CNTA-2019-0014 wls9-async 反序列化 rce 分析

orich1 / 2019-04-25 08:54:00 / 浏览数 7355 安全技术 漏洞分析 顶(0) 踩(0)

前言

漏洞编号：CNTA-2019-0014
大致是因为 wls9_async_response 包有个啥反序列化，上一次同样类型的漏洞在17年，那时候还不知道weblogic，刚好论文结尾了来学习下漏洞原理

XmlDecoder 相关安全不在此篇文章中介绍，也莫得poc，仅仅分享分析思路和漏洞触发流程

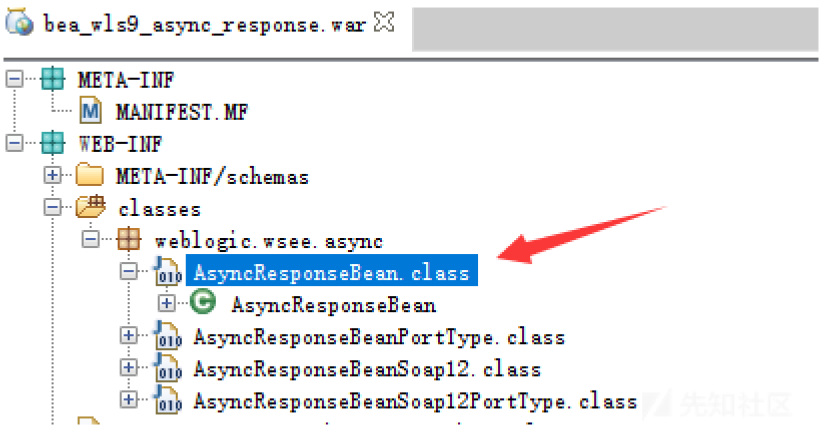
高版本weblogic如12.2.1.2默认不会部署该war包，我的测试版本是10.3.6

调用链

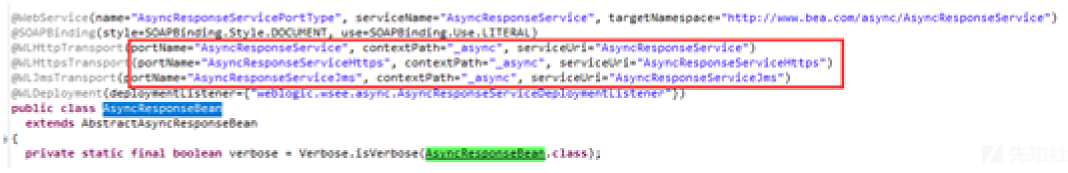
BaseWSServlet#service
->
SoapProcessor#process
->
ServerDispatcher#dispatch
->
HandlerIterator#handleRequest
->
WorkAreaServerHandler#handleRequest
->
WorkContextMapInterceptor#receiveRequest
->
WorkContextXmlInputAdapter#readUTF
->
XMLDecoder#readObject

起手式

莫得poc，莫得漏洞详情，就一则安全通告说 wls9_async_response 有问题，那就先直接看war包啥情况


bea_wls9_async_response.war
META-INF
MANIFEST.MF
WEB-INF
META-INF/schemas
classes
weblogic.wsee.async
AsyncResponseBean.class
AsyncResponseBean
AsyncResponseBeanPortType.class
AsyncResponseBeanSoap12.class
AsyncResponseBeanSoap12PortType.class

其实就四个class，而且路径全部指向 AsyncResponseBean ，查看一下内容如下：



```
@WebService(name="AsyncResponseServicePortType", serviceName="AsyncResponseService", targetNamespace="http://www.bea.com/async/AsyncResponseService")
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT, use=SOAPBinding.Use.LITERAL)
@WLHttpTransport(portName="AsyncResponseService", contextPath="_async", serviceUri="AsyncResponseService")
@WLHttpsTransport(portName="AsyncResponseServiceHttps", contextPath="_async", serviceUri="AsyncResponseServiceHttps")
@WLJmsTransport(portName="AsyncResponseServiceJms", contextPath="_async", serviceUri="AsyncResponseServiceJms")
@WLDeployment(deploymentListener={"weblogic.wsee.async.AsyncResponseServiceDeploymentListener"})
public class AsyncResponseBean
  extends AbstractAsyncResponseBean
{
  private static final boolean verbose = Verbose.isVerbose(AsyncResponseBean.class);
```

