Gregor Samsa: Exploiting Java's XML Signature Verification

Gregor Samsa: Exploiting Java's XML Signature Verification - Felix Wilhelm, projectzero.google.

- Published: date not stated
- Original: <https://googleprojectzero.blogspot.com/2022/11/gregor-samsa-exploiting-java-xml.html>
- Current location: <https://projectzero.google/2022/11/gregor-samsa-exploiting-java-xml.html>
- Preserved from: https://projectzero.google/2022/11/gregor-samsa-exploiting-java-xml.html (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

By Felix Wilhelm, Project Zero

Earlier this year, I discovered a surprising attack surface hidden deep inside Java's standard library: A custom JIT compiler processing untrusted XSLT programs, exposed to remote attackers during XML signature verification. This post discusses CVE-2022-34169, an integer truncation bug in this JIT compiler resulting in arbitrary code execution in many Java-based web applications and identity providers that support the SAML single-sign-on standard.

OpenJDK fixed the discussed issue in July 2022. The Apache BCEL project used by Xalan-J, the origin of the vulnerable code, released a patch in September 2022.

While the vulnerability discussed in this post has been patched , vendors and users should expect further vulnerabilities in SAML.

From a security researcher's perspective, this vulnerability is an example of an integer truncation issue in a memory-safe language, with an exploit that feels very much like a memory corruption. While less common than the typical memory safety issues of C or C++ codebases, weird machines still exist in memory safe languages and will keep us busy even after we move into a bright memory safe future.

Before diving into the vulnerability and its exploit, I'm going to give a quick overview of XML signatures and SAML. What makes XML signatures such an interesting target and why should we care about them?

Introduction

XML Signatures are a typical example of a security protocol invented in the early 2000's. They suffer from high complexity, a large attack surface and a wealth of configurable features that can weaken or break its security guarantees in surprising ways. Modern usage of XML signatures is mostly restricted to somewhat obscure protocols and legacy applications, but there is one important exception: SAML. SAML, which stands for Security Assertion Markup Language, is one of the two main Single-Sign-On standards used in modern web applications. While its alternative, the OAuth based OpenID Connect (OIDC) is gaining popularity, SAML is still the de-facto standard for large enterprises and complex integrations.

SAML relies on XML signatures to protect messages forwarded through the browser. This turns XML signature verification into a very interesting external attack surface for attacking modern multi-tenant SaaS applications. While you don't need a detailed understanding of SAML to follow this post, interested readers can take a look at Okta's Understanding SAML writeup or the SAML 2.0 wiki entry to get a better understanding of the protocol.

SAML SSO logins work by exchanging XML documents between the application, known as service provider (SP), and the identity provider (IdP). When a user tries to login to an SP, the service provider creates a SAML request. The IdP looks at the SAML request, tries to authenticate the user and sends a SAML response back to the SP. A successful response will contain information about the user, which the application can then use to grant access to its resources.

In the most widely used SAML flow (known as SP Redirect Bind / IdP POST Response) these documents are forwarded through the user's browser using HTTP redirects and POST requests. To protect against modification by the user, the security critical part of the SAML response (known as Assertion) has to be cryptographically signed by the IdP. In addition, the IdP might require SPs to also sign the SAML request to protect against impersonation attacks.

This means that both the IdP and the SP have to parse and verify XML signatures passed to them by a potential malicious actor. Why is this a problem? Let's take a closer look at the way XML signatures work:

XML Signatures

Most signature schemes operate on a raw byte stream and sign the data as seen on the wire. Instead, the XML signature standard (known as XMLDsig) tries to be robust against insignificant changes to the signed XML document. This means that changing whitespaces, line endings or comments in a signed document should not invalidate its signature.

An XML signature consists of a special Signature element, an example of which is shown below:

```
<Signature>
<SignedInfo>
<CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
<SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
```

<Reference URI="#signed-data">

<Transforms>

...

</Transforms>

<DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />

<DigestValue>9bc34549d565d9505b287de0cd20ac77be1d3f2c</DigestValue>

</Reference>

</SignedInfo>

<SignatureValue>....</SignatureValue>

<KeyInfo><X509Certificate>....</X509Certificate></KeyInfo>

</Signature>

| |

-

The SignedInfo child contains CanonicalizationMethod and SignatureMethod elements as well as one or more Reference elements describing the integrity protected data.

