

# XML Risks and Mitigations

---

Roger L. Costello

# Objective

---

This course examines XML to identify some **inherent vulnerabilities**, some of which could be maliciously exploited.

The course discusses the strengths and weaknesses of various ways to **mitigate the vulnerabilities**.

# Other Sources of Information

---

- **The CWE/CVE databases are excellent resources for information about security vulnerabilities.**
  - CVE provides vulnerabilities found in commercial and open source software, including information about patches
  - CWE provides a listing of the types of vulnerabilities
- **Searching the CVE database for “XML” returns thousands of hits. For more information about weaknesses and vulnerabilities related to XML, visit <http://cwe.mitre.org> and <http://cve.mitre.org>.**

# Toolbox

---

- **The material in these slides are a set of tools.**
- **Some tools may be useful to you, some may not.**
- **You will be exposed to the tools so that you know they exist and can be used, if desired.**

# Table of Contents

---

- **Security Terminology**
- **XML lacks inherent security**
- **A valid XML document can still cause trouble**
- **Security considerations when processing XML documents**
  - Reading inputs from external URLs and XInclude (external entities)
  - Attack surface
  - Pros and cons of spending the resources to check inputs
- **Miscellaneous security topics**
  - Expanding entities
  - Poorly constructed regular expressions
  - Manual editing of XML documents (problems with copying and pasting)
  - Unconstrained markup and data
- **A brief *introduction* to XML security tools and capabilities**
  - Canonicalization
  - XML Digital Signatures (Integrity)
  - XML Encryption (Confidentiality)

# Categories of Vulnerabilities

---

- **Many of the vulnerabilities described in the following slides fall into one or more of these categories:**
  - **Data attack**
  - **Data hiding**
  - **Data disclosure**

# Data Attack

---

This is where malicious code resides within the data, and the data has been crafted to compromise a vulnerability in the software responsible for parsing the content.

# Data Hiding

---

This is where information is purposely hidden within the file and never seen by the application.



# Data Disclosure

---

This is where information is accidentally leaked.

# CIA Triad

- **Three core goals of information security are confidentiality, integrity, and availability of the information [1]:**
  - ***Confidentiality*** refers to assurance that information is not disclosed to unauthorized users
  - ***Integrity*** means that information is protected against unauthorized modification, whether by accident or malicious activity
  - ***Availability*** means ensuring that data can be accessed when needed

[1] <http://csrc.nist.gov/publications/fips/fips199/FIPS-PUB-199-final.pdf>

# Confidentiality Examples

---

- **Information you want kept confidential:**
  - Your medical records
  - Your salary
  - Your credit card number
  - Your bank PIN number
  - A company's proprietary information
  - Classified information

# Integrity Examples

---

- **Information you want accurate (not accidentally or deliberately altered):**
  - Your retirement account
  - Your bank account
  - If your system crashes, you want the Word document that was open closed without corruption

# Availability Examples

---

- **Information you want available in a timely manner:**
  - You want to purchase tickets for a popular event that will sell out quickly
  - You are at an ATM machine and you need to know your current bank balance
  - You heard about a hot stock and you want to access the web site that is selling the stock

# Table of Contents

- **Security Terminology**
- **XML lacks inherent security**
- **A valid XML document can still cause trouble**
- **Security considerations when processing XML documents**
  - Expanding entities (XML bomb)
  - Reading inputs from external URLs and XInclude (external entities)
  - Attack surface
  - Pros and cons of spending the resources to check inputs
- **Miscellaneous security topics**
  - Poorly constructed regular expressions
  - Manual editing of XML documents (problems with copying and pasting)
  - Unconstrained markup and data
- **A brief *introduction* to XML security tools and capabilities**
  - Canonicalization
  - XML Digital Signatures (Integrity)
  - XML Encryption (Confidentiality)



You are here.

# XML is Text

---

- **An XML document is a plain text document.**
- **It's not a binary document (although there is a binary form of XML called EXI).**
- **Virtually every computing platform worldwide has at least one text-editing application.**

Consequence  Explosive growth of information sharing

Every Windows machine has Notepad and WordPad. Many have Word. They are “text” editors.





# Implications

---

- **The ease of viewing and editing text elevates the importance of mechanisms that support confidentiality and integrity**
- **The XML specification does not provide these mechanisms, so additional measures must be used to provide confidentiality and integrity**
  - XML Encryption can help protect against loss of confidentiality
  - XML Digital Signature can help protect against loss of integrity

# XML Characters are Encoded using an Encoding Scheme

`<?xml version="1.0" encoding="_____"?>`



ASCII, iso-8859-1,  
UTF-8, UTF-16, etc.

# Characters in Encoding Schemes

---

- **ASCII**: basically the characters on your keyboard
- **iso-8859-1**: all the ASCII characters plus characters in the European alphabets – À, Á, Â, Ã, Ä, etc.
- **UTF-8, UTF-16**: all the characters from every language in the world.  
*UTF-8 is the default and the predominate character encoding*

# Number of Characters in Encoding Schemes


---

- **ASCII: 128 characters**
- **iso-8859-1: 256 characters**
- **UTF-8, UTF-16: 1,112,064 characters**

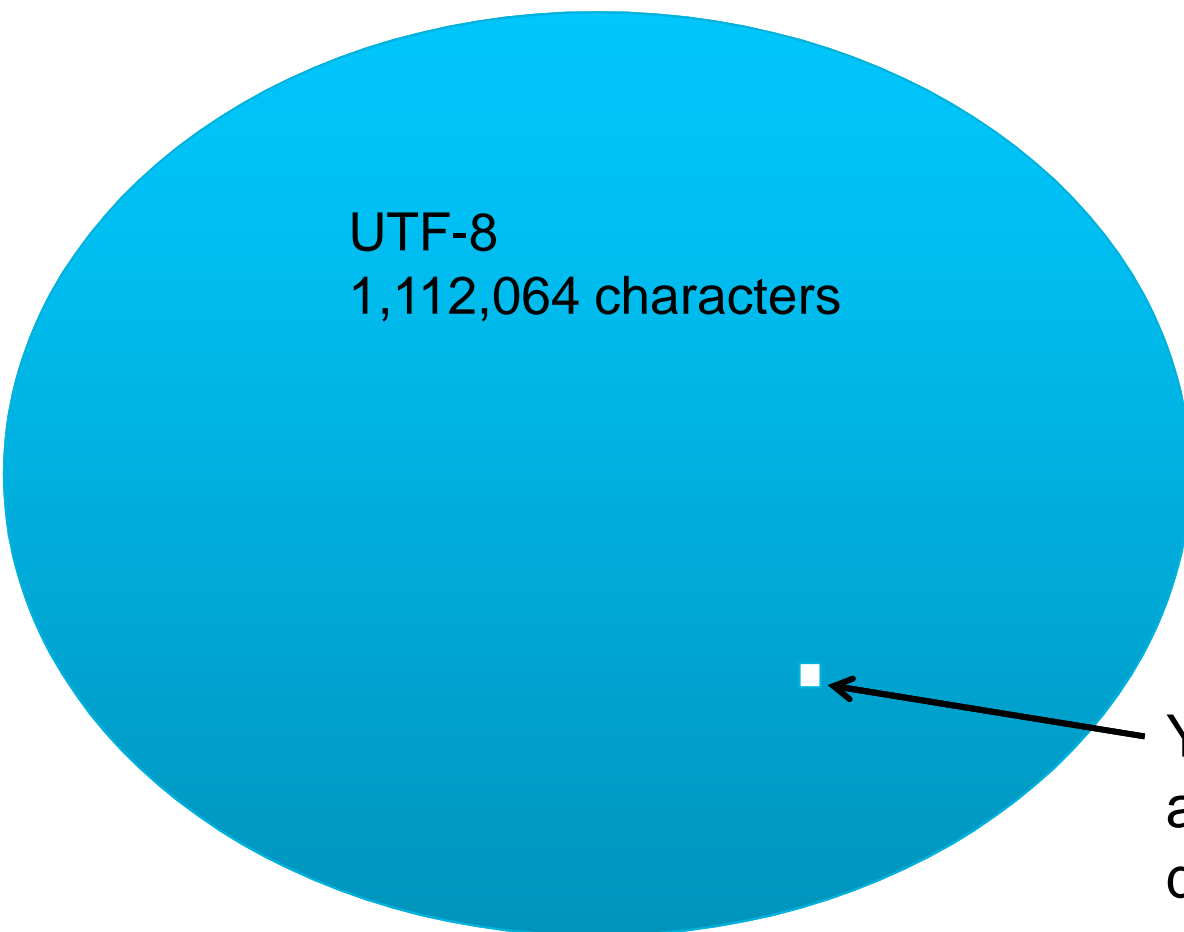
# UTF-8: A Boon to Internationalization

This XML document uses the Russian alphabet for markup and has Japanese content

```
<?xml version="1.0" encoding="utf-8" ?>
<Собрание версия="1.2-3">
  <Объект id="12">
    <НомерОбъекта>45-3454-123</НомерОбъекта>
    <ВНаличии>123</ВНаличии>
    <Описание xml:lang="ja">第二発電機</Описание>
  </Объект>
  <Объект id="64">
    <НомерОбъекта>45-7894-456</НомерОбъекта>
    <ВНаличии>123</ВНаличии>
    <Описание xml:lang="ja">手動ウォーター・ポンプ</Описание>
  </Объект>
</Собрание>
```

Consequence  Easy to share information across languages and cultures

# Applications that Process XML



UTF-8  
1,112,064 characters

Your English language application is likely to be designed to deal with only this subset of all the characters. (The English alphabet is about 0.01 percent of the size of the UTF-8 space.)

# Consequence of a Large Character Set

---

- **The XML specification does not provide mechanisms to validate that XML instance documents contain only a particular subset of characters, so additional measures must be used.**
  - Validation languages such as XML Schema or Relax NG can be used to validate that XML documents contain only the expected set of characters.

# Text is Flexible in Expressing Ideas

---

- The number 25000 can be expressed as *25000* or *25,000* or  $2.5 \times 10^4$  or *twenty five thousand*
- Humans can easily understand data in its various expressions
- Applications cannot process data in arbitrary expressions
- Most applications are designed to expect data in a given type (data type) with a standardized format



# Consequence of Flexible Data

---

- **The XML specification does not provide mechanisms to enforce standard data formats, so additional measures must be used.**
  - XML validation languages such as XML Schema or Relax NG can be used to enforce that XML documents contain data in an expected format.

# Lessons Learned

---

## XML lacks inherent security

**Therefore, XML must be supplemented with additional measures:**


- validation (XSD, RNG)
- confidentiality (XML Encryption)
- integrity (XML Digital Signature)

# Resources

---

- My tutorial on XML digital signatures and XML encryption: see the papers folder, in there is a Powerpoint document, *How-to-transfer-XML-documents-securely-with-integrity.pptx*
- The specification for the *XML Encryption Syntax and Processing* is available at <http://www.w3.org/TR/xmlenc-core/>
- The specification for the *XML Signature Syntax and Processing* is available at <http://www.w3.org/TR/xmldsig-core/>
- The ASCII characters: <http://en.wikipedia.org/wiki/ASCII>
- The iso-8859-1 characters: [http://en.wikipedia.org/wiki/ISO/IEC\\_8859-1](http://en.wikipedia.org/wiki/ISO/IEC_8859-1)
- The UTF-8 characters: <http://en.wikipedia.org/wiki/UTF-8>

# Table of Contents

- **Security Terminology**
- **XML lacks inherent security**
- **A valid XML document can still cause trouble** 
- **Security considerations when processing XML documents**
  - Expanding entities (XML bomb)
  - Reading inputs from external URLs and XInclude (external entities)
  - Attack surface
  - Pros and cons of spending the resources to check inputs
- **Miscellaneous security topics**
  - Poorly constructed regular expressions
  - Manual editing of XML documents (problems with copying and pasting)
  - Unconstrained markup and data
- **A brief *introduction* to XML security tools and capabilities**
  - Canonicalization
  - XML Digital Signatures (Integrity)
  - XML Encryption (Confidentiality)

You are here.

# Hidden Markup

---

# Five Reserved Characters

- The XML specification lists five reserved characters:  
**< > & " '**
- Those characters are *reserved* because they have a predefined meaning. For example, the '<' character means, “*Hey, this is the beginning of a start or end tag.*”

# Text with Escaped Characters

- If the reserved characters are to be used as plain text characters, they must break out of (escape) their predefined meaning.
- Suppose we want the content of a <Thermostat> element to be this text string:  
`if temperature < 32 then add heat`
- The '<' character is a reserved character. To use it as a plain text character in the string, it must be *escaped*.

# 3 Ways to Escape Reserved Characters

Reserved Character	XML Entity	Character Entity	CDATA Section
<	&lt;	&#x3C; or &#60;	<![CDATA[ < ]]>
>	&gt;	&#x3E; or &#62;	<![CDATA[ > ]]>
&	&amp;	&#x26; or &#38;	<![CDATA[ & ]]>
"	&quot;	&#x22; or &#34;	<![CDATA[" ]]>
'	&apos;	&#x27; or &#39;	<![CDATA[' ]]>



Hex codepoint

Decimal codepoint



# 3 Ways to Represent <Thermostat>

<b>XML Entity</b>	<b>&lt;Thermostat&gt; if temperature &amp;lt; 32 then add heat &lt;/Thermostat&gt;</b>
<b>Character Entity</b>	<b>&lt;Thermostat&gt; if temperature &amp;#x3C; 32 then add heat &lt;/Thermostat&gt;</b>
<b>CDATA Section</b>	<b>&lt;Thermostat&gt; &lt;![CDATA[ if temperature &lt; 32 then add heat ]]&gt; &lt;/Thermostat&gt;</b>

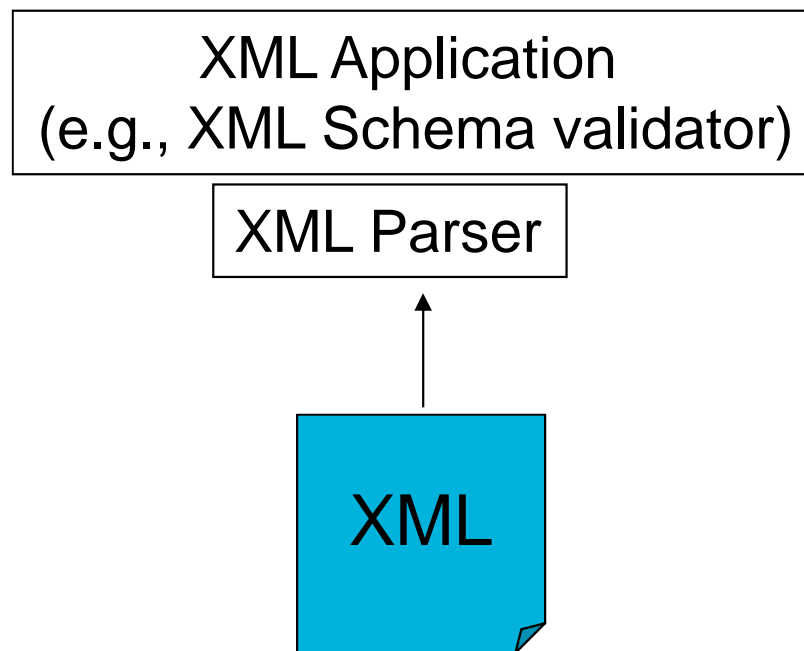
# Escaping is Good

---

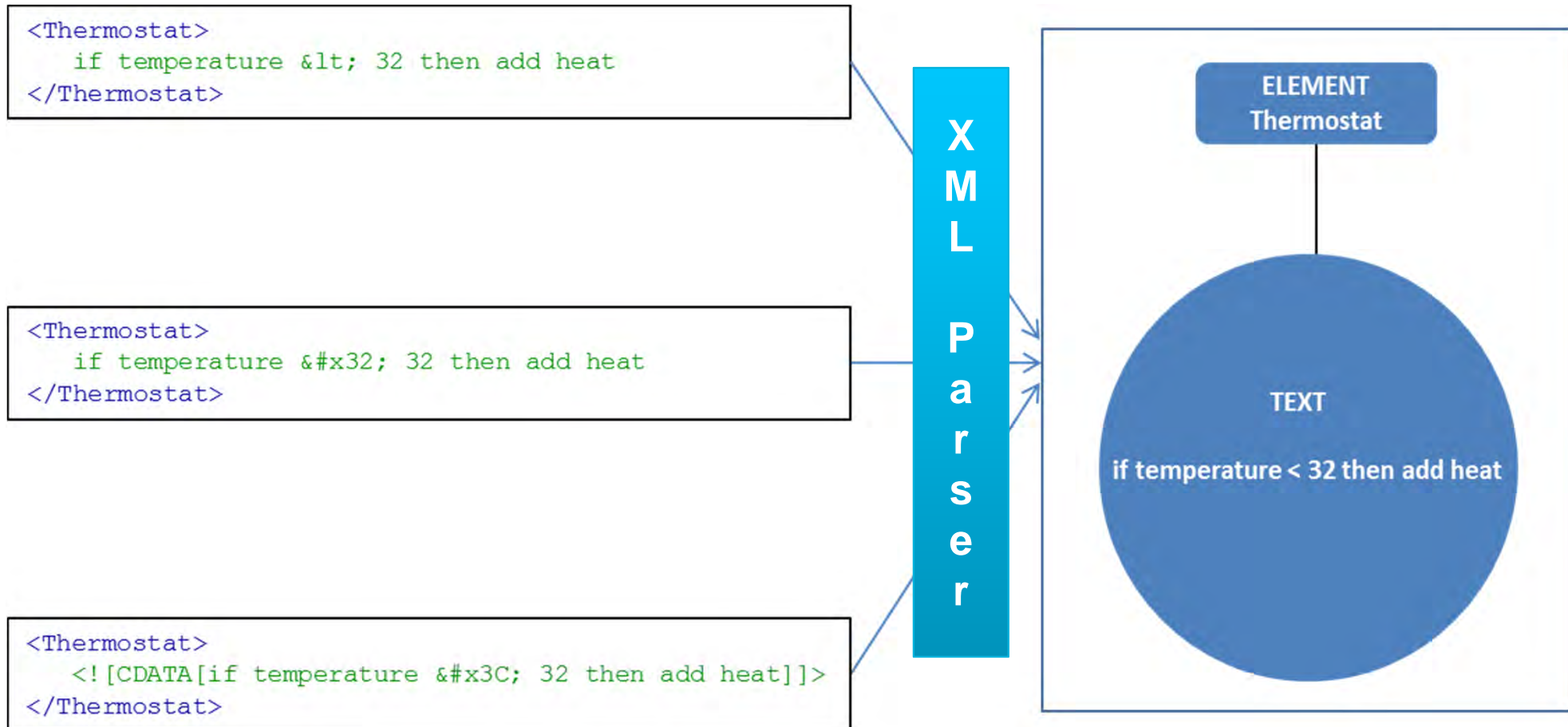
- XML entities, character entities, and CDATA sections are essential for creating text strings that contain reserved characters.
- They make it straightforward for XML processes and XML applications to process text strings without concern that the reserved characters will be acted upon in their predefined role.

# XML Parser

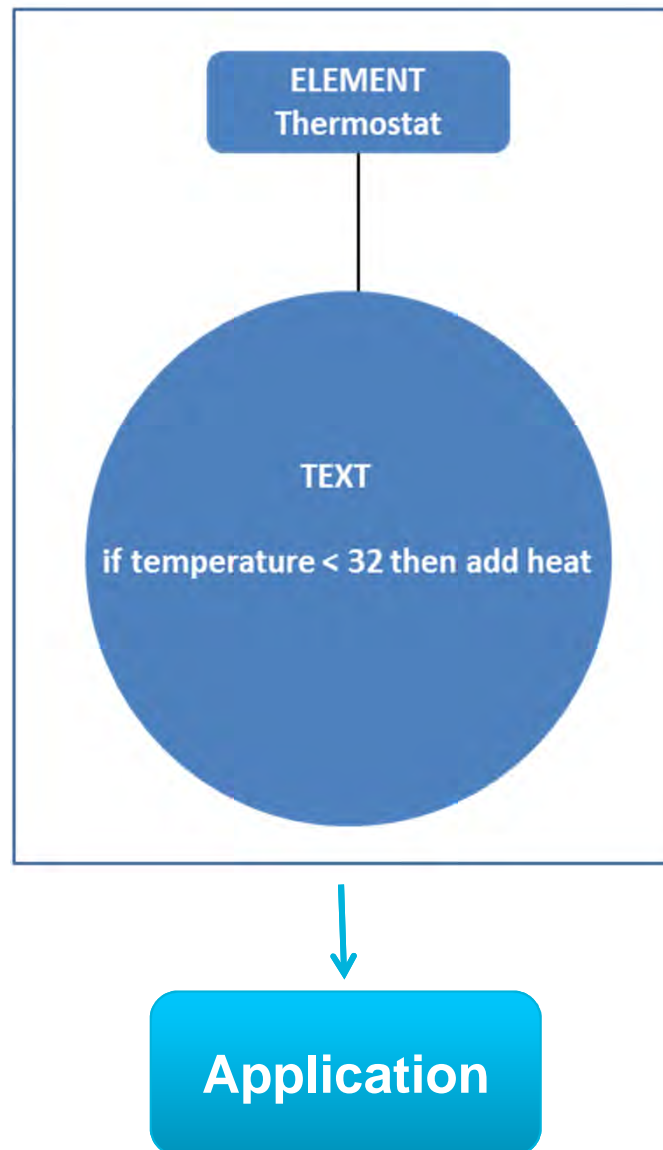
- An XML parser is a software program that parses XML documents, checking that they conform to the syntax rules of XML.
- XML applications build on top of XML parsers.



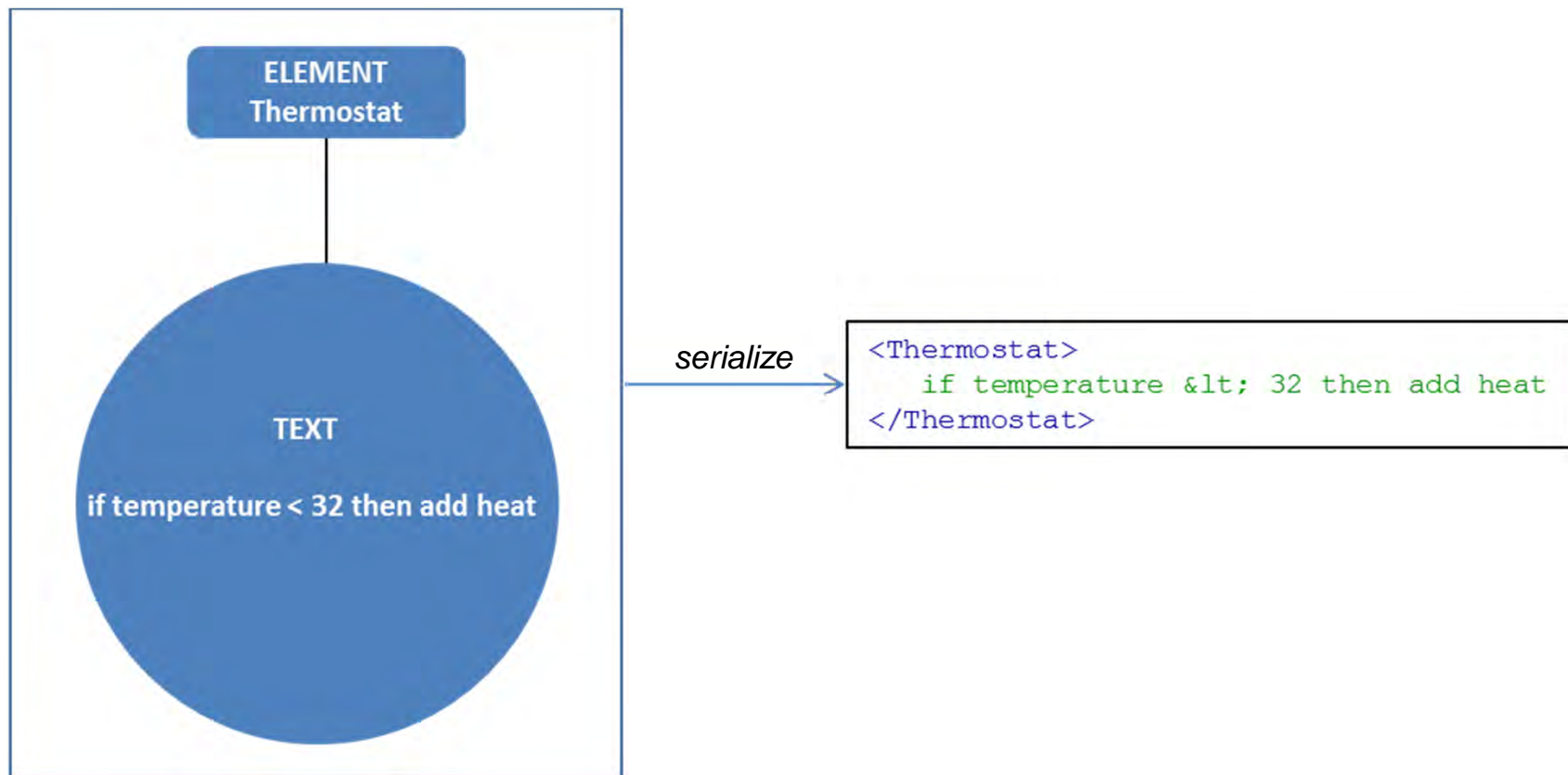
# All Map to the Same Parsed Representation



# Applications Operate on the Parsed Representation



# XML Serialization



When an XML application outputs a text node, the application can be designed to escape or not escape reserved characters. Well-behaved XML applications, such as XSLT processors, automatically escape any reserved characters in text nodes. This combination of outputting and escaping is called *XML serialization*.

# Caution

---

- ***There is no guarantee that every XML application will escape the reserved characters in text nodes when writing output.***
- **In fact, XSLT processors can be instructed to *not* escape reserved characters when writing output [1].**
- **Subsequent processing of the output would therefore attempt to parse the reserved characters as markup, causing unexpected or even malicious results.**

[1] disable-output-escaping="yes" See <http://www.w3.org/TR/xslt#disable-output-escaping>

# Could a Problem Occur if an Application Doesn't Escape Reserved Characters?

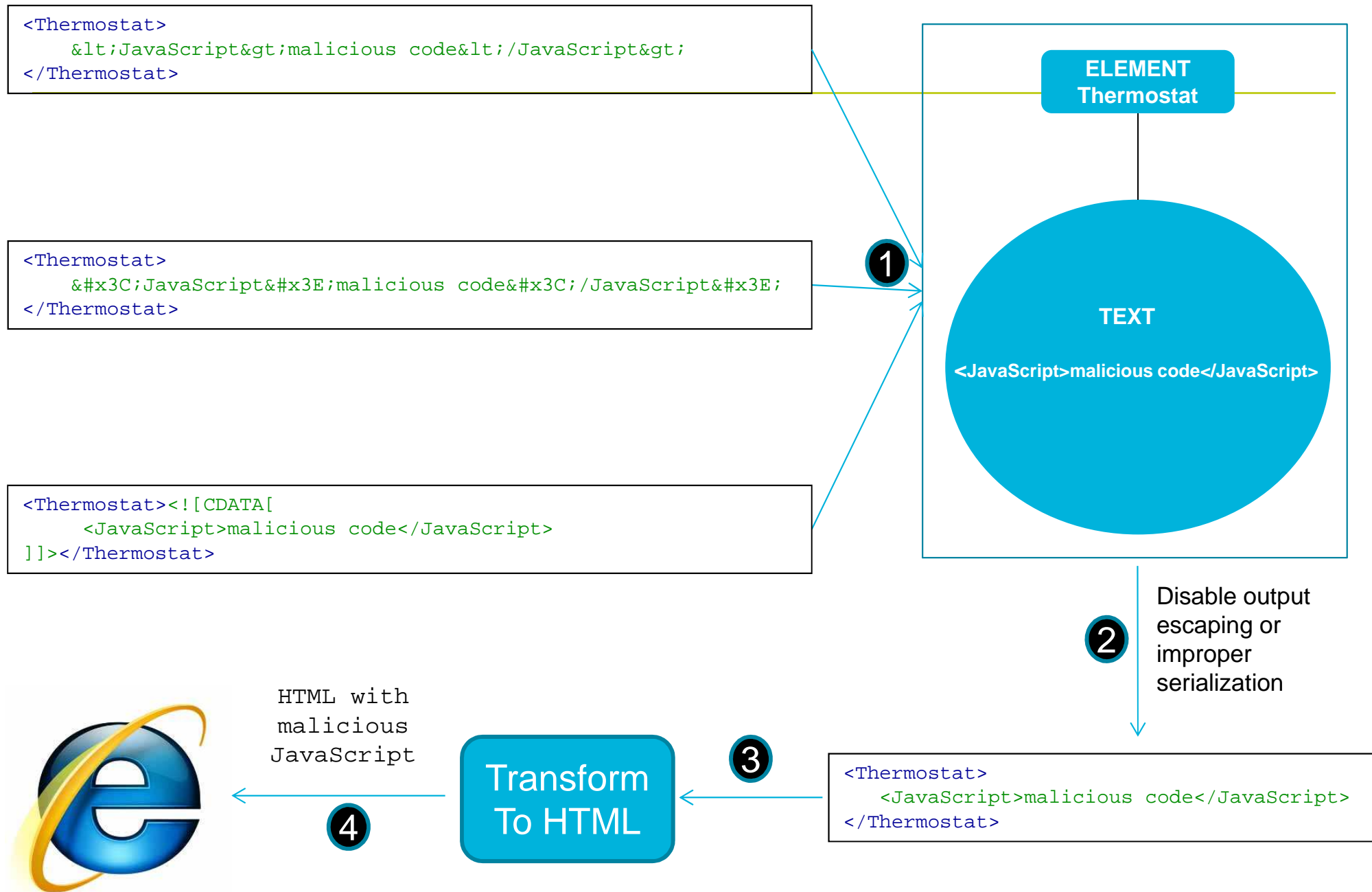
---

An attacker could exploit an XML application that does not escape reserved characters upon serializing.

