Blind XPath Injection

Amit Klein, Director of Security and Research

Abstract

This paper describes a Blind XPath Injection attack that enables an attacker to extract a complete XML document used for XPath querying - without prior knowledge of the XPath query. The attack is "complete" since all possible data is exposed. The attack makes use of two techniques – XPath crawling, and Booleanization of XPath queries.

Using this attack, it is possible to get hold of the XML "database" used in the XPath query. This can be most powerful against sites that use XPath queries (and XML "databases") for authentication, searching, and other uses.

Compared to the SQL injection attacks, XPath Injection has the following upsides:

- Since XPath is a standard (yet rich) language, it is possible to carry the attack 'as-is' for any XPath implementation. This is in contrast to SQL injection where different implementations have different SQL dialects (there is a common SQL language, but it is often too weak).
- The XPath language can reference practically all parts of the XML document without access control restrictions, whereas with SQL, a "user" (which is a term undefined in the XPath/XML context) may be restricted to certain tables, columns or queries. So the outcome of the Blind XPath Injection attack is guaranteed to consist of the complete XML document, i.e. the complete database.

These results enable an automated attack to fit any XPath based application provided that it possesses the basic security hole. Indeed, such pr oof of concept script was written and demonstrated on various XPath implementations.

About XPath

XPath 1.0 [1] is a language used to refer to parts of an XML [6] document. It can be used directly to query an XML document by an application, or as part of a larger operation such as applying an XSLT [2] transformation to an XML document, or applying an XQuery [3] to an XML document.

The syntax of XPath bears some resemblance to an SQL query, and indeed, it is possible to form SQL-like queries on an XML document using XPath. For example, assume an XML document that contains elements by the name "user", each of which contains 3 sub elements - "name", "password" and "account". The following XPath expression yields the account number of the user whose name is "jsmith" and whose password is "Demo1234" (or an empty string if no such user exists):

```
      string(//user[name/text()='jsmith' and
      password/text()='Demo1234']/account/text())
```

Security Implications

XPath is used, among other things, to query XML databases in a style similar to the example above. An XML document serves as the database, and the XPath query is analogous to an SQL query. There are benefits to using an XML database, for example portability, compatibility, re-use of the document, neatness of the document and the query results, and having a structure for the document and the query results.

Various products offer native XML databases with built-in XPath query facility [9]. Among these products are Software AG's Tamino, Apache Software Foundation XIndice, Sleepycat Software Berkeley DbXML, dbXML Group dbXML, An application that uses such a product may be vulnerable to Blind XPath Injection depending on how the XPath query is formulated by the application. In other contexts, XPath can be used directly (i.e. not using a database software), and is natively supported – e.g. in Microsoft's .NET framework and in Macromedia's ColdFusion/MX.
XPath queries are used for search requests, for login processing, for data retrieval, and in short, for flexible, lightweight database tasks.

An attacker, upon spotting an XPath Injection vulnerability in an XPath based application does not need to fully understand/guess the XPath query. Usually, within few attempts, the attacker can generate a "template" query data that can be used for Blind XPath Injection. As of that moment, an automated script (which is implemented, but not provided in this paper) can be used to extract the complete XML document that is the underlying database.

It should be stressed that the attacker does not need to know the exact structure of the XPath query, and moreover, since an XML document has no access control/privilege system associated with it, the attacker is able to extract the document (database) in its completeness -unlike in SQL injection, where the attacker is limited to the privileges of the database account used by the application.

Simple XPath Injection

Consider a Web application that uses XPath to query an XML document and retrieve the account number of a user whose name and password are received from the client. Such application may embed these values directly in the XPath query, thereby creating a security hole.

