察言观色也能挖到0day？在聊天记录中的漏洞挖掘

原创 队员编号030 酒仙桥六号部队 7月2日

这是 **酒仙桥六号部队** 的第 **30** 篇文章。

全文共计2229个字，预计阅读时长8分钟。

0x01 前言

大家在做代码审计或者学习代码审计的过程中，会有大量时间是对着代码的。有时候会觉得代码枯燥无聊，看代码看到怀疑自我。

这时候不妨通过其他思路，换个思维，看点有趣的相关事务。回过头来再看代码，也许会有意想不到的惊喜！

0x02 查看各种记录

更新日志

我在 GITHUB上找到了一个合适的开源 CMS项目。在其官网上可以看到该CMS的更新的日志。

查看更新日志（CHANGELOG）：

```
#### RELEASE NOTES
[redacted]-CMS V.1.2.4 (2019-10-27)
- Add setting on content page for member login can see (Choose User Group).
- Add setting on Article and Gallery plugin for member login can see (Choose User Group).
- Add div class on form builder.
- Add Vietnamese language on backend.
- Add Line notify libraries.
- Add copy_as on user groups.
- Fixed file manager upload vulnerability (Github Issue#20).
- Fixed more important bug.
- Improve performance.

[redacted]-CMS V.1.2.3 (2019-08-19)
- Add French language on backend.
- Update Bootstrap to 3.4.1.
- Fixed Time-based blind SQL injection Vulnerability (Github Issue#19).
- Fixed Private Message (PM) error.
- Fixed Article plugin pagination bug.
- Fixed not htaccess support bug.
- Fixed more important bug.
- Improve performance.
```

我们可以看到 1.2.3 版本修复了一个 SQL 盲注（Issue#19）和 1.2.4版本修复了一个文件管理器上传漏洞（Issue#20）。

查看该 CMS 的代码结构，发现是基于 CodeIgniter 框架(后文简称 CI)进行的二次开发。对 CI 的相关类进行了继承，并自己封装了函数，进行一些特殊处理。

ISSUE记录

我们先来看下这个 SQL 盲注，首先访问下 GITHUB 上 Issue#19 的页面看下详情。

satuer commented on 19 Jun • edited ...

Title: Time-based blind SQL injection Vulnerability in [redacted]/core/MY_Security.php on [redacted]-1.2.2
Discovered by: @satuer from ABT Labs
Security: high (dump database)
Software: https://github.com/[redacted]/archive/1.2.2.zip

Description:
There is a high-risk time-based SQL injection vulnerability on the frontend login page(http://website/[redacted]-1.2.2/member/login/check). Exploit this vulnerability could dump the whole database without login.
When login in the frontend login page, if the csrf_csz parameter is removed or padded any string, a log will be recorded in the table 'login_logs' with reason 'CSRF Protection Invalid'. But the 'HTTP_USER_AGENT' field is used directly without any check when inserting the recorded. By constructing a special 'HTTP_USER_AGENT' field, this vulnerability can be exploited.

在前端登录页面有一个高风险基于时间的SQL注入漏洞。利用这个漏洞可以在不登陆的情况下拖取整个数据库的漏洞。

当处在前台登录页面登录时，如果csrf_csz参数被移除或者被其他字符串填充。一条记录会以"CSRF保护无效"为理由记录到login_logs表中。但是HTTP_USER_AGENT没有经过任何检测直接插入到记录中。通过构造一个特殊的HTTP_USER_AGENT，这个漏洞就可以被利用。

files: [redacted]/core/MY_Security.php

User-Agent: '-(if(condition, sleep(5), 1))-'', '192.168.1.11','time') # ← payload
If the condition is true, the page will return after 5 seconds, otherwise it will return immediately. ← 如果条件正确，页面会在5秒之后返回，否则会立刻返回。

When UA is '-(if((substr((select email from user_admin limit 1), 1, 1)='1'),sleep(5),1))-'', '192.168.1.11','time') #
The sql string is INSERT INTO login_logs (email_login, note, result, user_agent, ip_address, timestamp_create) VALUES ('', 'CSRF Protection Invalid', 'CSRF_INVALID', ''-(if((substr((select email from user_admin limit 1),1,1)='1'),sleep(5),1))-'', '192.168.1.11','time') #', '192.168.62.1', '2019-06-19 21:17:34.000000')

