SoK: XML Parser Vulnerabilities

Christopher Späth
Ruhr-University Bochum

Christian Mainka
Ruhr-University Bochum

Vladislav Mladenov
Ruhr-University Bochum

Jörg Schwenk
Ruhr-University Bochum

Abstract

The Extensible Markup Language (XML) has become a widely used data structure for web services, Single-Sign On, and various desktop applications. The core of the entire XML processing is the XML parser. Attacks on XML parsers, such as the Billion Laughs and the XML External Entity (XXE) Attack are known since 2002. Nevertheless even experienced companies such as Google, and Facebook were recently affected by such vulnerabilities.

In this paper we systematically analyze known attacks on XML parsers and deal with challenges and solutions of them. Moreover, as a result of our in-depth analysis we found three novel attacks.

We conducted a large-scale analysis of 30 different XML parsers of six different programming languages. We created an evaluation framework that applies different variants of 17 XML parser attacks and executed a total of 1459 attack vectors to provide a valuable insight into a parser's configuration. We found vulnerabilities in 66 % of the default configuration of all tested parses. In addition, we comprehensively inspected parser features to prevent the attacks, show their unexpected side effects, and propose secure configurations.

1 Introduction

The Extensible Markup Language (XML) is a wide spread data structure used in many application areas ranging from desktop office tools which use it to save their documents (**.docx*), to XML-based databases (MarkLogic, eXist), and web protocol standards (SAML, SOAP).

On a technical level, the parser translates an input byte-stream into an XML document that can be accessed by APIs in different programming languages.

Security of XML parsing. By adding a Document Type Definition (DTD) directly on top of the XML document, the parser behavior can be influenced. Originally designed to define the structure (grammar) of an XML document, it also enables various attacks, such as Denial-of-Service (DoS), Server Side Request Forgery (SSRF), and File System Access (FSA).

In 2002, Steuck discovered the powerful XML External Entity (XXE) attack on XML parsers that allows FSA [60]. Leading companies like Google [15], Facebook [59, 53], Apple [8] and others [63, 9, 16, 17] have been recently affected by this attack.

The Open Web Application Security Project (OWASP) and other resources [47, 46] [71] only partially list vulnerabilities and slightly consider countermeasures. Morgan [40] provides till date the most complete compilation of available attack vectors. A systematic sampling of 13 parsers was conducted recently [57], however, with only one prevalent kind of FSA and DoS attack within scope. Attacks relying on the FTP [41] and netdoc protocol [22], as well as several bypasses [74] and novel attacks such as schemaEntity or XML Inclusion (XInclude) based SSRF are not addressed in any of these sources.

Systematic Parser Analysis. We contribute a comprehensive security analysis framework of 30 XML parsers in six popular programming languages: Ruby, .NET, PHP, Java, Python, Perl. We identify each parser's default behavior by using 17 core tests. This corresponds to all known attack vectors. Based on them, we introduce a metric enabling the comparison of all parsers regarding the security, by computing a Base Vulnerability Score (BVS).

Complex Attack Prevention. Finding countermeasures can be tedious, since the parser's documentation is outdated and a thorough inspection of the source code is necessary. We extended our core tests with parser-specific tests, to investigate the implication of security relevant parser features and their interaction with each other on the overall security. This results in a total of 1459 tests.

Contribution.

- ⋆ We systematically discuss the so-far largest number of state-of-the art XML attacks.
- ⋆ We develop three novel attack vectors.
- ⋆ We create an evaluation framework [10] considering *all* known attacks and apply a comprehensive evaluation to 30 parsers finding 66 % vulnerable in their default configuration.
- ⋆ We propose countermeasures (if possible) against all attacks and propose a secure configuration for each parser.
- ⋆ We apply our framework to Android and reveal a yet undiscovered attack.

2 XML Foundations

XML is a human-readable, structured document, which is subject to a set of rules. Documents adhering to these rules are called *well-formed*. Due to space limitations, we will only discuss two components of XML here - elements and DTDs. We release an extended version [11] and our evaluation framework [10] to support further research in this field.

2.1 XML Elements

Elements structure an XML document as in Listing 1.

```
<data class="products">4</data>
```

Listing 1: Example of an element and an attribute.

This document declares an element *data* with a text content *4* and an attribute *class* with a value *products*.

2.2 Document Type Definition

A DTD defines a grammar to reject invalid user input and is the first component declared in an XML document. In addition, DTDs allow the declaration of storage units, so called *entities*.

Entities. There are four different types of entities: *Internal General Entities* offer a neat way to define a value and reference it arbitrarily often within the document.

```
<!DOCTYPE data [
<!ENTITY a "Arachibutyrophobia">
]>
<data>&a;</data>
```

Listing 2: Example of an Internal General Entity

While processing the document, the parser replaces the reference "&a;" with the term "Arachibutyrophobia".

External General Entities facilitate the inclusion of external files.

```
<!DOCTYPE data [
<!ENTITY a SYSTEM "file:///C:/data/a.txt"> ]>
<data>&a;</data>
```

Listing 3: Example of an External General Entity

The file "a.txt" is a plain text file. The parser retrieves the file and replaces the reference as before.

```
Arachibutyrophobia
```

Listing 4: The content of a.txt

Internal Parameter Entities can be used to instantly modify the value of a General Entity.

```
<!DOCTYPE data [
<!ENTITY % m "majoris">
<!ENTITY a "Arachibutyrophobia %m;"> ]>
<data>&a;</data>
```

Listing 5: Example of an Internal Parameter Entity.

The value of Entity "a" instantly changes to "Arachibutyrophobia majoris".

External Parameter Entities can be used to include additional entity declarations, which are stored remotely.

```
<!DOCTYPE data [
<!ENTITY % extDTD SYSTEM "file:///C:/data/majoris.dtd"
    >
%extDTD;
<!ENTITY a "Arachibutyrophobia %m;"> ]>
<data>&a;</data>
```

Listing 6: Example of an External Parameter Entity.

The corresponding DTD "majoris.dtd" is shown in Listing 12.

```
<!ENTITY % m "majoris">
```

Listing 7: Example of an external DTD.

The parser first fetches the External Parameter Entity "extDTD", makes the declaration of the entity *m* available and finally, replaces this reference.

2.3 Other XML Technologies

XInclude. XInclude facilitates the inclusion of an external (XML) document into the source document. The following example shows how to include a file `other.xml` as a child node of the element *data*.

```
<data>
<xi:include xmlns:xi="http://www.w3.org/2001/XInclude"
      href="other.xml"/>
</data>
```

Listing 8: XML document containing a XInclude instruction.

XSLT. Extensible Stylesheet Language Transformations (XSLT) is commonly used to transform XML documents into other documents and formats, for example, into JSON or PDF [27] [42].

XML Schema. An XML Schema defines grammar using an XML style syntax. The XML Schema standard allows for the inclusion of external Schema files by using the `schemaLocation` and the `noNamespaceSchemaLocation` attributes.