Here's an example (assuming Microsoft ASP.NET and C#):

```
...
XmlDocument XmlDoc = new XmlDocument();
XmlDoc.Load("...");
...
XPathNavigator nav = XmlDoc.CreateNavigator();
XPathExpression expr =
      nav.Compile("string(//user[name/text()='"+TextBox1.Text+
      "' and password/text()='"+TextBox2.Text+
      "']/account/text())");
String account=Convert.ToString(nav.Evaluate(expr));
if (account=="")
{
      // name+password pair is not found in the XML document –
      // login failed.
      ...
}
else
{
      // account found -> Login succeeded.
      // Proceed into the application.
      ...
}
```

When such code is used, an attacker can inject XPath expressions (much like SQL injection), e.g. provide the following value as a user name:

```
' or 1=1 or ''='
```

This data causes the semantics of the original XPath to change, so that it always returns the first account number in the XML document. Such attack is called “XPath Injection” [4], an analogy to the “SQL injection” attacks, and results in having the attacker logged in (as the first user listed in the XML document), although the attacker did not provide any valid user name or password.

Although this attack grants the attacker access to the application, it does not necessarily grant him/her with access as the most privileged account. In fact, except for logging in, the attacker has acquired no information about the XML “account database”. In some cases, it might be possible to obtain information from the system if the XPath expression returns data from the XML document that is later displayed to the user (attacker). For example, the above code could have displayed the account number of the logged-in account in the HTML response. In this case, the attacker can further manipulate the XPath query to force the server to return various parts of the document. For example, the following injection (for the username parameter, while providing the value “`NoSuchPass`” as the password) would return the first element of the XPath nodeset designated by the node-set *`P`*:

```
NoSuchUser'] | P | //user[name/text()='NoSuchUser
```

This will form the following XPath query:

```
string(//user[name/text()='Foobar'] |
      p | //user[name/text()='NoSuchUser' and
      password/text()='NoSuchPass']/account/text())
```

Since the first and the third predicates are always false, the query returns the *string-value* [1] of the node-set *`P`*.

Note, however, that in this case, while it is possible to extract most of the data from the database, some prior knowledge on the XPath query is needed. Without knowing the exact format of the query, it's very hard to figure out exactly how to form the injection string. This is especially true if error messages are suppressed. Note it is also impossible, in this manner, to extract non-nodeset expressions (e.g. `count()` and `name()`).

There are many cases in which no XML data is sent directly to the user. In that case, the above attack (and similar attacks) is not applicable.

Blind XPath Injection

It is possible to take a more systematic approach to the XPath Injection problem. This approach is called "blind injection" (the foundation to which are laid in [5], in the SQL injection context). It assumes more or less nothing on the structure of the query except that the user data is injected in a Boolean expression context. It enables the attacker to extract a single bit of information per a single query injection. This bit is realized, for example, as "Login successful" or "Login failed".
This approach is even more powerful with XPath than it is with SQL, due to the following characteristics of XPath:

- XPath 1.0 is a standard language. SQL has many dialects all based on a common, relatively weak syntax.
- XPath 1.0 allows one to query all items of the "database" (XML object). In some SQL dialects, it is impossible to query for some objects of the database using an SQL `SELECT` query (e.g. MySQL does not provide a "table of tables").
- XPath 1.0 has no access control for the "database", while in SQL, some parts of the database may be inaccessible due to lack of privileges to the application.

The technique we use is as follows:
We first show how to crawl an XPath document, using only scalar queries (that is, queries whose return type is "string", "numeric" or "Boolean"). The crawling procedure assumes no knowledge of the document structure, yet at its end, the document, in its completeness is reconstructed.

We then show how a scalar XPath query can be replaced by a series of Boolean queries. This procedure is called a "Booleanization" of the query. A Boolean query is a query whose result is a Boolean value (true/false). So in a Booleanization process, a query whose result type is string or numeric is replaced with a series of queries whose result type is Boolean, and from which we can reconstruct the result of the original string or numeric query. This is explained in detail below.

Finally, each Boolean query can be resolved by a single "blind" injection. That is, we show how it is possible to form an injection string, including the Boolean query, that, when injected into an XPath query, causes the application to behave in one way if the Boolean query resolves into "true", and in another way if the query resolves into "false". This way, the attacker can determine a single bit – the Boolean query result.

The novelty in this approach towards XPath Injection is that it does not require much prior knowledge of the XPath query format unlike the “traditional” approach described above. It does not require that data from the XML document is embedded in the response and that the whole XML document is eventually extracted, regardless of the format of the XPath query used by the application. It uses only a difference in the application behavior resulting from a difference in the XPath query return value to extract a single information bit.