路径已经指出来了 /_async/.. 全部指向此 Bean，但是细看类成员函数的时候就只有俩：
`handleFault` 和 `handleResult`

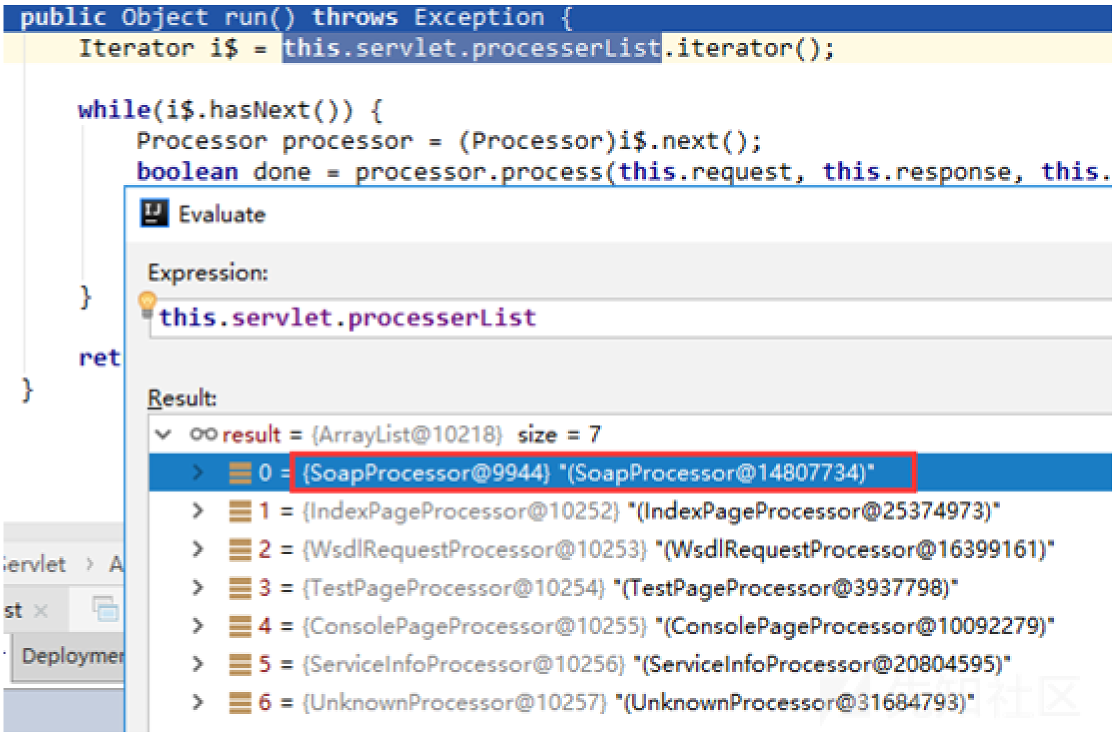
从这个名字来看，属于已经结束处理流程了，正在处理异常和结果，这里稍微想了想如果是soap过去的反序列化的话，那应该是处理流程中触发漏洞，为了确认仔细看了下
handleFault 和 handleResult 函数，确实没有触发点，既没有反序列化点

从底层摸起

那么这就奇怪了，难道不是war包的问题？找一找处理流程，但是weblogic没有详细分析过不知道整个生命周期，只能从 HttpServlet
开始下断点，中间的迷障也太多了，先整理下已知信息：

```
■■■■weblogic.wsee.async
■■■■■■■■■■■■ async ■■■■■ wsee ■■■■■■■■■■

■■■■■■soap■■■■■xml■■■■
```

