Struts2 漏洞利用原理及 OGNL 机制研究

Auth : Cryin'
Date : 2015.04.19

概述

在 MVC 开发框架中，数据会在 MVC 各个模块中进行流转。而这种流转，也就会面临一些困境，就是由于数据在不同 MVC 层次中表现出不同的形式和状态而造成的：

- **View 层—表现为字符串展示**

  数据在页面上是一个扁平的、不带数据类型的字符串，无论数据结构有多复杂，数据类型有多丰富，到了展示的时候，全都一视同仁的成为字符串在页面上展现出来。数据在传递时，任何数据都都被当作字符串或字符串数组来进行。

- **Controller 层—表现为 java 对象**

  在控制层，数据模型遵循 java 的语法和数据结构，所有的数据载体在 Java 世界中可以表现为丰富的数据结构和数据类型，你可以自行定义你喜欢的类，在类与类之间进行继承、嵌套。我们通常会把这种模型称之为复杂的对象树。数据在传递时，将以对象的形式进行。

可以看到，数据在不同的 MVC 层次上，扮演的角色和表现形式不同，这是由于 HTTP 协议与 java 的面向对象性之间的不匹配造成的。如果数据在页面和 Java 世界中互相传递，就会显得不匹配。所以也就引出了几个需要解决的问题：

1. 当数据从 View 层传递到 Controller 层时，我们应该保证一个扁平而分散在各处的数据集合能以一定的规则设置到 Java 世界中的对象树中去。同时，能够灵活的进行由字符串类型到 Java 中各个类型的转化。

2. 当数据从 Controller 层传递到 View 层时，我们应该保证在 View 层能够以某些简易的规则对对象树进行访问。同时，在一定程度上控制对象树中的数据的显示格式。

我们稍微深入思考这个问题就会发现，解决数据由于表现形式的不同而发生流转不匹配的问题对我们来说其实并不陌生。同样的问题会发生在 Java 世界与数据库世界中，面对这种对象与关系模型的不匹配，我们采用的解决方法是使用如 hibernate，iBatis 等框架来处理 java 对象与关系数据库的匹配。

现在在 Web 层同样也发生了不匹配，所以我们也需要使用一些工具来帮助我们解决问题。为了解决数据从 View 层传递到 Controller 层时的不匹配性，Struts2 采纳了 XWork 的一套完美方案。并且在此的基础上，构建了一个完美的机制，从而比较完美的解决了数据流转中的不匹配性。就是表达式引擎 OGNL。它的作用就是帮助我们完成这种规则化的表达式与 java 对象的互相转化(以上内容参考自 Struts2 技术内幕)。

关于 OGNL

OGNL（Object Graph Navigation Language）即对象图形导航语言，是一个开源的表达式引擎。使用 OGNL，你可以通过某种表达式语法，存取 Java 对象树中的任意属性、调用 Java 对象树的方法、同时能够自动实现必要的类型转化。如果我们把表达式看做是一个带有语义的字符串，那么 OGNL 无疑成为了这个语义字符串与 Java 对象之间沟通的桥梁。我们可以轻松解决在数据流转过程中所遇到的各种问题。

OGNL 进行对象存取操作的 API 在 Ognl.java 文件中，分别是 getValue、setValue 两个方法。getValue 通过传入的 OGNL 表达式，在给定的上下文环境中，从 root 对象里取值：

```
public static Object getValue(String expression, Map context, Object root) throws OgnlException {
    return getValue((String)expression, (Map)context, root, (Class)null);
}
```

setValue 通过传入的 OGNL 表达式，在给定的上下文环境中，往 root 对象里写值:

```
public static void setValue(String expression, Map context, Object root, Object value) throws OgnlException {
    setValue((Object)parseExpression(expression), (Map)context, root, value);
}
```

OGNL 同时编写了许多其它的方法来实现相同的功能，详细可参考 Ognl.java 代码。OGNL 的 API 很简单，无论何种复杂的功能，OGNL 会将其最终映射到 OGNL 的三要素中通过调用底层引擎完成计算，OGNL 的三要素即上述方法的三个参数 expression、root、context。

OGNL 三要素

- **Expression(表达式)：**

Expression 规定 OGNL 要做什么，其本质是一个带有语法含义的字符串,这个字符串将规定操作的类型和操作的内容。OGNL 支持的语法非常强大，从对象属性、方法的访问到简单计算，甚至支持复杂的 lambda 表达式。