3 Attacker Capabilities

For all attacks described in this paper, we assume that the attacker is able to generate XML messages and that the XML parser can process these messages. We assume a DTD to contain only such grammatical restrictions which do not hinder the attacker. For example, the attacker can create an XML config file and the targeted application parses this file on startup. In case of web applications, the attacker can control or upload an XML file that is processed by the business logic. Hence in our evaluation, we directly invoke the targeted parser and thus obviously control the parsed message.

4 Denial-of-Service

DoS attacks target system resources, such as network, storage, memory or CPU processing [29]. An efficient way to do this is to let the application process a "problem", thereby allocating a huge amount of resources. At the same time the attacker can generate and send the attack vector using much fewer resources. As a result, an offered service is unavailable for benign users or at least responds significantly slower than normal.

4.1 DoS: Recursive Entities

In the following example, the parser receives an XML document which declares two entities calling each other in an infinite loop (see Listing 9).

```
<!DOCTYPE data [
<!ENTITY a "&b;">
<!ENTITY b "&a;"> ]>
<data>&a;</data>
```

Listing 9: XML Infinite Recursion

The parser resolves the entity *a* to a reference of *b* and the entity *b* resolves to a reference of *a*. Therefore, the parser will loop indefinitely and consume CPU resources.

Limitation: Forbidden by XML Specification. The XML specification addresses this problem and forbids the processing of entities which call up each one in a loop. However, our evaluation on Android shows that not all parsers adhere to this rule.

4.2 DoS: Billion Laughs

Internal General Entities can be abused to create an *Exponential entity attack* (*Billion Laughs Attack*) [28]. The attack relies on a nested , but limited, level of entity recursions.

```
<!DOCTYPE data [
<!ENTITY a1 "dos">
<!ENTITY a2 "&a1;&a1;&a1;&a1;&a1;">
...
<!ENTITY a13 "&a12;&a12;&a12;&a12;&a12;"> ]>
<data>&a13;</data>
```

Listing 10: Example of the Billion Laughs Attack

By defining different nesting levels of Internal General Entities, a file of only 200 kilobytes is expanded to several gigabytes (3.5GB) during the parsing process.

Challenge: Thresholds. A number of parsers detect and counteract this attack by implementing a threshold to limit the total number of allowed entity references within a document.

4.3 DoS: Quadratic Blowup

Even if the parser implements such a threshold, there are other ways to execute a DoS attack by using an attack variant known as the *Quadratic Blowup Attack* [64]. Here, a single entity is created containing a large string (e.g. 10 MB). This entity is referenced multiple times within the document in order to achieve a similar result as before. Since less entity references are required than in Listing 10, the threshold limitation can be bypassed.

4.4 DoS with External General Entities

External General Entities can be misused for DoS attacks by pointing to a large external file [60, 47] which will be read during processing. As a result, the target system allocates memory resources. If this file is retrieved over the network, for example. from the attacker's server, the download speed can be reduced in order to improve the impact of this attack and to allocate additional network resources for a longer period of time.

Challenge: Not Applicable to Arbitrary Files. Our investigation shows that all parsers abort processing if the referenced file is not well-formed. We confirmed this for common attack vectors under both UNIX (`/dev/random`, `/dev/urandom` and `/dev/zero`) and Windows `C:/pagefile.sys`. Hence, we conclude that this attack is only feasible with large XML documents.

4.5 Countermeasures

Applicable countermeasures against these attacks are: (1.) **Prevention** by disabling insecure parser features. (2.) **Counteraction** by implementing custom thresholds. (3.) **Limitation** of the allocated resources

5 File System Access

A File System Access (FSA) is utilized to read out arbitrary files from a system. Steuck discovered an XML based FSA attack called XML External Entity (XXE) for the first time in 2002 [60]. XML External Entity (XXE) attacks are instances of injection attacks.

5.1 Classic XML External Entity

XXE attacks misuse a benign feature, namely External General Entities. In contrast to the benign usage of External General Entity, the attacker injects a path to an arbitrary resource (e.g. /etc/passwd) and the contents are returned.

Extension: No External Entity allowed in Attributes. The XML specification forbids the reference of External General Entities in attribute values. Yunusov et al. [74] showed how to adeptly bypass this limitation in 2013, namely an Internal General Entity is referenced within the attribute value. By means of an External Parameter Entity the content of an external resource is included into the Internal General Entity and hence the attribute value. This mimics the same functionality as an External General Entity. We later present a novel attack based on this bypass in Section 5.5.

Challenge: Well-formedness . The content of files referenced by an External General Entity which are not well-formed cause the parser to trigger an exception and abort processing. Some examples of not well-formed replacement text include a start-tag without a corresponding end-tag or characters forbidden in XML, such as the left angle bracket (<). Therefore, it is, for example, not possible to read out certain configuration files (e.g. /etc/fstab) with a classic XXE attack.

5.2 Parameter-based XXE

Internal Parameter Entities can be used to create a `CDATA` element and in this way escape the contents of the file. Consequently, the parser no longer triggers an exception. The first variation of this attack is mentioned by Morgan [40]. We developed a new modified version of this attack. For specific parser configurations, our evaluation results show that our attack vector succeeds when Morgan's attack does not and vice versa. Therefore, the two vectors complement each other.

```
<!DOCTYPE data SYSTEM "http://attacker.com/
     parameterEntity_doctype.dtd">
<data>&all;</data>
```

Listing 11: Our short Parameter-based XXE attack.

The parser first retrieves an external DTD with the contents as in Listing 12.

```
<!ENTITY % start "<![CDATA[">
<!ENTITY % file SYSTEM "file:///etc/fstab">
<!ENTITY % end "]]>">
<!ENTITY all '%start;%file;%end;'">
```

Listing 12: An external DTD contains further Entities.

In the example shown, three parameter entities are used: (1.) *start* – begins the escape sequence. (2.) *file* – contains the content of the referenced file - all characters are escaped; hence, the content is well-formed. (3.) *end* – closes the escape sequence. The Internal General Entity named *all* orders the Internal Parameter Entities (*start, file* and *end*). Parameter Entities can be used exclusively in an external DTD within General Entities .

Challenge: No direct Feedback Channel. All previous attacks assume that the XML content is echoed back to the attacker. This is not always the case. In a Single Sign-On system (e.g., SAML), the user sends his SAML token, which is an XML message, to a server and either gets logged in or blocked. In other words: the user receives a *true/false* answer instead of an *echoed* XML message. This scenario is comparable to blind SQL injection attacks where the attacker does not see the provoked error messages [45].

5.3 Blind XXE

Even if such a direct feedback channel is not available, a FSA attack is still feasible using blind XXE. (This term is coined analogously to blind SQLi). By referencing a non-existent file [54, 65], the parser aborts the processing and displays an error message.