打了个 HttpServlet 处的断点，跟进了 weblogic.wsee 包下的基础 Servlet ： BaseWSServlet



```
public Object run() throws Exception {
    Iterator i$ = this.servlet.processerList.iterator();

    while(i$.hasNext()) {
        Processor processor = (Processor)i$.next();
        boolean done = processor.process(this.request, this.response, this.
    }

    ret
}
```

Evaluate

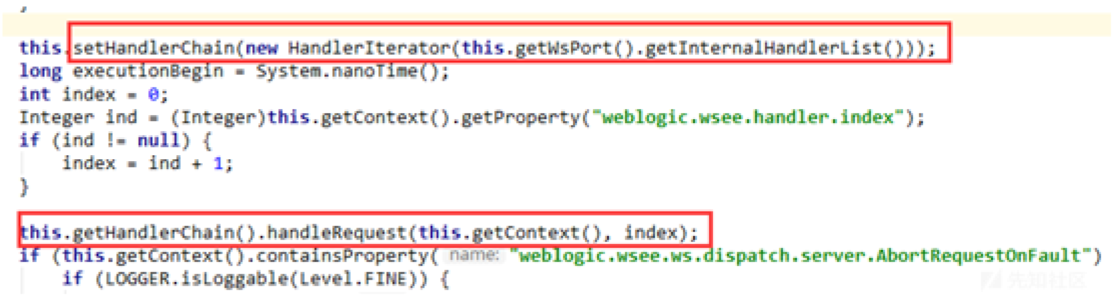
Expression:

```
this.servlet.processerList
```

Result:

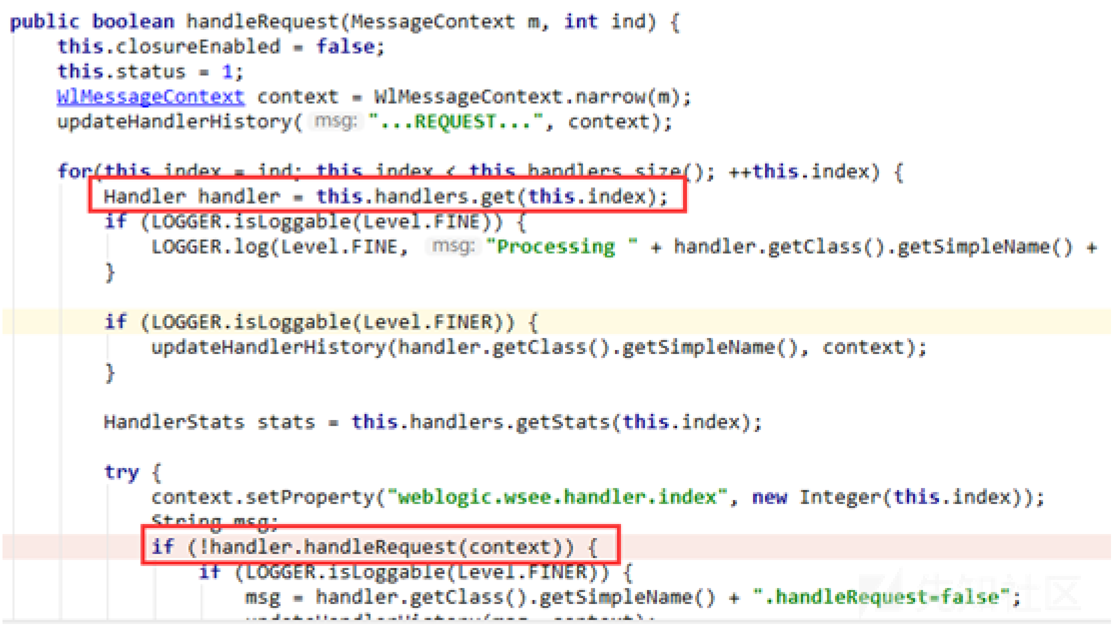
```
result = {ArrayList@10218}  size = 7
  0 = {SoapProcessor@9944} "(SoapProcessor@14807734)"
  1 = {IndexPageProcessor@10252} "(IndexPageProcessor@25374973)"
  2 = {WsdlRequestProcessor@10253} "(WsdlRequestProcessor@16399161)"
  3 = {TestPageProcessor@10254} "(TestPageProcessor@3937798)"
  4 = {ConsolePageProcessor@10255} "(ConsolePageProcessor@10092279)"
  5 = {ServiceInfoProcessor@10256} "(ServiceInfoProcessor@20804595)"
  6 = {UnknownProcessor@10257} "(UnknownProcessor@31684793)"
```