-

KeyInfo describes the signer key and can contain a raw public key, a X509 certificate or just a key id.

-

SignatureValue contains the cryptographic signature (using SignatureMethod) of the SignedInfo element after it has been canonicalized using CanonicalizationMethod.

At this point, only the integrity of the SignedInfo element is protected. To understand how this protection is extended to the actual data, we need to take a look at the way Reference elements work: In theory the Reference URI attribute can either point to an external document (detached signature), an element embedded as a child (enveloping signature) or any element in the outer document (enveloped signature). In practice, most SAML implementations use enveloped signatures and the Reference URI will point to the signed element somewhere in the current document tree.

When a Reference is processed during verification or signing, the referenced content is passed through a chain of Transforms. XMLDsig supports a number of transforms ranging from canonicalization, over base64 decoding to XPath or even XSLT. Once all transforms have been processed the resulting byte stream is passed into the cryptographic hash function specified with the DigestMethod element and the result is stored in DigestValue.

This way, as the whole Reference element is part of SignedInfo, its integrity protection gets extended to the referenced element as well.

Validating a XML signature can therefore be split into two separate steps:

-

Reference Validation: Iterate through all embedded references and for each reference fetch the referenced data, pump it through the Transforms chain and calculate its hash digest. Compare the calculated Digest with the stored DigestValue and fail if they differ.

-

Signature Validation: First canonicalize the SignedInfo element using the specified CanonicalizationMethod algorithm. Calculate the signature of SignedInfo using the algorithm specified in SignatureMethod and the signer key described in KeyInfo. Compare the result with SignatureValue and fail if they differ.

Interestingly, the order of these two steps can be implementation specific. While the XMLDsig RFC lists Reference Validation as the first step, performing Signature Validation first can have security advantages as we will see later on.

Correctly validating XML signatures and making sure the data we care about is protected, is very difficult in the context of SAML. This will be a topic for later blog posts, but at this point we want to focus on the reference validation step:

As part of this step, the application verifying a signature has to run attacker controlled transforms on attacker controlled input. Looking at the list of transformations supported by XMLDsig, one seems particularly interesting: XSLT.

XSLT, which stands for Extensible Stylesheet Language Transformations, is a feature-rich XML based programming language designed for transforming XML documents. Embedding a XSLT transform in a XML signature means that the verifier has to run the XSLT program on the referenced XML data.

The code snippet below gives you an example of a simple XSLT transformation. When executed it fetches each <data> element stored inside <input>, grabs the first character of its content and returns it as part of its <output>. So <input><data>abc</data><data>def</data></input> would be transformed into <output><data>a</data><data>d</data></output>

|

```
<Transform Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/strict" exclude-result-prefixes="foo"
version="1.0">
<xsl:output encoding="UTF-8" indent="no" method="xml" />
<xsl:template match="/input">
<output>
<xsl:for-each select="data">
<data><xsl:value-of select="substring(.,1,1)" /></data>
</xsl:for-each>
</output>
</xsl:template>
```

</xsl:stylesheet>

</Transform>

| |

Exposing a fully-featured language runtime to an external attacker seems like a bad idea, so let's take a look at how this feature is implemented in Java's OpenJDK.

XSLT in Java

Java's main interface for working with XML signatures is the java.xml.crypto.XMLSignature class and its sign and validate methods. We are mostly interested in the validate method which is shown below:

|

```
//
https://github.com/openjdk/jdk/blob/master/src/java.xml.crypto/share/classes/org/jcp/xml/dsig/internal/dom/DOMXMLSignature.java
@Override
public boolean validate(XMLValidateContext vc)
throws XMLSignatureException
{
[..]
// validate the signature
boolean sigValidity = sv.validate(vc); (A)
if (!sigValidity) {
validationStatus = false;
validated = true;
return validationStatus;
}
// validate all References
@SuppressWarnings("unchecked")
List<Reference> refs = this.si.getReferences();
boolean validateRefs = true;
for (int i = 0, size = refs.size(); validateRefs && i < size; i++) {
Reference ref = refs.get(i);
boolean refValid = ref.validate(vc); (B)
LOG.debug("Reference [{}] is valid: {}", ref.getURI(), refValid);
validateRefs &= refValid;
```

```
}
if (!validateRefs) {
LOG.debug("Couldn't validate the References");
validationStatus = false;
validated = true;
return validationStatus;
}
[..]
}
| |
```

As we can see, the validate method first validates the signature of the SignedInfo element in (A) before validating all the references in (B). This means that an attack against the XSLT runtime will require a valid signature for the SignedInfo element and we'll discuss later how an attacker can bypass this requirement.

