

漏洞信息

- 影响版本: <4.x
- 请求路径:

`http://127.0.0.1/cas/login;jsessionid=994275292E27F1C8AD104FC8B4CF221D[POST]`

源码分析

- 分析过程:

参考文章: 复现漏洞之前需要首先了解一下 spring-webflow 流程控制, 了解流程控制的基本原理, 以及中间参数的注入情况。

[Spring-webflow基础讲解](#)

[CAS单点登录开源框架解读（三）--CAS单点登录服务端认证之loginFlowRegistry流程](#)

源码中 `cas-server-webapp` 项目应该是主项目, 登录逻辑也在这个项目当中。项目整体是采用 `springmvc` 开发的, 首先分析 `mvc` 的主配置文件 `cas-servlet.xml` 文件。

```
<!-- 配置流程注册表, 其功能为: 负责加载流程定义-->
<!-- 所有 flow的定义文件它的位置在这里进行配置, flow-builder-services 用于配置 flow 的特性 -->
<webflow:flow-registry id="loginFlowRegistry" flow-builder-services="builder"
base-path="/WEB-INF/webflow">
    <webflow:flow-location-pattern value="/login/*-webflow.xml"/>
    <!-- 在这个声明中, 流程注册表会在该path下查找流程定义-->

    <!--flow-builder-services web Flow 中的视图通过 MVC 框架的视图技术来呈现 -->
</webflow:flow-registry>
<bean id="loginHandlerAdapter"
class="org.jasig.cas.web.flow.SelectiveFlowHandlerAdapter"
    p:supportedFlowId="login"
    p:flowExecutor-ref="loginFlowExecutor"
    p:flowUrlHandler-ref="loginFlowUrlHandler" /><!--loginFlowExecutor流程执行器-->

<bean id="loginFlowUrlHandler"
class="org.jasig.cas.web.flow.CasDefaultFlowUrlHandler" />

<bean name="loginFlowExecutor"
class="org.springframework.webflow.executor.FlowExecutorImpl"
    c:definitionLocator-ref="loginFlowRegistry"
    c:executionFactory-ref="loginFlowExecutionFactory"
    c:executionRepository-ref="loginFlowExecutionRepository" />

<bean name="loginFlowExecutionFactory"
class="org.springframework.webflow.engine.impl.FlowExecutionImplFactory"
    p:executionKeyFactory-ref="loginFlowExecutionRepository"/>

<bean id="loginFlowExecutionRepository"
class="org.jasig.spring.webflow.plugin.ClientFlowExecutionRepository"
    c:flowExecutionFactory-ref="loginFlowExecutionFactory"
    c:flowDefinitionLocator-ref="loginFlowRegistry"
    c:transcoder-ref="loginFlowStateTranscoder" />
```

```

<bean id="loginFlowStateTranscoder"
class="org.jasig.spring.webflow.plugin.EncryptedTranscoder"
c:cipherBean-ref="loginFlowCipherBean" />

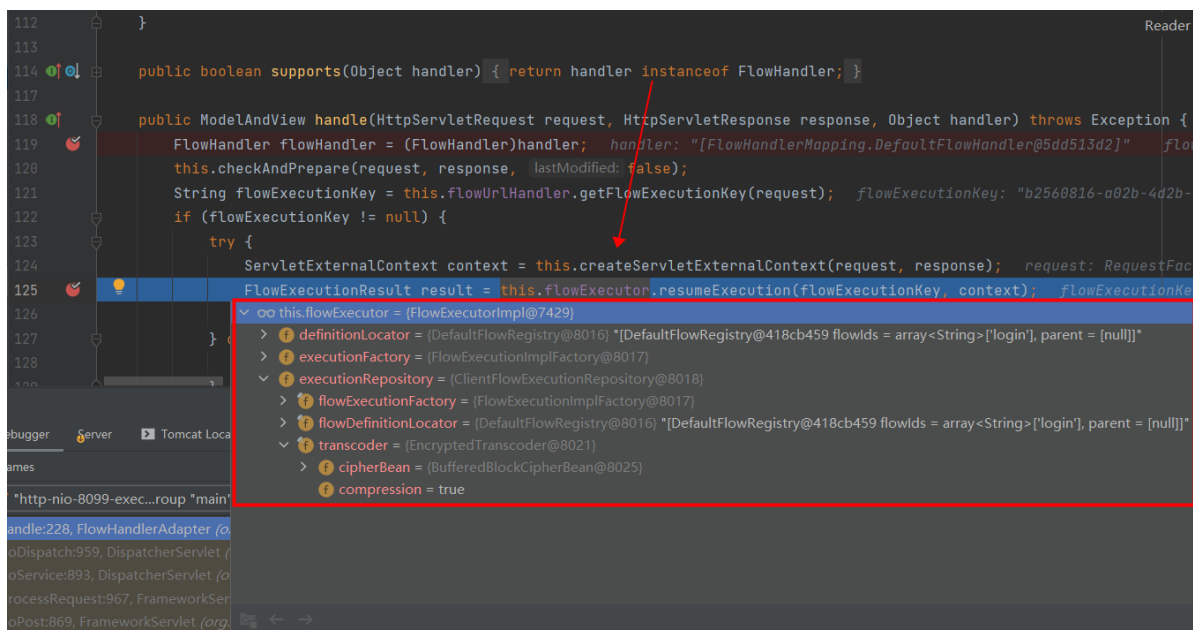
<bean id="loginFlowCipherBean"
class="org.cryptacular.bean.BufferedBlockCipherBean"
p:keyAlias="${cas.webflow.keyalias:aes128}"
p:keyStore-ref="loginFlowCipherKeystore"
p:keyPassword="${cas.webflow.keypassword:changeit}">
<property name="nonce">
<bean class="org.cryptacular.generator.sp80038a.RBGNonce" />
</property>
<property name="blockCiphersSpec">
<bean class="org.cryptacular.spec.BufferedBlockCiphersSpec"
c:algName="${cas.webflow.cipher.alg:AES}"
c:cipherMode="${cas.webflow.cipher.mode:CBC}"
c:cipherPadding="${cas.webflow.cipher.padding:PKCS7}" />
</property>
</bean>