The attacker injects malicious markup in the form of escaped text, hoping to bypass malware filtering.

Later, in the lifecycle of the XML document, the attacker intends for the XML application that doesn't serialize properly to convert the escaped text back into markup that is actually executable malicious code.





# Key Point

---

*If you are concerned about undesirable markup, know that it can be inserted into XML documents using any of the three escaping mechanisms.*

# Mitigating the Risk

- **Option 1: Use an XML Schema to ensure data element and attribute content does not contain reserved characters. Validate the XML document against the constrained XML Schema.**
- **Example of an XML Schema that prohibits the '<' character:**

```
<simpleType name="String-without-less-than">  
  <restriction base="string">  
    <minLength value="1" />  
    <maxLength value="100" />  
    <pattern value="[a-zA-Z 0-9]*" />  
  </restriction>  
</simpleType>
```

Allowed chars: a-z, A-Z, space, digits

# Mitigating the Risk

- **Option 2: Write a program that removes text containing XML entities, character entities, or CDATA sections:**

```
<Thermostat>  
    &lt;JavaScript>malicious code&lt;/JavaScript>  
</Thermostat>
```



*delete text*

```
<Thermostat></Thermostat>
```

# Disadvantage of Option 2

**Perfectly good data is removed:**

```
<Thermostat>if temperature < 32 then add heat</Thermostat>
```

*delete text*

```
<Thermostat></Thermostat>
```

# Mitigating the Risk

---

- **Option 3: Reject XML documents containing escaped reserved characters.**
- **This approach might be acceptable in certain high threat environments.**
- **Not acceptable in most normal situations.**

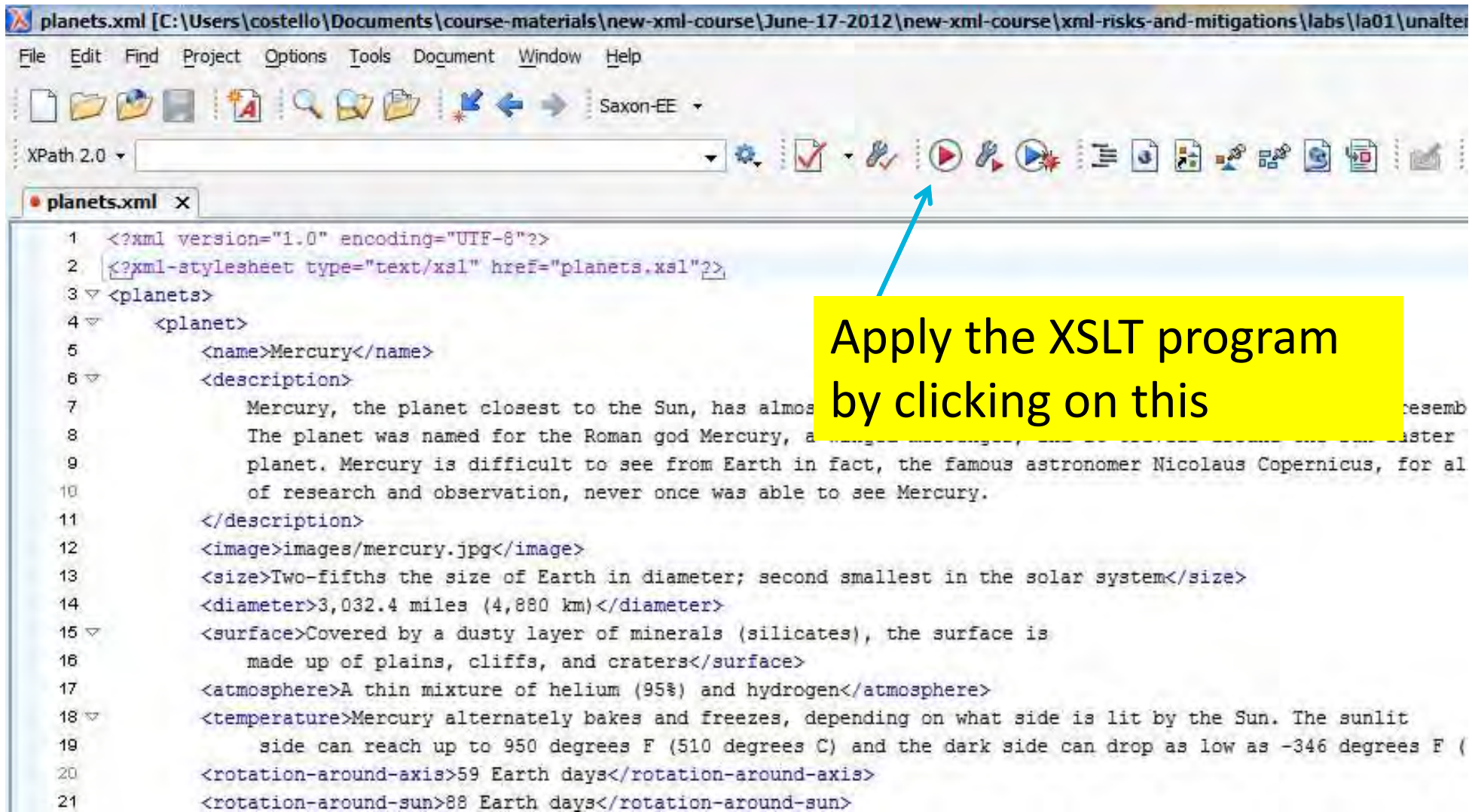
# Lab 1

---

- Using oXygen XML, open the planets.xml file in the lab01/unaltered folder.
- planets.xml contains data about the planets in our solar system.
- I created an XSLT program (planets.xsl) which extracts the data in planets.xml and embeds the data into an HTML document.

Continued →

# Lab 1 (cont.)



planets.xml [C:\Users\costello\Documents\course-materials\new-xml-course\June-17-2012\new-xml-course\xml-risks-and-mitigations\labs\la01\unalter

File Edit Find Project Options Tools Document Window Help

XPath 2.0

planets.xml

```

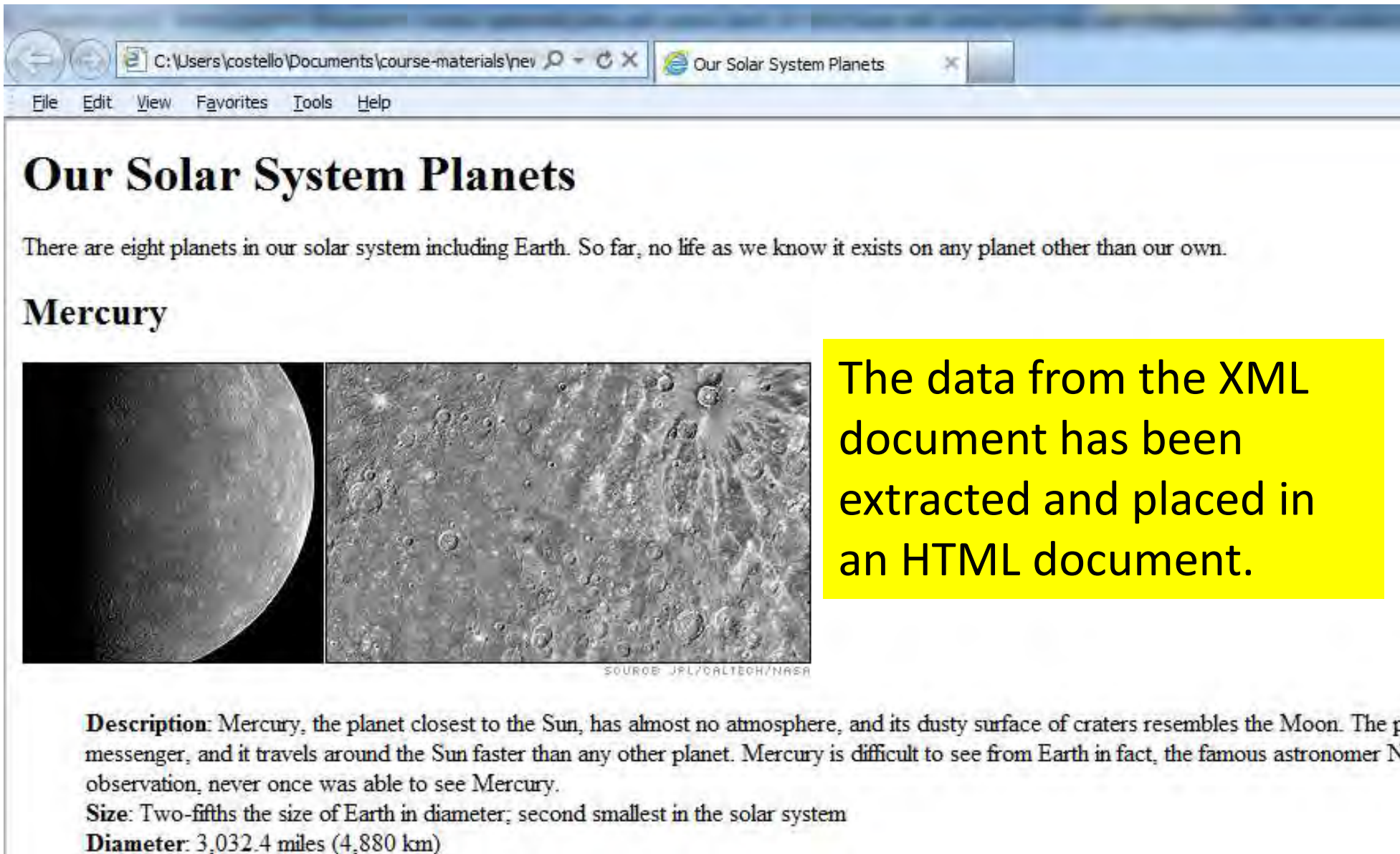
1 <?xml version="1.0" encoding="UTF-8"?>
2 <?xml-stylesheet type="text/xsl" href="planets.xsl"?>
3 <planets>
4   <planet>
5     <name>Mercury</name>
6     <description>
7       Mercury, the planet closest to the Sun, has almost no atmosphere. It resembles
8       The planet was named for the Roman god Mercury, a winged messenger, and is the smallest planet in the solar system.
9       planet. Mercury is difficult to see from Earth in fact, the famous astronomer Nicolaus Copernicus, for all
10      of research and observation, never once was able to see Mercury.
11    </description>
12    <image>images/mercury.jpg</image>
13    <size>Two-fifths the size of Earth in diameter; second smallest in the solar system</size>
14    <diameter>3,032.4 miles (4,880 km)</diameter>
15    <surface>Covered by a dusty layer of minerals (silicates), the surface is
16      made up of plains, cliffs, and craters</surface>
17    <atmosphere>A thin mixture of helium (95%) and hydrogen</atmosphere>
18    <temperature>Mercury alternately bakes and freezes, depending on what side is lit by the Sun. The sunlit
19      side can reach up to 950 degrees F (510 degrees C) and the dark side can drop as low as -346 degrees F (
20    <rotation-around-axis>59 Earth days</rotation-around-axis>
21    <rotation-around-sun>88 Earth days</rotation-around-sun>

```

Apply the XSLT program by clicking on this



# Lab 1 (cont.)




The screenshot shows a web browser window with the address bar displaying the file path: C:\Users\costello\Documents\course-materials\new. The browser has tabs for 'Our Solar System Planets' and another tab. The menu bar includes File, Edit, View, Favorites, Tools, and Help. The main content area has the title 'Our Solar System Planets' and a paragraph stating: 'There are eight planets in our solar system including Earth. So far, no life as we know it exists on any planet other than our own.' Below this is a section for 'Mercury' with two images: a crescent moon on the left and a cratered planetary surface on the right. A yellow text box on the right side of the page states: 'The data from the XML document has been extracted and placed in an HTML document.' Below the images is a description of Mercury, its size relative to Earth, and its diameter.

## Our Solar System Planets

There are eight planets in our solar system including Earth. So far, no life as we know it exists on any planet other than our own.

### Mercury



SOURCE: JPL/CALTECH/NASA

**Description:** Mercury, the planet closest to the Sun, has almost no atmosphere, and its dusty surface of craters resembles the Moon. The messenger, and it travels around the Sun faster than any other planet. Mercury is difficult to see from Earth in fact, the famous astronomer N observation, never once was able to see Mercury.

**Size:** Two-fifths the size of Earth in diameter; second smallest in the solar system

**Diameter:** 3,032.4 miles (4,880 km)

The data from the XML document has been extracted and placed in an HTML document.

# Lab 1 (cont.)

---

- Next, open planets.xml in the lab01/alterd folder.
- It is the same XML document, but an attacker has injected JavaScript in it.
- Can you spot the JavaScript?

Continued →

# Lab 1 (cont.)

planets.xml [C:\Users\costello\Documents\course-materials\new-xml-course\June-17-2012\new-xml-course\xml-risks-and-mitigations\labs\la01\altered-xml]

File Edit Find Project Options Tools Document Window Help

XPath 2.0

planets.xml x

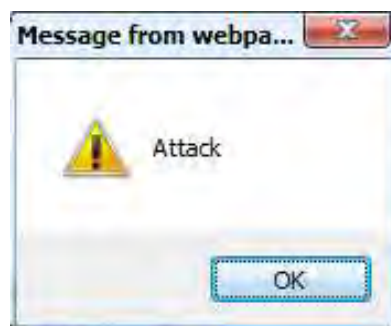
```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet type="text/xsl" href="planets.xsl"?>
3  <planets>
4    <planet>
5      <name>Mercury</name>
6      <description>
7        Mercury, the planet closest to the Sun, has almost no atmosphere, and its dusty surface of craters resemble
8        The planet was named for the Roman god Mercury, a winged messenger, and it travels around the Sun faster th
9        planet. Mercury is difficult to see from <script type="text/javascript">alert("Attack")</script> Earth
10       of research and observation, never once was able to see Mercury.
11     </description>
12     <image>images/mercury.jpg</image>
13     <size>Two-fifths the size of Earth in diameter; second smallest in the solar system</size>
14     <diameter>3,032.4 miles (4,880 km)</diameter>
15     <surface>Covered by a dusty layer of minerals (silicates), the surface is
16       made up of plains, cliffs, and craters</surface>
17     <atmosphere>A thin mixture of helium (95%) and hydrogen</atmosphere>
18     <temperature>Mercury alternately bakes and freezes, depending on what side is lit by the Sun. The sunlit
19       side can reach up to 950 degrees F (510 degrees C) and the dark side can drop as low as -346 degrees F (-2
20     <rotation-around-axis>59 Earth days</rotation-around-axis>
21     <rotation-around-sun>88 Earth days</rotation-around-sun>
22     <your-weight>If you weigh 100 pounds on Earth, you would weigh 38 pounds on Mercury.</your-weight>
23     <distance-from-earth>57 million miles, at the closest point in its orbit</distance-from-earth>
24     <mean-distance-from-sun>36 million miles (57.9 million km)</mean-distance-from-sun>

```

# Lab 1 (concluded)

- Apply the (same) XSLT program to planets.xml
- The browser will execute the JavaScript. (Click on “Allow blocked content at the bottom of the browser”)



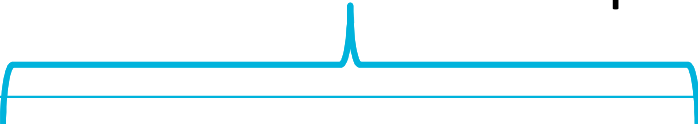
# Unused Namespaces

---

# Example of an Unused Namespace

- An unused namespace in an XML document is a namespace that is not associated with any element or attribute in the document.

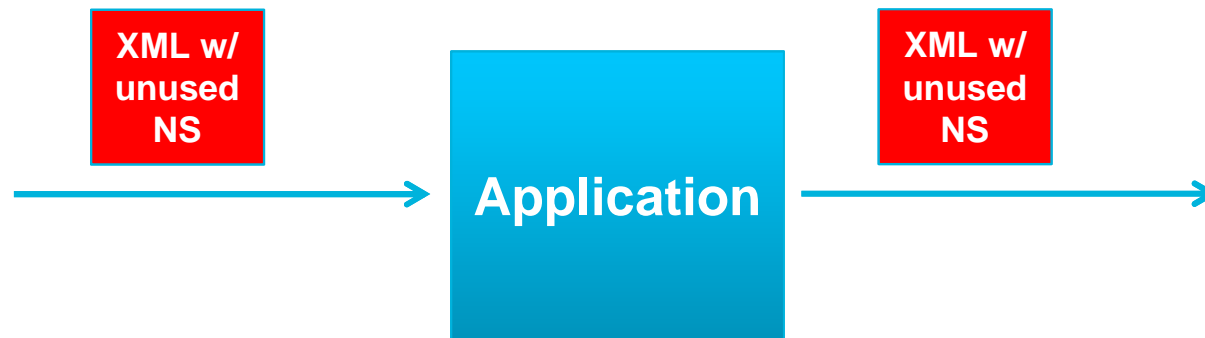
This namespace is unused



```
<Test xmlns:ns1="http://www.example.org">  
  <data>Value1</data>  
</Test>
```

# Applications Ignore Them

- The XML specification is silent on unused namespaces, so in general, applications—including validation programs—ignore them.





# Unconstrained

---

- **Unused namespaces can be of any length and can contain any legal XML characters.**
- **There can be duplicate namespaces.**



# Risk

---

- **Unused namespaces could be exploited by an adversary to embed unauthorized or malicious content that is not subject to schema validation.**
- **Even if unused namespaces do not contain malicious data, the presence of one or more very large namespaces could degrade performance or be used in a denial-of-service attack.**

# Mitigating the Risk

---

- **Delete all unused namespaces in the XML document.**

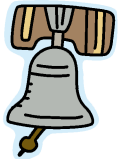
# Caution

---

- **Be sure the namespace really is unused before deleting it.**
- **Determining that a namespace is unused can be a bit tricky.**

# Where are Namespaces Used?

- An element name may be bound to a namespace:  
`<bk:Book xmlns:bk="...">`
- An attribute name may be bound to a namespace:  
`<Document icism:classification="..."  
xmlns:icism="...">`
- Data (name) may be bound to a namespace:  
`<fault>soap:client</fault>`



# QName

---

**QName = namespace-qualified name**

bk:Book (element name is a QName)

icism:classification (attribute name is a QName)

soap:client (data value is a QName)

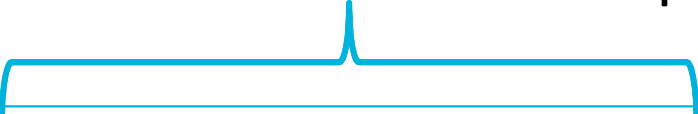
# QName Data Type

- XML Schema provides many data types: string, integer, boolean, etc.
- One data type it provides is: QName
- In your XML Schema you can declare, “*The value of this element must be a QName.*”

XML Schema: `<element name="fault" type="QName" />`

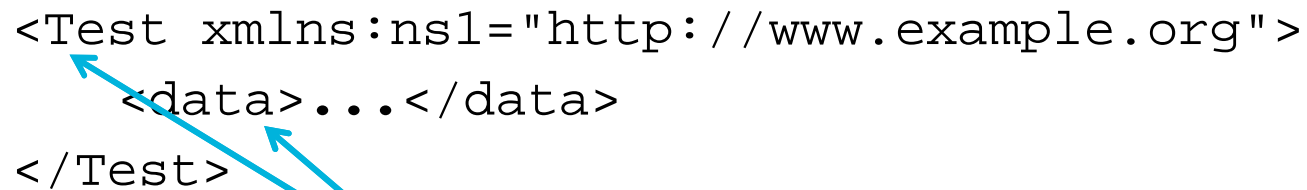
XML Instance: `<fault>soap:client</fault>`

Is this namespace unused?



```
<Test xmlns:ns1="http://www.example.org">  
  <data>...</data>  
</Test>
```


```
<Test xmlns:ns1="http://www.example.org">  
  <data>...</data>  
</Test>
```

The diagram shows an XML snippet enclosed in a light blue rectangular box. Two blue arrows originate from a point below the box and point to the 'xmlns:ns1' attribute in the opening tag and the '<data>' element.

The namespace isn't used by the elements,  
and there are no attributes.




```
<Test xmlns:ns1="http://www.example.org">  
  <data>...</data>  
</Test>
```



The namespace might be used in this data value.

Now what's your answer, is this namespace unused?

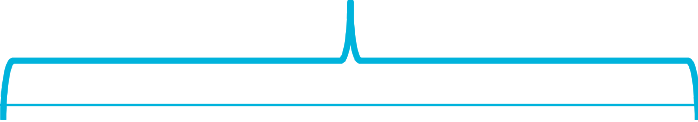


```
<Test xmlns:ns1="http://www.example.org">  
  <data>ns1:Foo</data>  
</Test>
```

Suppose the <data> element is declared like so:

```
<element name="data" type="string" />
```

Is this namespace unused?

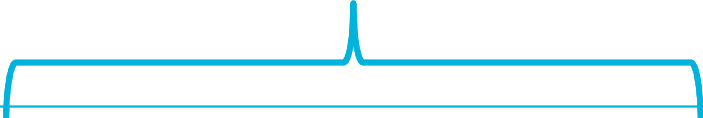


```
<Test xmlns:ns1="http://www.example.org">  
  <data>ns1:Foo</data>  
</Test>
```

Suppose the <data> element is declared like so:

```
<element name="data" type="QName" />
```

Is this namespace unused?



```
<Test xmlns:ns1="http://www.example.org">  
  <data>ns1:Foo</data>  
</Test>
```

# Answers

```
<element name="data" type="string" />
```

```
<Test xmlns:ns1="http://www.example.org">  
  <data>ns1:Foo</data>  
</Test>
```

**The namespace is unused when <data> is declared of type *string*.**

```
<element name="data" type="QName" />
```

```
<Test xmlns:ns1="http://www.example.org">  
  <data>ns1:Foo</data>  
</Test>
```

**The namespace is used when <data> is declared of type *QName*.**

# Lessons Learned

---

- **Determining if a namespace is unused is a bit tricky.**
- **Don't delete a namespace declaration unless you are sure it's unused.**
  - Refer to the XML Schema
- **Do delete all unused namespace declarations.**



# Canonical XML

- **“Canonicalization” means: put the XML into a standard format.**
  - Examples of things a canonicalizer tool does: delimit all attribute values with double quotes, convert all empty elements to paired start and end tags.
- **One thing a canonicalizer tool does is it removes unused namespaces.**
- **Caution:** some canonicalizers erroneously remove namespaces that are in fact being used within data values.

# The QName data type in XML Schema

---

XML Schema: `<element name="CountryCode" type="QName" />`

XML Instance:

```
<Document xmlns:iso="http://www.iso.org">  
    <CountryCode>iso:US</CountryCode>  
</Document>
```

The value of `<CountryCode>` is a QName,  
as required by the XML Schema

# Constrain the QName Value

XML Schema: `<element name="CountryCode" type="QName" />`

- That element declaration provides no constraints on the length of QName values nor on the set of characters used in values.
- To reduce risk, the data type should be constrained using facets.

## XML Schema:

```
<element name="CountryCode">
  <simpleType>
    <restriction base="QName">
      <maxLength value="10" />
      <pattern value="[a-zA-Z:]+" />
    </restriction>
  </simpleType>
</element>
```

## XML Instance:

```
<Document xmlns:iso="http://www.iso.org">
  <CountryCode>iso:US</CountryCode>
</Document>
```



The value of <CountryCode> is constrained to no more than 10 characters and the characters must be the letters of the alphabet and colon.

# Wrong!

- The maxLength facet is ignored for the QName data type!  
*1.3 if {primitive type definition} is QName or NOTATION, then any {value} is facet-valid.*  
<http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/#rf-maxLength>
- To constrain the length and characters of a QName value use the pattern facet:

```
<element name="CountryCode">
  <simpleType>
    <restriction base="QName">
      <pattern value="[a-zA-Z:]{1,10}" />
    </restriction>
  </simpleType>
</element>
```

1 to 10 characters

```
<Document xmlns:iso="http://www.iso.org">  
  <CountryCode>iso:US</CountryCode>  
</Document>
```

Okay, now we've constrained this

## Lab 2

- In the Lab 2 folder you will find **CountryCode.xsd**; open it in oXygen XML.
- It constrains the content of **<CountryCode>** to a QName that is no longer than 10 characters in length.