The insert statement does not actually execute because the last column is not in the correct time format. But the sub select statement will execute, and the sleep will execute if condition is true. ← 这条插入语句并不会被真的执行，因为最后一个字段是不正确的时间格式。但是语句中的select语句会被执行，如果条件正确sleep会执行。

So, the records in the login_logs table will not increase. The IP will not be added to the blacklist and condition($count < 10 in line 45) is always true. Then you can dump whole database one by one byte. ← 所以，在login_logs表中的记录并不会增加。IP也不会被添加到黑名单中，并且条件（在漏洞文件45行）一直回是true。你可以一个字节一字节的拖取整个数据库。

Suggest:
Check UA before inserting UA into sql statement ← 修复建议：在拼接到SQL语句之前，检查HTTP_USER_AGENT

这个是一个 1.2.2版本上报的 ISSUE 信息，从中我们可以看出这位漏洞上报者来自 ABT 实验室。漏洞信息非常全面，格式优美，图文并茂。

在 Issue 中不但给出了问题文件的路径是/core/MY_Security.php ，还给出了 Payload 细节：在前端登录时，User-Agent: '-(if(condition, sleep(5), 1))-'', '192.168.1.11','time')# 。并且将漏洞产生原因进行了详细阐述：由于参数缺失，该 CMS 会记录下该登录包的信息，以"无效的 CSRF 防护"为原因写入数据库。但是信息中的 HTTP_USER_AGENT 这个参数未作任何检测，就直接拼接到 SQL 语句中，故造成了漏洞可以被利用。

在最后漏洞上报者还给出了修复建议。

我们根据给出了文件名找到了 1.2.2 版本的 MY_Securtiy.php 文件。

MY_Securtiy 是继承 CI 框架的 CI_Security，并对其一些常用函数进行了封装扩展。

```php
<?php
defined('BASEPATH') OR exit('No direct script access allowed');
class MY_Security extends CI_Security {

    protected $ip_address = FALSE;
    protected $_enable_xss = FALSE;

```

我们的目标函数 csrf_show_error就是其中之一，我们先来看下该函数的调用过程：

Security.php

```php
    public function csrf_verify()
    {
        // If it's not a POST request we will set the CSRF cookie
        if (strtoupper($_SERVER['REQUEST_METHOD']) !== 'POST')
        {
            return $this->csrf_set_cookie();
        }

        // Check if URI has been whitelisted from CSRF checks
        if ($exclude_uris = config_item('csrf_exclude_uris'))
        {
            $uri = load_class('URI', 'core');
            foreach ($exclude_uris as $excluded)
            {
                if (preg_match('#^'.$excluded.'$#i'.(UTF8_ENABLED ? 'u' : ''), $uri->uri_string()))
                {
                    return $this;
                }
            }
        }

        // Check CSRF token validity, but don't error on mismatch just yet - we'll want to regenerate
        $valid = isset($_POST[$this->_csrf_token_name], $_COOKIE[$this->_csrf_cookie_name])
            && hash_equals($_POST[$this->_csrf_token_name], $_COOKIE[$this->_csrf_cookie_name]);

        // We kill this since we're done and we don't want to pollute the _POST array
        unset($_POST[$this->_csrf_token_name]);

        // Regenerate on every submission?
        if (config_item('csrf_regenerate'))
        {
            // Nothing should last forever
            unset($_COOKIE[$this->_csrf_cookie_name]);
            $this->_csrf_hash = NULL;
        }

        $this->_csrf_set_hash();
        $this->csrf_set_cookie();

        if ($valid !== TRUE)
        {
            $this->csrf_show_error();
        }
```

调用在CI框架中的 Security.php 文件：在 csrf 验证时，如果 csrf_token 不合法，则会调用csrf_show_error 函数。所以我们在构建请求报文时，需要将 csrf_token 参数不设置或者改动一下。