```

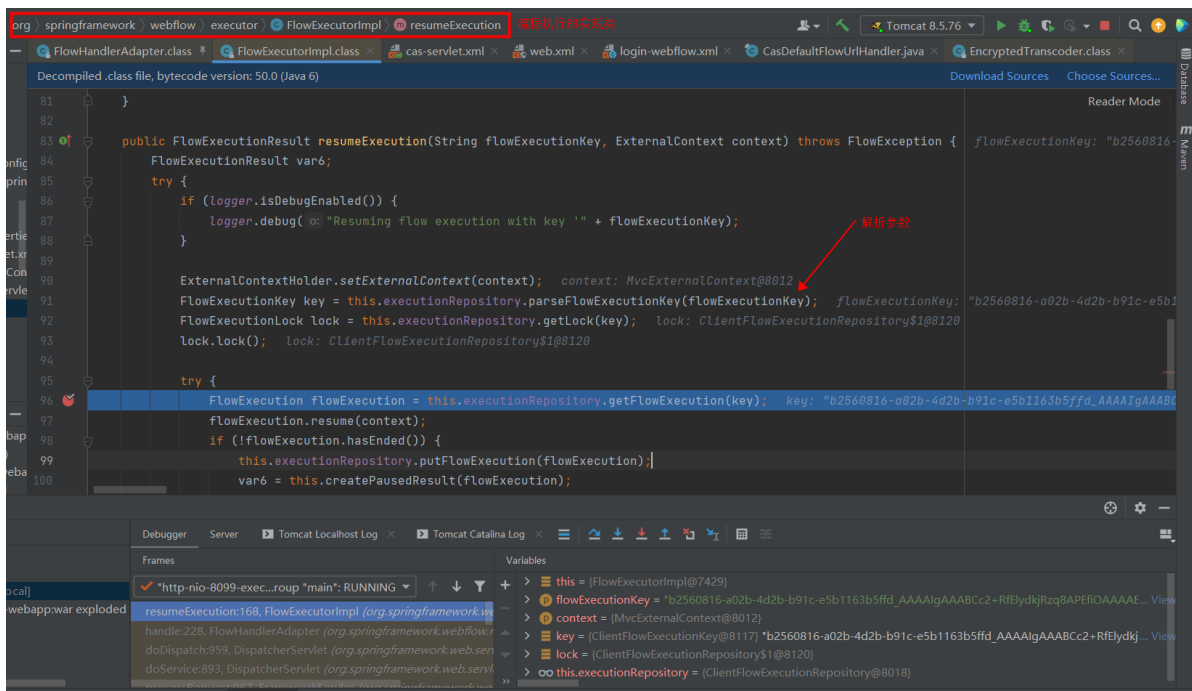
原先的配置文件内容较多，理解不方便，把相关的配置都抽取出来就很明了了。重点是看 `loginHandlerAdapter` 这个适配器的配置，这个也是处理流程的入口，适配器中注册的流程执行器是 `loginFlowExecutor`，然后流程执行器就是就是处理的主要逻辑。`loginHandlerAdapter` -> `loginFlowExecutor` -> `loginFlowExecutionRepository` -> `loginFlowExecutionRepository` -> `loginFlowStateTranscoder` -> `loginFlowCipherBean`。其中 `loginFlowStateTranscoder` 是处理加解密的逻辑，也是漏洞存在的点，注册的加解密配置是 `loginFlowCipherBean`。