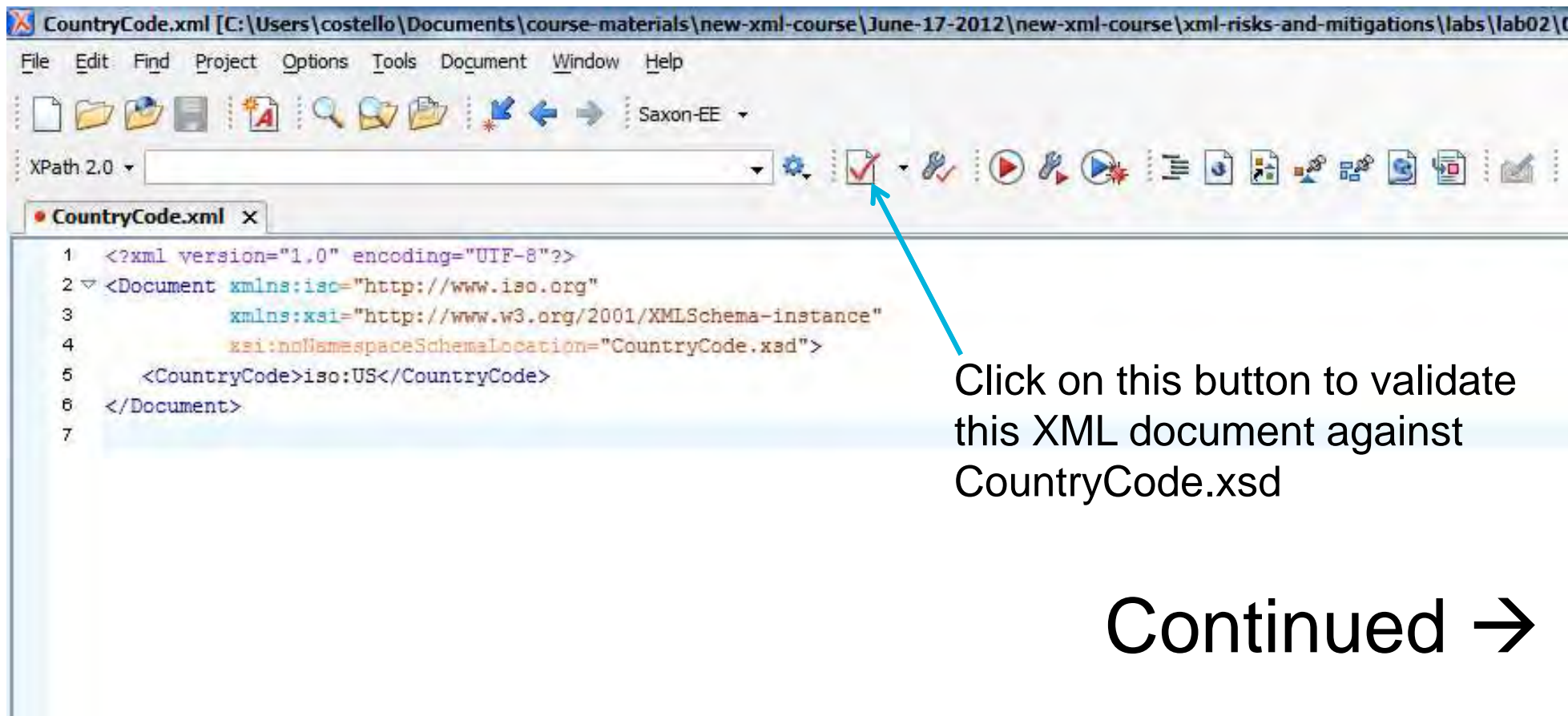
```
<element name="CountryCode">  
  <simpleType>  
    <restriction base="QName">  
      <pattern value="[a-zA-Z:]{1,10}" />  
    </restriction>  
  </simpleType>  
</element>
```

Continued →



## Lab 2 (cont.)

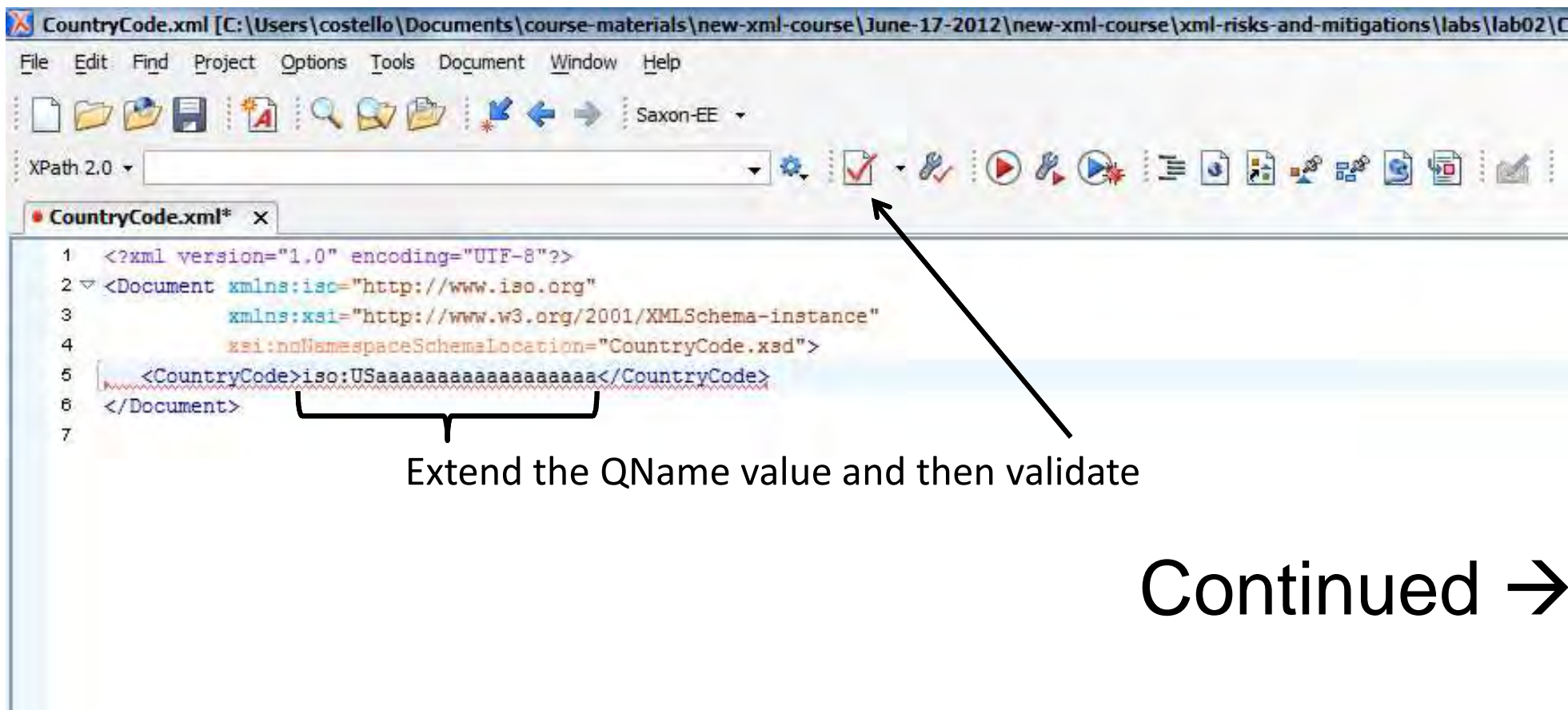
- Also in the Lab 2 folder is CountryCode.xml
- Open it in oXygen XML and validate it




Continued →

## Lab 2 (cont.)

- Extend the QName value so that it is longer than 10 characters in length and then validate



- 
- CountryCode.xml [C:\Users\costello\Documents\course-materials\new-xml-course\June-17-2012\new-xml-course\xml-risks-and-mitigations\labs\lab02\CountryCode.xml]
- File Edit Find Project Options Tools Document Window Help
- XPath 2.0
- CountryCode.xml\*
- ```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Document xmlns:iso="http://www.iso.orgxx"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:noNamespaceSchemaLocation="CountryCode.xsd">
5     <CountryCode>iso:US</CountryCode>
6 </Document>
7
```
- Set the QName value to iso:US, make the iso namespace long, and then validate

## What did you learn?

These are totally unconstrained



```
<Document xmlns:iso="http://www.iso.org">  
  <CountryCode>iso:US</CountryCode>  
</Document>
```

# Sensitive Data in the Namespace

---

```
<Document xmlns:iso="This is a long string containing sensitive data">
```

Passing unconstrained and unvalidated data into a process, even via a namespace, provides a way to attack the process.

# Sensitive Data in the Prefix

```
<Document xmlns:This_is_a_long_prefix_containing_sensitive_data="...">
```

In general, applications, including validation programs, ignore namespace prefixes.

Thus, namespace prefixes can conceivably be exploited to transmit sensitive information.

# How to Constrain the Namespace

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:iso="http://www.iso.org">
```

```
  <element name="Document">
```

```
    <complexType>
```

```
      <sequence>
```

```
        <element name="CountryCode">
```

```
          <simpleType>
```

```
            <restriction base="QName">
```

```
              <enumeration value="iso:DE"/>
```

```
              <enumeration value="iso:FR"/>
```

```
              <enumeration value="iso:US"/>
```

```
            </restriction>
```

```
          </simpleType>
```

```
        </element>
```

```
      </sequence>
```

```
    </complexType>
```

```
  </element>
```

```
</schema>
```

**Declare the namespace here, in the schema, at design time**

**Enumerate the QName values**

This is the only namespace that can be used



```
<Document xmlns:iso="http://www.iso.org">  
  <CountryCode>iso:US</CountryCode>  
</Document>
```



The only country codes allowed are: DE, FR, US



The namespace prefix is still unconstrained



```
<Document xmlns:iso="http://www.iso.org">  
  <CountryCode>iso:US</CountryCode>  
</Document>
```

# Prefix Rewrite

---

- **Check that namespace prefixes do not contain sensitive information.**
- **Where appropriate, rewrite namespace prefixes to neutral prefixes such as ns1, ns2, ns3.**
- **Namespaces can be rewritten using a simple search and replace tool or program.**

# Careful when Rewriting

- Rewriting what appears to be namespace prefixes in element or attribute values must be undertaken with care.
- For example, can we rewrite **ex1** in the following XML document?

```
<test xmlns:ex1="http://www.example.com">  
  <item>ex1:Value2</item>  
</test>
```

```
<test xmlns:ex1="http://www.example.com">  
  <item>ex1:Value2</item>  
</test>
```

The **ex1** on the test element is a namespace prefix. However, the **ex1** in the value of item might or might not be a prefix. If there is an XML Schema for the document that specifies the value of item is QName then **ex1** is a prefix and it can be rewritten in both places. If the value of item is not QName, then **ex1** in the value of item must not be changed.

# Lessons Learned

---

- **Valid XML documents can still cause trouble:**
  - There may be hidden markup
  - There may be unused namespaces
  - Unconstrained namespaces and namespace prefixes may be exploited

# Table of Contents

- **Security Terminology**
- **XML lacks inherent security**
- **A valid XML document can still cause trouble**
- **Security considerations when processing XML documents**
  - Reading inputs from external URLs and XInclude (external entities)
  - Attack surface
  - Pros and cons of spending the resources to check inputs
- **Miscellaneous security topics**
  - Expanding entities
  - Poorly constructed regular expressions
  - Manual editing of XML documents (problems with copying and pasting)
  - Unconstrained markup and data
- **A brief *introduction* to XML security tools and capabilities**
  - Canonicalization
  - XML Digital Signatures (Integrity)
  - XML Encryption (Confidentiality)



You  
are  
here

# XML Bombs

---

# Definition of “XML Bomb”

---

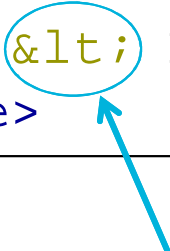
A block of XML that is both well-formed and valid according to the rules of an XML schema but which crashes or hangs a program when that program attempts to parse it.



# Definition of “XML Entity”

An XML entity is an abbreviation.

```
<?xml version="1.0" encoding="UTF-8"?>  
<Example>  
    if A &lt; B then ...  
</Example>
```



“1t” is a built-in entity. By “built-in” we mean that every XML parser recognizes this entity and knows that the replacement text for “1t” is “<”

```
<?xml version="1.0" encoding="UTF-8"?>
<Example>
    if A &lt; B then ...
</Example>
```

XML Parser

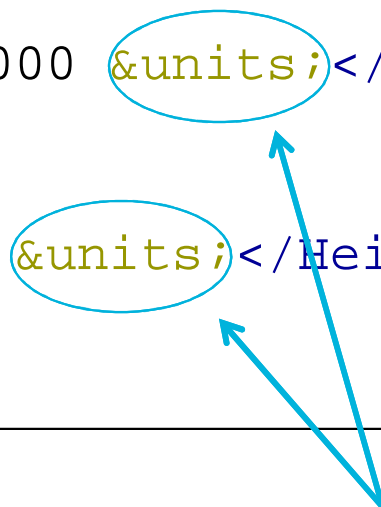
```
<?xml version="1.0" encoding="UTF-8"?>
<Example>
    if A < B then ...
</Example>
```

The XML parser has “resolved” the entity reference. That is, it replaced the entity reference with its replacement text.

XML Application  
(XSLT  
processor, XML  
Schema  
validator, etc.)

# You can Create your Own Entities

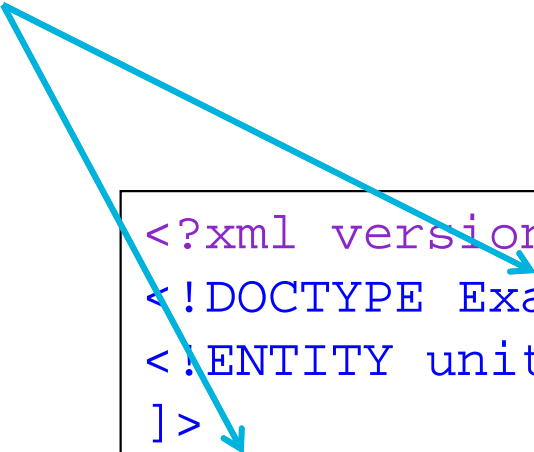
```
<?xml version="1.0" encoding="UTF-8"?>
<Example>
  <Airplane>
    <Altitude>12,000 &units;</Altitude>
  </Airplane>
  <Mountain>
    <Height>5,000 &units;</Height>
  </Mountain>
</Example>
```



User-defined entities are entities that you (the user) created. Thus, you must define it. Define an entity with an ENTITY declaration.

# ENTITIES are Declared in DOCTYPE


Must match



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Example [
  <!ENTITY units "feet">
]>
<Example>
  <Airplane>
    <Altitude>12,000 &units;</Altitude>
  </Airplane>
  <Mountain>
    <Height>5,000 &units;</Height>
  </Mountain>
</Example>
```

# Here's how to Declare an ENTITY

ENTITY  
declaration



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Example [
  <!ENTITY units "feet">
]>
<Example>
  <Airplane>
    <Altitude>12,000 &units;</Altitude>
  </Airplane>
  <Mountain>
    <Height>5,000 &units;</Height>
  </Mountain>
</Example>
```

# ENTITIES are Useful

- As the previous slide shows, you can avoid repeating yourself by declaring an entity once and reusing it multiple times.
- Here's another example that illustrates the value of entities: create a "disclaimer" entity:

```
<?xml version="1.0"?>
<!DOCTYPE letter [
    <!ENTITY disclaimer "DISCLAIMER GOES HERE">
]>
<letter>
    <salutation>
        Dear <customerName>Valued Customer</customerName>,
    </salutation>
    <body>
        ...
    </body>
    <closing>Sincerely,</closing>
    <signature>Customer Service Employee 334992</signature>
    &disclaimer;
</letter>
```

Example is from:  
<http://www.ibm.com/developerworks/xml/library/x-tipgentity/index.html>

# ENTITY Declaration

---

```
<!ENTITY name "replacement text">
```



The name of your ENTITY

The string that will be used by the XML  
parser to replace your ENTITY reference

# ENTITIES can be Used in any XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE schema [
<!ENTITY US_ASCII_CHARACTER "&#9;&#10;&#13;&#32;-&#127;">
]>
<schema xmlns="http://www.w3.org/2001/XMLSchema">

  <element name="Description">
    <simpleType>
      <restriction base="string">
        <pattern value=" [&US_ASCII_CHARACTER; ]*" />
      </restriction>
    </simpleType>
  </element>

</schema>
```

An ENTITY is used in this XML Schema pattern facet regular expression.



# ENTITIES can Use Other ENTITIES

The replacement text for date is the replacement text for month, day, and year.



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Example [
  <!ENTITY day "18">
  <!ENTITY month "February">
  <!ENTITY year "2013">
  <!ENTITY date "&month; &day;, &year;">
]>
<Example>
  <Today's-Date>&date;</Today's-Date>
</Example>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Example [
  <!ENTITY day "18">
  <!ENTITY month "February">
  <!ENTITY year "2013">
  <!ENTITY date "&month; &day;, &year;">
]>
<Example>
  <Todays-Date>&date;</Todays-Date>
</Example>
```



XML Parser

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Example [
  <!ENTITY day "18">
  <!ENTITY month "February">
  <!ENTITY year "2013">
  <!ENTITY date "&month; &day;, &year;">
]>
<Example>
  <Todays-Date>February 18, 2013</Todays-Date>
</Example>
```

# Lab 3

- In the lab03 folder you will find: Todays-datetime.xml
- Open it in oXygen XML and observe that it has some user-defined entities.
- Drag and drop it into a browser. The browser will display the resolved entities.
- Create another entity and call it datetime2. Set its replacement text to be two datetime entities. Then, within the <Todays-Datetime> element reference your datetime2 entity. Drag and drop the XML document into a browser.

# Alert!

---

The ability of an ENTITY to use other ENTITIES can result in exponential string length.

# Expanding ENTITY

```
<?xml version="1.0"
encoding="UTF-8"?>
<!DOCTYPE Example [
<!ENTITY ha1 'ha'>
<!ENTITY ha2 '&ha1;&ha1;'>
<!ENTITY ha3 '&ha2;&ha2;'>
<!ENTITY ha4 '&ha3;&ha3;'>
<!ENTITY ha5 '&ha4;&ha4;'>
]>
<Example>
    &ha5;
</Example>
```

ENTITY	Replacement text
ha1	ha
ha2	haha
ha3	hahahaha
ha4	hahahahahahaha
ha5	hahahahahahahahahahahahahah aha

The length of the replacement text is growing  
at an *exponential rate*.

# Billion Laughs Attack

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Example [
  <!ENTITY ha1 'ha'>
  <!ENTITY ha2 '&ha1;&ha1;'>
  <!ENTITY ha3 '&ha2;&ha2;'>
  <!ENTITY ha4 '&ha3;&ha3;'>
  <!ENTITY ha5 '&ha4;&ha4;'>
  ...
  <!ENTITY ha128 '&ha127;&ha127;'>
]>
<Example>
  &ha128;
</Example>
```

The length of the replacement text for **ha128** is a string of length:  $2^{128} =$   
6,800,000,000,000,000,000,000,000,000,000,000,000,000,000

---

The Billion Laughs Attack causes XML Parsers to consume lots of memory and CPU cycles as it resolves the ENTITY references.

The net effect is a Denial-of-Service attack or even a system crash.



# Other Names

---

- **The Billion Laughs Attack is also called:**
  - An XML bomb
  - Exponential entity expansion attack

# Quadratic Blowup Attack

**Instead of defining multiple expanding ENTITIES, create one long ENTITY and reference it many times:**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Example [
  <!ENTITY A "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ...">
]>
<Example>
  &A;&A;&A;&A;&A;&A;&A;&A;&A; ...
</Example>
```

# Internal vs. External ENTITIES

- In the previous examples we created an entity and specified its replacement text. That's called an *internal entity* declaration.
- You can also create an entity and provide a URL to its replacement text. That's called an *external entity* declaration:

<!ENTITY *name* **SYSTEM** "url">



Keyword, indicates that what follows is a URL.

# Google Web Service

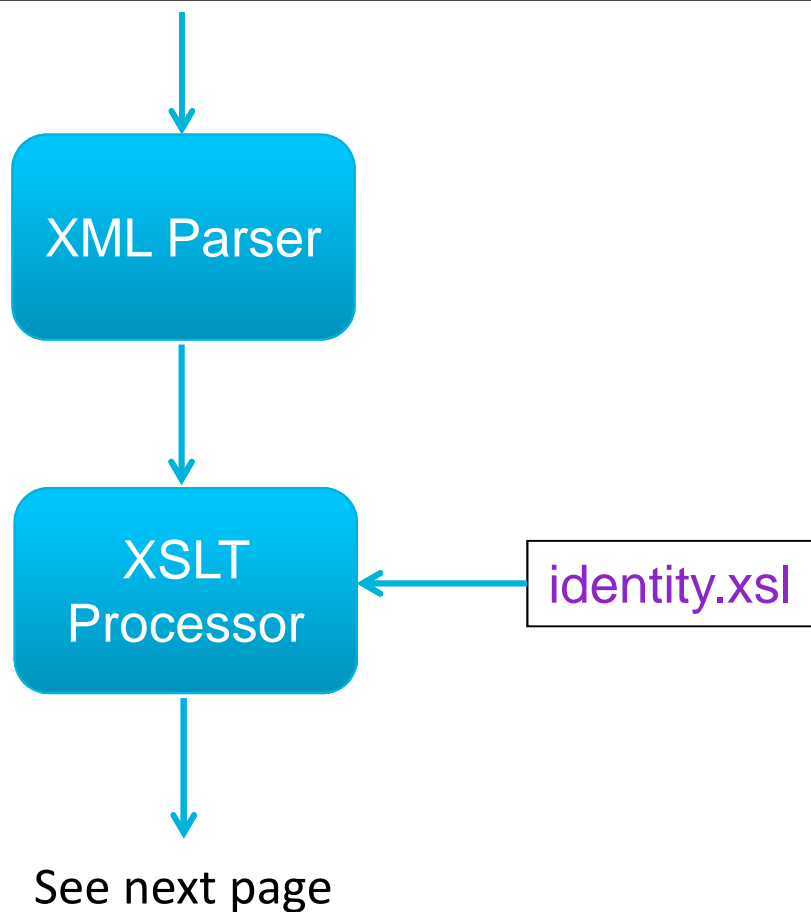
---

- **Google has a web service that you can invoke to get information about a city.**
- **Here's how to get info about Boston:**

<http://maps.googleapis.com/maps/api/geocode/xml?address=Boston,%20MA&sensor=false>

- **We can use external entities to pull into our XML documents data from web services (such as Google's web service)**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Example [
<!ENTITY Boston SYSTEM "http://maps.googleapis.com/maps/api/geocode/xml?address=Boston,%20MA&sensor=false">
]>
<Example>
    &Boston;
</Example>
```





```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<Example>
```

```
  <GeocodeResponse>
```

```
    <status>OK</status>
```

```
    <result>
```

```
      <type>locality</type>
```

```
      <type>political</type>
```

```
      <formatted_address>Boston, MA, USA</formatted_address>
```

```
      <address_component>
```

```
        <long_name>Boston</long_name>
```

```
        <short_name>Boston</short_name>
```

```
        <type>locality</type>
```

```
        <type>political</type>
```

```
      </address_component>
```

```
    ...
```

```
  </GeocodeResponse>
```

```
</Example>
```

Run the example: in the examples/expanding-entities folder  
apply identity.xsl to External-ENTITY.xml

# Know What is Addressed by the URL

`<!ENTITY name SYSTEM "url">`



If the resource at this URL is under the control of an attacker, then three types of attacks are possible:

1. The resource never returns, stalling your application.
2. A huge amount of data is returned.
3. Malicious code is returned.

# XML External Entity Attack (XXE)

---

- **XXE (Xml eXternal Entity) attack is an attack on an application via XML that brings in external input.**



# Mitigating the Risk

- **Configure your XML parser so that:**
  - Entity expansion is turned off (*“Hey parser, don’t resolve any ENTITIES.”*)
  - The number of entity expansions is set to a max limit (*“Hey parser, stop evaluating ENTITIES once N expansions have been done.”*)
  - The number of characters that entities can expand to is set to a max limit (*“Hey parser, stop expanding an ENTITY once you’ve output N characters.”*)
- **All, some, or none of these options may be available to you depending on what XML parsing API you are using.**

# Sample Code

```
XmlReaderSettings settings = new XmlReaderSettings();  
settings.ProhibitDtd = false;  
settings.MaxCharactersFromEntities = 1024;  
XmlReader reader = XmlReader.Create(stream, settings);
```

```
XercesDOMParser parser;  
parser.setValidationScheme(XercesDOMParser::Val_Always);  
parser.setDoNamespaces(true);  
parser.setDoSchema(true);  
SecurityManager sm;  
sm.setEntityExpansionLimit(100);  
parser.setSecurityManager(&sm);
```

# References

---

- **This blog nicely explains the attacks and contains code for configuring Apache Xerces to mitigate the risk of these attacks:**

<http://cytinus.wordpress.com/2011/07/26/37/>

- **This web site explains external entity attacks:**

<http://www.securityfocus.com/archive/1/297714>

# References

---

- **This StackOverflow post has info about available web services:**

<http://stackoverflow.com/questions/55901/web-service-current-time-zone-for-a-city>

- **Here's the original reporting of the Billion Laughs Attack:**

<http://www.securityfocus.com/archive/1/303509/2002-12-13/2002-12-19/0>

# References

---

- This is a good article: “XML Denial of Service Attacks and Defenses”:

<http://msdn.microsoft.com/en-us/magazine/ee335713.aspx>

# Attack Surface (External Inputs)

---

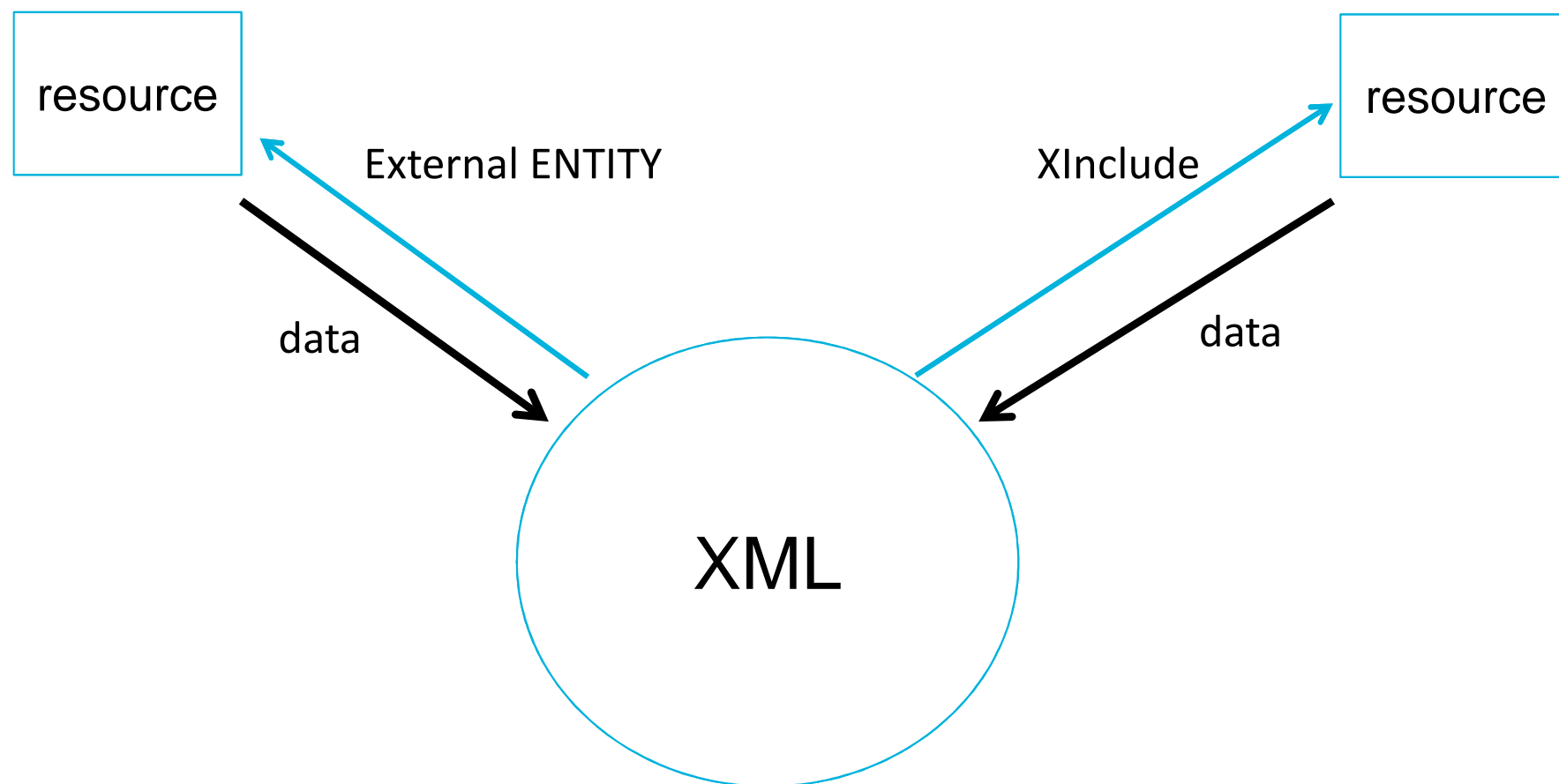
# Attack Surface

---

- The term “attack surface” refers to the ways that an adversary can get inside.
- Your home has an attack surface. It includes the windows and doors. But it also includes the water, gas, communication, and electrical connections.
- What are the ways that sensitive or malicious content could get inside your XML document?

## What is XML's attack surface?

# XML's Attack Surface





# XML's Attack Surface

- **There are two avenues for outside content to get inside your XML document:**
  1. XML/text obtained from external entities.
  2. XML/text obtained using XInclude elements.
- **The content may come from either local or network resources.**
- **The content is added by the XML parser as it resolves the external entity references and the <xi:include> elements.**

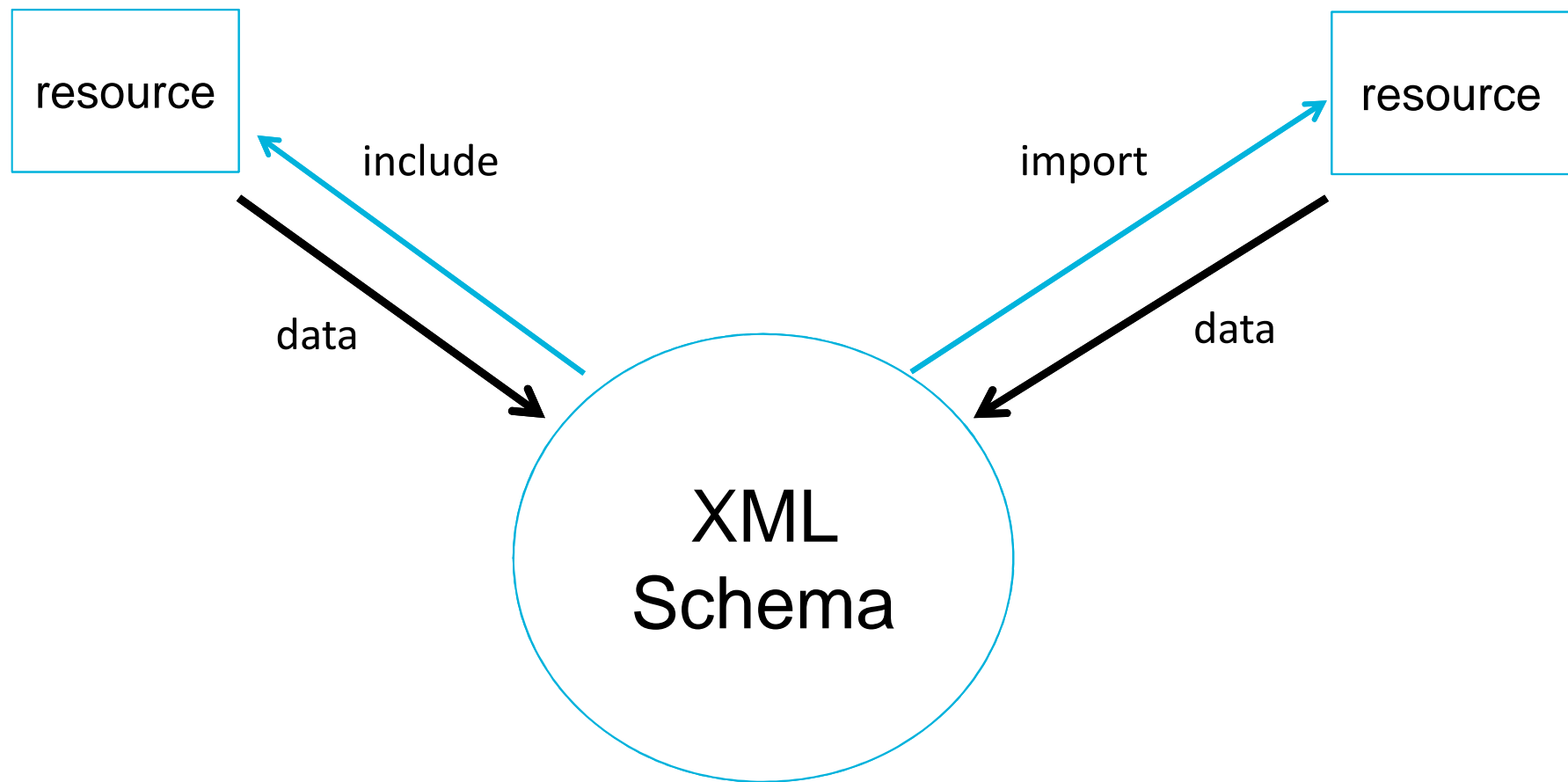
You need to be aware of some security implications of external entities and XInclude. Consider the following:

