[护网杯 2018]easy_tornado WriteUp（超级详细！）

原创

lunan0320 于 2021-04-22 21:08:43 发布 184 收藏 3

分类专栏： Web CTF 文章标签： web 安全漏洞

本文链接：https://blog.csdn.net/qq_51927659/article/details/116031923

Web 同时被 2 个专栏收录

14 篇文章 0 订阅

订阅专栏

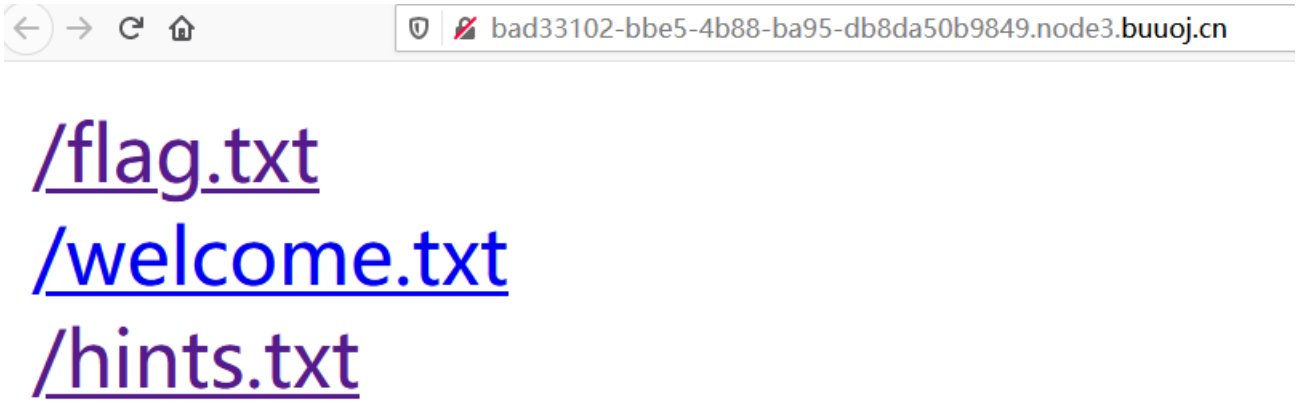
CTF

14 篇文章 0 订阅

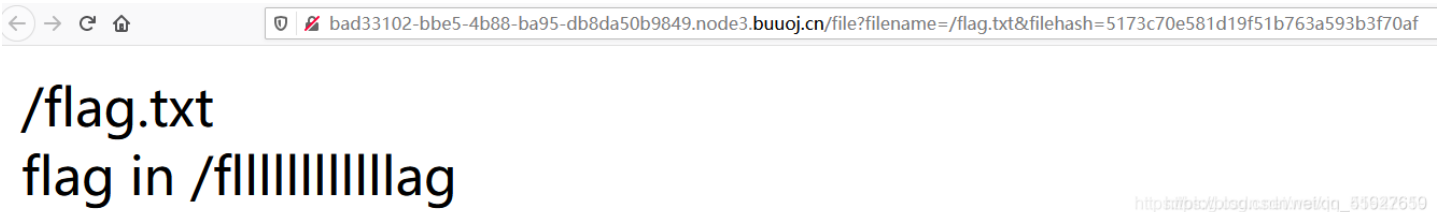
订阅专栏

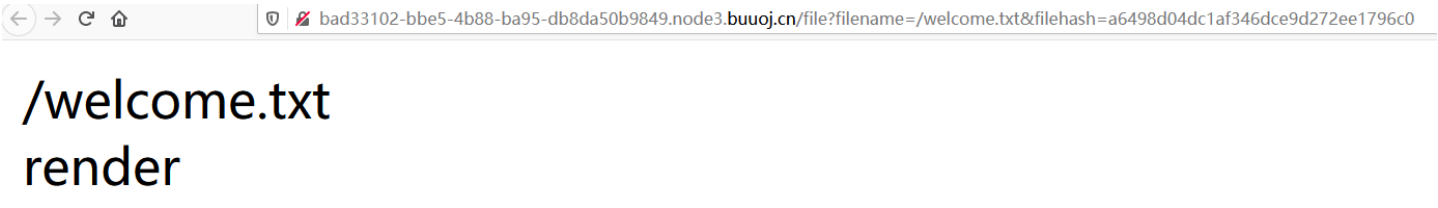
[护网杯 2018]easy_tornado

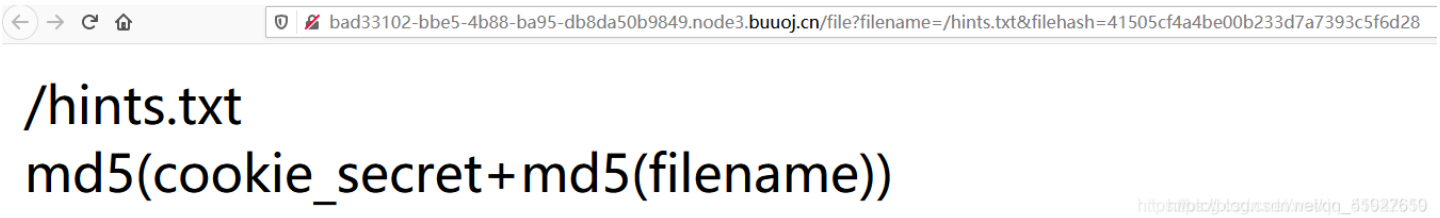
打开题目后，首先发现3个超链接



依次点开三个链接







网址里有参数`filename`和`filehash`推测这里flag应该是

`filename=/fllllllllllllag&filehash=md5(cookie_secret+md5(filename))`里面，filehash里hash就是提示为md5的hash加密。