XPath Crawling

Provided there is a path for the current element (*path*), we can easily proceed.

- The recursion starts with `path=''`.
- The element name (including namespace name) is given as `name(path)`, and the namespace value is `namespace-uri(path)`.
- The number of attributes of the element is `count(path/attribute::*)`, the *N*th attribute name is `name(path/attribute::*[position()=N])`, the *N*th attribute namespace value is `namespace-uri(path/attribute::*[position()=N])` and the *N*th attribute value is `path/attribute::*[position()=N]`.

There are 4 types of sub-nodes: text, processing-instruction (abbreviated as “PI”), element and comment. An annoying quirk of XPath 1.0 is that it’s possible to enumerate over sub-nodes, but it is impossible to directly retrieve the node type. However, there’s a simple work around, which involved some bookkeeping.

First, we need to know the number of various sub-nodes:

> `count(path/child::node())` - the count of all the nodes for the given *path*.
>
> `count(path/child::text())` - number of text fields children (up to 1...).
> `count(path/child::comment())` - number of comment nodes.
> `count(path/child::*)` - the number of element children.
> `count(path/child::processing-instruction())` - the number of PI nodes.

Now, the trick is to maintain at all times the current number of nodes from each type encountered so far. This enables us to know what index from each type is a candidate for the next node. We therefore have up to 4 candidates. It is mandatory for this technique not to list a candidate from a type that is exhausted – hence the need to first find out how many sub-nodes are expected from each type. Suppose we have counters *i*, *j*, *k* an *l* for text sub-nodes, comment sub-nodes, element sub-nodes and PI sub-nodes, respectively. And let us assume, for simplicity, than none of the types are exhausted, i.e.

```
i < count(path/child::text()) and
j < count(path/child::comment()) and
k < count(path/child::*) and
l < count(path/child::processing-instruction())
```

Now, we look at the following node-set expression:

```
path/child::node()[position()=((i+j+k+l+1)] |
path/child::text()[position()=(i+1)]
```

If the next sub-node (number *i+j+k+l*+1) is indeed the next text sub-node (number *i*+1), then the two node-sets that are united are in fact the same node-set, which contains exactly one node. However, if the next sub-node is not a text sub-node, then the union will produce two nodes. Therefore, we can know if the next sub-node is a text sub-node, by asking:

```
count(path/child::node()[position()=((i+j+k+l+1)] |
path/child::text()[position()=(i+1)])=1
```

A true value indicates that the next sub-node is a text sub-node, and a false value indicates that it is not. Likewise, the next 3 queries can be used to know if the sub-node is a comment sub-node, an element sub-node, or a PI sub-node:

```
count(path/child::node()[position()=((i+j+k+l+1)] |
path/child::comment ()=(j+1)])=1

count(path/child::node()[position()=((i+j+k+l+1)] |
path/child::*()[position()=(k+1)])=1

count(path/child::node()[position()=((i+j+k+l+1)] |
path/child::processing-instruction()[position()=(l+1)])=1
```

A text sub-node value is simply:

```
path/child::node()[position()=N]
```

A comment sub-node has the following data:

```
path/child::node()[position()=N]
```

In order to process an element sub-node, a recursion is needed (we implemented DFS). The procedure we describe here is thus called with the path

```
path/child::node()[position()=N]
```

And a processing instruction sub-node has a name:
`name(path/child::node()[position()=N])`, and data
`path/child::node()[position()=N]`

Booleanization of XPath Scalar Queries

We now describe how a query whose return type is string or number can be replaced with a sequence of XPath queries whose return type is Boolean.

Let us assume that `Q` is a numeric XPath query. Let us further assume that we handle 32-bit signed integers (which is sufficient in a 32 bit architecture). We first extract the sign bit of `Q`:

```
(Q>=0)
```

True value indicates that `Q` is positive (or zero), and false value indicates that `Q` is negative. We then use `-Q` (instead of `Q`) if it is negative, and proceed. Assuming that `Q` is positive, we extract its 31 bits, also assuming we already know the most significant N bits (of the 31 bits), we can then find the next bit: Let K be the number formed by the known N high bits, then the N+1 bit is set to 1, then the rest, 30-N bits, set to 0.

```
((Q-K)>=0)
```

Yields true if the N+1 bit (from the left) is 1, and false if that bit is 0.