```
<?xml version="1.0"?>
<!DOCTYPE BookStore [
<!ENTITY systemfile SYSTEM "/etc/passwd">
]>
<BookStore>
    &systemfile;
</BookStore>
```

If a system allows an external client to provide XML or XSLT stylesheets to it and the system will return the results of processing that XML or stylesheet, then the system shouldn't allow external entities or XInclude unless they are limited to some 'safe' sandbox.

Marc Hadley

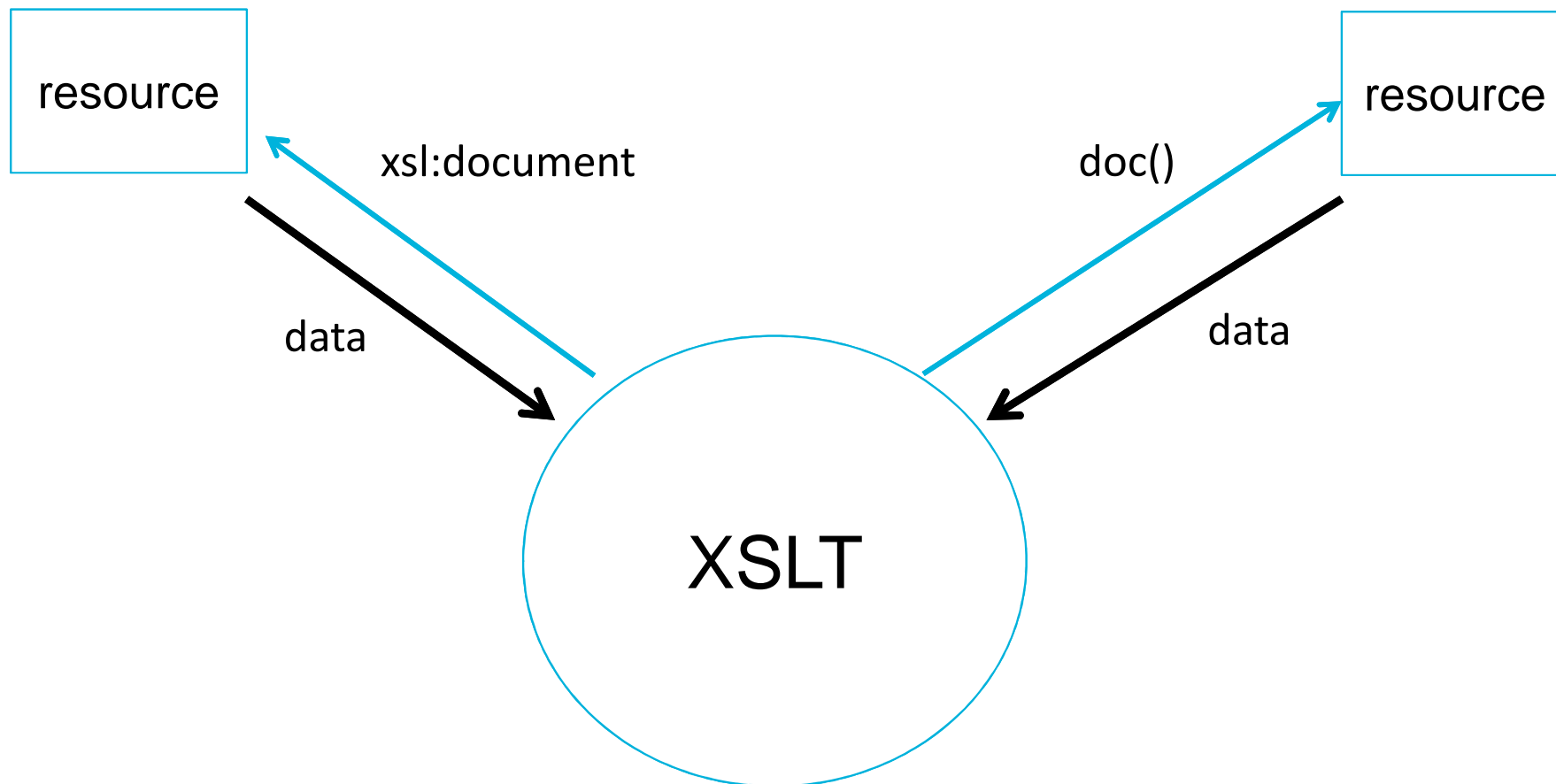
# XML Schema's Attack Surface



# XML Schema's Attack Surface

- **An XML Schema document is an XML document, so it has all the attack avenues of XML plus its own unique avenues:**
  - XML obtained from the `xs:import` element.
  - XML obtained from the `xs:include` element.
  - XML obtained from the `xs:redefine` element.
  - XML obtained from the `xs:override` element.
  - XML obtained from the XPath `doc()` function.
  - XML obtained from the XPath `collection()` function.
- **Content may come from either local or network resources.**
- **The content is added by the XML Schema processor as it executes the XML Schema.**

# XSLT's Attack Surface



# XSLT's Attack Surface

- **An XSLT document is an XML document, so it has all the attack avenues of XML plus its own unique avenues:**
  - The input XML document that is specified when the XSLT processor is invoked.
  - XML obtained from the document() function.
  - XML/text obtained from extension functions/elements.
  - XML obtained from the xsl:import and xsl:include elements.
  - XML/text obtained from global xsl:param elements.
  - Text obtained from the system-property() function.
  - XML obtained from the XPath doc() and collection() functions.
- **Content may come from either local or network resources.**
- **The content is added by the XSLT processor as it executes the XSLT program.**

# Mitigating the Risk

---

- **Check the location (URL) of all external inputs to ensure they are appropriate**
  - Validate each URL against a whitelist of approved external input locations
- **XML applications should not be able to write to any arbitrary file, whether on the local hard disk (such as a password file) or somewhere across the network**
  - Validate the output URL against a whitelist of approved locations

# References

---

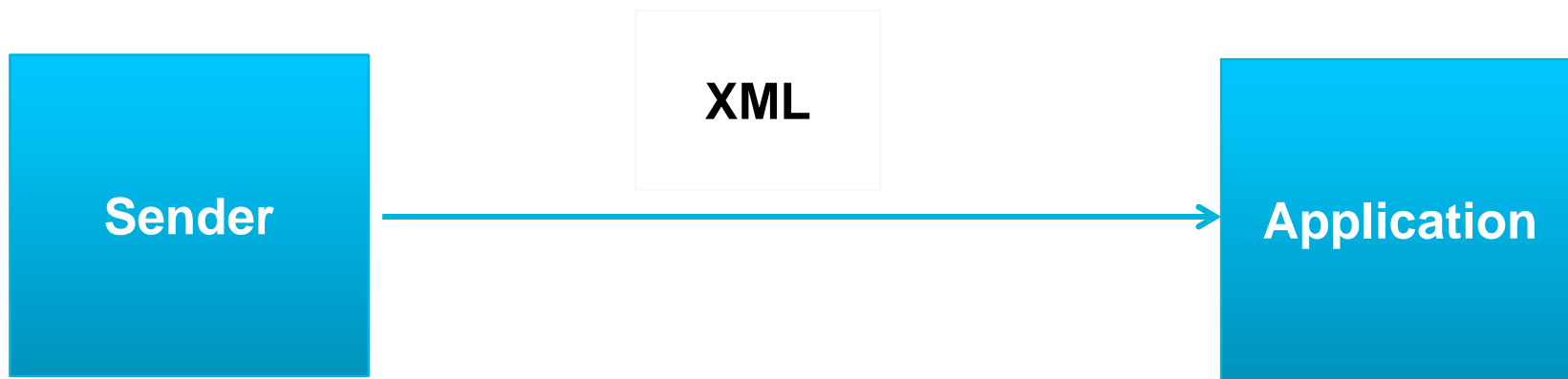
- **For a description of XInclude see:**  
<http://blogs.gnome.org/shaunm/2011/07/21/understanding-xinclude/>



# Should you check inbound XML?

---

# Web Service



Should the Application check the XML before processing it?

A web service needs to check inbound XML documents (for the presence of malware and other bad things) only if the documents come from unknown sources. So, online web services should have a strong set of checks since the data can come from anywhere. For trusted sources, however, there is no need to check inbound XML documents. So, a web service running on, say, SIPRNet doesn't need to check inbound XML documents.



The days of trusting inbound XML documents are over. All inbound XML documents must be checked, even if they come from a so-called trusted source. "Treat all systems as compromised. There's no such thing as 'secure' anymore." [Deborah Plunkett, NSA Information Assurance Directorate]. Every data exchange has been, will be, or could be tampered with by attackers. So, even if one is exchanging data exclusively on the SIPRNet, one must assume that the data may be compromised and therefore must be checked (for the presence of malware and other bad things).



# Wisdom from the infosec list

---

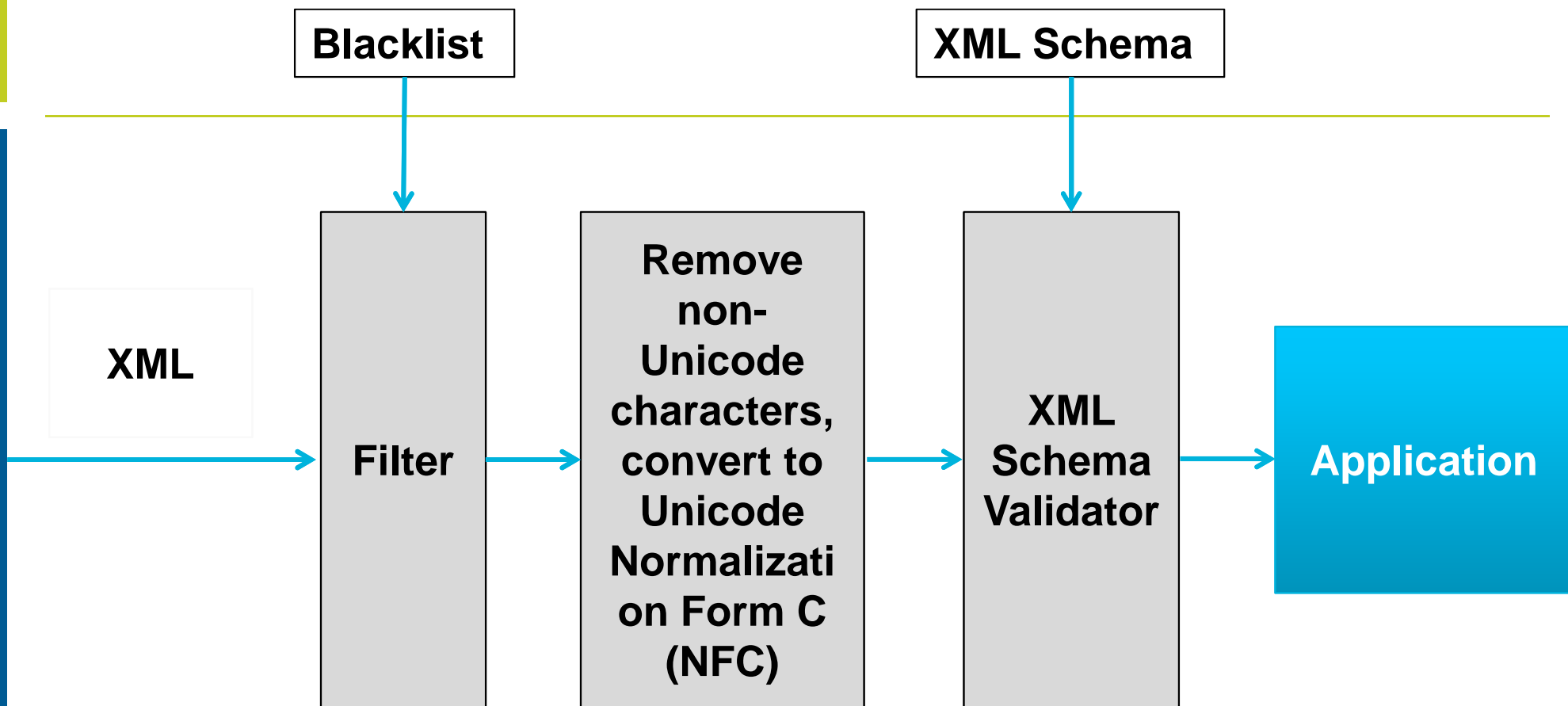
**Trust No One**

**Trust but Verify**

# How much Testing?


---

- **Cost/benefit analysis needed**
  - What is the potential impact of bad input in terms of cost and lives?
  - What is the cost of testing the input?
- **If the application is, say, a currency converter then perhaps little testing is needed**
- **If the application is determining an aircraft's flight path based on XML inputs from weather sensors, then lots of input testing is warranted**



Some applications determine whether or not to reject an input by scanning the input to see if it contains a prohibited string. The list of prohibited strings is called a *blacklist*.

# Table of Contents

- **Security Terminology**
- **XML lacks inherent security**
- **A valid XML document can still cause trouble**
- **Security considerations when processing XML documents**
  - Reading inputs from external URLs and XInclude (external entities)
  - Attack surface
  - Pros and cons of spending the resources to check inputs
- **Miscellaneous security topics**  **You are here.**
  - Expanding entities
  - Poorly constructed regular expressions
  - Manual editing of XML documents (problems with copying and pasting)
  - Unconstrained markup and data
- **A brief *introduction* to XML security tools and capabilities**
  - Canonicalization
  - XML Digital Signatures (Integrity)
  - XML Encryption (Confidentiality)

# Exponential Regular Expressions

---



# Ubiquity of Regular Expressions

---

- **The regular expression language is powerful. Regular expressions are used in many of the XML technologies:**
  - XML Schema: regular expressions are used in the pattern facet
  - XPath: regular expressions are used in these functions: matches(), tokenize(), and replace()
  - XSLT: regular expressions are used in the analyze-string element

# Example: Constraints on Surnames

English language family names are under 100 characters in length and consist of the characters a–z, A–Z, space, hyphen, apostrophe, and period.

Allowable Set of Characters	Example Family Name
The letters a–z, A–Z	Smith
Hyphen	Parsons-Kerns
Apostrophe	O'Donnel
Space	de La Cruz
Period	St. Ives

**Regex:** `[a-zA-Z' \.-]+`

# Understanding a Regex

**[a-zA-Z' \.-]+**

**[...]** means “pick one of the characters listed within the brackets”

**a-z** means range: the characters from ‘a’ to ‘z’

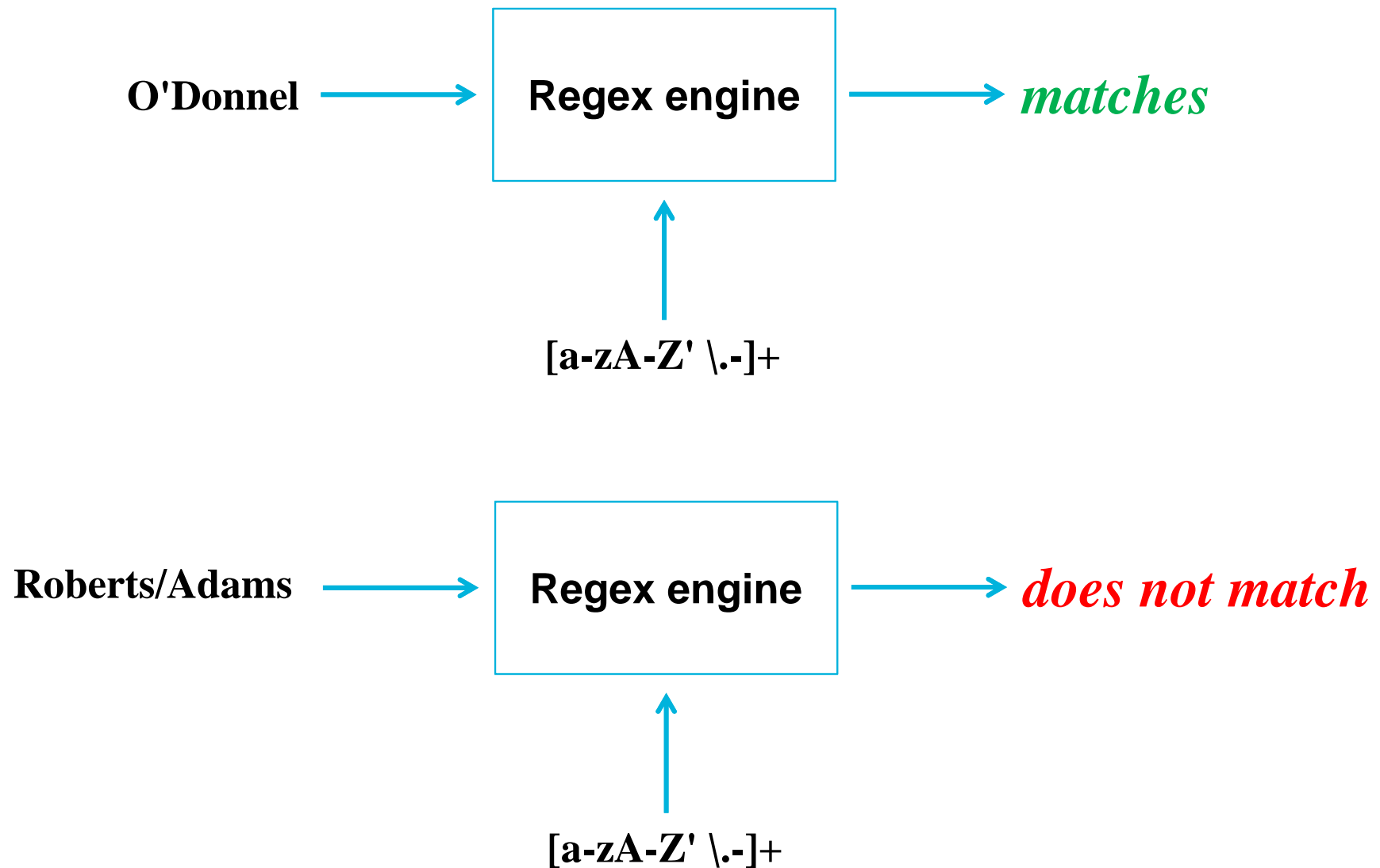
**A-Z** means range: the characters from ‘A’ to ‘Z’

**\.** means “the period character” (in regular expressions the period character has a special meaning; by preceding it with backslash we break out of (escape) its normal meaning)

**+** means “one or more occurrences”

So, the regex means “*One or more letters of the alphabet (upper and lowercase), space, period, and dash.*”

# “Does this string match that regex?”



# Example of a Regex in XML Schema

```
<simpleType name="English-language-family-name">
  <annotation>
    <documentation> The vast majority of English language
      family names are at least 1 character long, under 100 characters,
      and consist of the characters: a-z, A-Z, space, hyphen, period,
      and apostrophe.</documentation>
  </annotation>
  <restriction base="string">
    <minLength value="1" />
    <maxLength value="100" />
    <pattern value="[a-zA-Z' \.-]+" />
  </restriction>
</simpleType>
```

Regular expression

