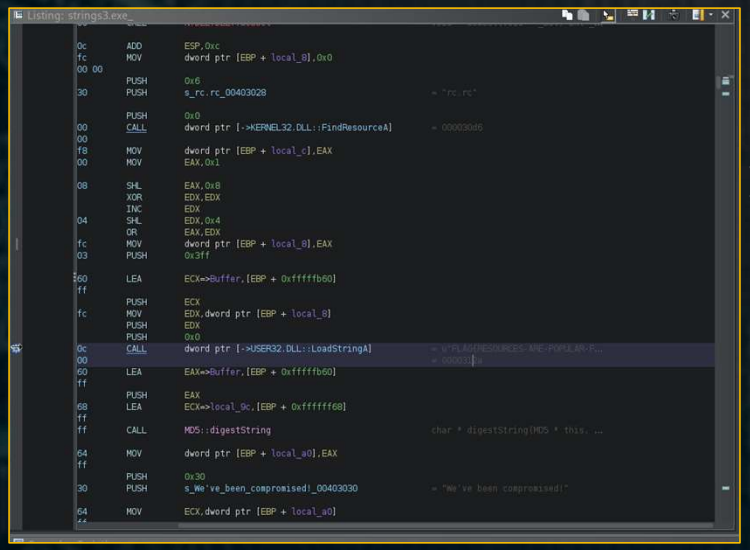
Ghidra: An Open-source Disassembler



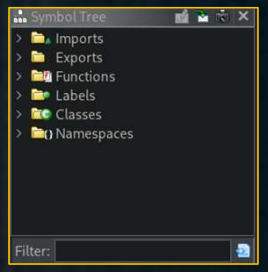
Decompiler: To view a pseudo source code in C



Program Tree: Displays the hierarchical organization of memory blocks and sections within a program's binary



Disassembly: To view Assembly code form the binary/exe files



Symbols Tree: Displays all the symbols within a program, such as functions, variables, and labels

Strings view: To find all strings available in the binary header

NOTE

- Ghidra is a powerful tool for analyzing binaries and understanding code execution flow through static analysis.
- By thoroughly examining a binary with Ghidra, we can uncover hidden functions that may not be apparent during runtime.
- Additionally, in CTF challenges, Ghidra's string analysis can quickly reveal function call references or even the flag itself in the Strings view, significantly simplifying the reversing process.

Techniques used for RE:

Static Analysis (Without Executing the Binary)

Disassembly: Using tools like Ghidra, IDA to analyze assembly code.

Decompilation: Converting binary code back into a high-level language (C, Python).

String Analysis: Extracting human-readable strings to find function calls and flag data.

Symbol Table Analysis: Checking for Function names, global variables, and debug symbols.

Control Flow Analysis: Understanding function calls and loops through CFG(Control Flow Graphs).

Dynamic Analysis (Executing and Observing Behaviour)

Debugger Usage: Using GDB, WinDBG or OllyDBG to set breakpoints and inspect registers/memory.

Tracing Execution: Using tools like strace (Linux) or API Monitor (Windows) to track system calls.

Memory Inspection: Dumping and analyzing process memory with tools like Volatility or Cheat Engine.

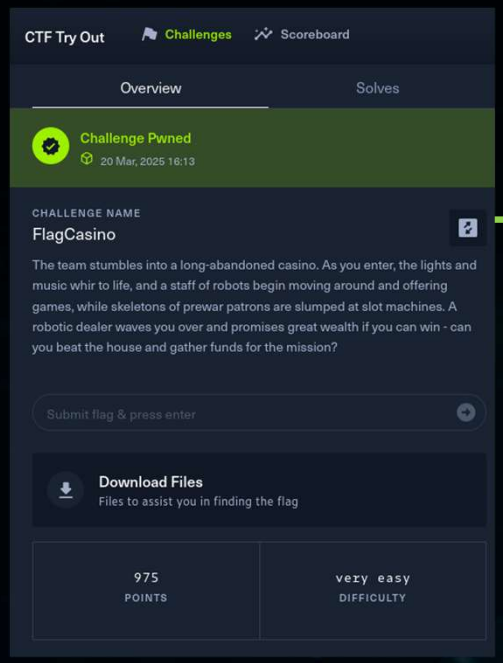
Hooking & Patching: Injecting custom code to modify behaviour at runtime.