根据已知信息那必然在 soapProcessor 中，一直跟到了 web.wsee.ws.dispatch.server.ServerDispatcher 里面，注意如下：



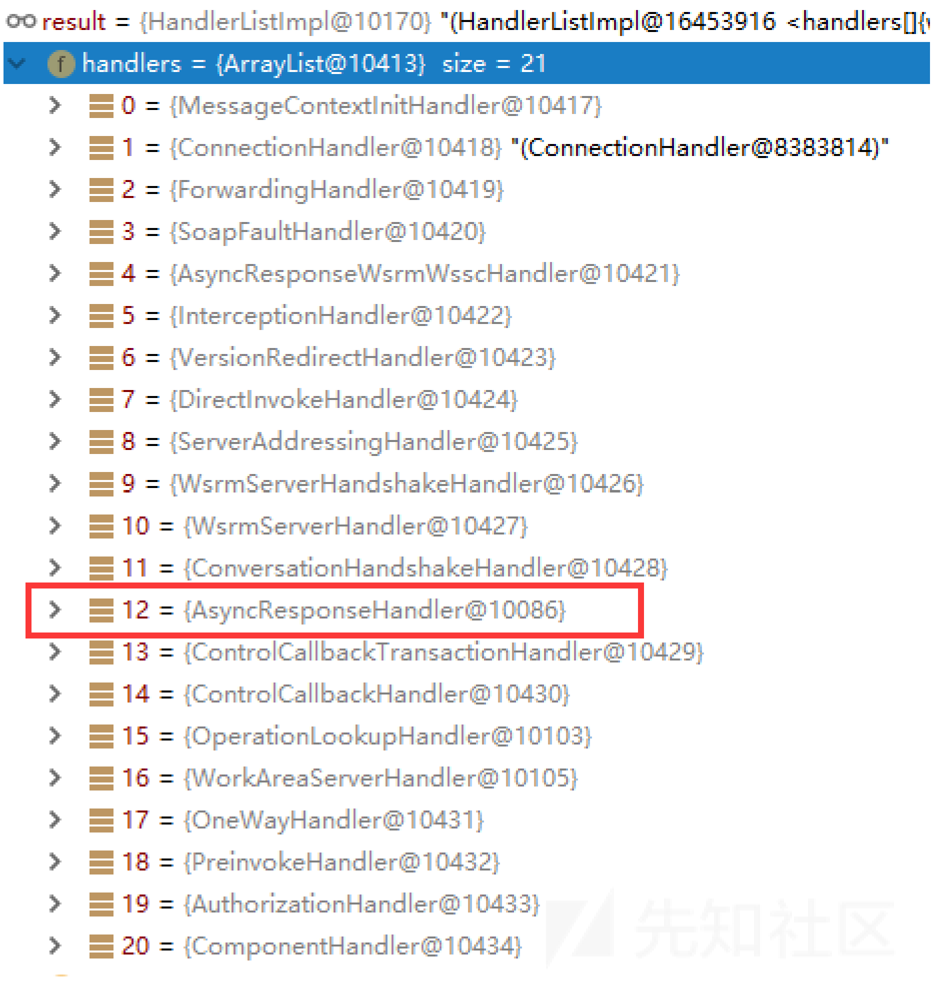
```
this.setHandlerChain(new HandlerIterator(this.getWsPort().getInternalHandlerList()));
long executionBegin = System.nanoTime();
int index = 0;
Integer ind = (Integer)this.getContext().getProperty("weblogic.wsee.handler.index");
if (ind != null) {
    index = ind + 1;
}

this.getHandlerChain().handleRequest(this.getContext(), index);
if (this.getContext().containsProperty( name: "weblogic.wsee.ws.dispatch.server.AbortRequestOnFault")
    if (LOGGER.isLoggable(Level.FINE)) {
```