```
<element name="Family-name" type="English-language-family-name" />
```

See Family-name.xsd in the regexes folder

# Instance Document

<Family-name>\_\_\_\_\_</Family-name>



The value is constrained to have a length of 1 – 100 characters and the characters are constrained to be a-z, A-Z, space, hyphen, apostrophe, and period

If the value of Family-name exceeds the length constraint or uses characters other than those defined by the regex then an XML Schema validator will report that the data is “invalid.”

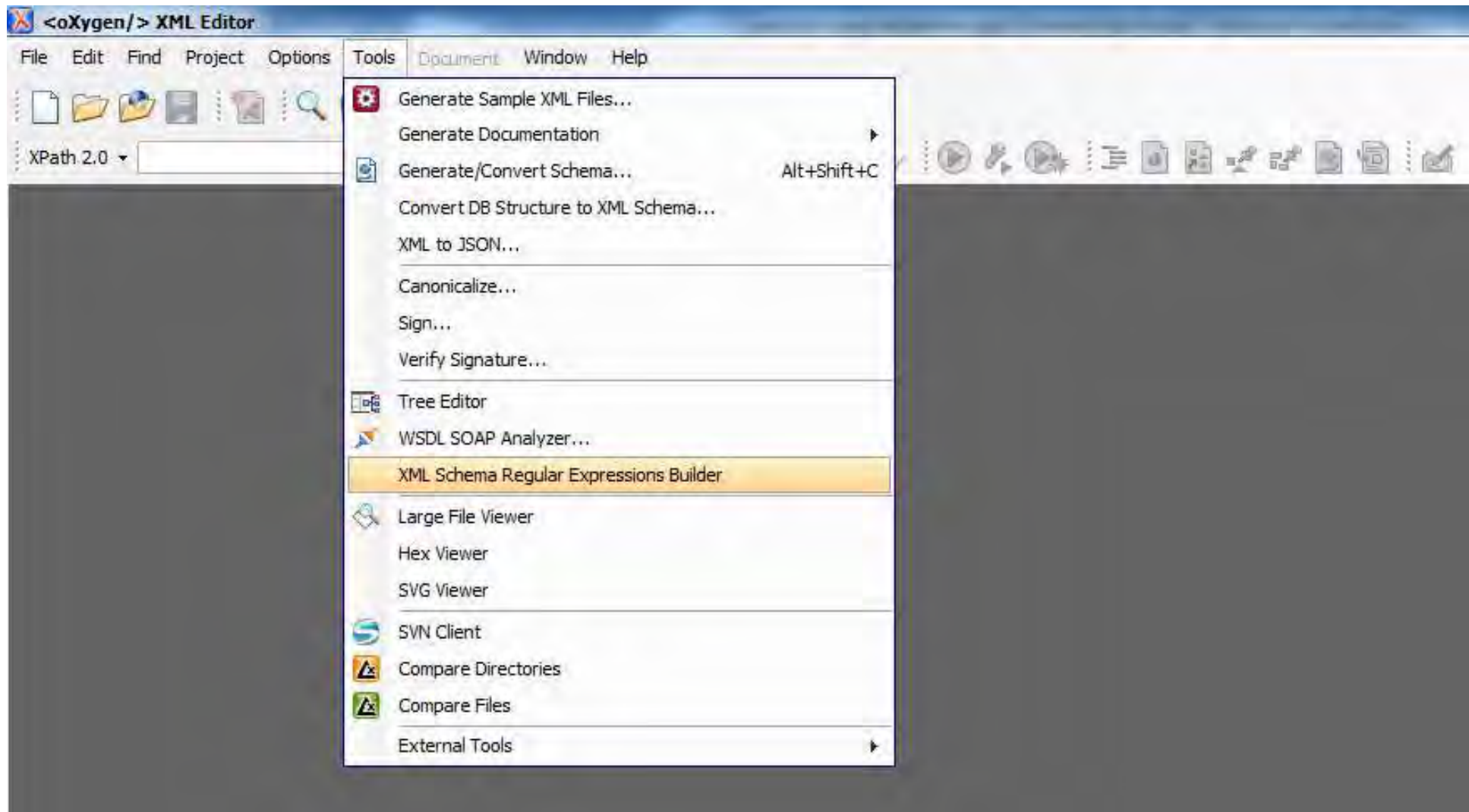
See Family-name.xml in the regexes folder

# Regex's are Good

---

- **The regular expression language is concise and powerful.**
- **With a regex you can precisely specify the set of strings – the language – that is permitted.**

# oXygen XML has a Tool for Testing Regexes

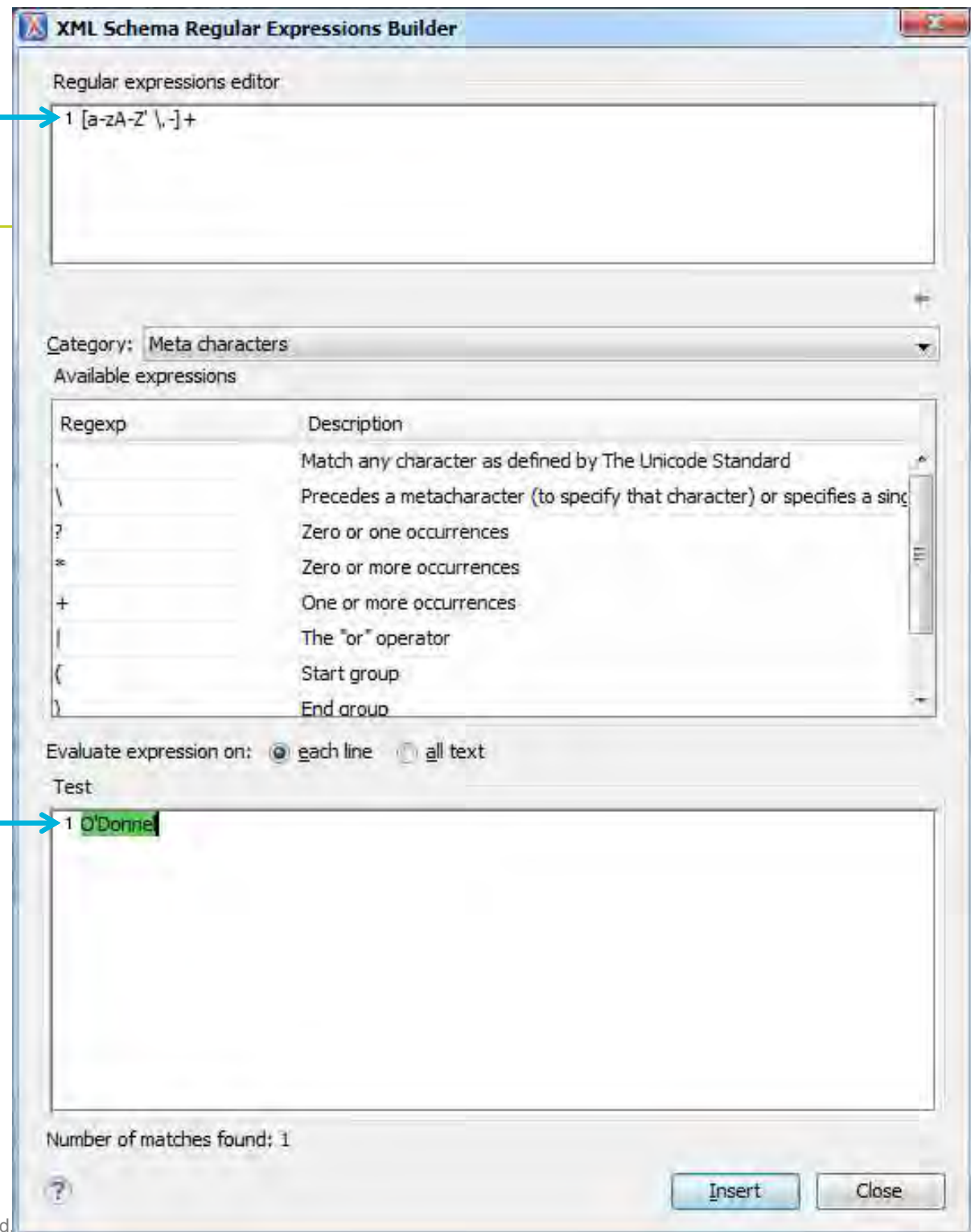




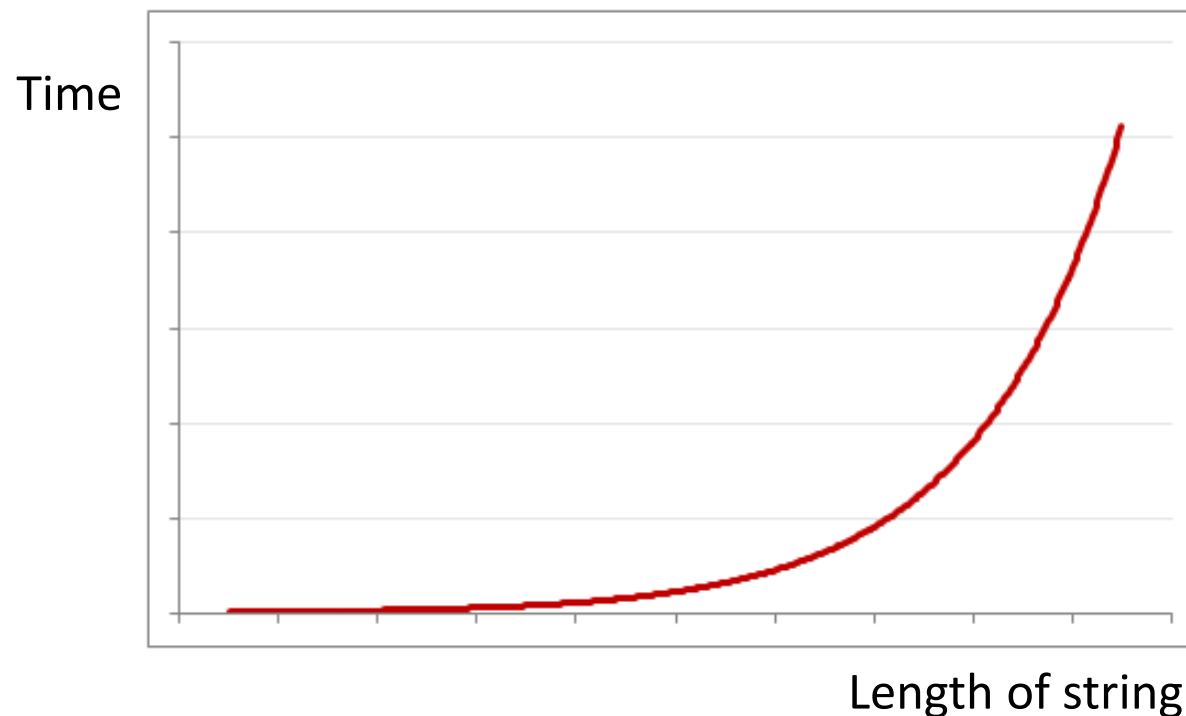
Type your regex here

Type your string here

**Green** means the string matches the regex



# Definition of “Exponential Regex”



A regular expression is said to be *exponential* if, as the length of an invalid string increases a little, the time it takes to determine that it is invalid increases a great deal (exponentially).

# Email Address

---

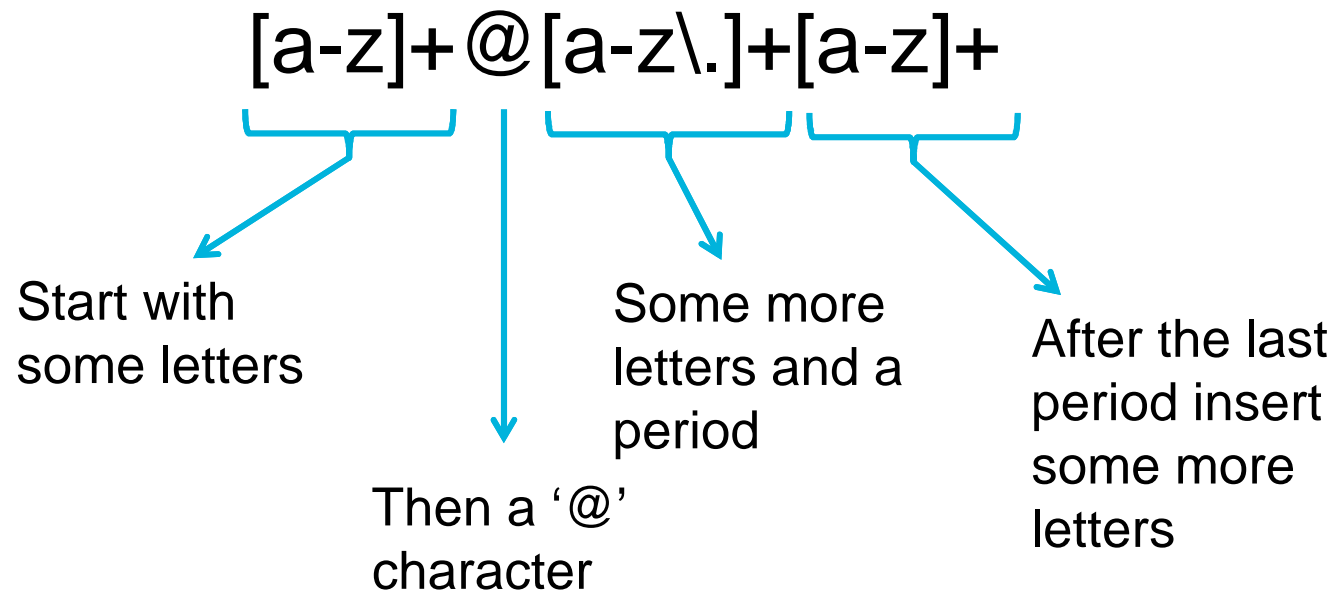
johndoe@somewhere.army.mil

Email addresses follow this pattern:

- Start with some letters
- Then a '@' character
- Then some more letters
- Repeat any number of times:
  - The "." character
  - Some more letters

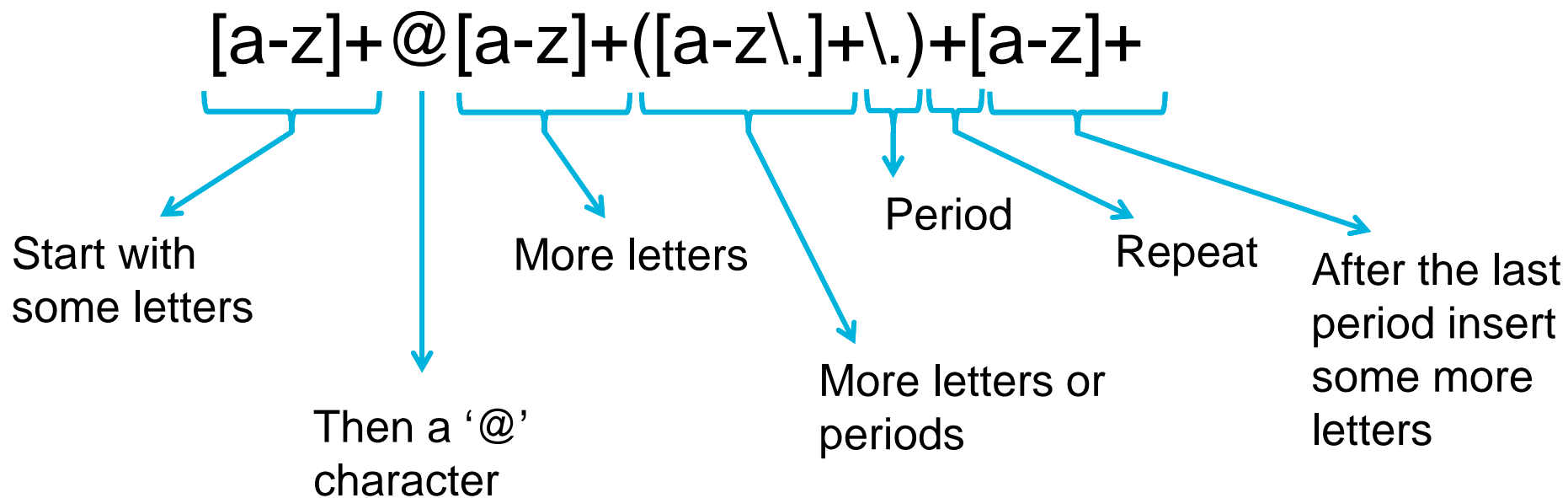
*Note: this is not a complete description of the pattern followed by email addresses.*

# Email Regex, Version #1



This regex is not exponential

# Email Regex, Version #2



This regex is exponential

# XML Schema with Exponential Regex

```
<element name="email">
  <simpleType>
    <restriction base="string">
      <pattern value="[a-z]+@[a-z]+([a-z\.]++\.)+[a-z]++" />
    </restriction>
  </simpleType>
</element>
```

# Short, Invalid Email

---

`<email>johndoe@somewhere.org.mil.biz.net0</email>`



Last character is invalid

# Quickly Generates an Error

The screenshot shows the Saxon-EE XML editor interface. The main window displays the XML file `Email-exponential-short.xml` with the following content:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <Example xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:noNamespaceSchemaLocation="Email-exponential.xsd">
4   <email>johndoe@somewhere.org.mil.biz.net0</email>
5 </Example>
6

```

A validation error is displayed in the status bar and the Errors pane. The error message is: "E [Saxon-EE 9.4.0.6] The content 'johndoe@somewhere.org.mil.biz....' of element <email> does not match the required simple type. Value 'johndoe@somewhere.org.mil.biz....' contravenes the simple type." The error is located at line 4, column 11.

The Errors pane shows the following details:

Info	Description - 2 items	Resource
✖	E [Saxon-EE 9.4.0.6] The content 'johndoe@somewhere.org.mil.biz....' of element <email> does not match the required simple type. Value 'johndoe@so...	Email-exponentia
✖	E [Saxon-EE 9.4.0.6] One or more validation errors were reported	Email-exponentia

The status bar at the bottom indicates: "C:\...\examples\regexes\Email-exponential-short.xml" and "Validation - failed. Errors: 2".



# Long, Invalid Email

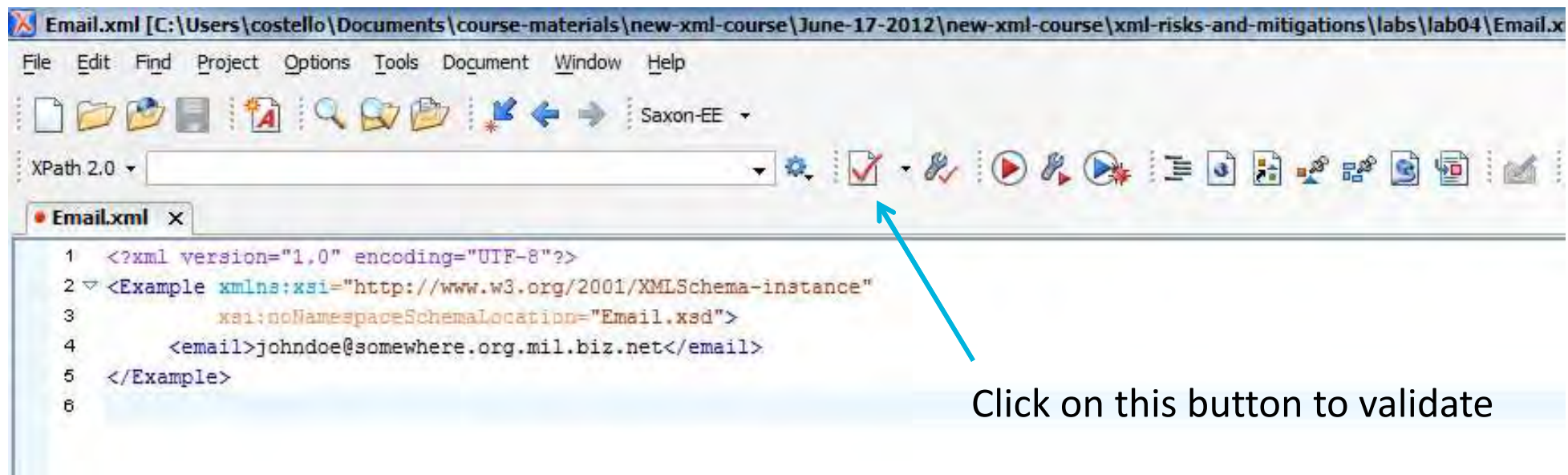
```
<email>johndoe@somewhere.org.mil  
.biz.net.com.edu.info.jobs.aero.  
cat.coop.org.mil.biz.net.com.edu  
.info.jobs.aero.cat.coop.org.mil  
.biz.net.com.edu.info.jobs.aero.  
cat.coop.org.mil.biz.net.com.edu  
.info.jobs.aero.cat.coop.org.mil  
.biz.net.com.edu0</email>
```



Last character is invalid

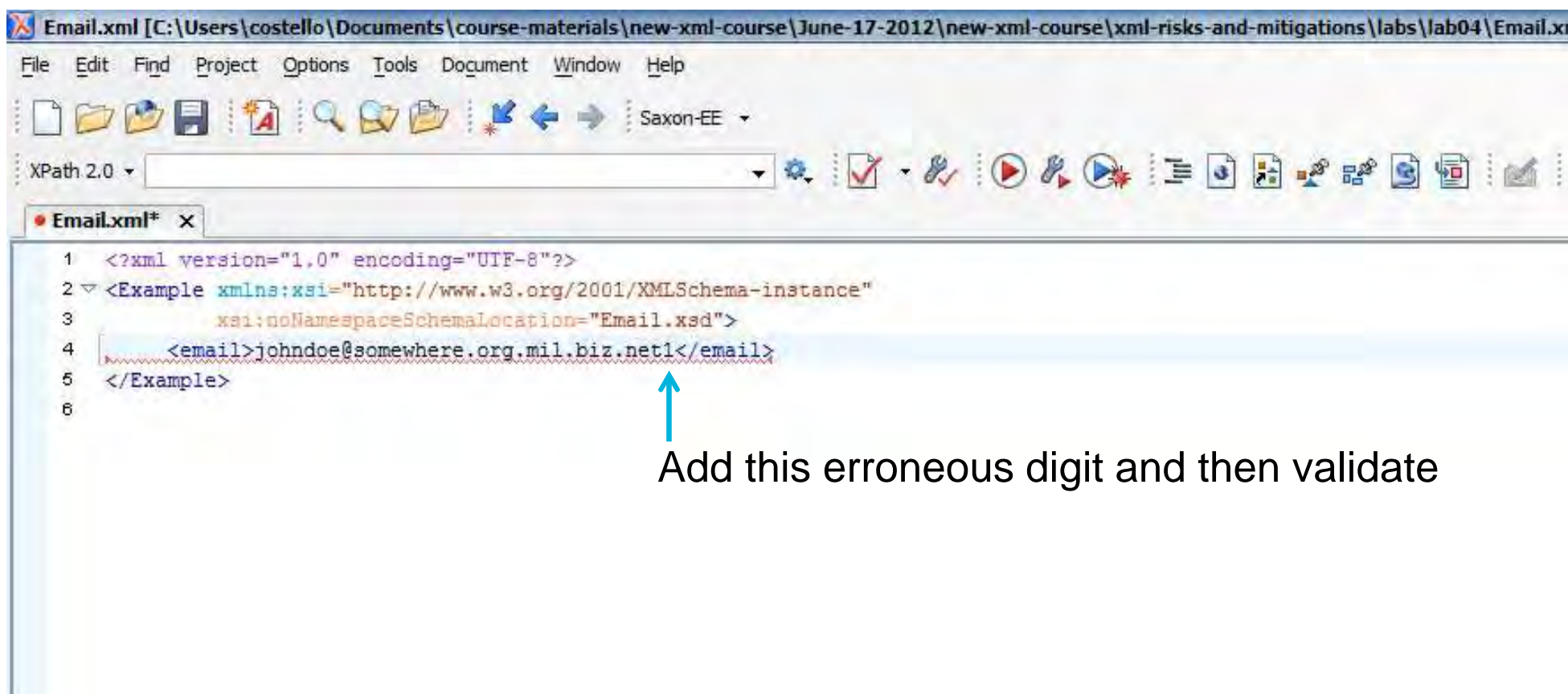
# Lab 4

- In the Lab04 folder you will find Email.xsd and Email.xml
- Email.xsd contains a regex that is exponential.
- Open Email.xml in oXygen XML and validate it.



## Lab 4 (cont.)

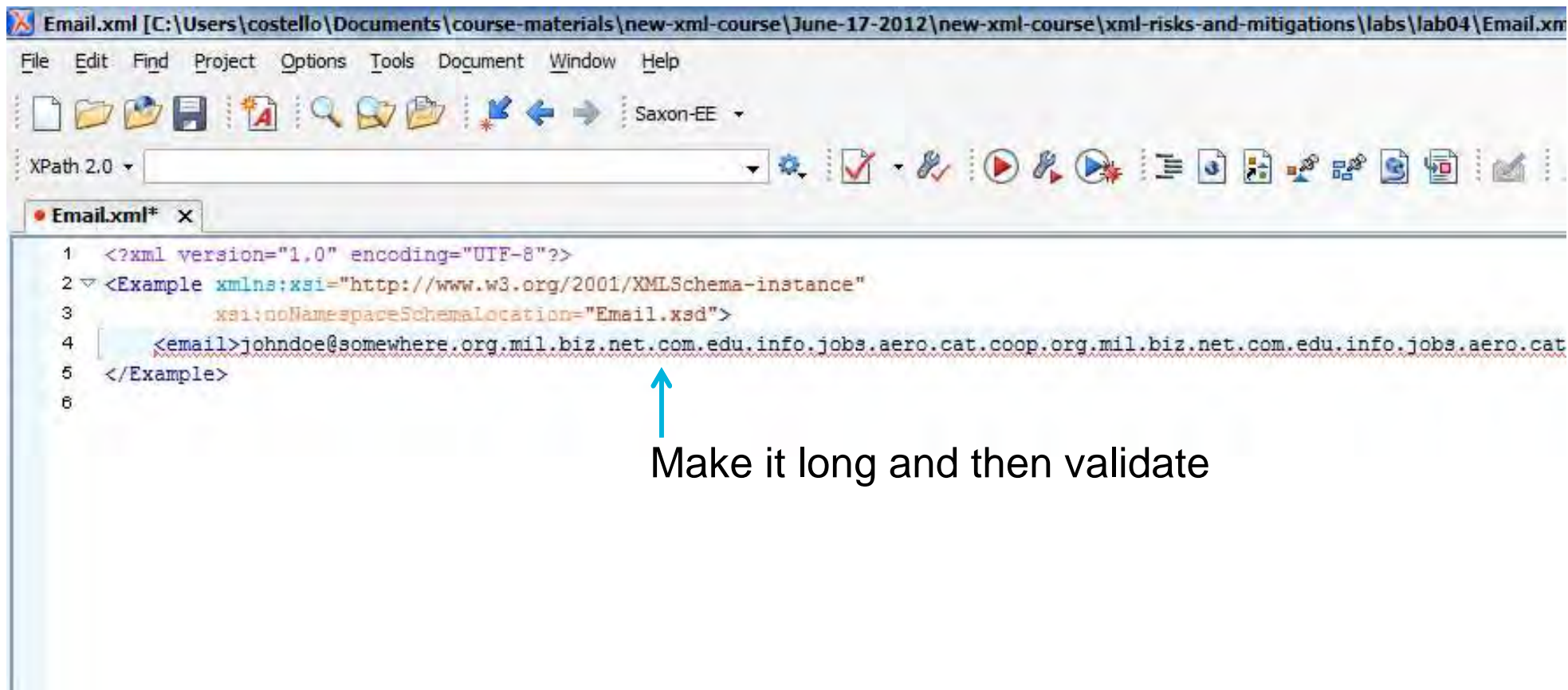
- Now add an erroneous character (a digit) at the end of the email address and then validate



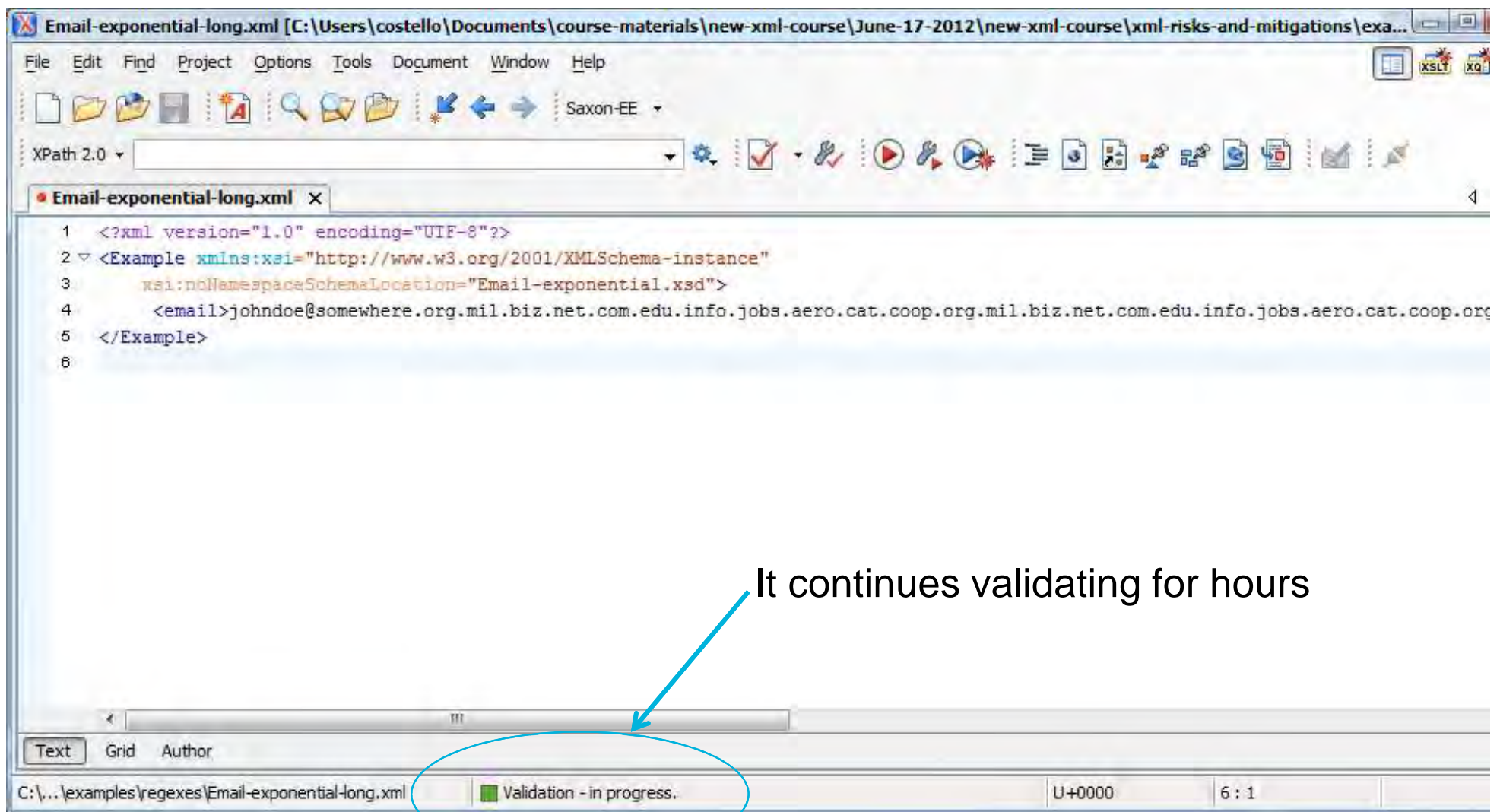
Add this erroneous digit and then validate

# Lab 4 (concluded)

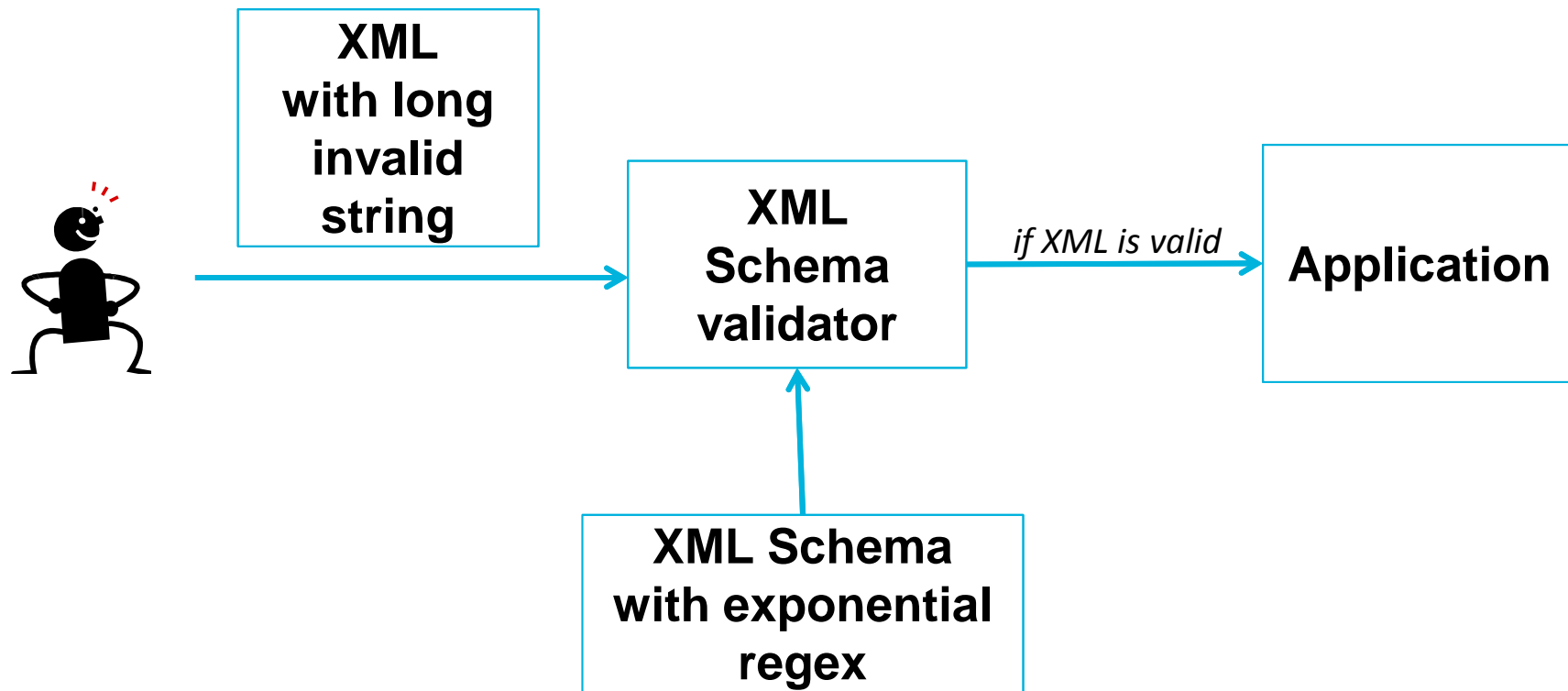
- Make the email address very long and then add an erroneous digit on its end