CI_Security中的csrf_show_error:只有错误显示。

```
290 ▼ ——/**
291 ——·*·Show·CSRF·Error
292 ——·*
293 ——·*·@return——void
294 ——·*/
295 ——public·function·csrf_show_error()
296 ——{
297 ————show_error('The·action·you·have·requested·is·not·allowed.',·403);
298 ——}
299
```

My_Security中的csrf_show_error:增加了对错误登录的记录入库并细分了错误种类。

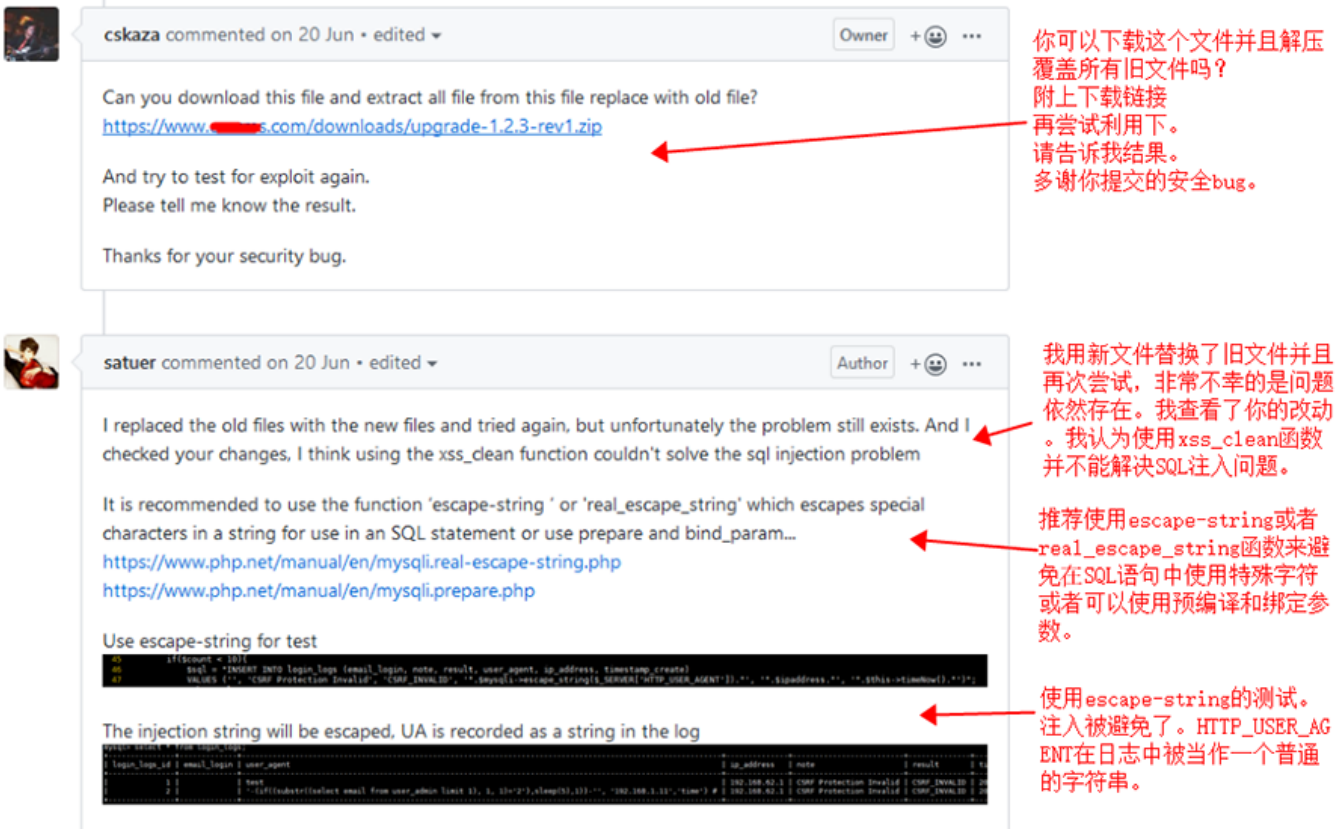
```
34  ····*·Show·CSRF·Error
35  ····*
36  ····*·@return——void
37  ····*/
38  ····public·function·csrf_show_error()·{
39  ········$ipaddress·=·$this->ip_address();
40  ········$mysqli·=·new·mysqli(DB_HOST,·DB_USERNAME,·DB_PASS,·DB_NAME);
41  ········$query·=·$mysqli->prepare("SELECT·ip_address·FROM·login_logs·WHERE·ip_address·=·'"·.·$ipaddress·."'·AND·
            result·=·'CSRF_INVALID'·AND·timestamp_create·>=·DATE_SUB('".$this->timeNow()."',INTERVAL·5·MINUTE)");
42  ········$query->execute();
43  ········$query->store_result();
44  ········$count·=·$query->num_rows;
45  ········if($count·<·10){
46  ············$sql·=·"INSERT·INTO·login_logs·(email_login,·note,·result,·user_agent,·ip_address,·timestamp_create)
47  ············VALUES·('',·'CSRF·Protection·Invalid',·'CSRF_INVALID',·'".$_SERVER['HTTP_USER_AGENT']."',·'".$ipaddress