责任链出来了，跟进去看看 HandlerIterator#handleRequest



```
public boolean handleRequest(MessageContext m, int ind) {
    this.closureEnabled = false;
    this.status = 1;
    WlMessageContext context = WlMessageContext.narrow(m);
    updateHandlerHistory( msg: "...REQUEST...", context);

    for(this.index = ind; this.index < this.handlers.size(); ++this.index) {
        Handler handler = this.handlers.get(this.index);
        if (LOGGER.isLoggable(Level.FINE)) {
            LOGGER.log(Level.FINE,  msg: "Processing " + handler.getClass().getSimpleName() +
        }

        if (LOGGER.isLoggable(Level.FINER)) {
            updateHandlerHistory(handler.getClass().getSimpleName(), context);
        }

        HandlerStats stats = this.handlers.getStats(this.index);

        try {
            context.setProperty("weblogic.wsee.handler.index", new Integer(this.index));
            String msg;
            if (!handler.handleRequest(context)) {
                if (LOGGER.isLoggable(Level.FINER)) {
                    msg = handler.getClass().getSimpleName() + ".handleRequest=false";
```

责任链中轮询调用 handleRequest 处理。
看一看这个 HandlerIterator 中有哪些 Handler

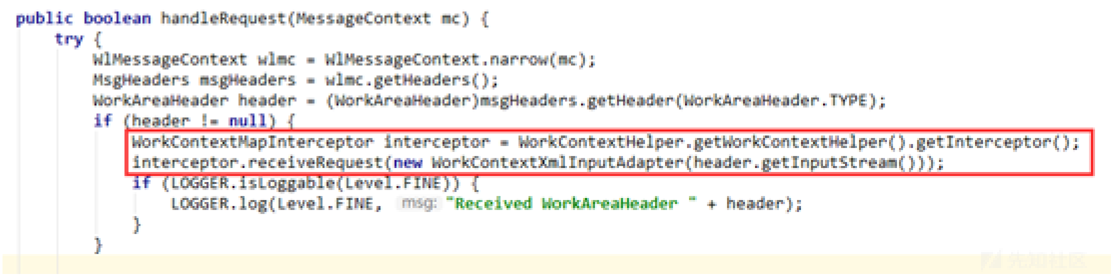


```
oo result = {HandlerListImpl@10170} "(HandlerListImpl@16453916 <handlers[]{
  v (f) handlers = {ArrayList@10413}  size = 21
      > 0 = {MessageContextInitHandler@10417}
      > 1 = {ConnectionHandler@10418} "(ConnectionHandler@8383814)"
      > 2 = {ForwardingHandler@10419}
      > 3 = {SoapFaultHandler@10420}
      > 4 = {AsyncResponseWsrmWsscHandler@10421}
      > 5 = {InterceptionHandler@10422}
      > 6 = {VersionRedirectHandler@10423}
      > 7 = {DirectInvokeHandler@10424}
      > 8 = {ServerAddressingHandler@10425}
      > 9 = {WsrmServerHandshakeHandler@10426}
      > 10 = {WsrmServerHandler@10427}
      > 11 = {ConversationHandshakeHandler@10428}
      > 12 = {AsyncResponseHandler@10086}
      > 13 = {ControlCallbackTransactionHandler@10429}
      > 14 = {ControlCallbackHandler@10430}
      > 15 = {OperationLookupHandler@10103}
      > 16 = {WorkAreaServerHandler@10105}
      > 17 = {OneWayHandler@10431}
      > 18 = {PreinvokeHandler@10432}
      > 19 = {AuthorizationHandler@10433}
      > 20 = {ComponentHandler@10434}
```

如图一共有21个，其中最让我起疑的就是 AsyncResponseHandler
但是仔细看了以后发现没有过于特殊的地方，并且需要前置条件太多，也就是需要用户填写的信息过多，其中很多信息不一定是每个服务器上都一样的。排除它。