# Churns for Hours



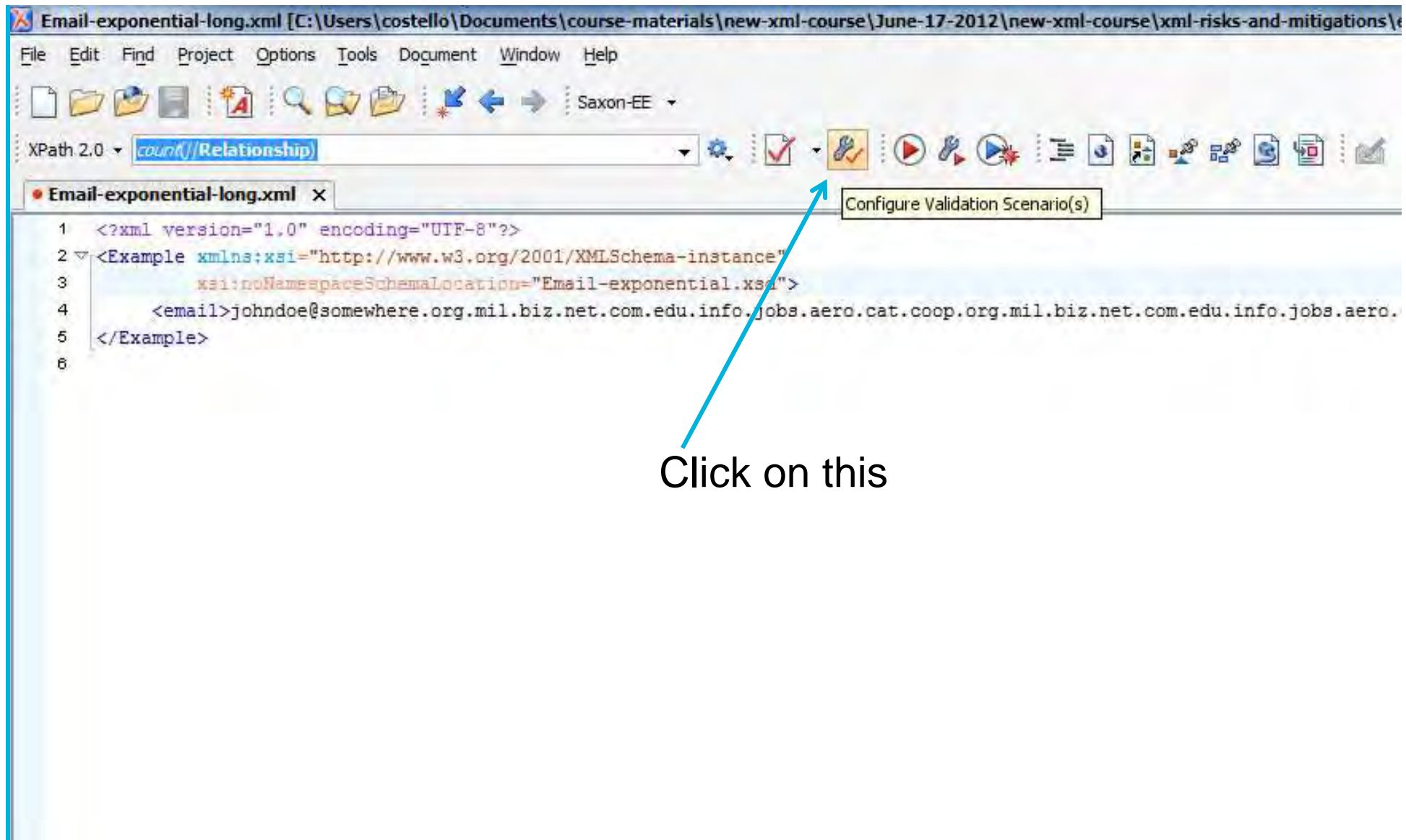
# Regular Expression Denial of Service (ReDos) Attack

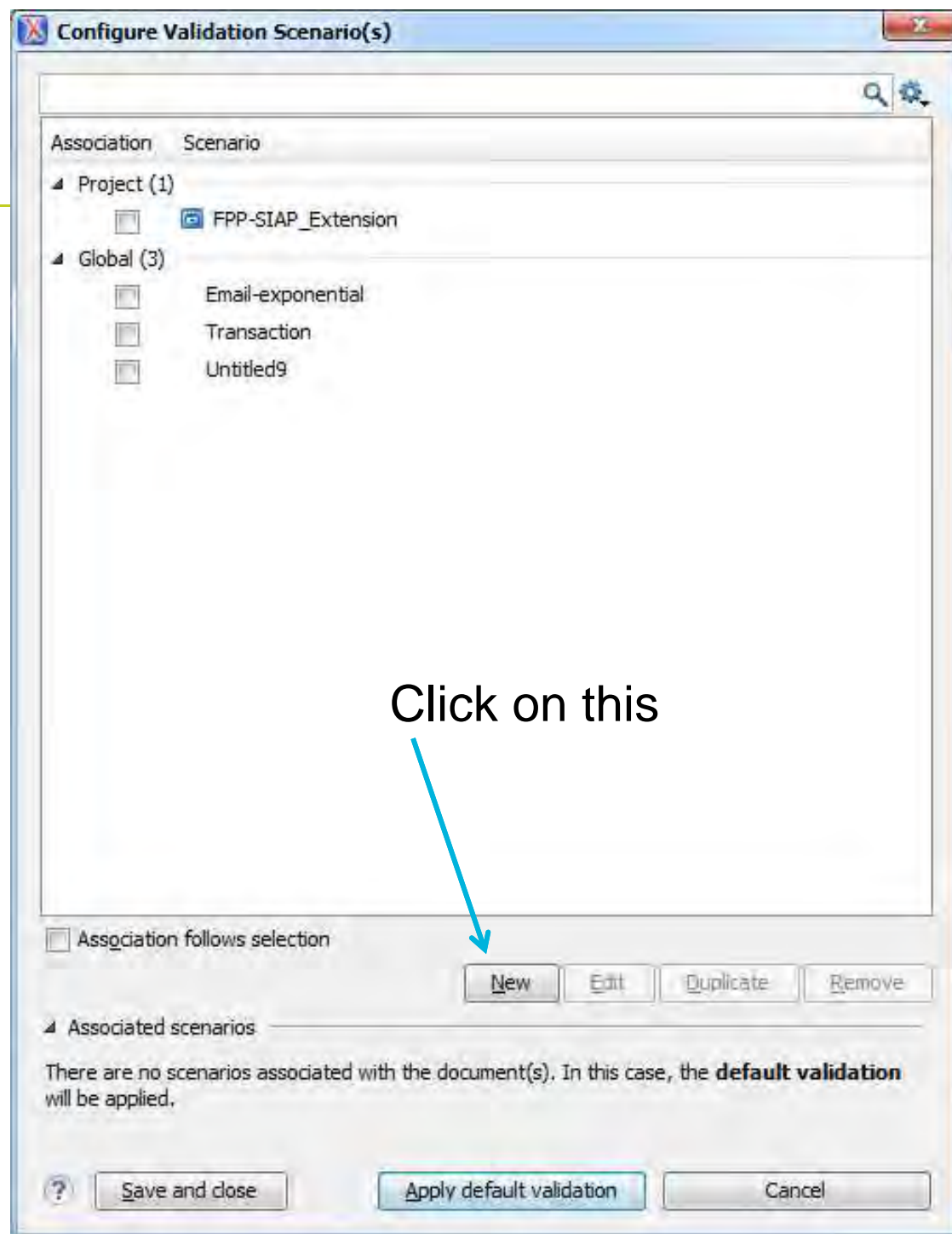


Scenario: A system validates inbound XML documents against an XML Schema. Only if the XML Schema validator indicates that the XML document is “valid” does the system process the document. The XML Schema is publicly available so that users know how to interact with the system. An adversary examines the XML Schema and discovers that it is using an exponential regex. He/she exploits this vulnerability in the XML Schema as follows: He/she sends the service an XML instance document. For the element with the exponential regex, he/she provides a value that causes the XML Schema validator to run for hours or days. Thus, the adversary has successfully accomplished a DoS attack.