Yunusov et al. [74] invoke an HTTP GET request to the attacker' server and includes the contents of the file with an External Parameter Entity as the path to the resource. Consequently, the content of the file corresponds to the requested file on the attacker's server. The attacker only has to review her log files in order to retrieve the content of the file.

Challenge: Reading out multi-line files. Line termination characters are not allowed as characters of a URL. If the parser does not automatically encode line termination characters, only the first line of a file can be transmitted by using this attack.

5.4 Blind XXE The FTP Protocol

Novikov reported a solution to this challenge (for Java) by relying on the FTP protocol [41]. The attacker simulates an FTP server that requests more commands from the client each time something has been sent. The commands correspond to a line within the document and the line termination character causes these commands to be sent. This way a multi-line file can be read out.

5.5 Blind XXE SchemaEntity

We too present a solution to this challenge with a novel Blind XXE attack called *schemaEntity*. We found three variations of this attack by using the (1.) *noNamespaceSchemaLocation* attribute, (2.) the *schemaLocation* attribute or (3.) an XInclude instruction.

Our attack relies on three building blocks. (1.) **Inclusion**: Parameter Entities are used to include an external resource into an attribute value [74, 75]. (2.) **Transformation**: The Attribute-Normalization algorithm converts line termination characters into whitespaces [13]. (3.) **Transmission**: XInclude or XML Schema attributes, such as `schemaLocation`, `noNamespaceSchemaLocation` are used to transmit the content (SSRF) to the attacker [40].

We will now discuss, by way of example, an instance of this attack based on the *noNamespaceSchemaLocation* attribute. The vector is shown in Listing 13.

```
<!DOCTYPE data [
<!ENTITY % remote SYSTEM "http://attacker.com/
      external_entity_attribute.dtd">
%remote; ]>
<data xmlns:xsi="http://www.w3.org/2001/XMLSchema-
      instance"
xsi:noNamespaceSchemaLocation="http://192.168.2.31/&
      internal;"></data>
```

Listing 13: SchemaEntity Attack with noNamespaceSchemaLocation attribute

The following listing shows the external DTD which is loaded from the attacker's server.

```
<!ENTITY % payload SYSTEM "file:///etc/passwd">
<!ENTITY % param1 "<!ENTITY internal '%payload;'>">
%param1;
```

Listing 14: External DTD for the schemaEntity Attack

First, we **include** the content of the file in order to store it in the attribute value using parameter entities. Second, we misuse the Attribute-Normalization algorithm to automatically **transform** line termination characters, such as #xD and #xA into whitespaces. This step is key because it enables the transmission of multi-line files. Finally, the transformed file is set as the path of a URL in the *noNamespaceSchemaLocation* attribute and hence the content of the file is **transmitted** to the attacker's server.

This attack requires an XML Schema to be processed. Of course, the attack based on XInclude requires XInclude processing.

5.6 Countermeasures

Applicable countermeasures against these attacks are: (1.) **Prevention** by disabling insecure parser features. (2.) **Filtering** by implementing input validation based on a whitelist/blacklist

6 Server Side Request Forgery

SSRF attacks send - in the context of XML - requests on behalf of the XML parser to other endpoints on the network [39]. Usually, these endpoints are not accessible from the Internet (e.g. they are protected by a firewall). SSRF attacks are used to port scan a host, inject malicious content (e.g. HTTP header injection) [43], use other URLs or steal Windows credentials [40].

6.1 Classic SSRF

The most prevalent SSRF attack, based on a DOCTYPE, has already been implemented in popular scanning tools such as Burp [61]. For our example, we suppose a host 192.168.0.11 on an internal network which offers several operations for remote administration, such as "shutdown". An example of this specific setup is shown in Listing 15.

```
<!DOCTYPE data SYSTEM "http://192.168.0.11/shutdown">
<data>4</data>
```

Listing 15: SSRF attack based on DOCTYPE.

An attacker can remotely invoke the shutdown operaton on this host by letting the parser send the request.

6.2 Innovative SSRF

We found a novel attack vector based on XInclude. The *schemaLocation/noNamespaceSchemaLocation* attributes can also faciliate this attack [40]. Other DTD based techniques are with External General Entities and External Parameter Entities.

Challenge: Parser Features. A parser might implement separate features to deactivate the processing of the DOCTYPE, External General Entities, External Parameter Entities, XML Schema and XInclude. If only a subset of these features is applied to harden the parser, the parser is still vulnerable to SSRF attacks.

Challenge: Firewall/missing HTTP support. A number of parsers do not implement network protocols, and in other scenarios network access is sometimes restricted. We found no suitable solution to this challenge.

6.3 Countermeasures

Applicable countermeasures against these attacks are: (1.) **Prevention** by disabling insecure parser features. (2.) **Filtering** by implementing input validation based on a whitelist/blacklist.

7 Additional Attack Techniques

The XML parsing process may consist of other optional processing steps which might introduce vulnerabilities. In addition to our current research, we also investigated other technologies, and we will shortly highlight them in this section.

XSLT and XInclude. We limited our tests to check the support of XSLT and XInclude processing in XML parsers. If a parser processes XSLT or XInclude, it is potentially vulnerable to all the attacks previously listed, namely DoS, FSA and SSRF.

XML Schema. The attributes `schemaLocation` and the `noNamespaceSchemaLocation` can be misused to conduct SSRF attacks (cf. Section 6).

8 Evaluation Framework

In this section we present our evaluation framework consisting of (1.) the selection of test vectors (2.) the parser selection (3.) the test methodology.

8.1 Selection of Test Vectors

Our test framework consists of 17 core test vectors which we have categorized into four groups.

Collection of known test vectors. Initially, we searched for research results and scientific papers considering XML based attacks and security problems of XML parsers by using different search engines. We also checked for vulnerabilities based on XML related technologies, such as XInclude and XSLT. As a result, we found well-known attacks such as the Billion Laughs and Quadratic Blowup Attack [28, 64], the XXE attack [60], the whitepaper of Morgan et. al. [40] and the bypass to include external content in attribute values [74].

We subscribed to multiple CVE newsletters like US-CERT [1] and Mitre [2] and observed recently reported vulnerabilities related to our topic. Each CVE contains information about the attack goals, used attack vectors and the affected vendor. This approach, however, did not reveal any new insights since all the reported issues were based on already known attacks.

We also subscribed to leading security professionals in the field of XML on Twitter. This way we learned about the Blind XXE attack based on the FTP protocol [41] and a netdoc based XXE attack in Java [22].

Addition of new Test Vectors. Our investigation showed that a DoS recursion attack and a parameter based Billion Laughs attack had not yet been included in any previous test set.

We created two Parameter-based XXE (Parameter-based XXE) test vectors to retrieve multi-line files, one with a direct feedback channel and one for Blind XXE (schemaEntity).