                ."',·'".$this->timeNow()."')";
48  ············$mysqli->query($sql);
49  ········}
50  ········$mysqli->close();
51  ········if·(!empty($_SERVER["HTTP_REFERER"]))·{
52  ············$referer_host·=·@parse_url($_SERVER['HTTP_REFERER'],·PHP_URL_HOST);
53  ············$own_host·=·parse_url(config_item('base_url'),·PHP_URL_HOST);
54  ············if·(($referer_host·&&·$referer_host·===·$own_host))·{
55  ················$this->clearCSRFcookie();
56  ················header('Refresh:2;url='·.·$_SERVER["HTTP_REFERER"]·.·'?nocache='·.·time());
57  ················show_error('The·action·is·not·allowed·by·CSRF·Protection.·Please·wait·2·seconds·to·redirect.',·403)
                    ;
58  ············}else{
59  ················$this->clearCSRFcookie();
60  ················show_error('The·action·is·not·allowed·by·CSRF·Protection.·Please·clear·your·browser·cookie·and·
                    cache.',·403);
61  ············}
62  ········}else{
63  ············$this->clearCSRFcookie();
64  ············show_error('The·action·is·not·allowed·by·CSRF·Protection.·Please·clear·your·browser·cookie·and·cache.',
                ·403);
65  ········}
66  ····}
```