柳暗花明

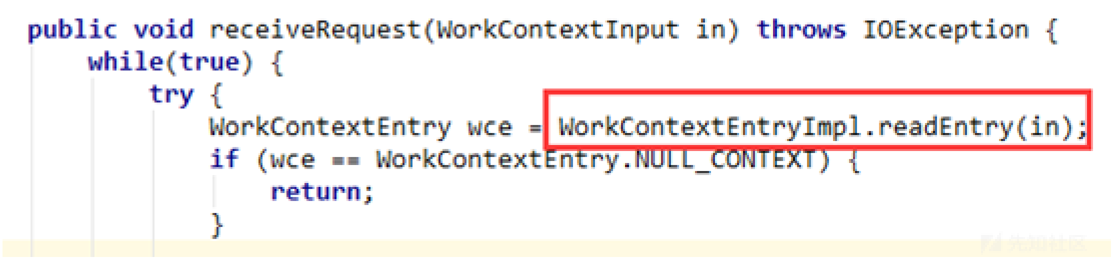
既然是责任链调用，那么他会从 Handler 0 一直执行到 Handler 20，挨个查阅了后，发现大多是对环境的各种值做存取操作，并没有特殊的地方，但是 `WorkAreaServerHandler` 这个handler除外，跟进去看看



```java
public boolean handleRequest(MessageContext mc) {
    try {
        WlMessageContext wlmc = WlMessageContext.narrow(mc);
        MsgHeaders msgHeaders = wlmc.getHeaders();
        WorkAreaHeader header = (WorkAreaHeader)msgHeaders.getHeader(WorkAreaHeader.TYPE);
        if (header != null) {
            WorkContextMapInterceptor interceptor = WorkContextHelper.getWorkContextHelper().getInterceptor();
            interceptor.receiveRequest(new WorkContextXmlInputAdapter(header.getInputStream()));
            if (LOGGER.isLoggable(Level.FINE)) {
                LOGGER.log(Level.FINE,  msg: "Received WorkAreaHeader " + header);
            }
        }
```

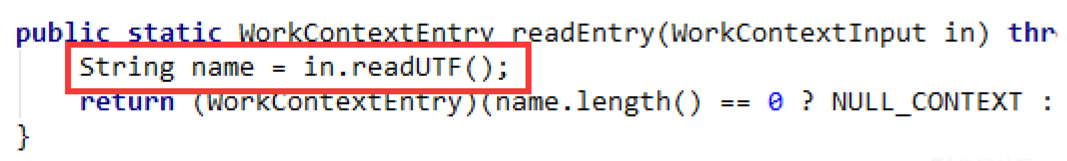
获取了一次header中的内容，这个header不是http header，是soap中的 http://schemas.xmlsoap.org/soap/envelope/ 内容里面的 Header，将其送入 WorkContextXmlInputAdapter 做初始化处理并且传入 receiveRequest 函数

跟进 receiveRequest 函数，如下：



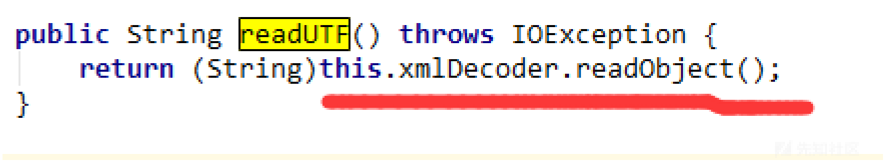
```java
public void receiveRequest(WorkContextInput in) throws IOException {
    while(true) {
        try {
            WorkContextEntry wce = WorkContextEntryImpl.readEntry(in);
            if (wce == WorkContextEntry.NULL_CONTEXT) {
                return;
            }
```

跟进 readEntry 函数，如下：



```
public static WorkContextEntry readEntry(WorkContextInput in) thr
    String name = in.readUTF();
    return (WorkContextEntry)(name.length() == 0 ? NULL_CONTEXT :
}
```

