**转载请注明出处谢谢：**[**http://blog.csdn.net/u011721501?viewmode=list**](http://blog.csdn.net/u011721501?viewmode=list)

**0、前言**

最近专业实训上需要添加一个struts2的批量利用程序，倒腾半天，我发现我对struts2系列的漏洞根本就不懂，用是会用，但是对我来说还是不够，网上也没有什么好的分析文章来老老实实分析每一个细节。所以我这里就结合每天的研究写一些博客share出来。这是第一篇，也是昨天研究的一部分。

**1、概述**

在Struts2中，使用Ognl表达式作为字符串与对象之间转换的一种实现，通过Ognl表达式可以轻松做到String与Object之间的沟通。

在Struts2工程中，ognl对应的jar包是ognl-2.7.3，附上源码来看一下Ognl类的实现。

在Ognl.java中，最基本的两个方法就是getValue与setValue，传入三个参数（表达式，上下文以及根对象）。其他的实现都是实现相同的功能。

Ognl的API很简单，无论什么复杂的功能，都会通过OGNL的三个要素完成。这三个要素分别是：

（1）表达式

表达式（Expression）规定Ognl做些什么，其本质是一个带有语法含义的字符串。支持的语法非常强大，从对象属性、方法的访问到简单计算，甚至可以进行对象的创建与lambda。

（2）Root对象

这个又称为根对象，是用来存放执行Ognl的对象。比如要利用Ognl表达式对User对象的name属性，这时候User对象就是本次操作的根对象（Root）。

（3）上下文环境

Context，这个对有Java开发经验的人来说并不算新奇，但是对于一般人来说很难理解，Context翻译为上下文，也可以理解为环境变量，是标明整个运行环境中的参数信息，默认值等。OGNL的上下文环境其实就是个Map结构，里面保存了一些值。

**2、基本操作**

（1）访问根对象

访问根对象的属性和方法可以不写对象名前面的"#"，使用点号进行访问即可。

如访问User根对象的name属性：

OGNL： user.name

（2）对上下文的访问

由于上下文是一个Map结构，访问这些参数需要通过#符号加上链式表达式进行。

如访问parameter中的name：

OGNL：#parameter.name

（3）静态方法/属性调用

通过@[class]@[field/method]访问，这里的类名要带着包名。

如调用User中的静态方法get

[@com.example.ynu.User@get()](mailto:@com.example.ynu.User@get())

（4）方法的调用

使用点号加方法名称进行调用，而且可以传递参数。

如访问Root对象中的group属性中的Users的size方法

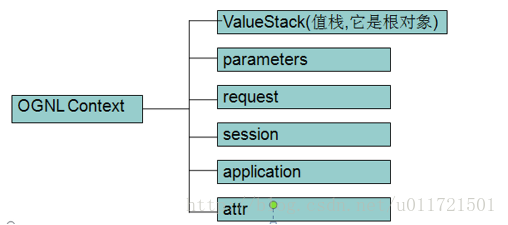
group.users.size()

此外，Ognl还支持简单计算，如9 mod 2 取模运算；支持对数组和容器的访问，可以按照数组下标进行访问，对于Map结构还可以直接使用键值来访问。

还有投影选择功能，用来筛选数据，这里不再赘述。

**3、深入研究**

（1）Context构成



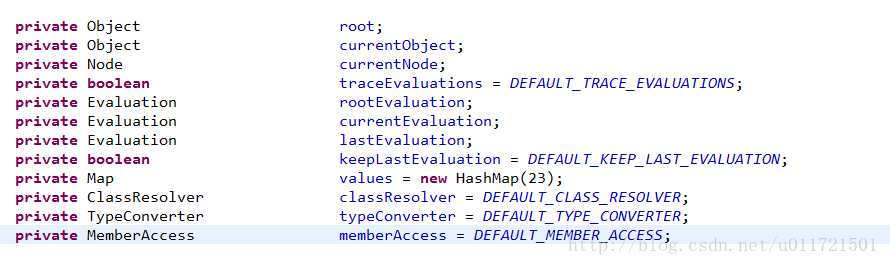
上面是OGNL上下文的结构，在OGNL中使用OgnlContext类进行实现。由上图可以看到OGNL Context的具体构成。其中的ValueStack叫做值栈，是根对象的实现，在OGNL中则是使用OgnlValueStack实现ValueStack接口来实现这个机制。

那么如何保存请求中的多个对象呢？Ognl中使用了CompoundRoot root;

来保存。CompoundRoot类继承了ArrayList类，里面有pop、push方法，很显然是被做成了一个栈结构，正好解释了为啥叫ValueStack（值栈）。在上下文中，除了ValueStack，其他都是Map结构，访问时直接#符号跟名称即可。

以前没有阅读源码，我一直认为根对象是独立与上下文环境而存在，知道看了上下文的实现（OgnlContext中的addDefaultContext方法），这个方法返回默认Map结构，作为Context。方法中还调用了setRoot方法将根对象也给附上去了，也就是说这个根对象（ValueStack）也是上下文的一部分。