Thus we can reconstruct a positive Q with 31 Boolean queries. We start with N=0 and iteratively extract the next bit until we get to N=30, inclusive.

A string query S is first factored into bytes (or, more accurately, Unicode symbols), as follows:

First, we query the string length using the XPath `string-length` function, which is a numeric query (covered above):

```
(string-length(S))
```

Then we can iterate over the symbols, reducing the query into a series of one byte (symbol) queries:

```
(substring(S,N,1))
```

Now, a single byte/symbol query B is in turn reduced into Boolean queries as follows: Let us assume that the list of possible symbols (excluding the double quote mark) in the document is known (denote it by C), and that the list's length is L. L is hopefully small, e.g. if it is known that the XML document is in fact comprised of printable ASCII characters, including CR, LF and HT, excluding double quotes, then L is 97. We index each possible symbol, starting from 0 and going to (L-1). Denote by K=ceiling(log2(L)) – this is the number of bits that are required to determine the symbol. Now, we prepare K strings of length L. The Nth string is a list of bits in position N of the symbols.Let us designate the Nth string as C_N.

First, we ensure that the byte is not a double quote:

```
(B='″')
```

If the expression returns true, then the byte is simply the double quote mark.
If the expression is false, we proceed as follows:
The Nth bit is extracted as following

```
(number(translate(B,″C″,″C_N″))=1)
```

If this yields true, then the Nth bit is 1, and if it yields false, the Nth bit is 0.
Note, we must exclude the double quote mark from C, or else the XPath syntax will be broken. Thus we are able to extract string queries using Boolean queries.

Mounting a Blind XPath Injection

Consider the following XPath query string (from the above C# code):

```
"string(//user[name/text()='"+TextBox1.Text+
"' and password/text()='"+TextBox2.Text+"']/account/text())"
```

Now, injecting a username of

```
NoSuchUser' or E or 'foobar'='
```

forces the predicate to yield true if `E` is a Boolean expression that evaluates to true, and to yield false if `E` is evaluated as false. Therefore, the application logs in successfully if `E` is true, and rejects the login attempt if `E` is false. We now have a mechanism that extracts a single bit from the system – the Boolean XPath expression `E`. As we saw above, this suffices to extract the entire underlying XML document.

In fact, a Perl script, implementing an algorithm that demonstrates this attack, was written by Ory Segal and Ronen Heled (both of Sanctum Inc). This algorithm successfully extracted a short XML document (from an XPath application provided by Chaim Linhart of Sanctum Inc), by recursively DFS crawling the document and sending Booleanized queries in form of blind XPath Injections. As can be analyzed from the above discussion, the algorithm has runtime complexity (number of queries) of *O* (*document size*). Query size is bounded (up to a constant overhead) by the size of the document effective alphabet, *L* (e.g., for ASCII only documents – 97).

It is interesting to note that there are very few variants of the above injection that cover most of the possible injection "space". Specifically, it does not require (in most cases) knowing exactly the structure of the XPath query. Therein lies the strength of the blind XPath Injection attack.

Shortcomings of the Current Algorithm

- *Non ASCII symbols and special characters*

  In the Booleanization of a single symbol query, we need to send the server an XPath query consisting of all possible symbols expected in the XML document. This may be problematic in some cases – the server may reject such requests due to specific character restrictions (e.g. "<", CR, LF, HT). When ASCII documents are expected, this is not a problem (at least not in ASP.NET 1.0).

- *Query size*

  When the document alphabet size (*L*) is large, the symbol Booleanization queries (whose size is slightly bigger than *L*) may be rejected due to length restrictions (in order to cover a generic Unicode document, some 65000+ symbols would be sent…).

  It is possible to replace the long strings of possible symbols with segments of possible symbols, thus trading off number of queries for query size. In the extreme, it is possible to send a multitude of simple Boolean expressions, such as (assume `s` is a single symbol):

  ```
  (B='s')
  ```

This, of course, affects the runtime complexity – in the extreme case, multiplying it by L.