第45行，即漏洞上报者提到的condition，如果语句不会被执行，则count也不会增加，所以条件一直为true。

第46行，可以看出直接将 $_SERVER['HTTP_USER_AGENT'] 拼接到 SQL 的 INSERT 语句中，并没有任何过滤，所以产生了SQL 注入。

看起来简单明了，这个漏洞应该会被完美的修复掉。但事实上这个漏洞修复的过程并没有这么简单。在这个Issue#19中后续有一段很有趣的CMS开发者和漏洞上报者之间的对话记录，引起了我的兴趣：

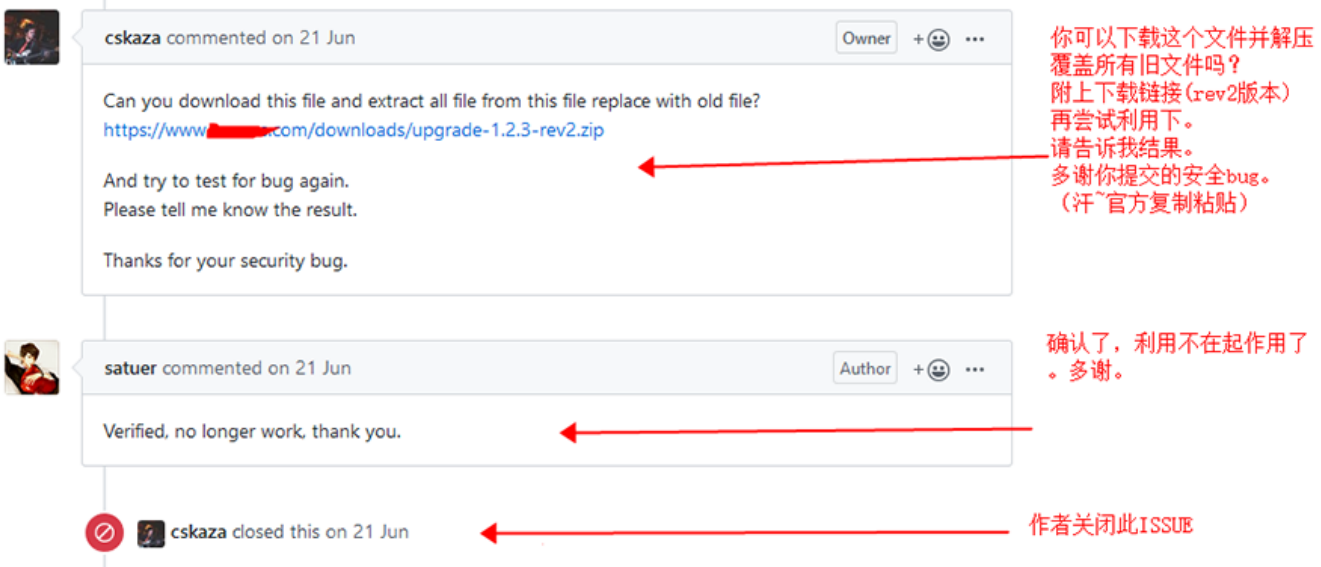
聊天记录



漏洞上报的版本为 1.2.2，而这时开发者已经开发出了 1.2.3-rev1版本，并且尝试修复这个 SQL 注入的问题，并希望漏洞上报者使用 1.2.3-rev1 来看看是否修复了此漏洞。

这个漏洞上报者非常有耐心，他在实际尝试后发现问题并没有被修复的同时，还查看 1.2.3-rev1 的 MY_Security.php 的代码。他告诉CMS开发者使用 xss_clean 函数并不能解决 SQL注入的问题，并且告诉他应该使用正确的函数为：escape-string 或者real_escape_string 这两个函数来防止 SQL 注入，并给出了这两个函数在 PHP.NET 的官方链接和以及修复之后的代码以及测试结果。

再来看之后的对话：



这个CMS开发者很勤奋，第二天就放出了 1.2.3-rev2 版本尝试修复了这个问题，并且再次邀请漏洞上报者测试是否修复此问题。

漏洞上报者于当日确认已经修复，Payload 不再起作用，然后开发者关闭了此问题。至此一个开源 CMS 的安全问题被修复了。

0x03 从中所得

我们可以从中得到什么？

看起来一切正常，但是我们能从这段对话中得到什么呢？

我们这个开发者是有基本安全意识的，该 CMS 包含 xxs_clean 函数，证明开发者在已经意识到 XSS 问题并且封装了相应的函数来做过滤处理。

但是该开发者对于安全问题的细节认识并不太清晰。在碰到 SQL 注入问题时企图用 xss_clean 来解决这个问题，这点说明他对于 SQL 注入产生的原因和修复方法并不太明确。

我们来看下GITHUB上 1.2.3-rev2 版本中开发者对于这个问题的修复方案：

```
46              $sql = "INSERT INTO login_logs (email_login, note, result, user_agent, ip_address, timestamp_create)
    -           VALUES ('', 'CSRF Protection Invalid', 'CSRF_INVALID', '".$_SERVER['HTTP_USER_AGENT']."', '".$ipaddress."', '".$this->timeNow()."')";
47  +           VALUES ('', 'CSRF Protection Invalid', 'CSRF_INVALID', '".$this->xss_clean($mysqli->escape_string($_SERVER['HTTP_USER_AGENT'])).."', '
48              $mysqli->query($sql);
```

开发者很听话，使用了 escape_string 来修复 SQL 注入的问题，做的很好。但是他并没有去掉 xss_clean 方法，而是在escape_string 调用之后仍然还继续调用 xss_clean 方法。