The call to ref.validate() in (B) ends up in the DomReference.validate method shown below:

```
|
public boolean validate(XMLValidateContext validateContext)
throws XMLSignatureException
{
if (validateContext == null) {
throw new NullPointerException("validateContext cannot be null");
}
if (validated) {
return validationStatus;
}
Data data = dereference(validateContext); (D)
calcDigestValue = transform(data, validateContext); (E)
[..]
validationStatus = Arrays.equals(digestValue, calcDigestValue); (F)
validated = true;
return validationStatus;
}
| |
```

The code gets the referenced data in (D), transforms it in (E) and compares the digest of the result with the digest stored in the signature in (F). Most of the complexity is hidden behind the call to transform in (E) which loops through all Transform elements defined in the Reference and executes them.

As this is Java, we have to walk through a lot of indirection layers before we end up at the interesting parts. Take a look at the call stack below if you want to follow along and step through the code:

at com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl.newTransformer(TemplatesImpl.java:584) at com.sun.org.apache.xalan.internal.xsltc.trax.TransformerFactoryImpl.newTransformer(TransformerFactoryImpl.java:818) at com.sun.org.apache.xml.internal.security.transforms.implementations.TransformXSLT.enginePerformTransform(TransformXSLT.java:130) at com.sun.org.apache.xml.internal.security.transforms.Transform.performTransform(Transform.java:316) at org.jcp.xml.dsig.internal.dom.ApacheTransform.transformIt(ApacheTransform.java:188) at org.jcp.xml.dsig.internal.dom.ApacheTransform.transform(ApacheTransform.java:124) at org.jcp.xml.dsig.internal.dom.DOMTransform.transform(DOMTransform.java:173) at org.jcp.xml.dsig.internal.dom.DOMReference.transform(DOMReference.java:457) at org.jcp.xml.dsig.internal.dom.DOMReference.validate(DOMReference.java:387) at org.jcp.xml.dsig.internal.dom.DOMXMLSignature.validate(DOMXMLSignature.java:281)

If we specify a XSLT transform and the org.jcp.xml.dsig.secureValidation property isn't enabled (we'll come back to this later) we will end up in the file src/java.xml/share/classes/com/sun/org/apache/xalan/internal/xsltc/trax/TransformerFactoryImpl.java which is part of a module called XSLTC.

XSLTC, the XSLT compiler, is originally part of the Apache Xalan project. OpenJDK forked Xalan-J, a Java based XSLT runtime, to provide XSLT support as part of Java's standard library. While the original Apache project has a number of features that are not supported in the OpenJDK fork, most of the core code is identical and CVE-2022-34169 affected both codebases.

XSLTC is responsible for compiling XSLT stylesheets into Java classes to improve performance compared to a naive interpretation based approach. While this has advantages when repeatedly running the same stylesheet over large amounts of data, it is a somewhat surprising choice in the context of XML signature validation. Thinking about this from an attacker's perspective, we can now provide arbitrary inputs to a fully-featured JIT compiler. Talk about an unexpected attack surface!

A bug in XSLTC

|

/**

- As Gregor Samsa awoke one morning from uneasy dreams he found himself
- transformed in his bed into a gigantic insect. He was lying on his hard,

- as it were armour plated, back, and if he lifted his head a little he
- could see his big, brown belly divided into stiff, arched segments, on
- top of which the bed quilt could hardly keep in position and was about
- to slide off completely. His numerous legs, which were pitifully thin
- compared to the rest of his bulk, waved helplessly before his eyes.
- "What has happened to me?", he thought. It was no dream....

*/

protected final static String DEFAULT_TRANSLET_NAME = "GregorSamsa";

| | |

Turns out that the author of this codebase was a Kafka fan.

| |

So what does this compilation process look like? XSLTC takes a XSLT stylesheet as input and returns a JIT'ed Java class, called translet, as output. The JVM then loads this class, constructs it and the XSLT runtime executes the transformation via a JIT'ed method.