We considered XInclude to conduct SSRF attacks, which was not in the scope of any previous research. We also explicitly used External Parameter Entities as another method to carry out SSRF attacks.

In brief, compared to existing work [40] [57], we created three novel attack vectors and included up to ten new test vectors. Our evaluation framework is therefore comprised of 17 tests in total.

Denial-of-Service. The parser is vulnerable if the entity references are completely expanded.

(1.) Recursion: Define a recursion: a -> b -> a ->...
(2.) Billion Laughs: Exceed a predefined threshold [64, 6] of Internal General Entity references.
(3.) Billion Laughs with Parameter Entities: Exceed a predefined threshold of Internal Parameter Entities.
(4.) Quadratic Blowup: Exceed a predefined threshold [64, 38] of Entity expansion, regarding the total size of the Entity.

File System Access. The parser is vulnerable if the content of the external file is included.

(5.) XXE: Return the content of a file from the system using a direct feedback channel.
(6.) Parameter-based XXE Classic: Return the content of a not well-formed file using a direct feedback channel [40].
(7.) Parameter-based XXE Small: Novel small test vector to achieve the same result.
(8.) Blind XXE schemaEntity: Return the content of a multi-line file. Novel test vector.
(9.) Blind XXE FTP: Transmit the contents of a multi-line file over FTP [41].

Server Side Request Forgery. The parser is vulnerable if a predefined HTTP [3] resource is invoked.

(10.) Doctype: Doctype based [40].
(11.) External General Entity: External General Entity based [60].
(12.) External Parameter Entity: External Parameter Entity based.
(13.) noNamespaceSchemaLocation: XML Schema attribute [40]

[1] `https://www.us-cert.gov/ncas/alerts`
[2] `https://cve.mitre.org`

[3] We limited our tests to the HTTP protocol. The SSRF bible [43] lists other interesting protocols for Java and PHP, which could also be tested.

(14.) schemaLocation: XML Schema attribute [40]
(15.) XInclude: Based on XInclude. Novel test vector.

Additonal Attacks.

(16.) XInclude:The parser is vulnerable if the content of a file from the system is returned to by using a direct feedback channel.
(17.) XSLT: The parser is vulnerable if the XSLT statement is processed.

8.2 Parser selection

We focused our work on the most popular parsers of wide-spread web development programming languages [20] [72], such as .NET, Java, Perl, PHP, Python and Ruby.

.NET. Microsoft resources [36, 35] list XmlReader as one of the recommended ways of parsing XML in .NET [37] and XmlDocument as a parser implementing the DOM API.

Java. Xerces (SAX/DOM) and its predecessor Crimson are widely employed, JDOM, dom4j and w3cdocument are popular among Java developers [67, 24, 19], Oracle (SAX/DOM) supposedly has support for XSLT, which would render it vulnerable to additional attacks [23], Piccolo is a small, non-validating and faster parser than any of the others [44] and KXml has been included because it is also used for parsing XML on Android.

Perl. Although XML::Simple seems by far the most popular parser, the developer [31] discourages its use for new projects. A popular Perl forum lists XML::Twig and XML::LibXml as currently being the best available parsers for Perl [25, 56, 58].

PHP. We selected SimpleXML, DOMDocument and XMLReader because they are part of the standard library [49].

Python. We selected etree, minidom, xml.sax and pulldom. These are included in the standard library [52, 18] lxml [12] is a fast parser based on expat. defusedxml [64]provides secure implementations of the aforementioned parsers. BeautifulSoup [55] has been used in various projects in the past.

Ruby. We selected REXML, included in the standard library, and Nokogiri, a third party module. [50, 73, 68, 70].

Therefore, our evaluation framework comprises of 30 parsers from six programming languages.

8.3 Testing Methodology

We implemented our evaluation framework using unit tests, allowing us to easily verify the findings and re-run all tests against each new version of a parser. All previously listed test vectors (see Section 8.1) are executed on all parsers and are therefore called "core tests".

Base Vulnerability Score (BVS). In order to reflect a parser's vulnerability when factory defaults are being used, we define the BVS as the sum of all core tests. Since each vulnerability adds 1 to this score, the highest possible score is 17. By consulting the BVS, parsers with secure factory defaults can be easily identified and it is possible to compare different parsers.

Additional tests. Many parsers implement unique features which change the processing of a DTD. We identify DTD security related features by thoroughly checking the documentation, API and even the source code of each parser. We contribute "additional tests" which check both the effects on processing if a single feature or multiple features are set simultaneously. Finally, we propose features which can be used to counteract vulnerabilities.

Remaning Vulnerability Score (RVS). After applying the proposed countermeasures, we re-evaluate the security of the parser and summarize these results in the Remaning Vulnerability Score (RVS).

9 Evaluation

In this section we present our evaluation results of 30 tested parsers. Our claims are based on a total of 1,459 tests.

Here we also discuss the implemented countermeasures for each parser. These correspond for DoS in Section 4.5, for FSA in Section 5.6 and for SSRF in Section 6.3.