这是为什么呢？我们来尝试猜测开发者的想法：在1.2.3-rev2版本中使用xss_clean后未修复，于是加上了escape_string来修复该漏洞。xss_clean 是处理危险字符的方法，escape_string也是处理危险字符的方法，两个过滤危险字符的方法叠加起来，理应是更安全的。就像两个 WAF 串联叠加，不应该是难上加难吗？

但是事实可能并非如此，也许正是这种情况给了我们绕过机会！

绕过修复方案

首先看一下 escape_string 到底转义了哪些字符：

下列字符受影响：

- \x00
- \n
- \r
- \
- '
- "
- \x1a

由于这段 SQL 注入是字符型，我们需要 ' 来闭合语句，但是 escape_string 会将 ' 变为 \' 从而阻止 SQL 注入。

我们来看下 xss_clean 中的代码：

```
354     public function xss_clean($str, $is_image = FALSE)
355     {
356         // Is the string an array?
357         if (is_array($str))
358         {
359             foreach ($str as $key => &$value)
360             {
361                 $str[$key] = $this->xss_clean($value);
362             }
363
364             return $str;
365         }
366
367         // Remove Invisible Characters
368         $str = remove_invisible_characters($str);
369
370         /*
371         * URL Decode
372         *
373         * Just in case stuff like this is submitted:
374         *
375         * <a href="http://%77%77%77%2E%67%6F%6F%67%6C%65%2E%63%6F%6D">Google</a>
376         *
377         * Note: Use rawurldecode() so it does not remove plus signs
378         */
379         if (stripos($str, '%') !== false)
380         {
381             do
382             {
383                 $oldstr = $str;
384                 $str = rawurldecode($str);
385                 $str = preg_replace_callback('#%(?:\s*[0-9a-f]){2,}#i', array($this, '_urldecodespaces'), $str);
386             }
387             while ($oldstr !== $str);
388             unset($oldstr);
389         }
```

第 354-365 行是一个递归调用。

第 368行是一个移除不可见字符的方法，如：\x00、\x01 等等。

第 379-389 行是我们的关键代码，这段代码是判断 $str 参数中是否包含 % 字符，如果有的话就判定为需要 URL 解码并且调用 rawurldecode 来进行 URL 解码。