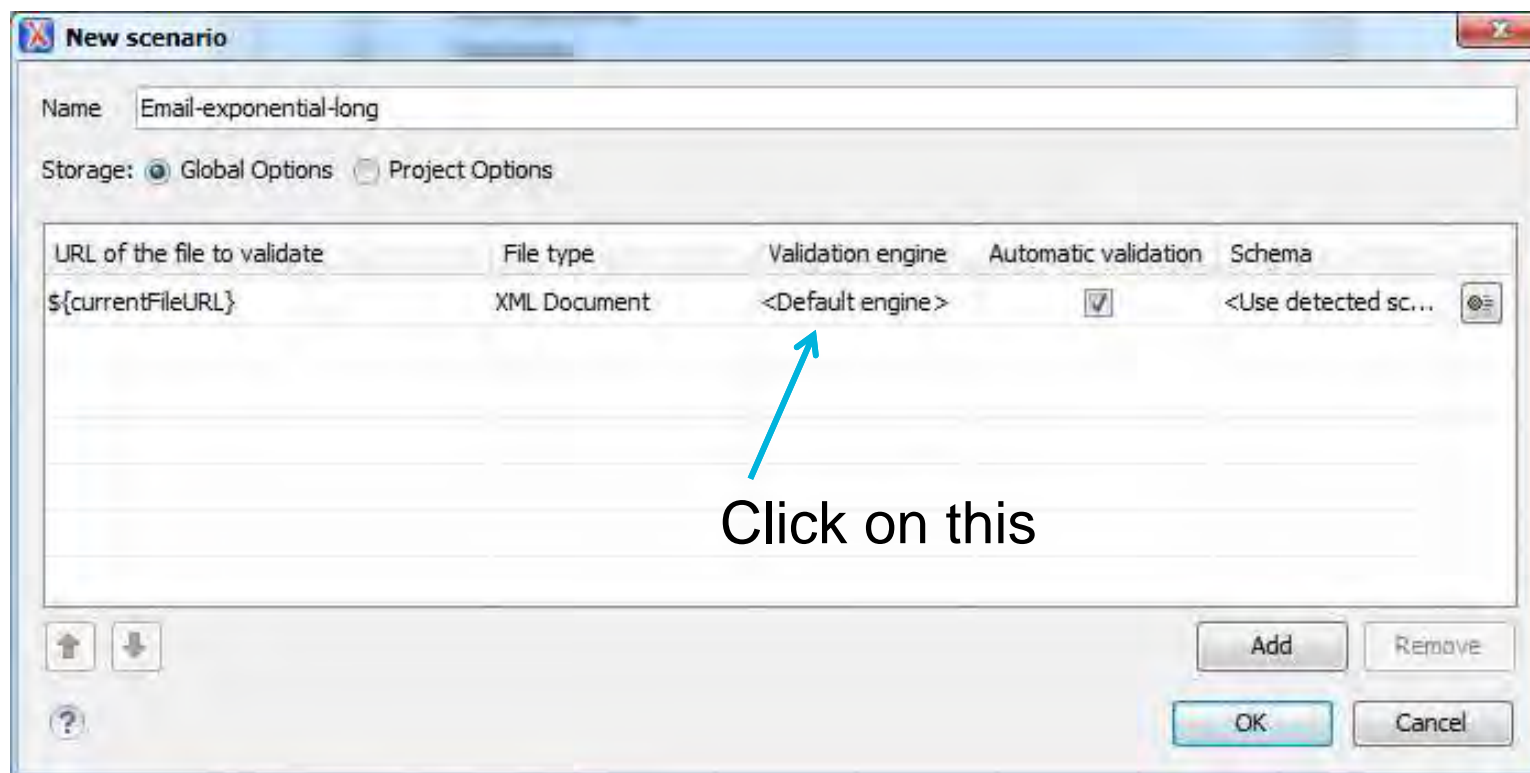


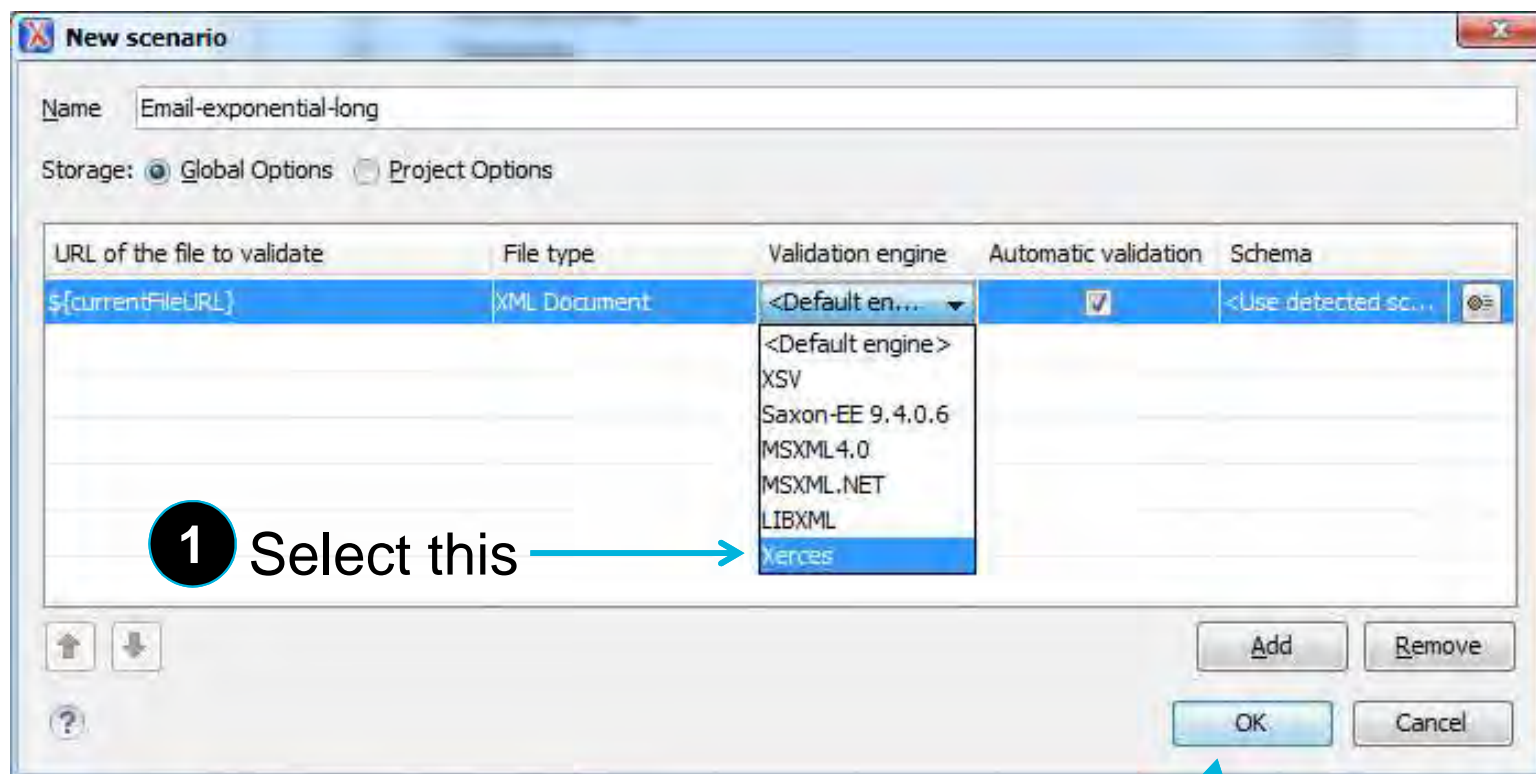
# Change to Another Validator

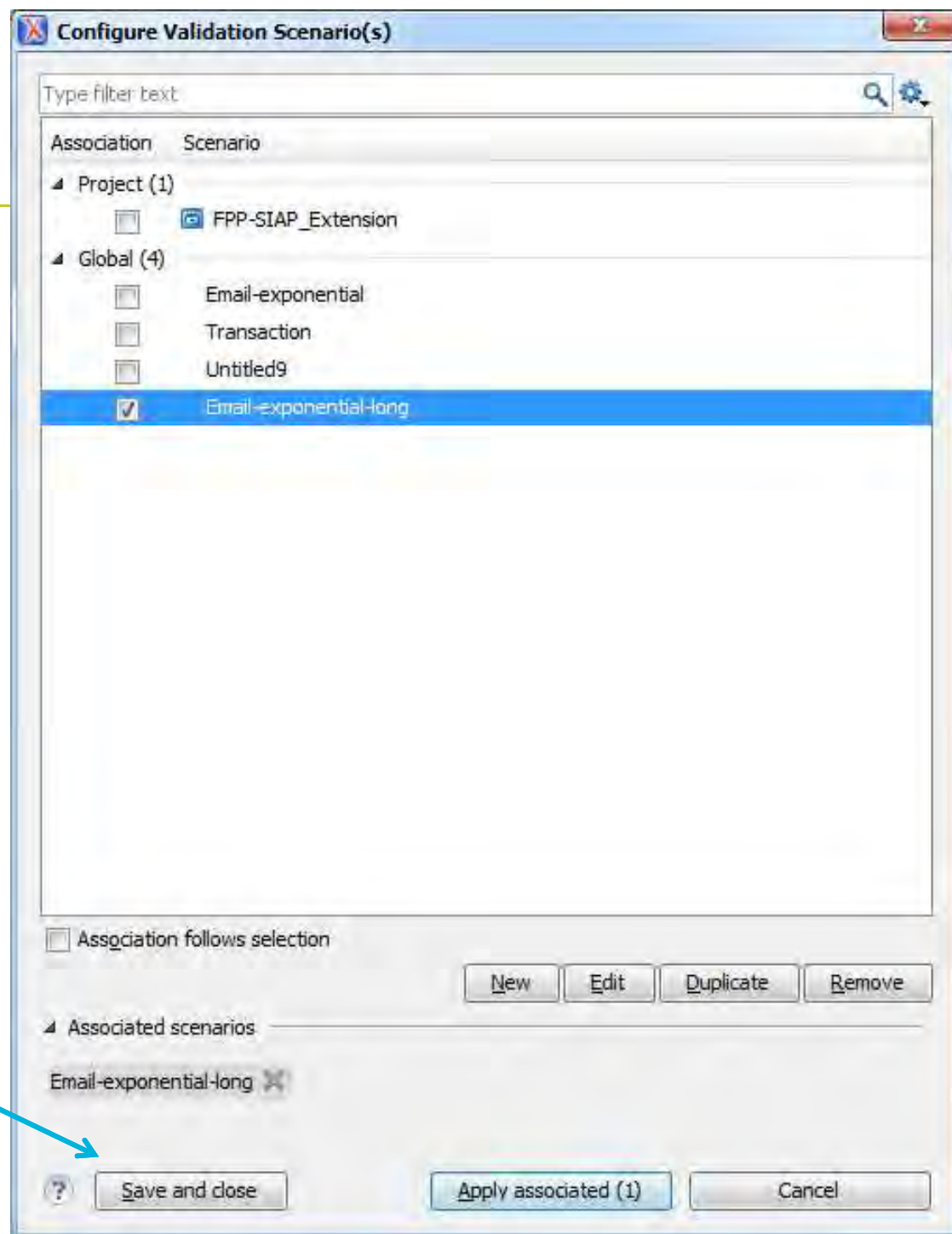




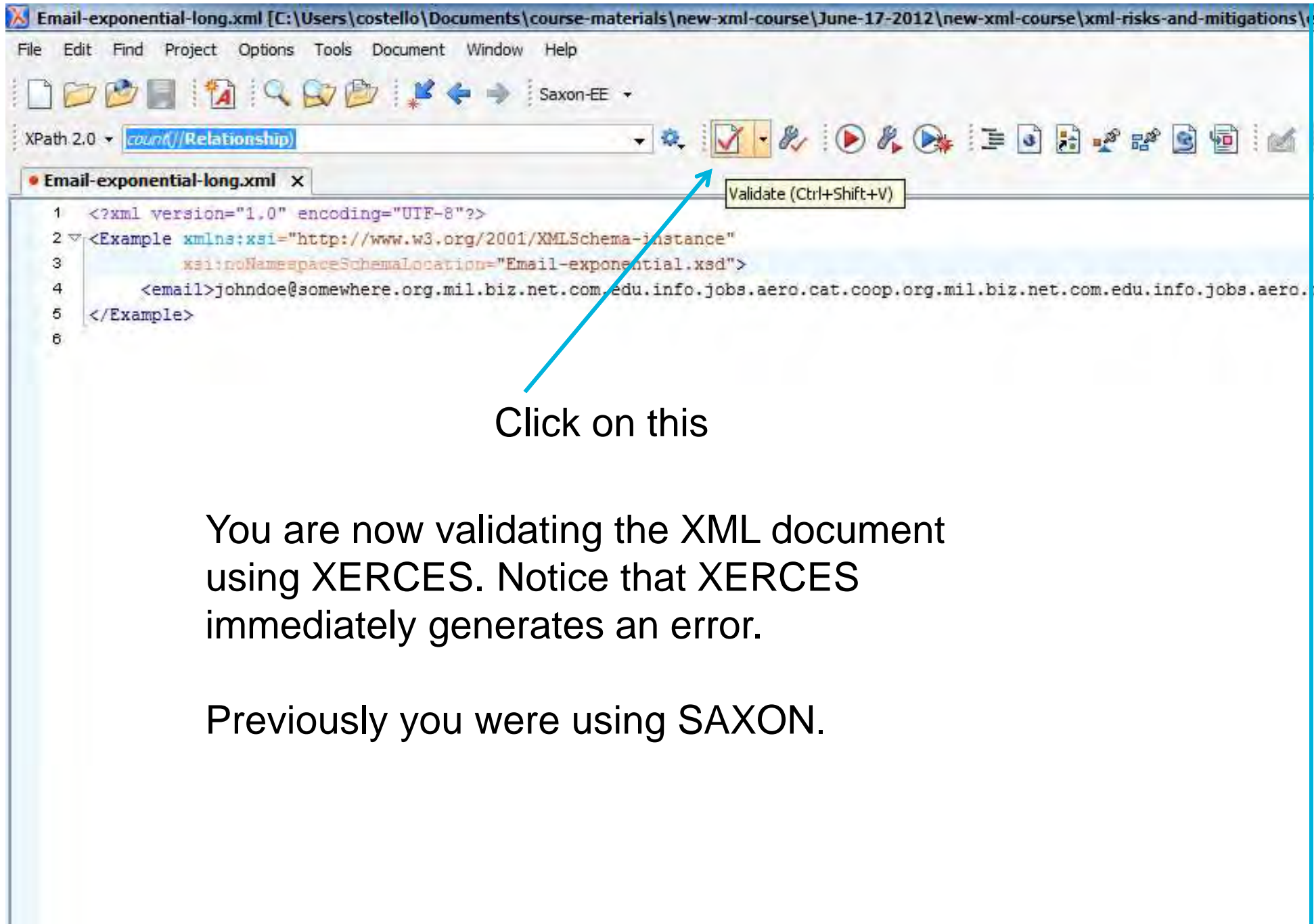








Click on this



The screenshot shows the Saxon-EE XML editor interface. The title bar indicates the file is 'Email-exponential-long.xml' located at 'C:\Users\costello\Documents\course-materials\new-xml-course\June-17-2012\new-xml-course\xml-risks-and-mitigations\'. The menu bar includes File, Edit, Find, Project, Options, Tools, Document, Window, and Help. The toolbar contains various icons for file operations, XPath evaluation, and validation. The XPath 2.0 dropdown is set to 'count(//Relationship)'. The main text area displays the XML content:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Example xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:noNamespaceSchemaLocation="Email-exponential.xsd">
4     <email>johndoe@somewhere.org.mil.biz.net.com.edu.info.jobs.aero.cat.coop.org.mil.biz.net.com.edu.info.jobs.aero.
5 </Example>
6
```

A blue arrow points from the text 'Click on this' to the 'Validate (Ctrl+Shift+V)' button in the toolbar.

Click on this

You are now validating the XML document using XERCES. Notice that XERCES immediately generates an error.

Previously you were using SAXON.

# Uses Different Regex Engines

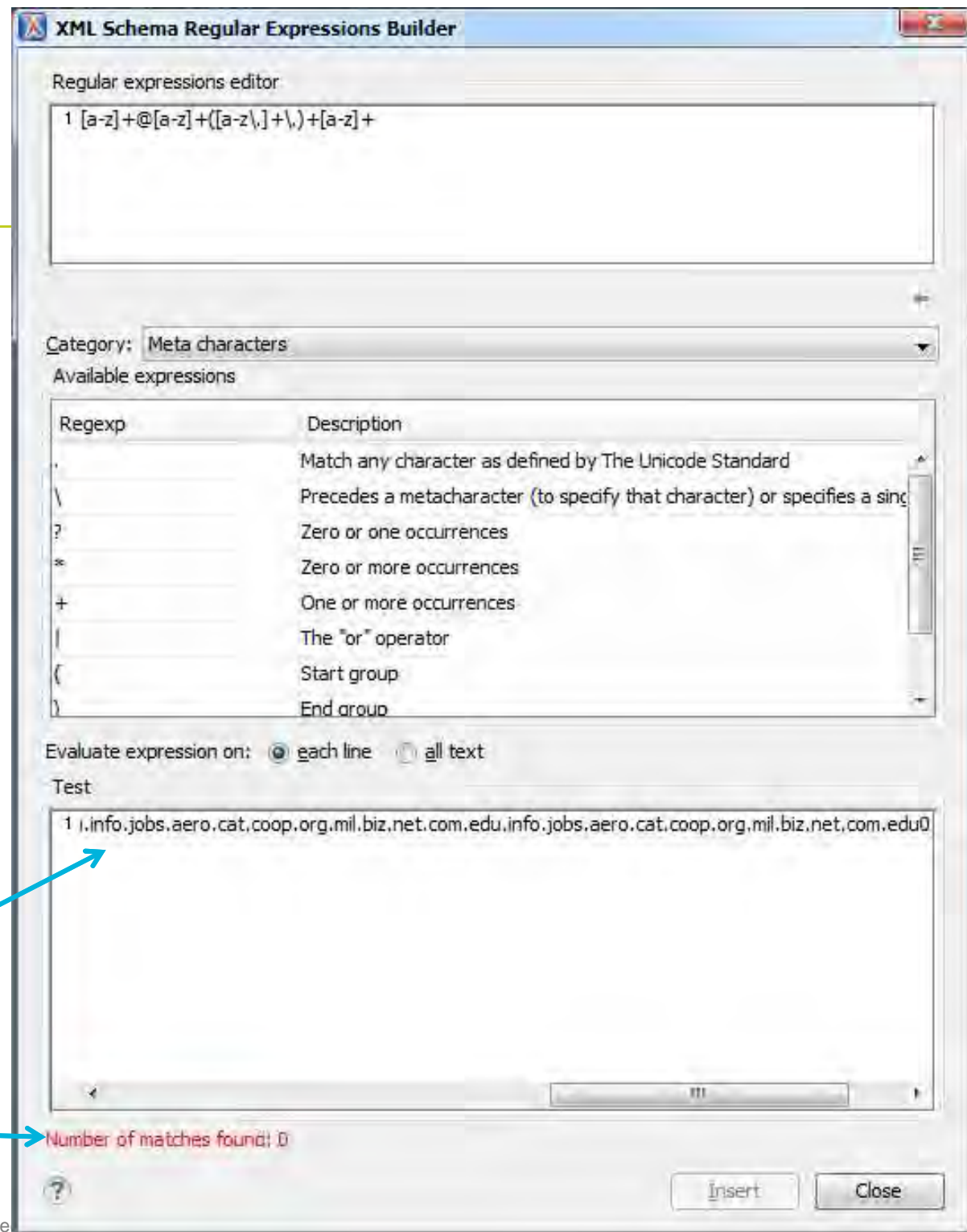
---

SAXON

Regex  
engine  
A

XERCES

Regex  
engine  
B



With a long, invalid string  
the oXygen regex tool  
immediately determines  
it doesn't match

Why?

**Answer: The oXygen regex tool uses XERCES's regex engine**

---

# Simple Regex

---

`a+` ← Not exponential

`(a+)+` ← Exponential



# Patterns of Regexes that are Exponential

---

$(a^+)^+$

$(a^*)^*(a^+)^+$

$(a^+)^*$

When you see regexes with the same form, suspect exponential behavior.

# Cause of Exponential Time

---

- **Backtracking is the cause of the exponential time. Here's how some regex engines work:**
  - “I took this path and it failed so I'll backup and try another path. Oh, that didn't work, so I'll back up and try still another path. And on and on.”
- **The number of possible paths grows exponentially.**

# Mitigating the Risk

---

- **Craft your regexes carefully**
  - See the first paper referenced on the next slide
- **Use an XML Schema validator that doesn't backtrack, such as XERCES**

# References

---

- In the folder, papers, see this:  
**Guidelines\_for\_Regular\_Expressions\_with\_XML\_Schemas-29-June-2012.pdf**
- The real cause of exponential time: some regex engines have too much computation power  
See the paper titled, *Use Minimal Computational Power* in the papers folder

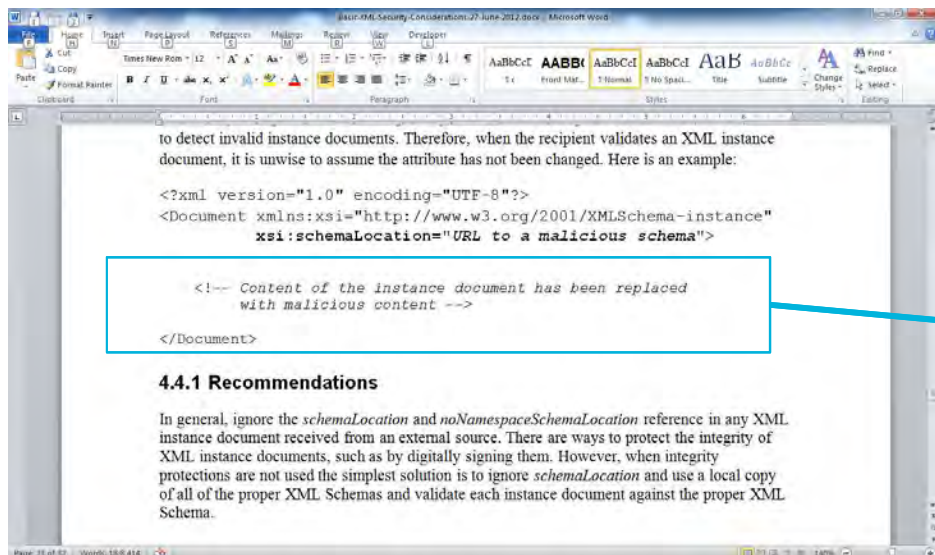
# Copying and Pasting Text into XML

---

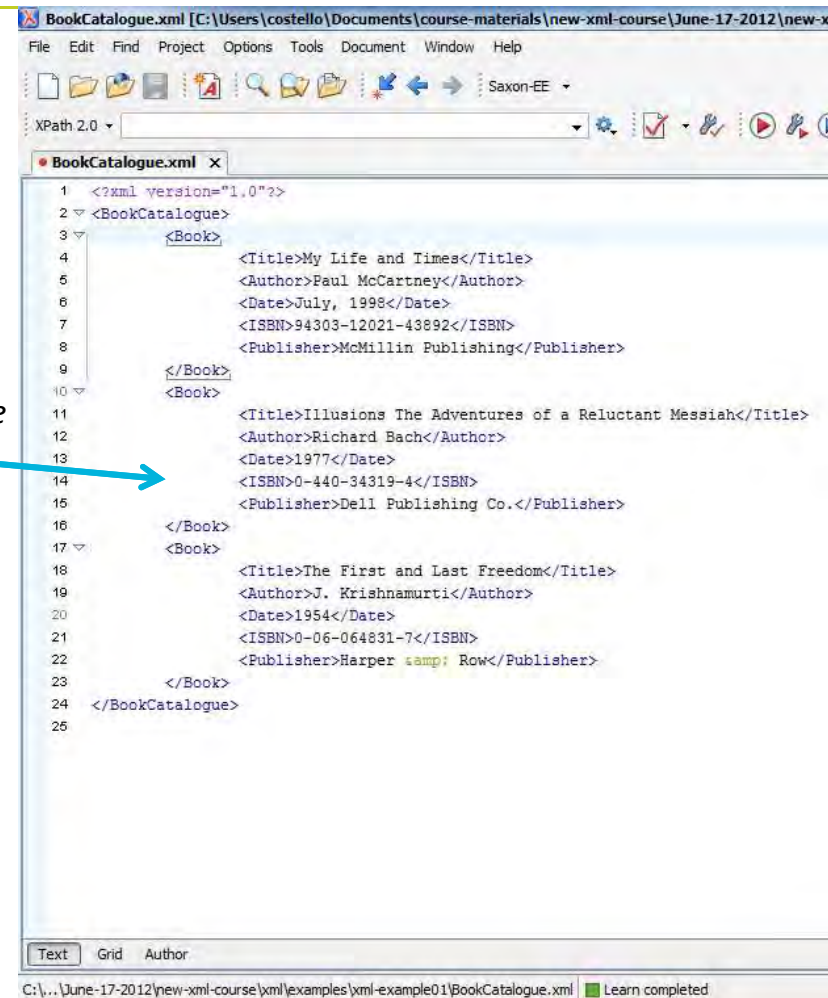
# Manual Editing of XML is Common

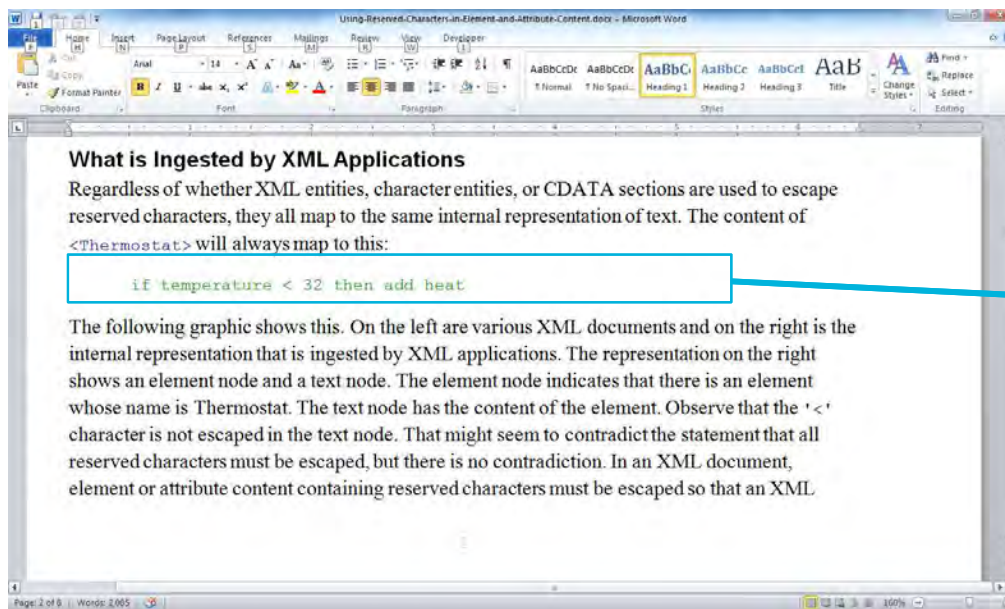
---

- XML documents are intended for machine processing.
- However, XML documents are also easy to edit manually because they are text documents and text editors are ubiquitous.
- Problems can arise when manually copying text from an arbitrary document and pasting it into an XML document.

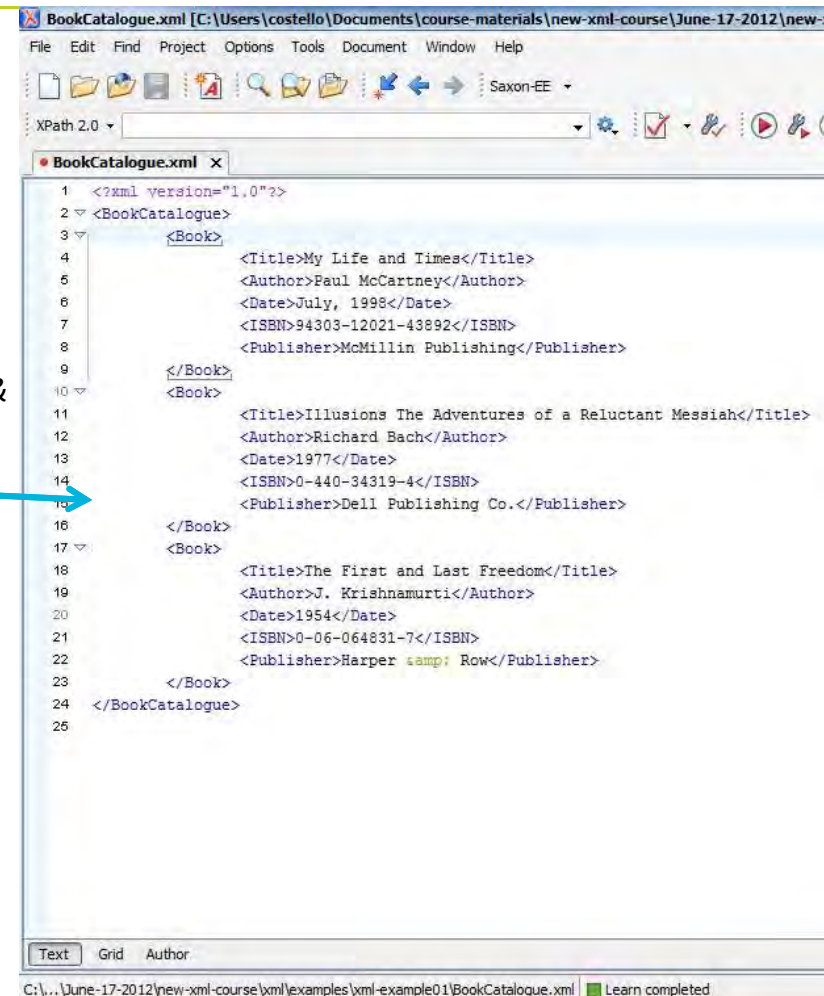


The copied text includes an end-tag but not the start-tag. Consequently, the resulting XML document is **not well-formed**.



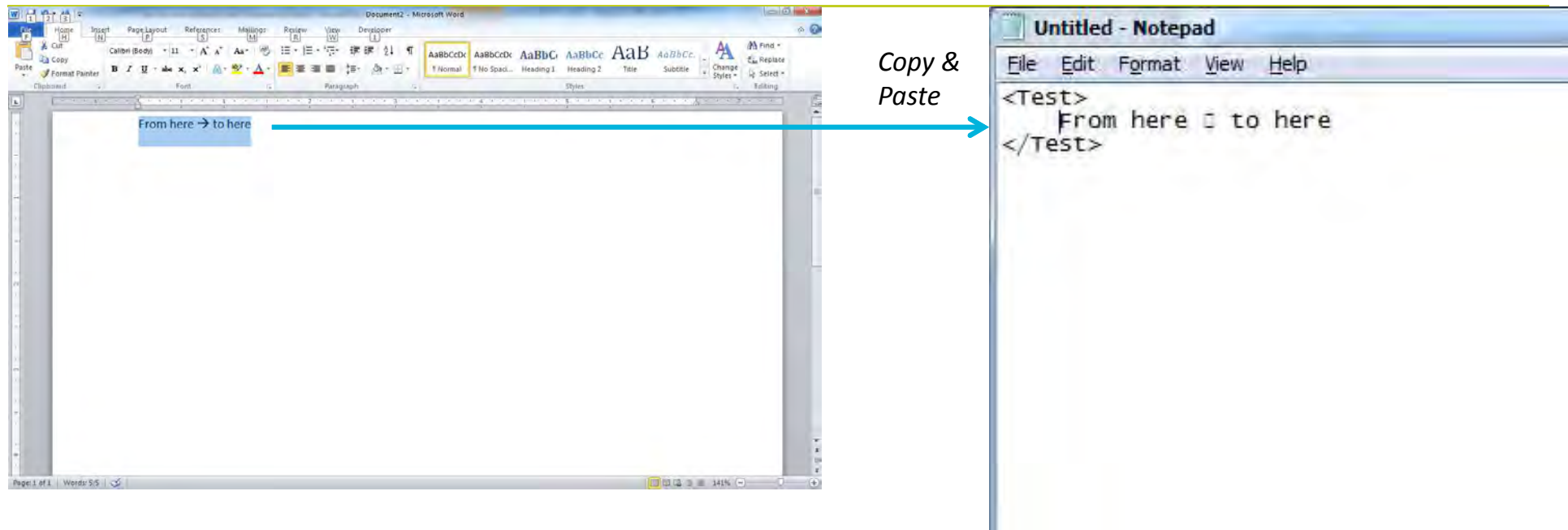


Copy &  
Paste



The copied text contains a character that is reserved in XML. Consequently, the resulting XML document is **not well-formed**.





If the copied text comes from a document or application that uses a **different character encoding** than the XML document, then the text that is pasted could differ from the text that was copied. For example, if the right arrow character → from a Word 2010 document is copied and pasted into a text editor such as Notepad, then the box character will result instead of the arrow.

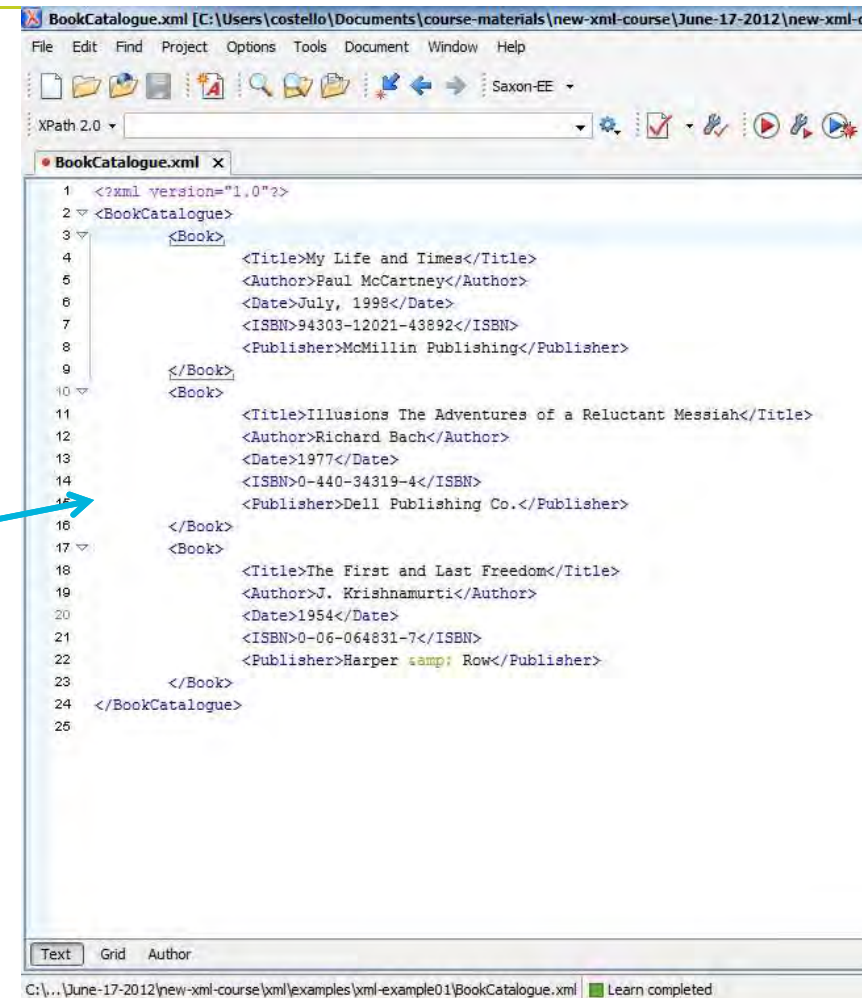
```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Test [
  <!ENTITY IMHO "In my humble opinion">
]>
<MovieReview>
  &IMHO; it was a great movie.
</MovieReview>

```

Copy &  
Paste

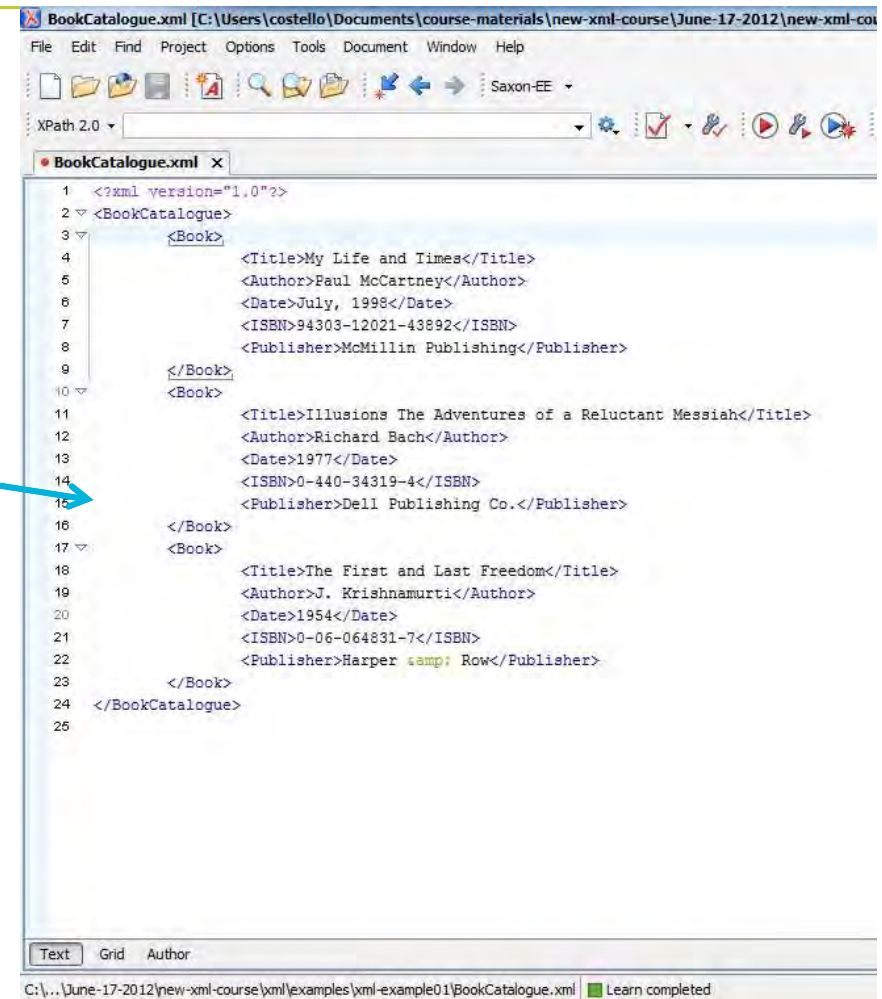
The copied text might contain user-defined **XML entities** that cannot be resolved in the destination XML document.



```
<?xml version="1.0" encoding="UTF-8"?>
<mag:Magazine xmlns:mag="http://www.magazine.com">
  <mag:Title>Scientific American</mag:Title>
</mag:Magazine>
```

**Namespace prefix in the copied text might not be resolvable.**

*Copy & Paste*



```

<?xml version="1.0"?>
<Library>
  <BookCatalogue>
    <Book isbn="94303-12021-43892">
      <Title>My Life and Times</Title>
      <Author>Paul McCartney</Author>
      <Date>1998</Date>
      <Publisher>McMillan Publishing</Publisher>
    </Book>
    <Book isbn="0-440-34319-4">
      <Title>Illusions The Adventures of a Reluctant Messiah</Title>
      <Author>Richard Bach</Author>
      <Date>1977</Date>
      <Publisher>Dell Publishing Co.</Publisher>
    </Book>
    <Book isbn="0-06-064831-7">
      <Title>The First and Last Freedom</Title>
      <Author>J. Krishnamurti</Author>
      <Date>1954</Date>
      <Publisher>Harper & Row</Publisher>
    </Book>
  </BookCatalogue>
  <GuestAuthors>
    <GuestAuthor>
      <BookSigning isbn_ref="0-440-34319-4"></BookSigning>
    </GuestAuthor>
  </GuestAuthors>
</Library>

```

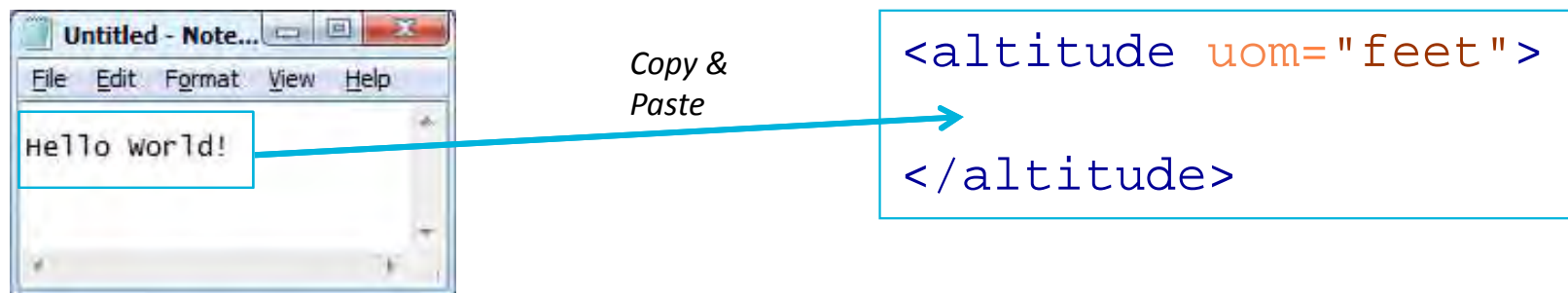
Copy &  
Paste

```

1 <?xml version="1.0"?>
2 <BookCatalogue>
3   <Book>
4     <Title>My Life and Times</Title>
5     <Author>Paul McCartney</Author>
6     <Date>July, 1998</Date>
7     <ISBN>94303-12021-43892</ISBN>
8     <Publisher>McMillan Publishing</Publisher>
9   </Book>
10  <Book>
11    <Title>Illusions The Adventures of a Reluctant Messiah</Title>
12    <Author>Richard Bach</Author>
13    <Date>1977</Date>
14    <ISBN>0-440-34319-4</ISBN>
15    <Publisher>Dell Publishing Co.</Publisher>
16  </Book>
17  <Book>
18    <Title>The First and Last Freedom</Title>
19    <Author>J. Krishnamurti</Author>
20    <Date>1954</Date>
21    <ISBN>0-06-064831-7</ISBN>
22    <Publisher>Harper & Row</Publisher>
23  </Book>
24 </BookCatalogue>
25

```

If the copied text contains an IDREF value without the matching ID value, then the target document has a **dangling reference**, which is an error.



The copied text might not be of an appropriate **data type** for the destination XML document.

# Mitigating the Risk

---

- **If manual copying and pasting of XML content is necessary, check for any of the problems listed on the previous slides.**

# Visual Spoofing

---

# Manual Inspection of XML is Common


---

- XML documents are intended for machine processing.
- However, XML documents are often manually inspected. Or, the data in XML documents are transferred to a Web page and the Web page is manually inspected.
- If the inspection results in a “thumbs up” approval, then the XML is sent downstream for usage/processing.
- Problems can arise when data is manually inspected.



# Homograph Attack

Dear Customer:  
There is a problem with your account.  
Check with your bank: <http://www.citibank.com>



This is not the Latin letter 'c', it is the Cyrillic letter "es". Message recipients have no way to discern that by looking at the address. Customers who click on the web address will not go to the citibank.com web site but rather to the Cyrillic 'c' itibank.com web site. If customers do not recognize the web site as bogus, they might give away their user names and passwords.

**Homoglyph:** the identical appearance of two or more glyphs (characters). For example, the Latin small letter 'c' (U+0063) and the Cyrillic small letter 'es' (U+0441) have the same glyph: c.

# Visual Spoofing

---

Visual spoofing occurs when a malicious actor deliberately replaces a character with another character that is similar in appearance. In other words, the two characters have identical or nearly identical glyphs. The intent is to deceive users visually so that they take unsafe actions.

# Direction of Text

---

- Many languages are written and read from *left to right*, but some languages, such as Arabic and Hebrew, have an inherent *right-to-left* direction.
- Unicode makes it possible to represent both types of languages in a document.

# Bidirectional Text Spoofing

An XML document was opened in a browser and here's what the user saw:

```
<Part-of-Czechoslovakia-Annexed-by-Germany>  
  http://www.example.org/annexe.jpg  
</Part-of-Czechoslovakia-Annexed-by-Germany>
```

The value of the element appears to be the URL to a harmless JPG file.

However, when the XML is viewed in a text editor a completely different story is seen. It is not a URL to a JPG file; instead, it is a URL to an exe file. The Unicode Right-to-Left Override (RLO) and Pop Directional Formatting (PDF) characters bracket gpj.exe which instructs the browser to reverse the characters. Here is the XML viewed in a plain text editor:

Continued →

# Bidirectional Text Spoofing

<Part-of-Czechoslovakia-Annexed-by Germany>

http://www.example.org/ann&#x202e;gpj.exe&#x202c;

</Part-of-Czechoslovakia-Annexed-by-Germany>

RLO – “Start reversing the text”

POP – “Stop reversing the text”

# Lab 5

---

- In the lab 05 folder you will find INVOICE.xml
- Open a browser, drag and drop INVOICE.xml into the browser. Note the URL.
- Next, open oXygen XML, drag and drop INVOICE.xml into oXygen. Notice that the URL is not what you thought it was.

# Mitigating the Risk

---

- If manual inspection of XML content is necessary:
  - Display the content in different fonts to help discern different characters.
  - Display the content in a plain text editor that doesn't resolve entities.

# References

---

- **Unicode Technical Report #36, *Unicode Security Considerations*; <http://www.unicode.org/reports/tr36/>**



# Unconstrained Markup and Data

---

# Common Practice

- XML Schema developers regularly declare XML elements with a string data type.
- And they use the XML Schema `<any>` element to allow for extensions.
- They do this to maximize flexibility.