这里的加解密主要是处理 `execution` 参数，这个参数的作用在参考文章中有写到：此参数用于指定一个唯一的流程实例，在页面提交时此处的值可以直接通过 `${flowExecutionKey}` 获得。

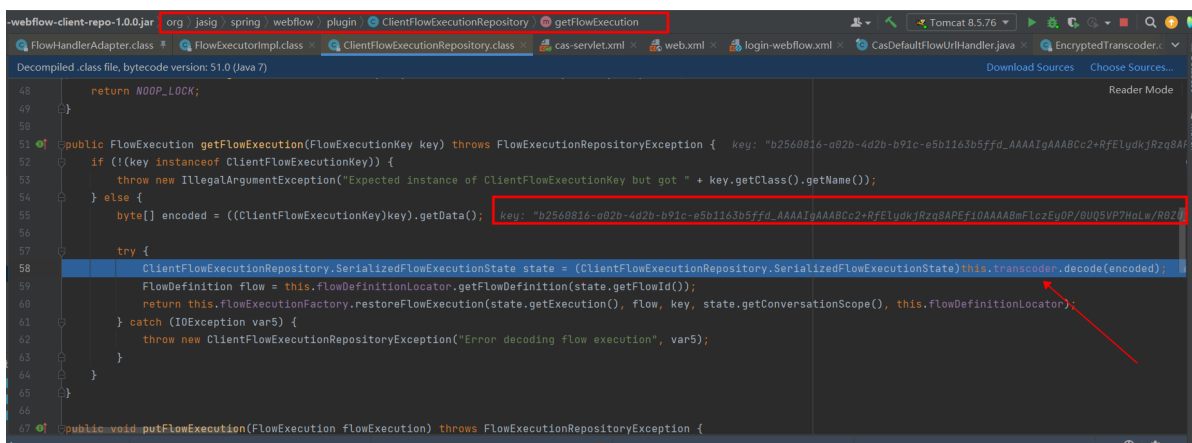
首先是 `loginHandlerAdapter` 这个入口类，其中处理逻辑在 `handle` 方法当中。通过 `this.flowUrlHandler.getFlowExecutionKey(request)` 获取到 `execution` 参数的值，然后通过流程执行器 `this.flowExecutor` 对参数进行处理。这个流程执行器就是我们通过 `ioc` 机制自动注入的 `loginFlowExecutor` 这个执行器，属性信息如图，和上面分析的保持一致。