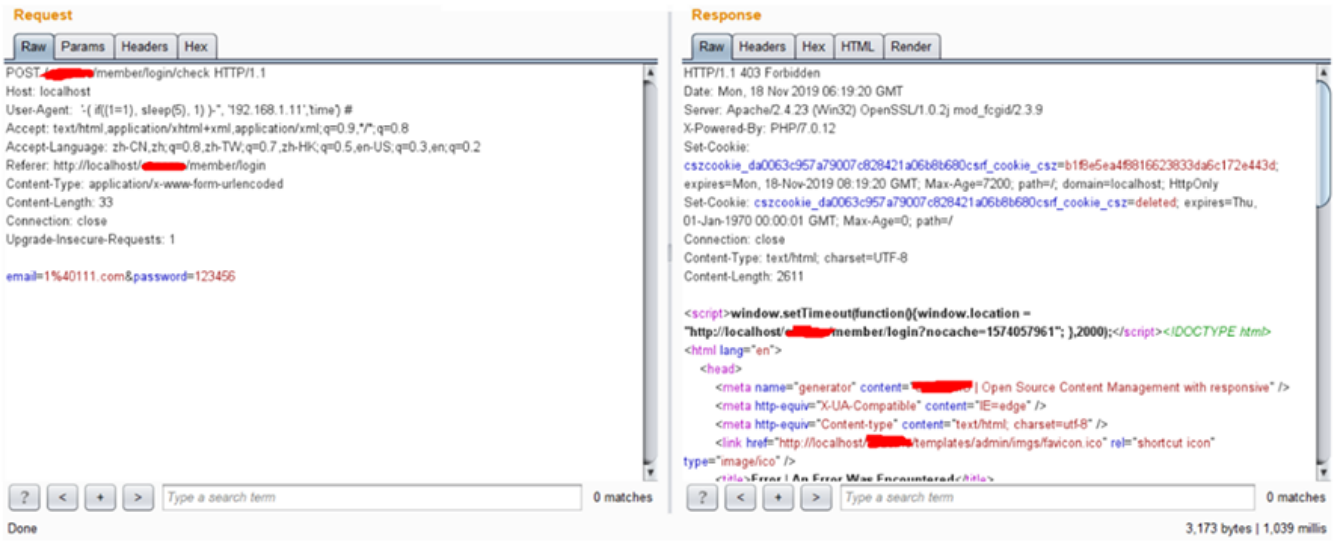
这个不就是我们要找的代码吗？！

escape_string 方法并不会转义 % ，所以我们将 Payload 进行 URL 编码后并不会被 escape_string 方法改变任何内容。而当到达 xss_clean 时 Payload 将被 URL 解码，从而绕过了对 SQL 注入的过滤。

漏洞验证

我们先来测试下原来的 Payload：

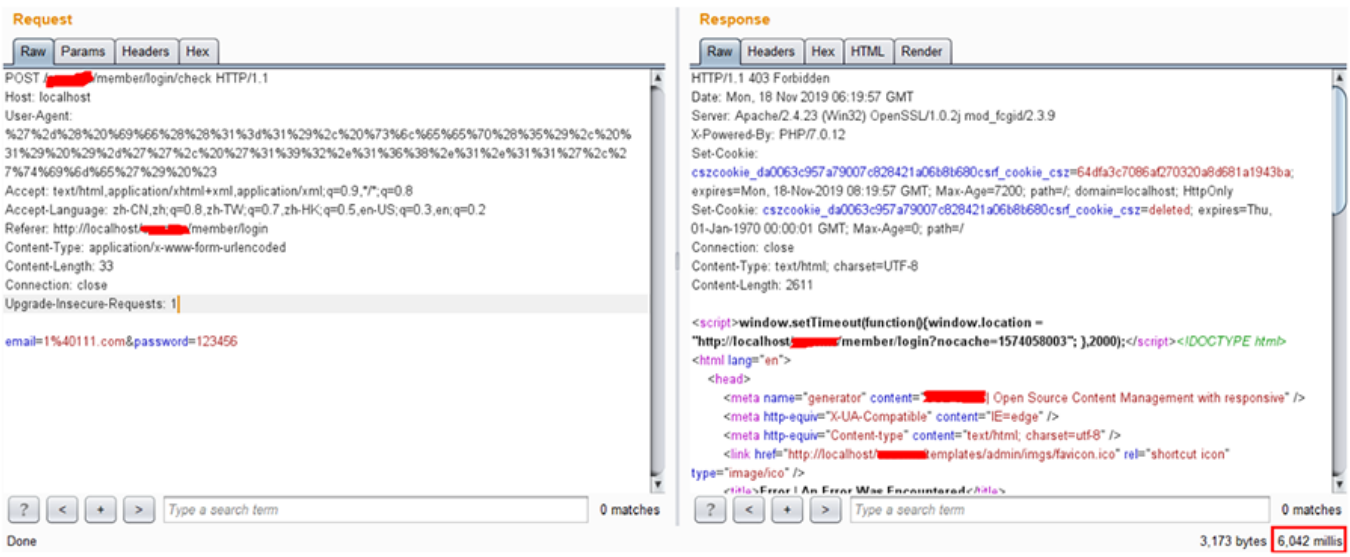
```
**'-( if((1=1), sleep(5), 1) )-'', '192.168.1.11','time') \#**
```