During the past few days:

Basic Malware RE	Medium	Reverse	Solved		FLAG{CAN-MAKE-IT-ANYMORE	March 23, 2025	March 24, 2025
EncryptedScroll [CyberApocalypse]	Very Easy	Reverse	Solved	900	HTB{s1mpl3_fl4g_4r1thm3t1c}	March 22, 2025	March 22, 2025
SealedRune	Very Easy	Reverse	Solved	850	HTB{run3_m4g1c_r3v34l3d}	March 22, 2025	March 22, 2025
Endlesscycle	Easy	Reverse	Need Help	950		March 22, 2025	
Flag Casino [Try-Outs]	Very Easy	Reverse	Solved	975	HTB{needle_in_the_stack} some	March 17, 2025	March 20, 2025

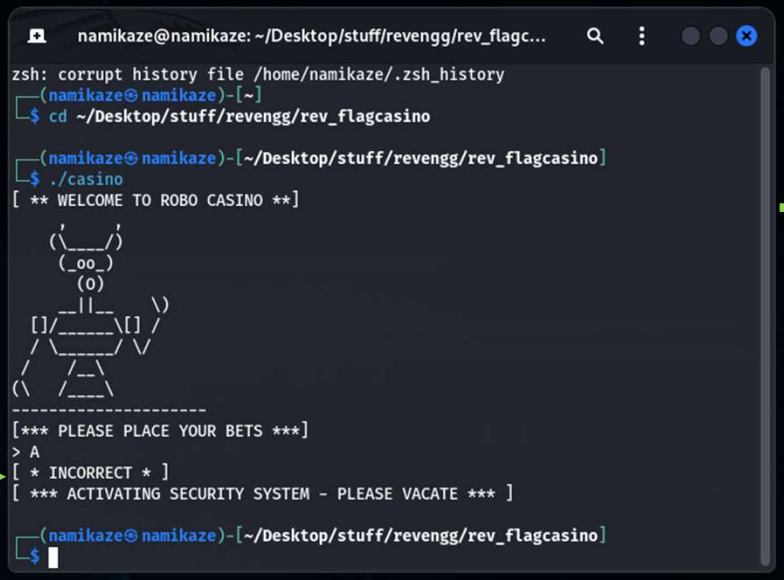
and many more ...

Flag Casino (HTB Try-outs): RE challenge



A problem from CTF Try-out on HackTheBox platform.
Reversing category;
Difficulty: Very Easy

Provided a zip file containing a single binary named casino.



On running the binary, I was prompted for an input used a random char ‘A’, to checkout how the program goes.
No clue found from this output yet.

Flag Casino: Continued...

Decompile: main - (casino)

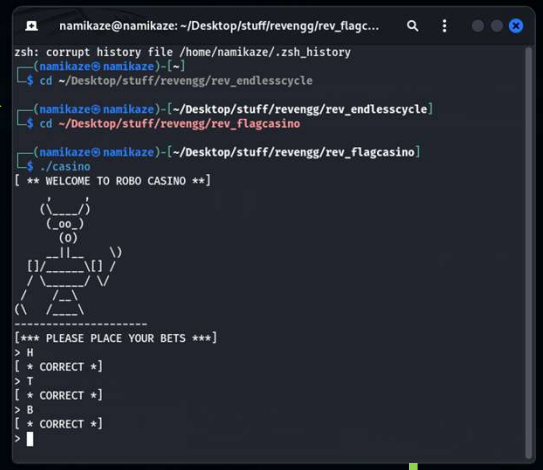
```
undefined8 main(void)

{
  int buffer;
  char input;
  uint num;

  puts("[ ** WELCOME TO ROBO CASINO **]");
  puts(
      "     ,     ,\n    (\\____/)\n     (_oo_)\n       (O)\n     __||__    \\)\n  []/_____\\[]\n     / \\_____/ \\/\\n /    /__\\\\n(\\   /____\\\\n----------"
      );
  puts("[*** PLEASE PLACE YOUR BETS ***]");
  num = 0;
  while( true ) {
    if (28 < num) {
      puts("[ ** HOUSE BALANCE $0 - PLEASE COME BACK LATER ** ]");
      return 0;
    }
    printf("> ");
    buffer = __isoc99_scanf(&DAT_001020fc,&input);
    if (buffer != 1) break;
    srand((int)input);
    buffer = rand();
    if (buffer != *(int *)(check + (long)(int)num * 4)) {
      puts("[ * INCORRECT * ]");
      puts("[ *** ACTIVATING SECURITY SYSTEM - PLEASE VACATE *** ]");
                    /* WARNING: Subroutine does not return */
      exit(-2);
    }
    puts("[ * CORRECT *]");
    num = num + 1;
  }
                    /* WARNING: Subroutine does not return */
  exit(-1);
}
```