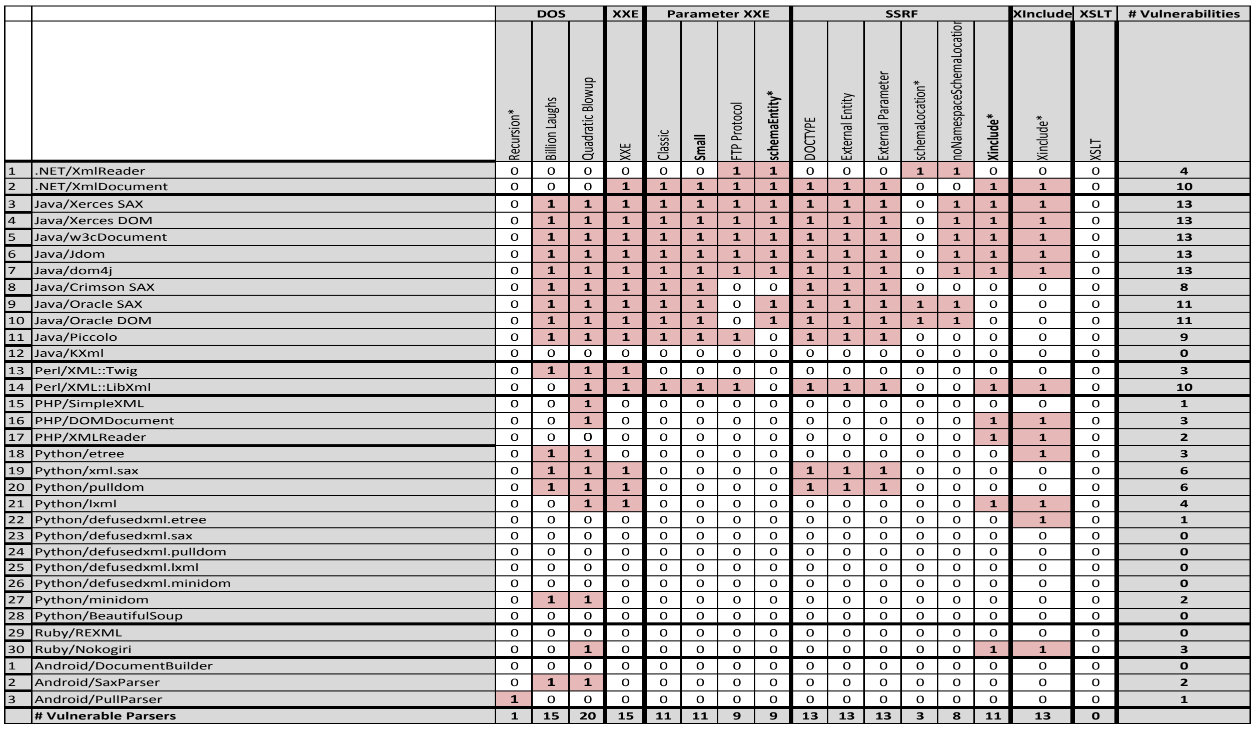
9.1 .NET

Overview. XmlReader ($BVS = 0$) is not vulnerable to any attack vector. XmlDocument ($BVS = 7$) is susceptible to FSA and SSRF attacks.

Countermeasures. XmlDocument should be invoked with an *XmlReader* [62] to mitigate all attacks ($RVS = 0$) (see Listing 19).

Caveats. XmlReader supports XInclude processing. Setting the feature *DtdProcessing* (*DtdProcessing.Parse*) renders the parser insecure.

SchemaEntity: Both XmlReader and XmlDocument are vulnerable if XML Schema processing is enabled (*ValidationType* (*ValidationType.Schema*)).



		DOS			XXE	Parameter XXE				SSRF						XInclude	XSLT	# Vulnerabilities
		Recursion*	Billion Laughs	Quadratic Blowup	XXE	Classic	**Small**	FTP Protocol	**schemaEntity***	DOCTYPE	External Entity	External Parameter	schemaLocation*	noNamespaceSchemaLocation	**XInclude***	XInclude*	XSLT	
1	.NET/XmlReader	0	0	0	0	0	0	1	1	0	0	0	1	1	0	0	0	4
2	.NET/XmlDocument	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1	0	10
3	Java/Xerces SAX	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	13
4	Java/Xerces DOM	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	13
5	Java/w3cDocument	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	13
6	Java/Jdom	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	13
7	Java/dom4j	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	13
8	Java/Crimson SAX	0	1	1	1	1	1	0	0	1	1	1	0	0	0	0	0	8
9	Java/Oracle SAX	0	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0	11
10	Java/Oracle DOM	0	1	1	1	1	1	0	1	1	1	1	1	1	0	0	0	11
11	Java/Piccolo	0	1	1	1	1	1	1	0	1	1	1	0	0	0	0	0	9
12	Java/KXml	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	Perl/XML::Twig	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	3
14	Perl/XML::LibXml	0	0	1	1	1	1	1	0	1	1	1	0	0	1	1	0	10
15	PHP/SimpleXML	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
16	PHP/DOMDocument	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	3
17	PHP/XMLReader	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	2
18	Python/etree	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	3
19	Python/xml.sax	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	6
20	Python/pulldom	0	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	6
21	Python/lxml	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	4
22	Python/defusedxml.etree	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
23	Python/defusedxml.sax	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	Python/defusedxml.pulldom	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	Python/defusedxml.lxml	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	Python/defusedxml.minidom	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	Python/minidom	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2
28	Python/BeautifulSoup	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	Ruby/REXML	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	Ruby/Nokogiri	0	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	3
1	Android/DocumentBuilder	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	Android/SaxParser	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2
3	Android/PullParser	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	# Vulnerable Parsers	1	15	20	15	11	11	9	9	13	13	13	3	8	11	13	0	

Figure 1: Results of our evaluation framework; 1 = parser is vulnerable to the attack; Novel attacks are highlighted in **bold font**; * = When certain prerequisites are met, otherwise default settings;

Blind XXE FTP: Both XmlReader (*DtdProcessing* (*DtdProcessing.Parse*)) and XmlDocument initiate a connection to the FTP server. No data is sent though. This might be due to an implementation flaw of the FTP server in use.

9.2 Java

Overview. All tested Java parsers are vulnerable to instances of DoS, FSA and SSRF ($BVS = 8$) except KXml which is not vulnerable to any attack vector ($BVS = 0$).

Our tests show that w3cDocument, JDOM and dom4j are different implementations of the DOM in Java. All these implementations rely on an underlying parser to process the XML document (here: Xerces DOM). Therefore, the detected vulnerabilities directly correlate to the configuration of this parser.

We confirmed the report of Goldshlager [22] to misuse the netdoc protocol [14] for XXE attacks.

Countermeasures. An *EntityResolver* mitigates FSA and some SSRF attacks by filtering the input. A *DeclHandler* mitigates DoS attacks. Secure implementations of the interfaces *EntityResolver* [33, 30] and *DeclHandler* [32] trigger exceptions for the methods *resolveEntity()*, *externalEntityDecl()* and *internalEntityDecl()* and hence abort the processing. If both countermeasures are applied, the resulting configuration is secure ($RVS = 0$).

Additionally, (i) Xerces, (ii) Oracle and (iii) Piccolo have parser-specific features to mitigate these attacks. (i) Setting the feature *disallow-doctype-decl* (*true*) aborts the processing if a DTD is found, thus mitigating all attacks resulting in a secure configuration ($RVS = 0$). This is an instance of the countermeasure **Prevention** and it mitigates all attacks. Listing 17 shows how to apply this feature on a SAX and DOM parser.

(ii) Setting the feature *EXPAND_ENTITYREF* (*false*) mitigates DoS, FSA and some SSRF attacks. A secure implementation of an *EntityResolver* is still necessary in order to secure the parser. Listing 18 shows how to apply this parser-specific countermeasure.

(iii) Piccolo implements the SAX features *external-general-entities* and *external-parameter-entities* [34]. These can be used as an alternative to an *EntityResolver*. As shown in Listing 17, the features have to be applied analogously in an instance of Piccolo.

Caveats. When applying countermeasures, the available features might not work as expected. A *DeclHandler* does not mitigate a Doctype based SSRF attack.

The features *external-general-entities* and *external-parameter-entities* are part of the SAX API; however, parsers are not required to implement them. **Crimson**, for instance, does not.

Piccolo reports the value of *external-general-entities* for both features, and the feature *external-general-entities* (false) inhibits the loading of an external DTD. This is clearly an implementation flaw.

Xerces implements the features *load-dtd-grammar* and *load-external-dtd* [7]. However, they do not prevent the anticipated attack vectors (e.g. SSRF). A *Security-Manager* prevents External General Entity based FSA attacks. However, it does not mitigate External General Entity based SSRF attacks.