（2）Context其他重要参数



root（\_root）：作为根对象的引用。

values（\_values）：如果自己传入一个Map作为上下文，那么也要用OgnlContext进行包装，这个values就被看做为真正的容器。

ClassResolver（\_classResolver）：指定类加载机制的处理类，默认使用反射机制动态加载类。

TypeConvert（\_typeConverter）：处理类型转化的类。处理字符串转为java类型。

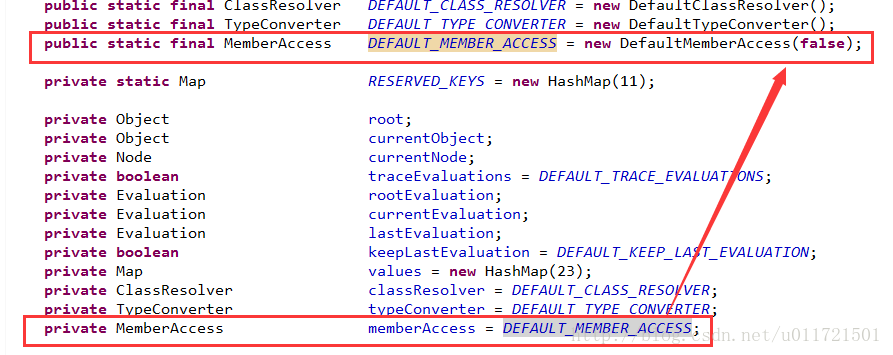
MemeberAccess（\_memberAccess）：指定了访问策略的处理方式。

（3）方法/属性访问策略MemberAccess

我们知道，在Struts2漏洞中的s2-005的攻击代码中有**('\u0023\_memberAccess[\'allowStaticMethodAccess\']')(meh)=true**这么一句。

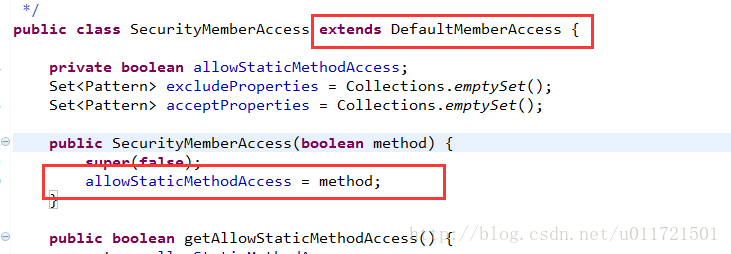
这里就是为了修改上下文中的\_memberAccess为true，为啥设置为true呢？  因为默认为false........

源码如下图：



MemberAccess在OGNL中的实现是DefaultMemberAccess，在Struts2中是SecurityMemberAccess。

这里说一下SecurityMemberAccess：



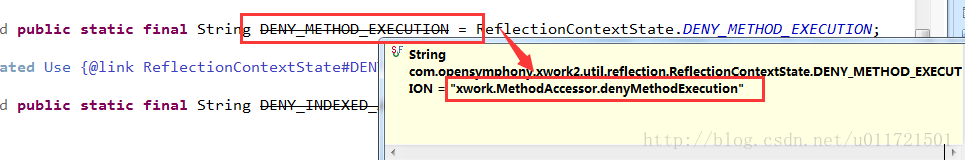
Struts2为了自身机制实现，又把这个memberAccess在OGNL上的实现封装了一遍，可以其中的allStaticMethodAccess默认是false，也就是说默认不支持静态方法的执行。

（4）方法/属性访问机制-------MethodAccessor和PropertyAccessor

这俩规定了OGNL访问方法和属性时的实现方式。在Struts2中，MethodAcccessor的实现类是XWorkMethodAccessor。

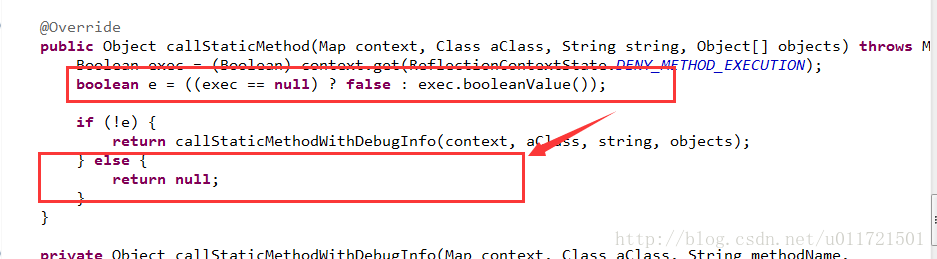
**(('\u0023context[\'xwork.MethodAccessor.denyMethodExecution\']\u003d\u0023foo')(\u0023foo\u003dnew%20java.lang.Boolean("false")))**

这句也是攻击代码的一部分，意思是将XWorkMethodAccessor的denyMethodExecution设置为false。



读源码，只有当ReflectionContextState.DENY\_INDEXED\_ACCESS\_EXECUTION为false，方法才可以执行。

具体控制在callStaticMethod中，这里从上下文中获取这个值**xwork.MethodAccessor.denyMethodExecution** ，转化为Boolean的类型，可以看到，如果没有设置，默认为False。如果e为false，就return null，方法不会成功执行。



**4、总结**

OGNL机制是建立在三个基本元素根对象、上下文、表达式的基础上，其他的实现都是提供一套解析引擎。由于OGNL及其灵活，所以造成了不少的漏洞，想必做安全的大家都会知道，接下来的几篇我就会从源码角度探索Struts2的各个漏洞。

**参考资料**

Struts2技术内幕

**转载请注明出处谢谢：**[**http://blog.csdn.net/u011721501?viewmode=list**](http://blog.csdn.net/u011721501?viewmode=list)