- *Imperfect reconstruction of the XML document*

  An XML document may contain xmlns attributes (designating namespaces), xml: space attributes (governing how spaces are handled) and xml:lang attributes (designating languages). Also, the XML declaration is not accessible. Furthermore, what is accessible is the logical document, where DTD, default attributes and values, XML entities and CDATA are already expanded.

- *Performance*

  The algorithm described above is not optimized for speed. There are some areas where optimization can be applied, for example, predicting "reasonable" number of bits per integer (usually much less than 31), and parallelizing some queries. The Booleanization of the numeric query can be done in a way that enables sending all queries in parallel, using the numeric division and "mod" operators, and the byte Booleanization and string to byte factoring are definitely parallelizable. Sending several queries in parallel makes much better use of the attacker resources.

XPath 2.0

XPath 2.0 [7] is the next version of XPath, currently at status "Working Draft" (as of February 15th, 2004). XPath 2.0 expands many of the capabilities of XPath 1.0. In theory, with XPath 2.0 an attacker can gain much more than with XPath 1.0:

- XPath 2.0 can access documents by URL and is not restricted to the "current" document, using the `doc` function or the `collection` function [8]. This enables the attacker to retrieve any XML document whose location on the host is known (using a URL with a file scheme format, e.g. `file:///c:/data/database.xml`) and is accessible by the server.
- XPath 2.0 has a function (`string-to-codepoints`, [8]) that converts a string (a single symbol, in our case) to a sequence of numbers that represent the Unicode codes for that string. This simplifies the retrieval of symbols, and hence of strings.

We did not experiment with XPath 2.0.

Defending against XPath Injection

Defending against XPath Injection is essentially similar to defending against SQL injection. The application must sanitize user input. Specifically, the single and double quote characters should be disallowed. This can be done either in the application itself, or in a third party product (e.g. application firewall.)

Testing application susceptibility to XPath Injection can be easily performed by injecting a single quote or a double quote, and inspecting the response. If an error has occurred, then it's likely that an XPath Injection is possible.

Conclusion

We showed that it's possible to extract the complete XML document used in an XPath query, with very little knowledge of the XPath query, and without the need to obtain data from the response (other than different "behavior") using a method we call "Blind XPath Injection". This attack is feasible in terms of runtime, and was implemented in Perl and successfully mounted on some XPath/XML examples.

Applications that embed user data into XPath queries in an insecure manner are, therefore, susceptible to this kind of attack, and the XML documents used by the XPath queries may be compromised. Defense and testing measures are possible to protect applications against XPath Injections.

Acknowledgements

To Chaim Linhart (Sanctum), for triggering this research.
To Ory Segal (Sanctum), for implementing the XPath crawler.
To Ronen Heled (Sanctum), for implementing the Booleanization functions.

References

[1] "XML Path Language (XPath) Version 1.0 - W3C Recommendation, 16 November 1999", http://www.w3.org/TR/xpath
[2] "XSL Transformations (XSLT) Version 1.0 - W3C Recommendation, 16 November 1999", http://www.w3.org/TR/xslt
[3] "XQuery 1.0: An XML Query Language - W3C Working Draft, 12 November 2003", http://www.w3.org/TR/xquery/
[4] "Encoding a Taxonomy of Web Attacks with Different-Length Vectors", G. Alvarez and S. Petrovic, http://arxiv.org/PS_cache/cs/pdf/0210/0210026.pdf
[5] "Using Binary Search with SQL Injection", Sverre H. Huseby, http://shh.thathost.com/text/binary-search-sql-injection.txt
[6] "Extensible Markup Language (XML) 1.0 (Second Edition) - W3C Recommendation, 6 October 2000", http://www.w3.org/TR/REC-xml
[7] "XML Path Language (XPath) 2.0 – W3C Working Draft, 12 November 2003", http://www.w3.org/TR/xpath20/
[8] "XQuery 1.0 and XPath 2.0 Functions and Operators – W3C Working Draft, 12 November 2003", http://www.w3.org/TR/xpath-functions/
[9] "XML Database Products – Native XML Databases", Ronald Bourret, http://www.rpbourret.com/xml/XMLDatabaseProds.htm#native