SchemaEntity: Both Xerces and Oracle are vulnerable if XML Schema processing is enabled (Xerces: *validation/schema* (*true*) and *namespaces* (*true*), Oracle: *setValidationMode* (*SCHEMA_VALIDATION*)).

Blind XXE FTP: Both Xerces and Piccolo are vulnerable by default.

9.3 Perl

Overview. XML::Twig ($BVS = 3$) is vulnerable to DoS and FSA attacks. XML::LibXml ($BVS = 8$) is exposed to DoS, FSA and SSRF attacks.

Countermeasures. Setting the feature *NoExpand* (*false*) in XML::Twig mitigates all attacks ($RVS = 0$). The application is shown in Listing 20.

```
$t= XML::Twig−>new();
$t−>parsefile('../../xml_files_windows/dos_core.xml', NoExpand => 1);
```

Listing 16: Applying countermeasures for XML::Twig.

Setting the feature *load_ext_dtd* (*false*) in XML::LibXml mitigates all but DoS attacks ($RVS = 2$). The application is shown in Listing 21.

Caveats. XML::LibXml supports XInclude. The feature *expand_entities* only affects the processing of External General Entities but not External Parameter Entities. The feature *validation* has precedence over *expand_entities*, that is if both features are set simultaneously; then only the feature validation affects the processing.

According to the API of **XML::Twig** [56], features can be spelled either using Java CamelCase style (NoExpand) or Perl style (no_expand). Also, features can be set either in the *new()* constructor or in the method *parsefile()*. There are two implementation flaws regarding these features. First, the same feature affects the processing of entities differently when used in different methods. Second, using a different spelling renders a feature useless. Table 1 exemplary summarizes the behavior for the feature *NoExpand*. Other features are also affected by this problem.

The *file://* protocol is not implemented in Twig; hence an XXE attack must be conducted without it.

Blind XXE FTP: LibXML initiates a connection to the FTP server. However, no data is sent. This might be due to an implementation flaw of the FTP server in use.

9.4 PHP

Overview. XMLReader ($BVS = 0$) is not vulnerable to any attack vector. SimpleXML and DOMDocument ($BVS = 1$) are both susceptible to DoS attacks.

Countermeasures. No countermeasures are available for any of these parsers ($BVS = RVS$).

Caveats. Both XmlReader and DOMDocument support XInclude. Using the proposed countermeasure *disable_entity_loader* (*true*) [47, 48] is neither suitable for SimpleXML nor for DOMDocument because it either does not affect the parsing process or it prevents the parser from loading the input XML document. All tested PHP parsers are based on the libxml2 library and hence offer the corresponding features DTDATTR, DTDLOAD, DTDVALID, NOENT. We advise readers to not set any of these features primarily because it renders the corresponding parser vulnerable to a plethora of attacks. Additionally, in XmlReader there are two implementation flaws.

(1.) Features behave differently for different input files (e.g. XMLReader::VALIDATE and DTDVALID for External General Entities) and (2.) The parser-specific features are not implemented analogously to the libxml2 features (e.g. LOADDTD/DEFAULTATTRS vs DTDLOAD/DTDATTR)

This is counterintuitive because usually one would expect that (1.) the same feature affects the processing of the same underlying data structure in an identical way and (2.) that parser-specific features modify the processing in an identical way as the features of the underlying library.

Blind XXE FTP: All parsers are vulnerable if DTD processing is enabled (DTDLOAD/VALIDATION).

9.5 Python

Overview. defusedxml and BeautifulSoup are not vulnerable to any attack vector ($BVS = 0$). etree and minidom ($BVS = 2$) are susceptible to DoS attacks. lxml is exposed to DoS and FSA attacks ($BVS = 2$). xml.sax and pulldom ($BVS = 6$) are vulnerable to DoS, FSA and SSRF attacks.

Countermeasures. If applicable, an instance of defusedxml [64] should be used ($RVS = 0$). If another parser must be used, we propose the following advice:

Setting the feature *resolve_entities* (*False*) in lxml mitigates all attacks ($RVS = 0$). The application of this countermeasure is shown in Listing 22. Applying a secure *EntityResolver* (see Section 9.2) to xml.sax and pulldom leaves the parser vulnerable DoS attacks ($RVS = 2$). Other countermeasures are not available.

Caveats. Both etree and lxml support XInclude. Setting the feature *no_network* (*false*) in lxml to activate network access renders the parser vulnerable to Parameter-based XXE and SSRF attacks. Disabling the built-in protection for Billion Laughs Attacks (*huge_tree* (*true*)) renders the parser vulnerable to exactly this attack.

Blind XXE FTP: lxml initiates a connection to the FTP server. However, no data is sent. This might be due to an implementation flaw of the FTP server in use.

9.6 Ruby

Overview. REXML ($BVS = 0$) is not vulnerable to any attack vector. Nokogiri ($BVS = 1$) is vulnerable to DoS attacks.

Countermeasures. Setting the features *entity_expansion_limit* and *entity_expansion_text_limit* [69] of REXML can be used to restrict the number and size of Entities even more ($BVS = RVS$).

Nokogiri's underlying libxml2 library is configured identically to the PHP implementation of DOMDocument and SimpleXML. Therefore, the same advice applies. No countermeasures are available ($RVS = 1$).

Blind XXE FTP: Nokogiri initiates a connection to the FTP server if DTD processing is enabled (DTDLOAD/-VALIDATION). However, no data is sent. This might be due to an implementation flaw of the FTP server in use.

10 Android

Six months after we created our evaluation framework, we applied it without any modifications to Android (API 23). XML processing on Android is based on Java [66, 4].

Overview. DocumentBuilder [2] and XmlPullParser [3] ($BVS = 0$) are not vulnerable to any attack vector. SaxParser [5] is vulnerable to DoS attacks ($BVS = 2$).

Countermeasures. There applicable countermeasures from 9.2 are not implemented on Android and hence not applicable.

Caveats. XmlPullParser is based on KXml ($BVS = 0$); however on Android, methods and features specifically for processing DTDs have been implemented. This is particularly infelicitous since setting the feature *PROCESS_DOCDECL* (`true`) renders the parser vulnerable to the DoS Recursion Attack. Therefore, processing only stops if the App crashes or quits. Obviously, this is an implementation flaw. Other attacks, such as FSA or SSRF, are not feasible because neither External General nor Parameter Entities are implemented.

11 Conclusion

DTD attacks are still a prevalent problem in popular XML parsers. We found that multiple parsers are vulnerable to DoS, FSA and SSRF attacks in their default configuration. We also showed, how our attack framework can be used to evaluate new systems by the example of Android and thus revealed a vulnerability that has not been found on any other parser before. The security of other parsers, especially if contained in a closed source system, such as iOS , IBM DataPower or Axway Security Gateway is an interesting research area. Therefore we released an extended version [11] and our evaluation framework [10] to support further research in this field.