Decompile: main × | Defined Strings × | Functions ×

Looking at the decompiled pseudo-code in ghidra, the following points can be noted:

- We are supposed to enter a single character for each loop
- The loop is supposed to run for 29 times (maybe the length of flag?)
- We see that the input char is converted to int and used to seed the rand() function.
- Then the number from rand() is compared to the int value stored in resource check

Knowing that all HTB flags have the structure HTB{flag}. I guessed maybe, it's supposed to check flag as the input. So, I tried the characters H, T, B ... Clue spotted!

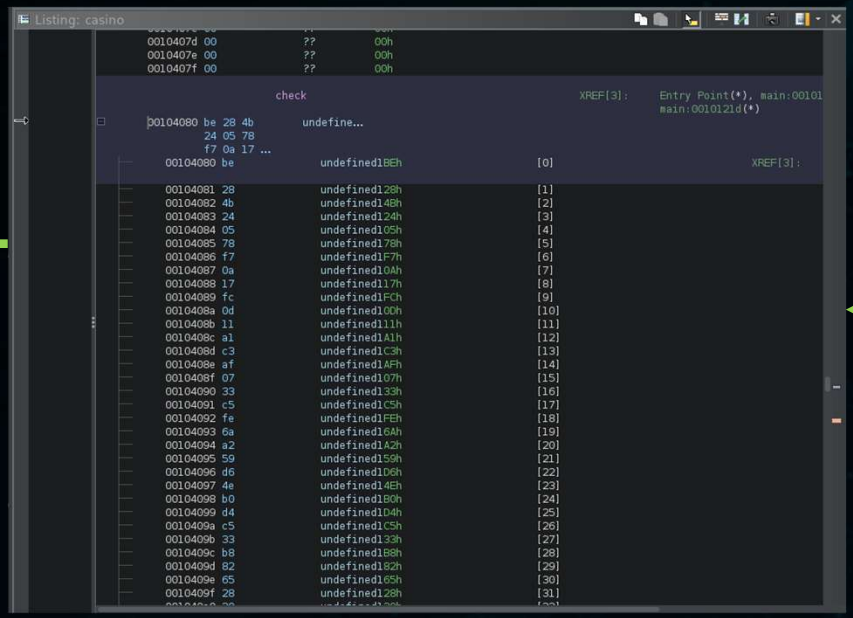


Flag Casino: Continued...

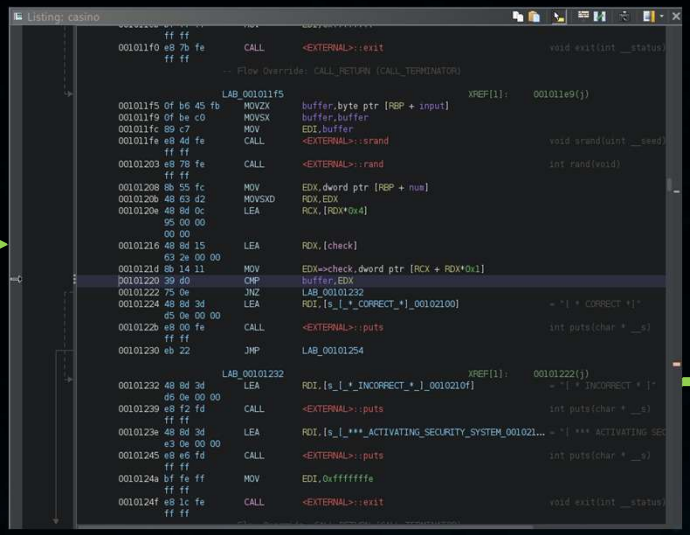
Analyzing the while loop further to understand how the input is checked:

- we know that the rand() gives a fixed value for a constant seeding.
- the rand() value is stored in buffer which is then compared to check
- notice that for each successful iteration, the next comparison is made by moving 4 steps ahead in check.

```
while( true ) {
  if (28 < num) {
    puts("[ ** HOUSE BALANCE $0 - PLEASE COME BACK LATER ** ]");
    return 0;
  }
  printf("> ");
  buffer = __isoc99_scanf(&DAT_001020fc,&input);
  if (buffer != 1) break;
  srand((int)input);
  buffer = rand();
  if (buffer != *(int *)(check + (long)(int)num * 4)) {
    puts("[ * INCORRECT * ]");
    puts("[ *** ACTIVATING SECURITY SYSTEM - PLEASE VACATE *** ]");
                    /* WARNING: Subroutine does not return */
    exit(-2);
  }
  puts("[ * CORRECT *]");
  num = num + 1;
}
                    /* WARNING: Subroutine does not return */
exit(-1);
}
```



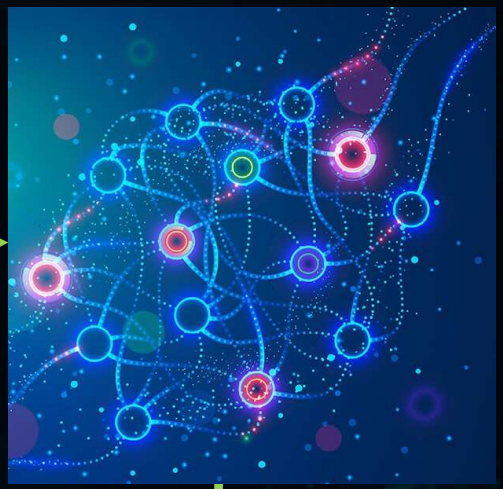
The check resource contains a series of hex values of undefined data type (this is typecasted to (int *)) in the code.
So, in each loop the comparison value is moved by 4 bytes in check.



The one thing that comes to mind is to change the source code in such a way that we obtain the values in check, but for that I'd have to change assembly code … which is a rather difficult endeavour .

So, can it be done in any other way?

- ✓ The approach is to make a map of all characters as per their respective rand() index obtained by using the character itself as seed.
- ✓ Then, all I need to do is get the integer value that check is supposed to compare buffer with, because the character in the map at that index will lead to the flag string (by combining all such characters).



test.sh level5.py lvl5.txt decrypt.py enc.py

rev_flagcasino > script.py > ...

```python
from pwn import *
import ctypes as c

libc = c.CDLL('libc.so.6')
map={}
for seed in range(255):
    libc.srand(seed)
    map[libc.rand()]=chr(seed)

flag = ""
casino = ELF("./rev_flagcasino/casino", checksec=False)

for num in range(29):
    key = casino.u32(casino.sym["check"] + num*4)
    flag += map[key]

print(flag)
```