Java class files contain the JVM bytecode for all class methods, a so-called constant pool describing all constants used and other important runtime details such as the name of its super class or access flags.

|

// https://docs.oracle.com/javase/specs/jvms/se18/html/jvms-4.html

ClassFile {

u4 magic;

u2 minor_version;

u2 major_version;

u2 constant_pool_count;

cp_info constant_pool[constant_pool_count-1]; // cp_info is a variable-sized object

u2 access_flags;

u2 this_class;

u2 super_class;

u2 interfaces_count;

u2 interfaces[interfaces_count];

u2 fields_count;

field_info fields[fields_count];

u2 methods_count;

method_info methods[methods_count];

u2 attributes_count;

attribute_info attributes[attributes_count];

}

| |

XSLTC depends on the Apache Byte Code Engineering Library (BCEL) to dynamically create Java class files. As part of the compilation process, constants in the XSLT input such as Strings or Numbers get translated into Java constants, which are then stored in the constant pool. The following code snippet shows how an XSLT integer expression gets compiled: Small integers that fit into a byte or short are stored inline in bytecode using the bipush or sipush instructions. Larger ones are added to the constant pool using the cp.addInteger method:

|

// org/apache/xalan/xsltc/compiler/IntExpr.java

public void translate(ClassGenerator classGen, MethodGenerator methodGen) {

ConstantPoolGen cpg = classGen.getConstantPool();

InstructionList il = methodGen.getInstructionList();

il.append(new PUSH(cpg, _value));

}

// org/apache/bcel/internal/generic/PUSH.java

public PUSH(final ConstantPoolGen cp, final int value) {

if ((value >= -1) && (value <= 5)) {

instruction = InstructionConst.getInstruction(Const.ICONST_0 + value);

} else if (Instruction.isValidByte(value)) {

instruction = new BIPUSH((byte) value);

} else if (Instruction.isValidShort(value)) {

instruction = new SIPUSH((short) value);

} else {

instruction = new LDC(cp.addInteger(value));

}

}

| |

The problem with this approach is that neither XSLTC nor BCEL correctly limits the size of the constant pool. As constant_pool_count, which describes the size of the constant pool, is only 2 bytes long, its maximum size is limited to 2**16 - 1 or 65535 entries. In practice even fewer entries are possible, because some constant types take up two entries. However, BCELs internal constant pool representation uses a standard Java Array for storing constants, and does not enforce any limits on its length.

When XSLTC processes a stylesheet that contains more constants, and BCELs internal class representation is serialized to a class file at the end of the compilation process the array length is truncated to a short, but the complete array is written out:

This means that constant_pool_count will now contain a small value and that parts of the attacker-controlled constant pool will get interpreted as the class fields following the constant pool, including method and attribute definitions.

Exploiting a constant pool overflow

To understand how we can exploit this, we first need to take a closer look at the content of the constant pool. Each entry in the pool starts with a 1-byte tag describing the kind of constant, followed by the actual data. The table below shows an incomplete list of constant types supported by the JVM (see the official documentation) for a complete list). No need to read through all of them but we will come back to this table a lot when walking through the exploit.

|

Constant Kind

|

Tag

|

Description

|

Layout

| | |

CONSTANT_Utf8

|

1

|

Constant variable-sized UTF-8 string value

|

CONSTANT_Utf8_info {

u1 tag;

u2 length;

u1 bytes[length];

}

| | |

CONSTANT_Integer

|

3

|

4-byte integer

|

CONSTANT_Integer_info {

u1 tag;

u4 bytes;

}

| | |

CONSTANT_Float

|

4

|

4-byte float

|

CONSTANT_Float_info {

u1 tag;

u4 bytes;

}

| | |

CONSTANT_Long

|

5

|

8-byte long

|

CONSTANT_Long_info {

u1 tag;

u4 high_bytes;

u4 low_bytes;

}

| | |

CONSTANT_Double

|

6

|

8-byte double

|

CONSTANT_Double_info {

u1 tag;

u4 high_bytes;

u4 low_bytes;

}

| | |

CONSTANT_Class

|

7

|

Reference to a class. Links to a Utf8 constant describing the name.

|

CONSTANT_Class_info {

u1 tag;

u2 name_index;

}

| | |

CONSTANT_String

|

8

|

A JVM “String”. Links to a UTF8 constant.

|

CONSTANT_String_info {

u1 tag;

u2 string_index;