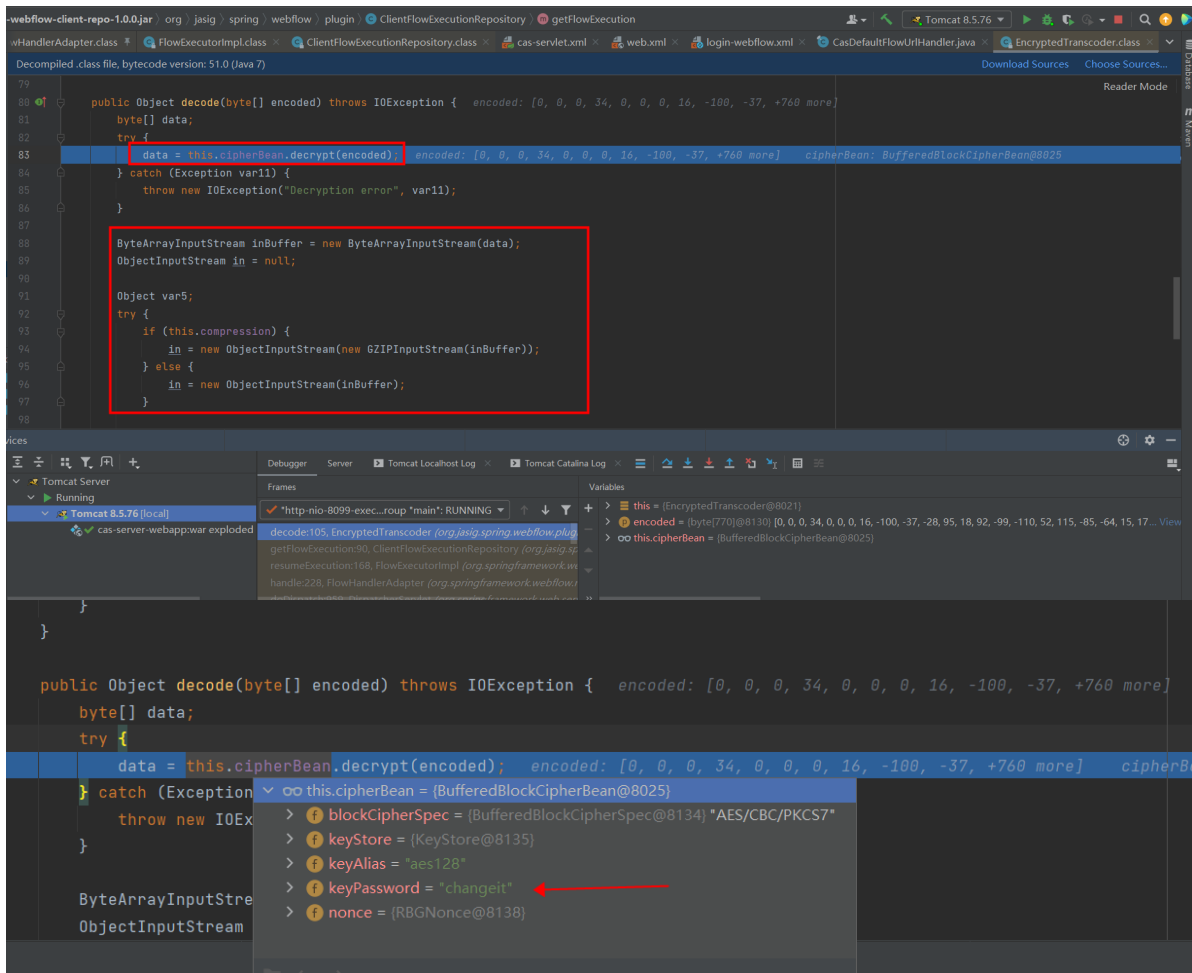
之后便是进入流程执行器的 `resumeExecution` 方法，首先解析 `execution` 参数，然后将参数传递给 `this.executionRepository.getFlowExecution(key)`，这个 `executionRepository` 是通过 `ioc` 机制自动注入的 `LoginFlowExecutionRepository`-->`org.jasig.spring.webflow.plugin.ClientFlowExecutionRepository`。



进入 `org.jasig.spring.webflow.plugin.ClientFlowExecutionRepository#getFlowExecution()` 方法之后通过 `this.transcoder.decode()` 方法对参数进行解密。然后就进入到 `LoginFlowStateTranscoder`-->`org.jasig.spring.webflow.plugin.EncryptedTranscoder` 类当中，触发反序列化漏洞。以上过程和我们分析的保持一致。





















首先是对参数进行解密，然后序列化的流是经过 `gzip` 压缩的，所以要先解压缩，然后反序列化。



大于等于4.1.7版本之后的修复方案，便是修改了默认的加密方法，采用自设的密钥。



漏洞的利用，默认的依赖包中发现有 `commons-collection4-4.0` 和 `hibernate`，所以应该可以直接利用这两个构造链。参考文章：[Apereo CAS 反序列化漏洞分析及回显利用](#)，[Apereo-CAS-4-X 反序列化漏洞分析及复现](#)

	<code>cas-server-support-generic-4.1.6.jar</code>	De
	<code>cas-server-webapp-support-4.1.6.jar</code>	7
	<code>cdi-api-1.0-SP4.jar</code>	7
	<code>classmate-1.0.0.jar</code>	7
	<code>commons-codec-1.10.jar</code>	7
	<code>commons-collections4-4.0.jar</code>	7
	<code>commons-io-2.4.jar</code>	7
	<code>commons-jexl-1.1.jar</code>	7
	<code>commons-lang3-3.4.jar</code>	7
	<code>commons-logging-1.2.jar</code>	7
	<code>cryptacular-1.0.jar</code>	7
	<code>dom4j-1.6.1.jar</code>	7
	<code>FastInfoset-1.2.12.jar</code>	8
	<code>guava-18.0.jar</code>	8
	<code>hibernate-commons-annotations-4.0.5.Final.jar</code>	8
	<code>hibernate-core-4.3.10.Final.jar</code>	8
	<code>hibernate-jpa-2.1-api-1.0.0.Final.jar</code>	8
	<code>hibernate-validator-5.1.3.Final.jar</code>	8