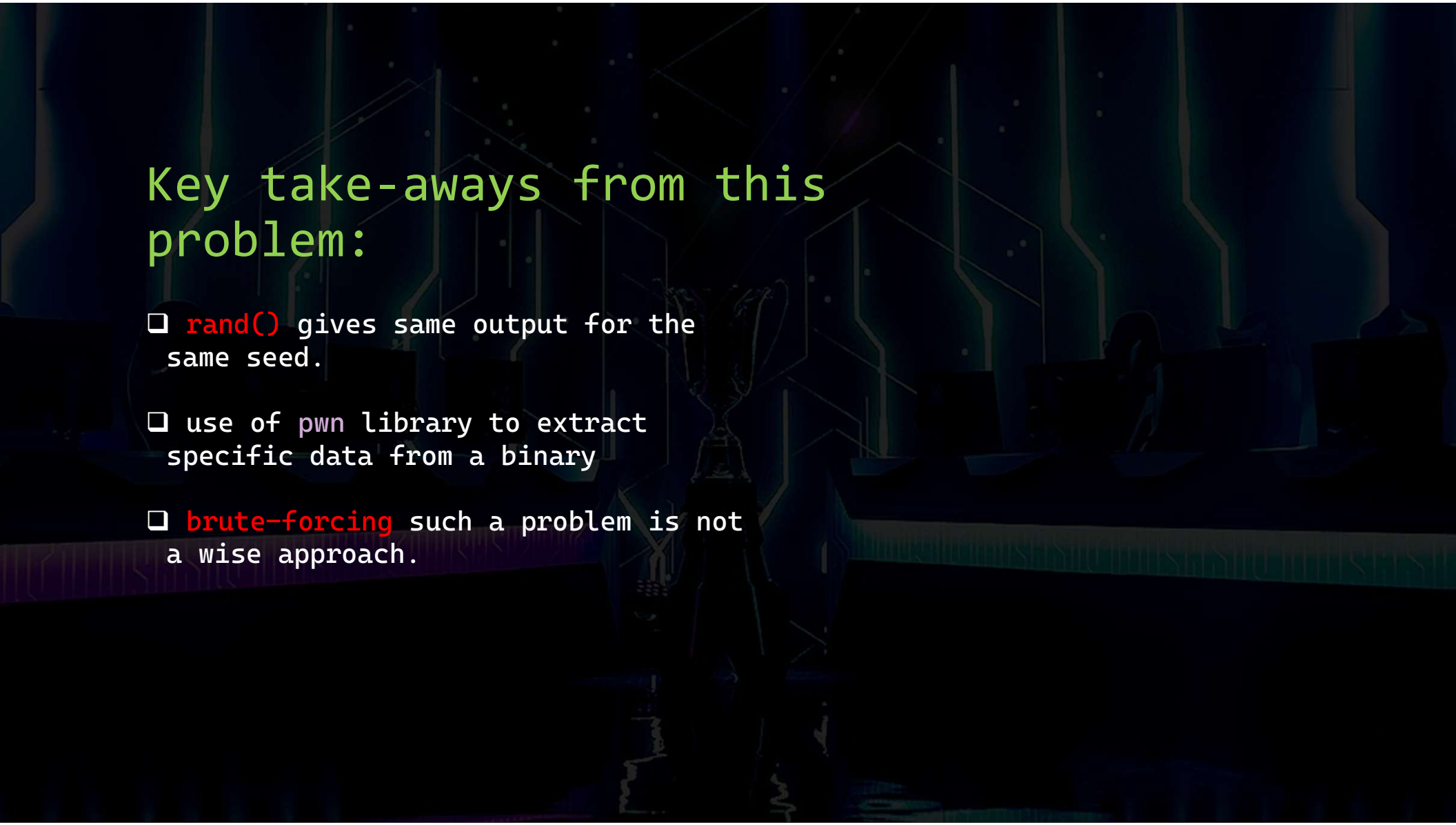
This python script does that task. Using the pwn library, the data check can be extracted from the binary file. Moreover, the ctypes library enables the use of C functions to emulate the features of obfuscation of string using rand() function.

Running the script, the flag itself is directly revealed!

Flag: HTB{r4nd_1s_v3ry_pr3d1ct4bl3}

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
  File "/home/namikaze/Desktop/stuff/rev_flagcasino/script.py", line 15, in <module>
    flag += map[key]
            ~~~^^^^^
KeyError: 0

┌──(nami/bin/python3.13 /home/namikaze/Desktop/stuff/rev_flagcasino/script.py
Traceback (most recent call last):
  File "/home/namikaze/Desktop/stuff/rev_flagcasino/script.py", line 15, in <module>
    flag += map[key]
            ~~~^^^^^
KeyError: 0

┌──(nami/bin/python3.13 /home/namikaze/Desktop/stuff/rev_flagcasino/script.py
HTB{r4nd_1s_v3ry_pr3d1ct4bl3}

┌──(namikaze㉿namikaze)-[~/Desktop/stuff]
└─$
```



Key take-aways from this problem:

- ❑ rand() gives same output for the same seed.
- ❑ use of pwn library to extract specific data from a binary
- ❑ brute-forcing such a problem is not a wise approach.

Resources used:

- ❑ Ghidra for disassembly, Decompilation
- ❑ HackTheBox for the challenge.
- ❑ binqqer.com and medium.com for problem writeups.