没有造成延时，原 Payload 失败。

新的 Payload 就是将原 Payload 进行 URL 编码，新 Payload：

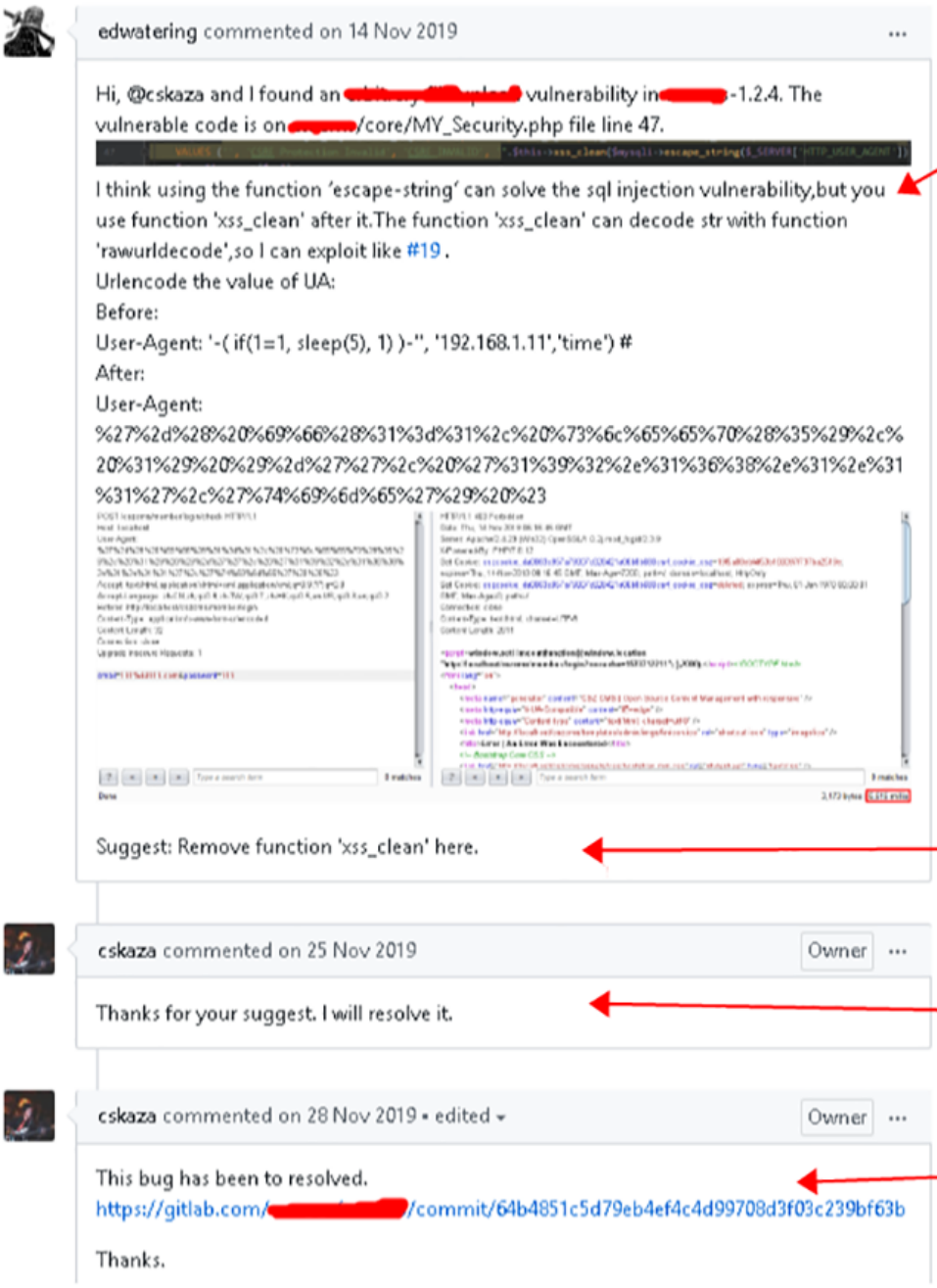
```
**%27%2d%28%20%69%66%28%28%31%3d%31%29%2c%20%73%6c%65%65%70%28%35%29%2c%
```



造成延时成功，绕过修复方案，触发 SQL 时间盲注漏洞。

0x04 后记

此漏洞在GITHUB中上报给了作者团队。



知其黑 守其白

分享知识盛宴，闲聊大院趣事，备好酒肉等你

长按二维码关注 酒仙桥六号部队

文章已于2020-07-02修改