- **Root(根对象)：**

OGNL 的 root 对象可以理解为 OGNL 要操作的对象，表达式规定 OGNL 要干什么，root 则指定对谁进行操作。OGNL 的 root 对象实际上是一个 java 对象，是所有 OGNL 操作的实际载体。

- **Context(上下文)：**

有了表达式和根对象，已经可以使用 OGNL 的基本功能了。例如，根据表达式对 root 对象进行 getvalue、setvalue 操作。

不过事实上在 OGNL 内部，所有的操作都会在一个特定的数据环境中运行，这个数据环境就是 OGNL 的上下文。简单说就是上下文将规定 OGNL 的操作在哪里进行。

OGNL 的上下文环境是一个 MAP 结构，定义为 OgnlContext，root 对象也会被添加到上下文环境中，作为一个特殊的变量进行处理。

OGNL 基本操作

- **对 root 对象的访问：**

针对 OGNL 的 root 对象的对象树的访问是通过使用‘点号’将对象的引用串联起来实现的。通过这种方式，OGNL 实际上将一个树形的对象结构转化为一个链式结构的字符串来表达语义，如下：

```
//获取 root 对象中的 name 属性值
name
//获取 root 对象 department 属性中的 name 属性值
department.name
//获取 root 对象 department 属性中的 manager 属性中的 name 属性值
department.manager.name
```

- **对上下文环境的访问：**

Context 上下文是一个 map 结构，访问上下文参数时需要通过#符号加上链式表达式来进行，从而表示与访问 root 对象的区别，如下：

```
//获取上下文环境中名为 introduction 的对象的值
#introduction
```

//获取上下文环境中名为 parameters 的对象中的 user 对象中名为 name 属性的值

#parameters.user.name

- **对静态变量、方法的访问：**

在 OGNL 中，对于静态变量、方法的访问通过@[class]@[field/method]访问，这里的类名要带着包名。如下

//访问 com.example.core.Resource 类中名为 img 的属性值

@com.example.core.Resource@img

//调用 com.example.core.Resource 类中名为 get 的方法

@com.example.core.Resource@get()

- **访问调用：**

//调用 root 对象中的 group 属性中 users 的 size()方法

group.users.size()

//调用 root 对象中 group 中的 containsUser 方法，并传递上下文环境中名为 user 值作为参数

group.containsUser(#user)

- 构造对象

OGNL 支持直接通过表达式来构造对象，构造的方式主要包括三种：

1. 构造 List：使用{}，中间使用逗号隔开元素的方式表达列表
2. 构造 map：使用#{},中间使用逗号隔开键值对，并使用冒号隔开 key 和 value
3. 构造对象：直接使用已知的对象构造函数来构造对象

可以看到 OGNL 在语法层面上所表现出来的强大之处，不仅能够直接对容器对象构造提供语法层面支持，还能够对任意 java 对象提供支持。然而正因为 OGNL 过于强大，它也造成了诸多安全问题。恶意攻击者通过构造特定数据带入 OGNL 表达式解析并执行，而 OGNL 可以用来获取和设置 Java 对象的属性，同时也可以对服务端对象进行修改，所以只要绕过沙盒恶意攻击者甚至可以执行系统命令进行系统攻击。

深入 OGNL 实现类

在 OGNL 三要素中，context 上下文环境为 OGNL 提供计算运行的环境，这个运行环境在 OGNL 内部由 OgnlContext 类实现，它是一个 map 结构。在 OGNL.java 中可以看到 OgnlContext 会在 OGNL 计算时动态创建，同时传入 map 结构的 context，并根据传入的参数设定 OGNL 计算的一些默认行为以及设置 root 对象。

```
public static Map addDefaultContext(Object root, ClassResolver classResolver, TypeConverter converter, MemberAccess memberAccess, Map context) {
    OgnlContext result;
    if(!(context instanceof OgnlContext)) {
        result = new OgnlContext();
        result.setValues(context);
    } else {
        result = (OgnlContext)context;
    }

    if(classResolver != null) {
        result.setClassResolver(classResolver);
    }

    if(converter != null) {
        result.setTypeConverter(converter);
    }

    if(memberAccess != null) {
        result.setMemberAccess(memberAccess);
    }

    result.setRoot(root);
    return result;
}
```

这里看到 MemberAccess 属性，在 struts2 漏洞利用中经常看到。转到 OgnlContext 类中查看。

```
public static final MemberAccess DEFAULT_MEMBER_ACCESS = new DefaultMemberAccess(false);
```