}

| | |
CONSTANT_Fieldref
CONSTANT_Methodref
CONSTANT_InterfaceMethodref
|
9
10
11
|
Reference to a class field or method. Links to a Class constant
|
CONSTANT_Fieldref_info {
u1 tag;
u2 class_index;
u2 name_and_type_index;
}
CONSTANT_Methodref_info {
u1 tag;
u2 class_index;
u2 name_and_type_index;
}
CONSTANT_InterfaceMethodref_info {
u1 tag;
u2 class_index;
u2 name_and_type_index;
}
| | |
CONSTANT_NameAndType
|
12
|
Describes a field or method.
|

CONSTANT_NameAndType_info {

u1 tag;

u2 name_index;

u2 descriptor_index;

}

| | |

CONSTANT_MethodHandle

|

15

|

Describes a method handle.

|

CONSTANT_MethodHandle_info {

u1 tag;

u1 reference_kind;

u2 reference_index;

}

| | |

CONSTANT_MethodType

|

16

|

Describes a method type. Points to a UTF8 constant containing a method descriptor.

|

CONSTANT_MethodType_info {

u1 tag;

u2 descriptor_index;

}

| |

A perfect constant type for exploiting CVE-2022-34169 would be dynamically sized containing fully attacker controlled content. Unfortunately, no such type exists. While CONSTANT_Utf8 is dynamically sized, its content isn't a raw string representation but an encoding format JVM calls "modified UTF-8". This encoding introduces some significant restrictions on the data stored and rules out null bytes, making it mostly useless for corrupting class fields.

The next best thing we can get is a fixed size constant type with full control over the content. CONSTANT_Long seems like an obvious candidate, but XSLTC never creates attacker-controlled long constants during the compilation process. Instead we can use large floating numbers to create CONSTANT_Double entry with (almost) fully controlled content. This gives us a nice primitive where we can corrupt class fields behind the constant pool with a byte pattern like 0x06 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0xXX 0x06 0xYY 0xYY 0xYY 0xYY 0xYY 0xYY 0xYY 0x06 0xZZ 0xZZ 0xZZ 0xZZ 0xZZ 0xZZ 0xZZ 0xZZ.

Unfortunately, this primitive alone isn't sufficient for crafting a useful class file due to the requirements of the fields right after the constant_pool:

|

cp_info constant_pool[constant_pool_count-1];

u2 access_flags;

u2 this_class;

u2 super_class;

| |

access_flags is a big endian mask of flags describing access permissions and properties of the class.

While the JVM is happy to ignore unknown flag values, we need to avoid setting flags like ACC_INTERFACE (0x0200) or ACC_ABSTRACT (0x0400) that result in an unusable class. This means that we can't use a CONSTANT_Double entry as our first out-of-bound constant as its tag byte of 0x06 will get interpreted as these flags.

this_class is an index into the constant pool and has to point to a CONSTANT_Class entry that describes the class defined with this file. Fortunately, neither the JVM nor XSLTC cares much about which class we pretend to be, so this value can point to almost any CONSTANT_Class entry that XSLTC ends up generating. (The only restriction is that it can't be a part of a protected namespace like java.lang.)

super_class is another index to a CONSTANT_Class entry in the constant pool. While the JVM is happy with any class, XSLTC expects this to be a reference to the org.apache.xalan.xsltc.runtime.AbstractTranslet class, otherwise loading and initialization of the class file fails.

After a lot of trial and error I ended up with the following approach to meet these requirements:

CONST_STRING CONST_DOUBLE

0x08 0x07 0x02 0x06 0xXX 0xXX 0x00 0x00 0x00 0x00 0xZZ 0xZZ

access_flags this_class super_class ints_count fields_count methods_count

-

We craft a XSLT input that results in 0x10703 constants in the pool. This will result in a truncated pool size of 0x703 and the start of the constant at index 0x703 (due to 0 based indexing) will be interpreted as access_flags.

-

During compilation of the input, we trigger the addition of a new string constant when the pool has 0x702 constants. This will first create a CONSTANT_Utf8 entry at index 0x702 and a CONSTANT_String entry at 0x703. The String entry will reference the preceding Utf8 constant so its value will be the tag byte 0x08 followed by the index 0x07 0x02. This results in an usable access_flags value of 0x0807.