这里调用了 WorkContextXmlInputAdapter 的 readUTF 函数，跟进，如下：



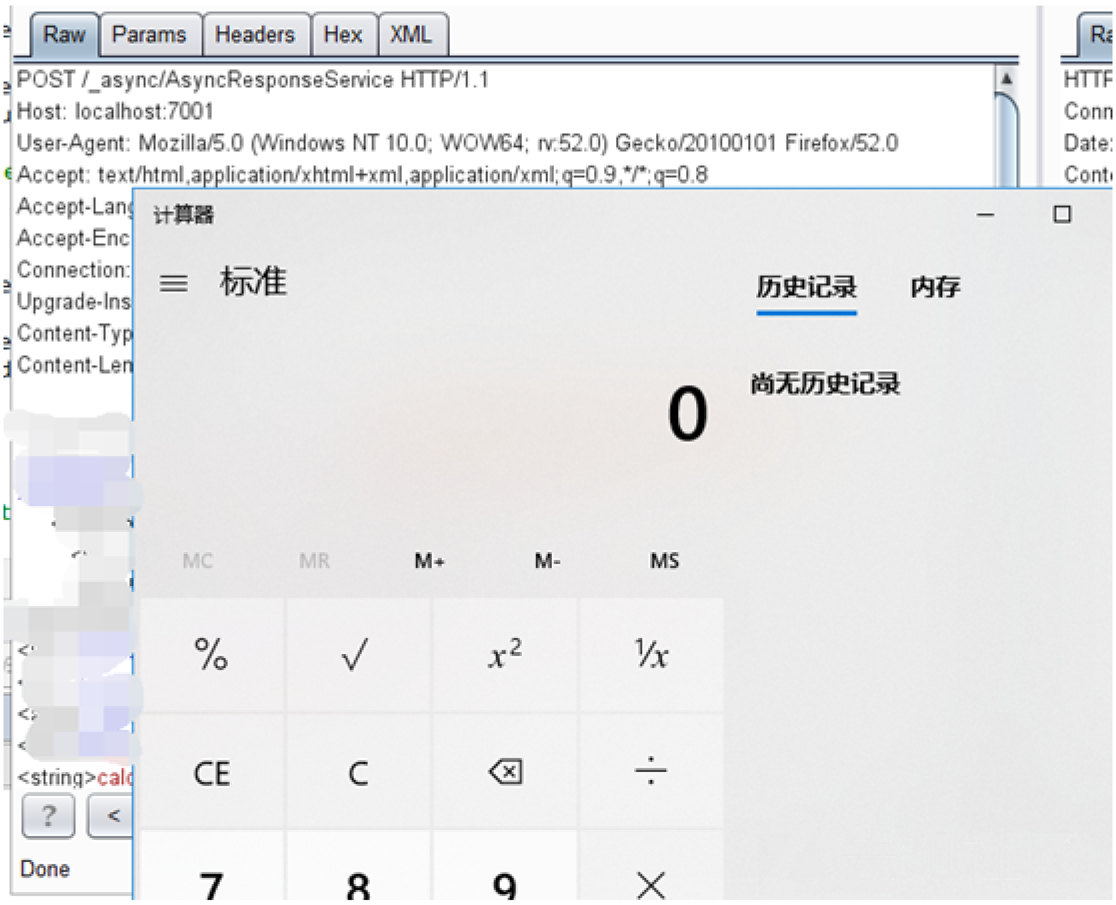
```
public String readUTF() throws IOException {
    return (String)this.xmlDecoder.readObject();
}
```

readObject 映入眼帘

分析流程结束

效果

尝试构造了一下poc，10.3.6 本地未加任何补丁，win10



```
POST /_async/AsyncResponseService HTTP/1.1
Host: localhost:7001
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Lang
Accept-Enc
Connection:
Upgrade-Ins
Content-Typ
Content-Len
<string>calc
```

点击收藏 | 0 关注 | 1

上一篇：Android反调试——从源码入手 下一篇：Django中的OWASP防御手段...

1. 3 条回复

chybeta 2019-04-25 09:00:44

加上绕过补丁后才算真正的0day

0 回复Ta

---

orich1 2019-04-25 10:52:11

@chybeta 放了0day就不能发文了23333

0 回复Ta

---

lucifaer 2019-04-25 11:03:32

想要可用的话，限制还是有点大

0 回复Ta

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