变量 filename 的值总是为要访问的文件，再根据提示三和 filehash 三个不同的值猜测 filehash 的值为MD5加密后的字符串。

filename知道了，`cookie_secret`在哪呢？hints提示`render`

又根据题目`easy_tornado`可推测是**服务器模板注入。**

> **扩展：SSTI注入**
>
> SSTI就是**服务器端模板注入**(Server-Side Template Injection)，也给出了一个注入的概念。
> **服务端模板**：相当于很多公式，根据变量输出结果。这里的模板就是模板引擎根据数据自动生成前端页面。
> 常见的注入有：SQL 注入，XSS 注入，XPATH 注入，XML
> 注入，代码注入，命令注入等等。sql注入已经出世很多年了，对于sql注入的概念和原理很多人应该是相当清楚了，SSTI也是注入类的漏洞，其成因其实是可以类比于sql注入的。
> sql注入是从**用户获得一个输入，然后又后端脚本语言进行数据库查询，所以可以利用输入来拼接我们想要的sql语句**，当然现在的sql注入防范做得已经很好了，然而随之而来的是更多的漏洞。
> SSTI也是**获取了一个输入，然后在后端的渲染处理上进行了语句的拼接，然后执行**。错误的执行了用户输入。类比于 sql
> 注入。当然还是和sql注入有所不同的，SSTI利用的是现在的网站模板引擎(下面会提到)，主要针对python、php、java的一些网站处理框架，比如Python的jinja2
> mako tornado django，php的smarty twig，java的jade
> velocity。当这些框架对运用渲染函数生成html的时候会出现SSTI的问题。

因为**render（）是tornado里的函数，可以生成html模板**。是一个**渲染函数**，就是一个公式，能输出前端页面的公式。

tornado是用Python编写的Web服务器兼Web应用框架，简单来说就是用来生成模板的东西。和Python相关，和模板相关，就可以推测这可能是个ssti注入题了。

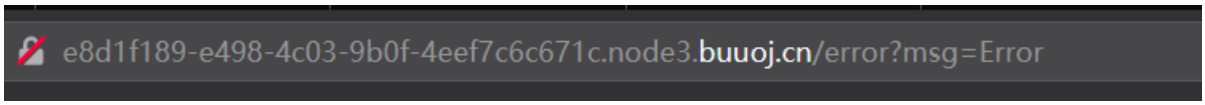
那我们开始初步尝试：

```
/file?filename=/flllllllllllag&filehash={{1}}
```

为什么使用双花括号？

> Tornado templates support control statements and expressions. Control statements are surrounded by {% %}, e.g. {% if len(items) > 2 %}.
> Expressions are surrounded by {{ }}, e.g. {{ items[0] }}.

得到关键报错信息：



模板注入必须通过传输型如{{xxx}}的执行命令。探测方式很简单，给一个参数赋值`{{22*22}}`返回484则必然存在模板注入。

但是当我们输入`error?msg={{1}}`就可以得到回显，说明此处是存在SSTI注入漏洞的。

当我构造的payload为：error?msg={{2*2}}的时候，回显的结果是orz。因此我们可以猜测出，此处是出现了过滤。

此时我们需要找的是`cookie_secret`，

搜素百度得Tornado框架的附属文件handler.settings中存在cookie_secret

Handler这个对象，Handler指向的处理当前这个页面的RequestHandler对象

> RequestHandler中并没有settings这个属性，与RequestHandler关联的Application对象（Requestion.application）才有setting这个属性
> handler 指向RequestHandler
> 而RequestHandler.settings又指向self.application.settings
> 所有handler.settings就指向RequestHandler.application.settings了！

此时构造payload:

```
http://e8d1f189-e498-4c03-9b0f-4eef7c6c671c.node3.buuoj.cn/error?msg={{handler.settings}}
```

得到提示信息，得到cookie_secret: 7837cb65-a58d-4897-9e1e-efdebe9b75b5

{'autoreload': True, 'compiled_template_cache': False, 'cookie_secret': '7837cb65-a58d-4897-9e1e-efdebe9b75b5'}

此时的payload应该就是如下通过md5的结果：

```
file?filename=/fllllllllllllag&filehash=md5(cookie_secret+md5(/fllllllllllllag))
```

/fllllllllllllag通过md5加密后：3bf9f6cf685a6dd8defadabfb41a03a1

md5(cookie_secret+md5(/fllllllllllllag))通过md5加密后的结果:77a1d0572298d9f79da44fd36511802c

此时构造payload:

```
/file?filename=/fllllllllllllag&filehash=77a1d0572298d9f79da44fd36511802c
```

回显得到最终的flag{a97dd1f2-6848-4eb9-86b1-c4f2306216c4}

/fllllllllllllag
flag{a97dd1f2-6848-4eb9-86b1-c4f2306216c4}