-

Add a CONSTANT_Double entry at index 0x704. Its 0x06 tag byte will be interpreted as part of the this_class field. The following 2 bytes can then be used to control the value of the super_class field. By setting the next 4 bytes to 0x00, we create an empty interface and fields arrays, before setting the last two bytes to the number of methods we want to define.

The only remaining requirement is that we need to add a CONSTANT_Class entry at index 0x206 of the constant pool, which is relatively straightforward.

The snippet below shows part of the generated XSLT input that will overwrite the first header fields. After filling the constant pool with a large number of string constants for the attribute fields and values, the CONST_STRING entry for the `jEb` element ends up at index 0x703. The XSLT function call to the `ceiling` function then triggers the addition of a controlled CONST_DOUBLE entry at index 0x704:

|

<jse jsf='jsg' ... jDL='jDM' jDN='jDO' jDP='jDQ' jDR='jDS' jDT='jDU' jDV='jDW' jDX='jDY' jDZ='jEa' /><jEb />

<xsl:value-of select='ceiling(0.0000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
08344026969402015)'/>

| |

We constructed the initial header fields and are now in the interesting part of the class file definition: The methods table. This is where all methods of a class and their bytecode is defined. After XSLTC generates a Java class, the XSLT runtime will load the class and instantiate an object, so the easiest way to achieve arbitrary code execution is to create a malicious constructor. Let’s take a look at the methods table to see how we can define a working constructor:

|

ClassFile {

[...]

u2 methods_count;

method_info methods[methods_count];

[...]

}

method_info {

u2 access_flags;

u2 name_index;

u2 descriptor_index;

u2 attributes_count;

attribute_info attributes[attributes_count];

}

attribute_info {

u2 attribute_name_index;

u4 attribute_length;

u1 info[attribute_length];

}

Code_attribute {

u2 attribute_name_index;

u4 attribute_length;

u2 max_stack;

u2 max_locals;

u4 code_length;

u1 code[code_length];

u2 exception_table_length;

{ u2 start_pc;

u2 end_pc;

u2 handler_pc;

u2 catch_type;

} exception_table[exception_table_length];

u2 attributes_count;

attribute_info attributes[attributes_count];

}

| |

The methods table is a dynamically sized array of method_info structs. Each of these structs describes the access_flags of the method, an index into the constant table that points to its name (as a utf8 constant), and another index pointing to the method descriptor (another CONSTANT_Utf8).

This is followed by the attributes table, a dynamically sized map from Utf8 keys stored in the constant table to dynamically sized values stored inline. Fortunately, the only attribute we need to provide is the Code attribute, which contains the actual bytecode of the method.

Going back to our payload, we can see that the start of the methods table is aligned with the tag byte of the next entry in the constant pool table. This means that the 0x06 tag of a CONSTANT_Double will clobber the access_flag field of the first method, making it unusable for us. Instead we have to create two methods: The first one as a basic filler to get the alignment right, and the second one as the actual constructor. Fortunately, the JVM ignores unknown attributes, so we can use dynamically sized attribute values. The graphic below shows how we use a series of CONST_DOUBLE entries to create a constructor method with an almost fully controlled body.

CONST_DOUBLE: 0x06 0x01 0xXX 0xXX 0xYY 0xYY 0x00 0x01 0xZZ

CONST_DOUBLE: 0x06 0x00 0x00 0x00 0x05 0x00 0x00 0x00 0x00

CONST_DOUBLE: 0x06 0x00 0x01 0xCC 0xCC 0xDD 0xDD 0x00 0x03

CONST_DOUBLE: 0x06 0x00 0x00 0x00 0x00 0x04 0x00 0x00 0x00

CONST_DOUBLE: 0x06 0xCC 0xDD 0xZZ 0xZZ 0xZZ 0xZZ 0xAA 0xAA

CONST_DOUBLE: 0x06 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA

CONST_DOUBLE: 0x06 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA

CONST_DOUBLE: 0x06 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA

CONST_DOUBLE: 0x06 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA 0xAA

|

First Method Header

access_flags 0x0601

name_index 0xXXXX

desc_index 0xYYYY

attr_count 0x0001

| | |

Attribute [0]

name_index 0xZZ06

length 0x00000005

data “\x00\x00\x00\x00\x06”

| | | | | | |

Second Method Header

access_flags 0x0001

name_index 0xCCCC -> <init>

desc_index 0xDDDD -> ()V

attr_count 0x0003

| | |

Attribute [0]

name_index 0x0600

length 0x00000004

data “\x00\x00\x00\x06

| | |

Attribute [1]

name_index 0xCCDD -> Code

length 0xZZZZZZZZ

data PAYLOAD

| | |

Attribute [2] ...