Our evaluation is focused on XML, but its conclusion is valid for structured document parsers in general. In order to mitigate such existing risks, we advise the developers of an parser to: (1.) Turn off all security critical features by default. An application developer using the parser must be able to decide if he should turn on the according feature or not. (2.) In addition to the previous aspect, make the enabling of security critical features possible (instead of the need to disable security critical features that are enabled as default) (3.) Document the risks of security critical features and thus make other developers aware of them.

This is especially important when it comes to more recently developed parsers, for example JSON, as the attacks known from XML can be adapted. Examples are: (1.) JSLT is a JavaScript alternative to XSLT [1]. (2.) JSON Include, which is comparable to XInclude [51, 21]. (3.) JSON Schema [26].

This leads to the research question whether JSON (or other) parsers are also vulnerable to DoS, SSRF, and FSA attacks.

Acknowledgements

The research was supported by the *German Ministry of research and Education (BMBF)* as part of the `VERTRAG` and `OpenC3S` research project and by the `FutureTrust` project funded by the European Commission (Grant agreement number: *700542-Future-Trust-H2020-DS-2015-1*).

References

[1] ajaxian: Jslt: A javascript alternative to xslt (2007), http://ajaxian.com/archives/jslt-a-javascript-alternative-to-xslt

[2] android.com: Documentbuilder. https://developer.android.com/reference/javax/xml/parsers/DocumentBuilder.html (2016)

[3] android.com: Documentbuilder. https://developer.android.com/reference/org/xmlpull/v1/XmlPullParser.html (2016)

[4] android.com: Parsing xml data. https://developer.android.com/training/basics/network-ops/xml.html (2016)

[5] android.com: Saxparser. https://developer.android.com/reference/javax/xml/parsers/SAXParser.html (2016)

[6] apache.org: Class securitymanager. https://xerces.apache.org/xerces2-j/javadocs/xerces2/org/apache/xerces/util/SecurityManager.html#getEntityExpansionLimit%28%29 (2010)

[7] apache.org: Parser features. https://xerces.apache.org/xerces2-j/features.html (2010)

[8] apple.com: Office viewer (Aug 2015), http://lists.apple.com/archives/security-announce/2015/Aug/msg00002.html

[9] authors of this submission: Slashdot acknowledgement (2014), http://beta.slashdot.org/journal/1083427

[10] Authors of this submission: Core tests and parser specific tests (source code). Zip, Google Drive (May 2016), https://goo.gl/nfKuaL

[11] Authors of this submission: Security implications of dtd attacks against a wide range of xml parsers. Pdf, Google Drive (May 2016), https://goo.gl/qGMlpw

[12] Behnel: lxml - xml and html with python. http://lxml.de/index.html (2015)

[13] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (xml) 1.0 (fifth edition) (November 2008), http://www.w3.org/TR/2008/REC-xml-20081126/

[14] Byrne: netdoc. http://www.docjar.com/html/api/sun/net/www/protocol/netdoc/Handler.java.html (2015)

[15] detectify: How we got read access on Google's production servers (Nov 2014), http://blog.detectify.com/post/82370846588/how-we-got-read-access-on-googles-production-servers

[16] erpscan.com: SAP Mobile Platform 2.3 – XXE in application import (Aug 2015), http://erpscan.com/advisories/erpscan-15-020-sap-mobile-platform-2-3-xxe-in-application-import/

[17] erpscan.com: SAP NetWeaver 7.4 – XXE (2015), http://erpscan.com/advisories/erpscan-15-018-sap-netweaver-7-4-xxe/

[18] etutorials.org: 5.4 understanding xml. http://etutorials.org/Programming/Python.+Text+processing/Chapter+5.+Internet+Tools+and+Techniques/5.4+Understanding+XML/ (2015)

[19] FilipJirsak: Dom4j. http://dom4j.sourceforge.net/ (2015)

[20] fromdev.com: 5 best programming languages for web developers (Sep 2013), http://www.fromdev.com/2013/09/Best-Programming-Languages-Web-Development.html

[21] github: composer-merge-plugin (2016), https://github.com/wikimedia/composer-merge-plugin

[22] Goldshlager: Pro tip. https://twitter.com/Nirgoldshlager/status/618417178505814016 (2015)

[23] Harold: Sax conformance testing. http://cafeconleche.org/SAXTest/ (2004)

[24] Hunter: Jdom. http://jdom.org/ (2015)

[25] ikegami: Re: best xml parser in 5.18. http://www.perlmonks.org/?node_id=1127488 (2015)

[26] Kashyap: An introduction to json schema (2014), http://crypt.codemancers.com/posts/2014-02-11-An-introduction-to-json-schema/

[27] Kay, M.: XSL Transformations (XSLT) Version 2.0 (Second Edition). W3C proposed edited recommendation, W3C (Apr 2009),

http://www.w3.org/TR/2009/PER-xslt20-20090421/

[28] Klein: Multiple vendors xml parser (and soap/webservices server) denial of service attack using dtd. http://www.securityfocus.com/archive/1/303509 (2002)

[29] Liverani: Defending against application level dos attacks. https://www.owasp.org/images/0/04/Roberto_Suggi_Liverani_OWASPNZDAY2010-Defending_against_application_DoS.pdf (2010)

[30] McLaughlin: Tip: Using an entity resolver. http://www.ibm.com/developerworks/library/x-tipent/index.html (2001)

[31] McLean: Xml::simple. http://search.cpan.org/~grantm/XML-Simple-2.20/lib/XML/Simple.pm (2002)

[32] Megginson: Interface declhandler. http://www.saxproject.org/apidoc/org/xml/sax/ext/DeclHandler.html (2015)

[33] Megginson: Interface entityresolver. http://www.saxproject.org/apidoc/org/xml/sax/EntityResolver.html (2015)

[34] Megginson: Package org.xml.sax. http://www.saxproject.org/apidoc/ (2015)

[35] Meier: Chapter 9 — improving xml performance. https://msdn.microsoft.com/en-us/library/ff647804.aspx (2004)

[36] microsoft.com: Xml processing. https://msdn.microsoft.com/en-us/library/aa478996.aspx#aspnet-jspmig-xmlprocessing_topic3 (2003)

[37] microsoft.com: Xmlreader class. https://msdn.microsoft.com/en-us//library/system.xml.xmlreader%28v=vs.110%29.aspx (2015)

[38] microsoft.com: Xmlreadersettings.maxcharactersfromentities property. https://msdn.microsoft.com/en-us/library/system.xml.xmlreadersettings.maxcharactersfromentities%28v=vs.110%29.aspx (2015)

[39] mitre: Cwe-918: Server-side request forgery (ssrf). http://cwe.mitre.org/data/definitions/918.html (2013)