```
<element name="Book">
  <complexType>
    <sequence>
      <element name="Title" type="string"/>
      <element name="Author" type="string"/>
      <element name="Date" type="string"/>
      <element name="ISBN" type="string"/>
      <element name="Publisher" type="string"/>
      <any minOccurs="0"/>
    </sequence>
  </complexType>
</element>
```

XML elements  
with a string  
data type

“Hey, you can have  
anything after Publisher.”

# Problem #1

---

- **Erroneous data not caught:**

- One purpose of schema validation is to catch erroneous data.
- If the data can be any string or any markup and data, then that purpose is defeated – erroneous data is not caught.

- **Consequence:**

- Developers insert additional checks inside procedural code.
- These checks are hard to change and difficult for managers to know what checks are implemented.

## Problem #2

---

- The “string” data type can contain any number of UTF-8 characters, including Chinese, Cyrillic, and Arabic.
- So that means those characters are valid in an XML document.
- This is an easy avenue for passing undesirable, sensitive, or malicious data.

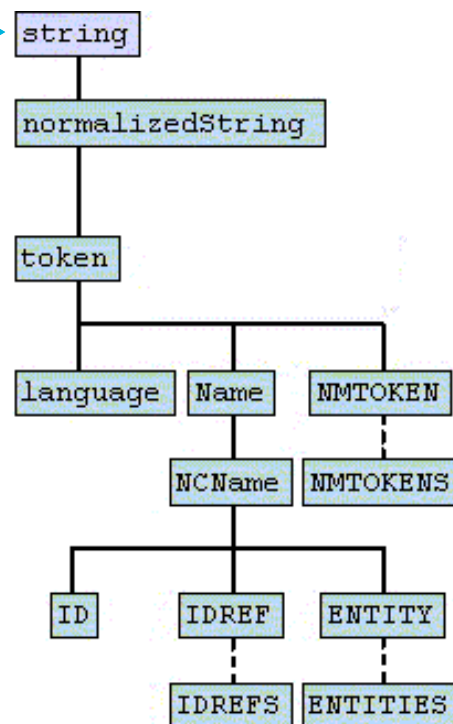
# Problem #3

---

- **There is no limit to the number of characters.**
- **XML documents with huge amounts of text are valid.**
- **Recall from the section on expanding entities that the long strings could be dynamically generated.**

# Mitigating the Risk

- Constrain the XML Schema data types.
- Don't use an unconstrained string data type or any of its derived types
- Don't use the `<any>` element, the `<anyAttribute>` element, or the `anyType` data type.



# How much Constraint?

---

- So you decide to constrain the length of strings.
- What should be the maximum allowable length of strings?
- A max length of, say, 20 is very constrained (and low risk).
- Perhaps, however, that is too constrained (your data is longer than 20 chars).
- You decide that 256 chars is a good max length.

# Two Constraint Axes

---

- **Length is just one of the constraint axes. The other is character set. Be sure to constrain the set of allowable characters.**
- **256 ASCII characters may be of acceptable risk, but 256 characters that may include Arabic, Cyrillic, and Chinese characters may not.**
- **To constrain the character set, use the XML Schema pattern facet.**



# References

---

- In the papers folder you will find the NSA Publication: ***Security Guidance for the use of XML Schema 1.0/1.1 and RELAX NG***

# Table of Contents

- **Security Terminology**
- **XML lacks inherent security**
- **A valid XML document can still cause trouble**
- **Security considerations when processing XML documents**
  - Reading inputs from external URLs and XInclude (external entities)
  - Attack surface
  - Pros and cons of spending the resources to check inputs
- **Miscellaneous security topics**
  - Expanding entities
  - Poorly constructed regular expressions
  - Manual editing of XML documents (problems with copying and pasting)
  - Unconstrained markup and data
- **A brief *introduction* to XML security tools and capabilities**
  - Canonicalization
  - XML Digital Signatures (Integrity)
  - XML Encryption (Confidentiality)



You  
are  
here

# Canonical XML

# Are these the Same?

```
<Publisher>Harper & Row</Publisher>
```

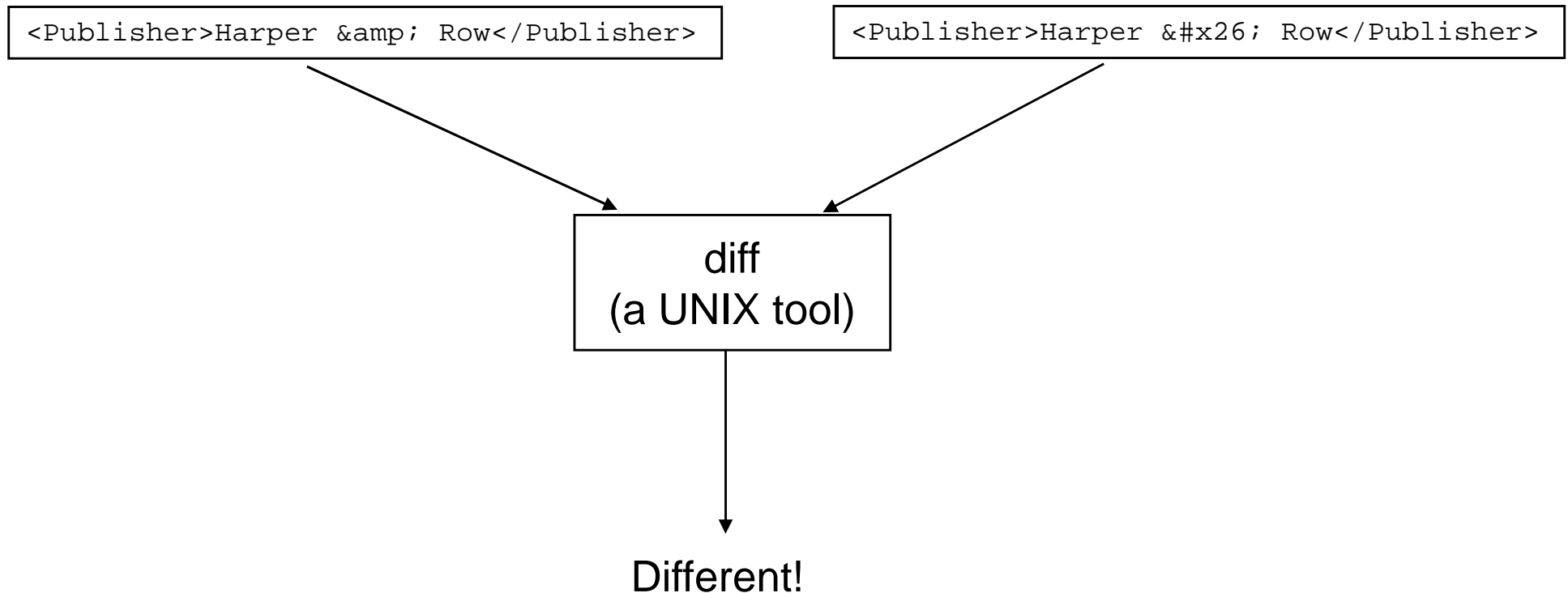
```
<Publisher>Harper &#x26; Row</Publisher>
```

The hex value for the ampersand sign is 26.

```
<Publisher><![CDATA[Harper & Row]]></Publisher>
```

All 3 forms indicate that the publisher is Harper & Row. So, intuitively, they seem to be the same. However, non-XML-aware tools would not be able to recognize that they are the same.

# Comparing XML Documents Using a Non-XML-Aware Tool

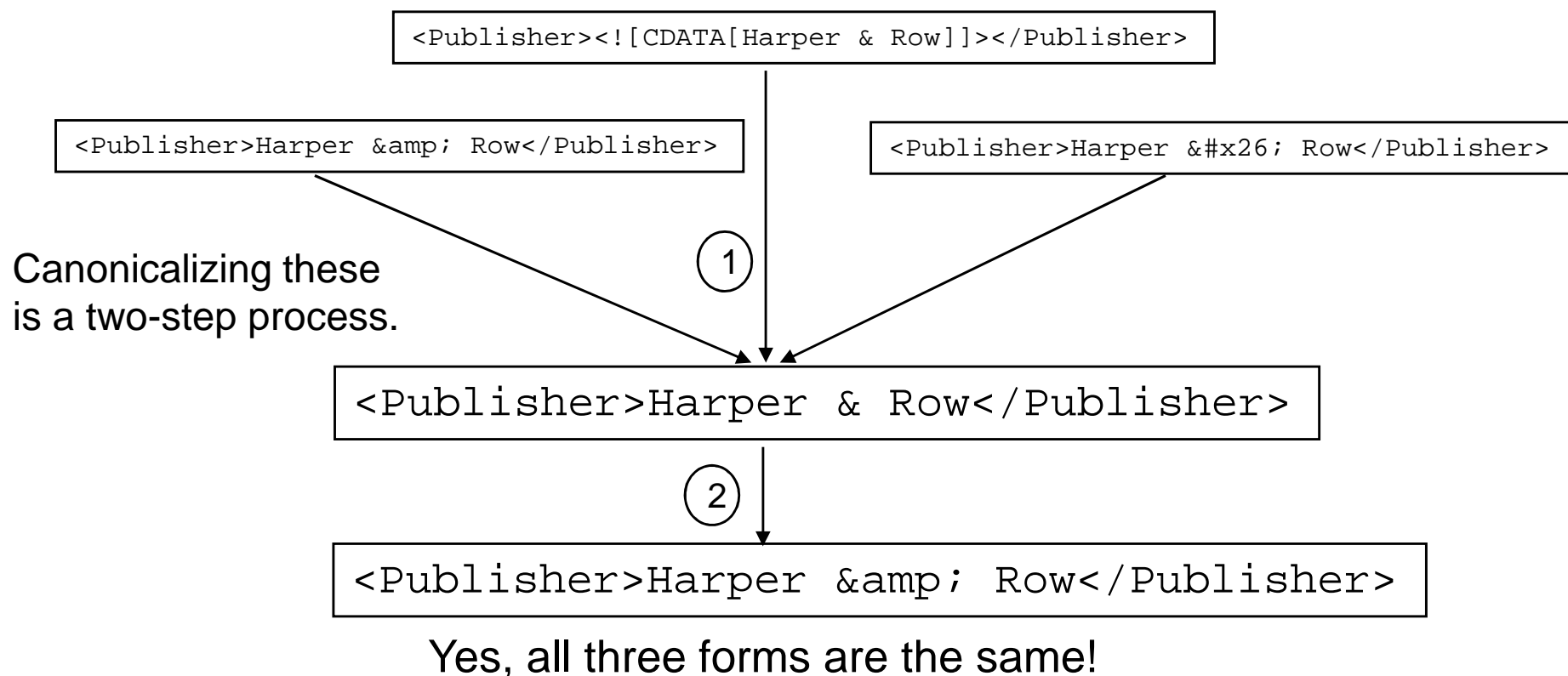


# Purpose of Canonical XML

---

- **The purpose of canonical XML is to define a standard (canonical) format of an XML document.**
- **Thus, to determine if two XML documents are the same, we convert the two XML documents to their canonical format. If their canonical formats are identical, byte for byte, then the two XML documents are the same.**

# Canonical Form



Canonicalization algorithm:

- Replace all entity references with their replacement text.
- Replace all character entity references with their replacement text.
- Replace CDATA sections with their content.
- Then, replace all illegal characters (e.g., &) with an entity reference

# Canonicalization Algorithm

Step 1. Encode the document in UTF-8

Step 2. Change each line break to a single linefeed.

Step 3. Normalize attribute values:

- replace character and entity references by their replacement text
- replace tabs, carriage returns, and linefeeds with a single space
- for all non-CDATA type attributes trim all leading/trailing spaces

Step 4. Replace character and entity references by their replacement text.

Step 5. Replace CDATA sections by their content.

Step 6. Delete the XML declaration.

Step 7. Convert empty tags to start-tag end-tag.

Step 8. Delete blank lines before and after the root element.

Step 9. Normalize white space inside tags:

- Example: `<cost      currency = "USD">` is normalized to: `<cost currency="USD">`

Step 10. Change all attribute value delimiters to double quote marks.

Step 11. Replace all illegal characters in attribute values and element content with entity references

Step 12. Sort the attributes.



# Canonicalizer Tool

---

- **Oxygen XML has a canonicalizer tool.**  
Tools >> Canonicalize ...
- **Apache provides a canonicalizer tool with their XML security tool.**

# Resources

---

- My tutorial on canonicalization: see the papers folder, in there is a Powerpoint document, *canonical-xml.ppt*
- The specification for the *Canonical XML Version 1.1* is available at <http://www.w3.org/TR/xml-c14n11/>

# XML Digital Signature

<http://www.w3.org/TR/xmlidsig-core/>

---

File Edit Find Project Perspective Options Tools Document Window Help

XPath 2.0

stripped-BookCatalogue.xml x canonicalized-stripped-BookCatalogue.xml x ...onicalized-stripped-BookCatalogue.xml x

```

1 <?xml version="1.0"?>
2 <BookCatalogue>
3   <Book ID="B000000001">
4     <Title>Huckleberry Finn</Title>
5     <Author>Mark Twain</Author>
6     <Date>1877</Date>
7     <ISBN>0-440-34319-4</ISBN>
8     <Publisher>Dell Publishing Co.</Publisher>
9   </Book>
10  <Book>
11    <Title>The First and Last Freedom</Title>
12    <Author>J. Krishnamurti</Author>
13    <Date>1954</Date>
14    <ISBN>0-00-084821-7</ISBN>
15    <Publisher>Harper & Row</Publisher>
16    <Publisher>Harper & Row</Publisher>
17    <Publisher>Harper & Row</Publisher>
18    <Publisher>Harper & Row</Publisher>
19  </Book>
20  <Signature><value>"http://www.w3.org/2000/09/xmldsig#">
21    <SignatureInfo><value>"http://www.w3.org/2000/09/xmldsig#">
22    <CanonicalizationMethod><value>"http://www.w3.org/TR/2001/REC-xml-c14n/20011219/#<value>"http://www.w3.org/2000/09/xmldsig#">
23    <SignatureMethod><value>"http://www.w3.org/2000/09/xmldsig#rsa-sha1" xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
24    <Reference><value>"http://www.w3.org/2000/09/xmldsig#">
25    <Transformations><value>"http://www.w3.org/2000/09/xmldsig#">
26    <Transformations><value>"http://www.w3.org/2000/09/xmldsig#canonicalization" xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
27    </Transformations>
28    <DigestMethod><value>"http://www.w3.org/2000/09/xmldsig#sha1" xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
29    <DigestValue><value>"http://www.w3.org/2000/09/xmldsig#sha1B4C231WpTt0SmV80003Secc"</DigestValue>
30    </Reference>
31    </SignatureInfo>
32    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
33    <SignatureInfo><value>"http://www.w3.org/2000/09/xmldsig#">
34    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
35    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
36    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
37    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
38    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
39    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
40    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
41    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
42    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
43    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
44    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
45    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
46    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
47    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
48    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
49    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
50    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
51    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
52    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
53    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
54    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
55    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
56    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
57    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
58    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
59    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
60    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
61    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
62    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
63    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
64    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
65    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
66    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
67    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
68    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
69    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
70    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
71    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
72    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
73    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
74    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
75    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
76    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
77    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
78    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
79    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
80    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
81    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
82    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
83    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
84    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
85    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
86    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
87    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
88    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
89    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
90    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
91    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
92    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
93    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
94    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
95    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
96    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
97    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
98    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
99    <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">
100   <SignatureValue><value>"http://www.w3.org/2000/09/xmldsig#">

```

A digital signature has been added to the canonicalized BookCatalogue XML document

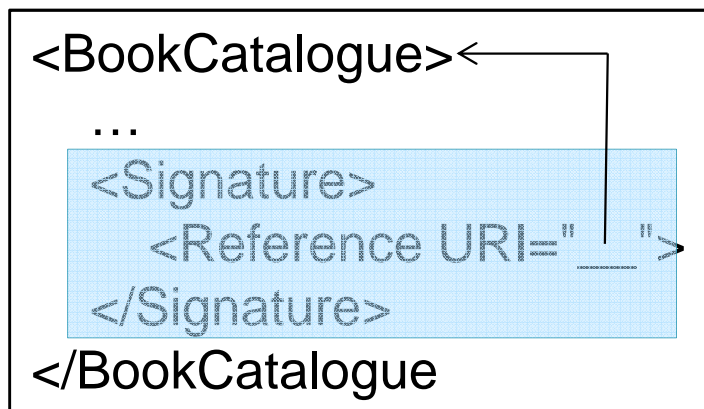
# 3 Methods of Signing

---

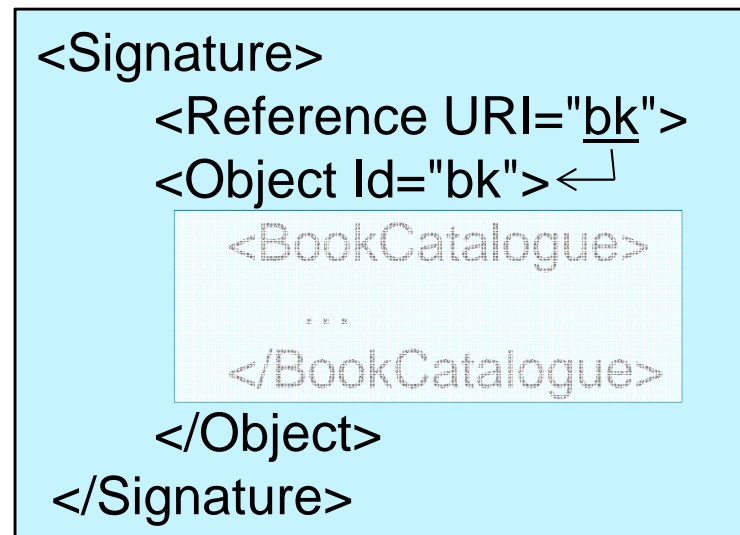
- In the example on the previous slide the digital signature was stuffed inside the XML document, at the bottom of the root element. That is called an enveloped digital signature.
- An enveloping digital signature has the digital signature as the root element, and your XML document is stuffed inside one of its elements.
- A detached digital signature is separate from the XML document – in a totally separate document, or in a header section.

# Contents of XML Signature

- The root element of the XML Signature is the **<Signature>** element
- The signed element is referenced through a **<Reference>**.
- The **<Object>** is used only in Enveloping XML



Enveloped



Enveloping

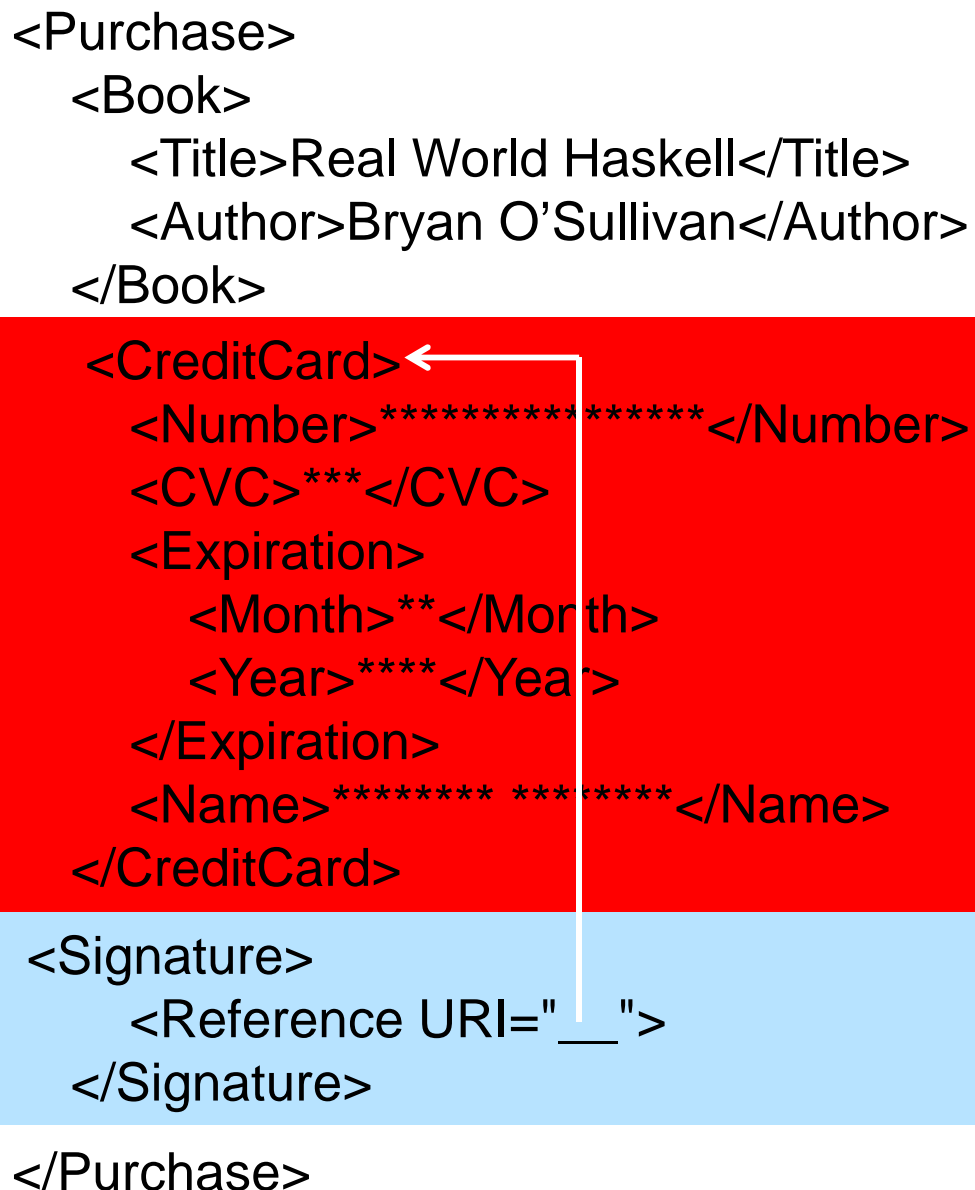
# Sign only a Portion of the XML

- You do not have to sign the entire XML document
- You may sign only a portion of it
- In this document only the credit card data is signed →

```

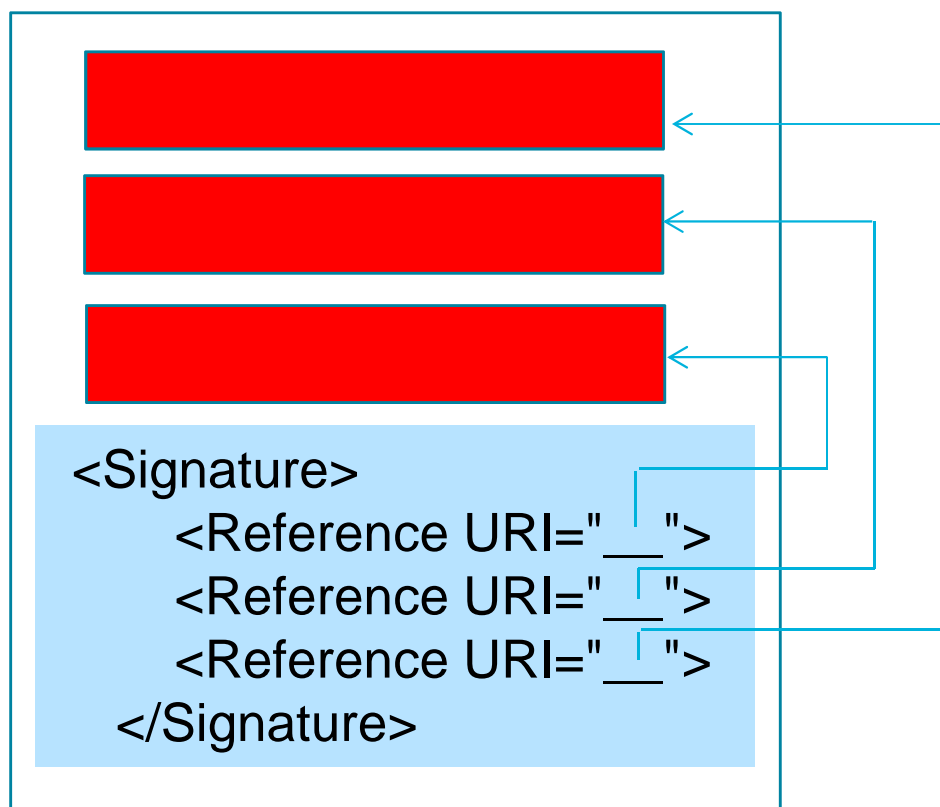
<Purchase>
  <Book>
    <Title>Real World Haskell</Title>
    <Author>Bryan O'Sullivan</Author>
  </Book>
  <CreditCard>
    <Number>*****</Number>
    <CVC>***</CVC>
    <Expiration>
      <Month>**</Month>
      <Year>****</Year>
    </Expiration>
    <Name>*****</Name>
  </CreditCard>
  <Signature>
    <Reference URI="__">
  </Signature>
</Purchase>

```



# Sign Several Parts

- You can sign multiple parts of the XML document
- There is a `<Reference>` element for each part





# Sign Whole Document

---

- If the entire XML document is signed then there is one **<Reference>** element and the value of the URI attribute is empty:

`<Reference URI="" />`

# XML Encryption

<http://www.w3.org/TR/xmlenc-core/>

---

# Great Tandem

- XML DigSig and XML Encryption are meant to be used together.
- Use XML Encryption to encrypt the data (*Confidentiality*)
- Use XML DigSig to verify that the data wasn't altered (*Integrity*)
- WS-Security = XML DigSig + XML Encryption

# Why use XML Encryption?

- Secure HTTP (i.e. https or SSL) can be used to encrypt data exchanges, so why use XML Encryption?
- SSL encrypts the data during the exchange, but once the data gets to the recipient it becomes exposed (“in the clear”).
- Suppose the data is processed by several intermediaries before arriving at its final destination. You may want to keep certain data—such as credit card data, passwords, usernames—encrypted until the final destination. By using XML Encryption the intermediate nodes will not be able to see the sensitive data.
- Encrypting the data is called *message-level encryption*. Contrast with SSL, which encrypts the connection and is called *network-level encryption*.

XML Encryption is useful for:

- Encrypting a part of a document
- Keeping a document (or parts of the document) encrypted across more than one point-to-point exchange

Example: Consider this document:

```
<purchaseOrder>
  <Order>
    <Item>book</Item>
    <Id>123-958-74598</Id>
    <Quantity>12</Quantity>
  </Order>
  <Payment>
    <CardId>123654-8988889-9996874</CardId>
    <CardName>visa</CardName>
    <ValidDate>12-10-2004</ValidDate>
  </Payment>
</purchaseOrder>
```

With XML Encryption we can encrypt:

- The whole document
- An element and its content
- An element's content

In the above document it is important to keep the credit card data confidential. It's okay to expose the item purchased.

The <Payment> element and its content has been encrypted:

```
<?xml version='1.0' ?>
<PurchaseOrder>
  <Order>
    <Item>book</Item>
    <Id>123-958-74598</Id>
    <Quantity>12</Quantity>
  </Order>
  <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'
    xmlns='http://www.w3.org/2001/04/xmlenc#'>
    <CipherData>
      <CipherValue>A23B45C564587</CipherValue>
    </CipherData>
  </EncryptedData>
</PurchaseOrder>
```

Prior to all this occurring there will be exchanges between the sender and receiver regarding what keys to use to perform the encryption and decryption.

# Good Article on XML Encryption

---

<http://www.ibm.com/developerworks/xml/library/x-encrypt/>

# Resources

---

- My tutorial on XML digital signatures and XML encryption: see the papers folder, in there is a Powerpoint document, *How-to-transfer-XML-documents-securely-with-integrity.pptx*
- The specification for the *XML Encryption Syntax and Processing* is available at <http://www.w3.org/TR/xmlenc-core/>
- The specification for the *XML Signature Syntax and Processing* is available at <http://www.w3.org/TR/xmldsig-core/>



# Summary

---

# Summary

---

- **XML is just text.**
- **It has no inherent security.**
- **Security is bolted onto XML.**
- **Just because an XML document validates against an XML Schema, doesn't mean it is risk-free.**
- **There may be hidden markup lurking inside.**
- **Unused namespaces may be exploited to carry malicious code or sensitive data.**

# Summary (concluded)

---

- It is not necessary for an XML document to be large to result in a DoS attack – entities may be expanded by the parser and result in huge documents.
- Be careful constructing regular expressions.
- Poorly constructed regexes may be exploited by attackers to carry out a ReDoS attack.
- Copying and pasting into XML may introduce errors into your XML.