MemberAccess 指定 OGNL 的对象属性、方法访问策略，这里初始默认情况下是 flase，即默认不允许访问。在 struts2 中 SecurityMemberAccess 类封装了 DefaultMeberAccess 类并进行了扩展，指定是否支持访问静态方法以及通过指定正则表达式来规定某些属性是否能够被访问。其中 allowStaticMethodAccess 默认为 flase，即默认禁止访问静态方法。

```
public SecurityMemberAccess(boolean method) {
    super(false);
    allowStaticMethodAccess = method;
}
```

另外在 OGNL 实现了 MethodAccessor 及 PropertyAccessor 类规定 OGNL 在访问方法和属性时的实现方式，详细可参考代码文件。

```
public interface MethodAccessor {
    Object callStaticMethod(Map var1, Class var2, String var3, Object[] var4) throws MethodFailedException;

    Object callMethod(Map var1, Object var2, String var3, Object[] var4) throws MethodFailedException;
}
```

在 struts2 中，XWorkMethodAccessor 类对 MethodAccessor 进行了封装，其中在 callStaticMethod 方法中可以看到，程序首先从上下文获取到 DENY_METHOD_EXECUTION 值并转换为布尔型，如果为 false 则可以执行静态方法，如果为 true 则不执行静态发放并返回 null。

```
@Override
public Object callStaticMethod(Map context, Class aClass, String string, Object[] objects) throws MethodF
    Boolean exec = (Boolean) context.get(ReflectionContextState.DENY_METHOD_EXECUTION);
    boolean e = ((exec == null) ? false : exec.booleanValue());

    if (!e) {
        return callStaticMethodWithDebugInfo(context, aClass, string, objects);
    } else {
        return null;
    }
}
```

Struts2 漏洞利用原理

上文已经详细介绍了 OGNL 引擎，因为 OGNL 过于强大，它也造成了诸多安全问题。恶意攻击者通过构造特定数据带入 OGNL 表达式即可能被解析并执行，而 OGNL 可以用来获取和设置 Java 对象的属性，同时也可以对服务端对象进行修改，所以只要绕过 Struts2 的一些安全策略，恶意攻击者甚至可以执行系统命令进行系统攻击。如 struts2 远程代码执行漏洞 s2-005。

XWork 通过 getters/setters 方法从 HTTP 的参数中获取对应 action 的名称，这个过程是基于 OGNL 的。OGNL 是怎么处理的呢？如下：

user.address.city=Bishkek&user['favoriteDrink']=kumys

会被转化成：

action.getUser().getAddress().setCity("Bishkek")

action.getUser().setFavoriteDrink("kumys")

这个过程是由 ParametersInterceptor 调用 ValueStack.setValue()完成的，它的参数是用户可控的，由 HTTP 参数传入。虽然 Struts2 出于安全考虑，在 SecurityMemberAccess 类中通过设置禁止静态方法访问及默认禁止执行静态方法来阻止代码执行。即上面提及的 denyMethodExecution 为 true，MemberAccess 为 false。但这两个属性都可以被修改从而绕过安全限制执行任意命令。POC 如下：

http://mydomain/MyStruts.action?('\u0023_memberAccess[\'allowStaticMethodAccess\']')(meh)=true&(aaa)(('\u0023context[\'xwork.MethodAccessor.denyMethodExecution\']\u003d\u0023foo')(\u0023foo\u003dnew%20java.lang.Boolean("false")))&(asdf)(('\u0023rt.exec('ipconfig')')(\u0023rt\u003d@java.lang.Runtime@getRuntime()))=1

在 POC 中\u0023 是因为 S2-003 对#号进行了过滤，但没有考虑到 unicode 编码情况，而通过#_memberAccess[\'allowStaticMethodAccess\']')(meh)=true，可以设置允许访问静态方法，因为 getRuntime 方法是静态的，通过 context[\'xwork.MethodAccessor.denyMethodExecution\']=#foo')(\#foo=new%20java.lang.Boolean("false")设置拒绝执行方法为 false，也就是允许执行静态方法。最后调用 rt.exec('ipconfig') 执行任意命令。

参考

[1] Struts2 技术内幕-深入解析 Struts2 架构设计与实现原理

[2] http://blog.csdn.net/u011721501/article/details/41610157

[3] http://www.freebuf.com/articles/web/33232.html

[4] http://blog.csdn.net/three_feng/article/details/60869254

[5] https://github.com/apache/struts/

[6] http://www.secbox.cn/hacker/program/205.html

[7] http://struts.apache.org/docs/s2-005.html

[8] http://archive.apache.org/dist/struts/