[40] Morgan: Xml schema, dtd, and entity attacks. http://vsecurity.com/download/papers/XMLDTDEntityAttacks.pdf (2014)

[41] Novikov: Xxe oob exploitation at java 1.7+. http://lab.onsec.ru/2014/06/xxe-oob-exploitation-at-java-17.html (2014)

[42] Onder, R., Bayram, Z.: XSLT version 2.0 is turing-complete: A purely transformation based proof. In: Implementation and Application of Automata, pp. 275–276. Springer (2006)

[43] @ONsec_Lab: Ssrf bible. cheatsheet. https://docs.google.com/document/d/1v1TkWZtrhzRLy0bYXBcdLUedXGb9njTNIJXa3u9akHM/edit# (2014)

[44] Oren: Sourceforge logo sax parser benchmarks. http://piccolo.sourceforge.net/bench.html (2004)

[45] owasp.org: Blind sql injection. https://www.owasp.org/index.php/Blind_SQL_Injection (2013)

[46] owasp.org: Testing for denial of service. https://www.owasp.org/index.php/Testing_for_Denial_of_Service (2013)

[47] owasp.org: Xml external entity (xxe) processing. https://www.owasp.org/index.php/XML_External_Entity_%28XXE%29_Processing (2015)

[48] php.net: libxml_disable_entity_loader. http://php.net/manual/en/function.libxml-disable-entity-loader.php (2015)

[49] php.net: Xml manipulation. http://php.net/manual/en/refs.xml.php (2015)

[50] ruby portal.de: Xml und ruby (Nov 2015), http://wiki.ruby-portal.de/XML_und_Ruby

[51] Python: json-include (2015), https://pypi.python.org/pypi/json-include

[52] python.org: Python and xml. https://wiki.python.org/moin/PythonXml (2012)

[53] Ramadan, M.: How I hacked Facebook with a Word Document (Oct 2015), http://www.attack-secure.com/blog/hacked-facebook-word-document

[54] Rantasaari: Forcing xxe reflection through server error messages. https://blog.netspi.com/forcing-xxe-reflection-server-error-messages/ (2015)

[55] Richardson: Beautiful soup. https://www.crummy.com/software/BeautifulSoup/ (2016)

[56] Rodriguez: Xml::twig. http://search.cpan.org/~mirod/XML-Twig-3.49/Twig.pm (2015)

[57] Sadeeq: Known xml vulnerabilities are still a threat to popular parsers and open source systems. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7272938&tag=1 (2015)

[58] Sergeant: Xml::libxml::parser. http://search.cpan.org/~shlomif/XML-LibXML-2.0121/lib/XML/LibXML/Parser.pod (2015)

[59] Silva, R.: XXE in OpenID: one bug to rule them all, or how I found a Remote Code Execution flaw affecting Facebook's servers (Jan 2014), http://www.ubercomp.com/posts/2014-01-16_facebook_remote_code_execution

[60] Steuck: Xxe (xml external entity) attack. http://www.securityfocus.com/archive/1/297714/2002-10-27/2002-11-02/0 (2002)

[61] Stuttard: Burp suite now reports blind xxe injection. http://blog.portswigger.net/2015/05/burp-suite-now-reports-blind-xxe.html (2015)

[62] Sullivan: Security briefs - xml denial of service attacks and defenses. https://msdn.microsoft.com/en-us/magazine/ee335713.aspx (2009)

[63] threatpost.com: Adobe Patches XXE Vulnerability in LiveCycle Data Services (Aug 2015), https://threatpost.com/adobe-patches-xxe-vulnerability-in-livecycle-data-services/114331

[64] tiran: defusedxml 0.4.1. https://pypi.python.org/pypi/defusedxml/ (2013)

[65] Tran: Advisory: Xxe injection in oracle database (cve-2014-6577). https://blog.netspi.com/advisory-xxe-injection-oracle-database-cve-2014-6577/ (2015)

[66] tutorialspoint: Android - xml parser tutorial. http://www.tutorialspoint.com/android/android_xml_parsers.htm (2016)

[67] Ullenboom: Java ist auch eine Insel, chap. 16.3. Galileo Computing (2011), http://openbook.rheinwerk-verlag.de/javainsel/javainsel_16_003.html#dodtp80ec559d-9ea1-435d-9b81-e786274f1786

[68] Unsung: Xml parsing in ruby (Jan 2012), http://stackoverflow.com/questions/8798179/xml-parsing-in-ruby

[69] usa: Entity expansion dos vulnerability in rexml (xml bomb, cve-2013-1821). https://www.ruby-lang.org/en/news/2013/02/22/rexml-dos-2013-02-22/ (2013)

[70] Vervloesem: Rexml: Processing xml in ruby (Nov 2005), http://www.ml.com/pub/a/2005/11/09/rexml-processing-xml-in-ruby.html?page=1

[71] vsespb: Best xml library to validate xml from untrusted source. http://www.perlmonks.org/?node_id=1104296 (2014)

[72] w3techs.com: Usage of server-side programming languages for websites (Nov 2015), http://w3techs.com/technologies/overview/programming_language/all

[73] yahoo.com: Parse xml using ruby (Nov 2015), https://developer.yahoo.com/ruby/ruby-xml.html

[74] Yunusov: Xml out-of-band data retrieval. https://media.blackhat.com/eu-13/briefings/Osipov/bh-eu-13-XML-data-osipov-slides.pdf (2013)

[75] Yunusov: Xml data retrieval. https://media.blackhat.com/eu-13/briefings/Osipov/bh-eu-13-XML-data-osipov-wp.pdf (2014)

A Appendix

```
factory.setFeature("http://apache.org/xml/features/
    disallow-doctype-decl", true);
```

Listing 17: Applying countermeasures for Xerces.

```
parser.setAttribute(SAXParser.EXPAND_ENTITYREF, false)
    ;
```

Listing 18: Applying countermeasures for Oracle.

```
String filename = "/home/user/someFile.xml";
XmlReaderSettings settings = new XmlReaderSettings();
XmlReader reader = XmlReader.Create(filename, settings);
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(reader);
```

Listing 19: Example of using XmlDocument with XmlReader.

Feature	**new()**	**parsefile()**	**Comment**
NoExpand	working	not working	External General Entity
NoExpand	not working	working	Internal General Entity
no_expand	not working	not working	Internal General Entity

Table 1: Processing of XML::Twig when feature NoExpand is used in different contexts and on different input files.

```
$t= XML::Twig−>new();
$t−>parsefile('../../xml_files_windows/dos_core.xml', NoExpand => 1);
```

Listing 20: Applying countermeasures for XML::Twig.

```
$dom = XML::LibXML−>load_xml(
location => $file,
load_ext_dtd => 0);
```

Listing 21: Applying countermeasures for XML::LibXml.

```
parser = XMLParser(resolve_entities=False)
```

Listing 22: Applying countermeasures for lxml.