| |

We still need to bypass one limitation: JVM bytecode does not work standalone, but references and relies on entries in the constant pool. Instantiating a class or calling a method requires a corresponding constant entry in the pool. This is a problem as our bug doesn’t give us the ability to create fake constant pool entries so we are limited to constants that XSLTC adds during compilation.

Luckily, there is a way to add arbitrary class and method references to the constant pool: Java’s XSLT runtime supports calling arbitrary Java methods. As this is clearly insecure, this functionality is protected by a runtime setting and always disabled during signature verification.

However, XSLTC will still process and compile these function calls when processing a stylesheet and the call will only be blocked during runtime (see the corresponding code in FunctionCall.java). This means that we can get references to all required methods and classes by adding a XSLT element like the one shown below:

|

<xsl:value-of select="rt:exec(rt:getRuntime(),'...')" xmlns:rt="java.lang.Runtime"/>

| |

There are two final checks we need to bypass, before we end up with a working class file:

-

The JVM enforces that every constructor of a subclass, calls a superclass constructor before returning. This check can be bypassed by never returning from our constructor either by adding an endless loop at the end or throwing an exception, which is the approach I used in my exploit Proof-of-Concept. A slightly more complex, but cleaner approach is to add a reference to the AbstractTranslet constructor to the object pool and call it. This is the approach used by thanat0s in their exploit writeup.

-

Finally, we need to skip over the rest of XSLTC's output. This can be done by constructing a single large attribute with the right size as an element in the class attribute table.

Once we chain all of this together we end up with a signed XML document that can trigger the execution of arbitrary JVM bytecode during signature verification. I've skipped over some implementation details of this exploit, so if you want to reproduce this vulnerability please take a look at the heavily commented proof-of-concept script.

Impact and Restrictions

In theory every unpatched Java application that processes XML signatures is vulnerable to this exploit. However, there are two important restrictions:

As references are only processed after the signature of the SignedInfo element is verified, applications can be protected based on their usage of the KeySelecto r%3B%0A%20%20%20%20%7D-,Using%20KeySelectors,-KeySelectors%20are) class. Applications that use a allowlist of trusted keys in their KeySelector will be protected as long as these keys are not compromised. An example of this would be a single-tenant SAML SP configured with a single trusted IdP key. In practice, a lot of these applications are still vulnerable as they don't use KeySelector directly and will instead enforce this restriction in their own application logic after an unrestricted signature validation. At this point the vulnerability has already been triggered. Multi-tenant SAML applications that support customer-provided Identity Providers, as most modern cloud SaaS do, are also not protected by this limitation.

SAML Identity Providers can only be attacked if they support (and verify) signed SAML requests.

Even without CVE-2022-34169, processing XSLT during signature verification can be easily abused as part of a DoS attack. For this reason, the property org.jcp.xml.dsig.secureValidation can be enabled to forbid XSLT transformation in XML signatures. Interestingly this property defaults to false for all JDK versions <17, if the application is not running under the Java security manager. As the Security Manager is rarely used for server side applications and JDK 17 was only released a year ago, we expect that a lot of applications are not protected by this. Limited testing of large java-based SSO providers confirmed this assumption. Another reason for a lack of widespread usage might be that org.jcp.xml.dsig.secureValidation also disables use of the SHA1 algorithm in newer JDK versions. As SHA1 is still widely used by enterprise customers, simply enabling the property without manually configuring a suitable jdk.xml.dsig.secureValidationPolicy might not be feasible.

Conclusion

XML signatures in general and SAML in particular offer a large and complex attack surface to external attackers. Even though Java offers configuration options that can be used to address this vulnerability, they are complex, might break real-world use cases and are off by default.

Developers that rely on SAML should make sure they understand the risks associated with it and should reduce the attack surface as much as possible by disabling functionality that is not required

for their use case. Additional defense-in-depth approaches like early validation of signing keys, an allow-list based approach to valid transformation chains and strict schema validation of SAML tickets can be used for further hardening.
