SFX-SQLi

SELECT FOR XML SQL INJECTION

Fast data extraction using SQL injection and XML statements

Daniel Kachakil (dani@kachakil.com)

5th February, 2009

Contents

Abstract

As the database management systems are being released with more functionality, new attack methods are arising, taking advantages of these features. This document describes a method to dump all the information of a **Microsoft SQL Server** database, using a very efficient technique based on the **FOR XML** clause. A tool which implements this technique will also be described here.

This technique does not discover a new kind of vulnerability, but it is based in a better use of a known technique that has been used for 10 years: **SQL injection**. Despite its oldness and the publication of many attack techniques and automated tools based on this vulnerability, it remains today as one of the most widely present in web applications, compromising the confidentiality, integrity and availability of a large amount of data.

Assuming a scenario in which it is possible to apply SQL injection techniques, we will try to get an injection with a valid union sentence which would be able to append a row into the results of some existing query of the web application. The main prerequisite is that we can read at least one of their records, preferably a string (via HTML, as a web service response, exported into a file, and so on).

If we can show even a single character under our control, in the general case it will be possible to replace this character with the whole contents of a table, provided that we can dump it into a single string. In order to achieve this objective, we will use the **FOR XML** clause of the **SELECT** statement which is incorporated into the syntax of **Microsoft SQL Server** since its version 2000.

Introduction: SQL injection

The first document with references to SQL injection appears at the end of the year 1998 [1], introducing the problem with some basic examples, but highlighting also the largest potential impact if we know how to take advantage of the powerful features of some stored procedures (for instance, sending data via e-mail). In the same paper it was also described how to protect our applications against this kind of attacks, and these protection mechanisms are still perfectly valid and they remain unchanged nowadays.

Over the years, the number of new techniques exploiting these vulnerabilities has been growing, taking advantage of all kind of features that make things easier for developers, but also being very useful to the attacker at the same time. From a simple syntax that allows commenting part of the query to some complex predefined stored procedures that are able to send e-mails, to dump information into the disk drive or to execute commands of the underlying operating system… everything could be useful for an auditor and for a potential attacker as well.

This type of generic vulnerability is not only present in Microsoft SQL Server, but it also affects other DBMS (Oracle, MySQL, and so on), because it is neither an issue of the database manager itself nor the development technology used, but it is caused by an incorrect or a

nonexistent way of filtering the user inputs by the developers of the web application. A bad security policy at server level can also amplify their impact.

The first appeared techniques related to massive information dump, to send e-mails or to execute commands are hardly exploitable if the server permissions are well configured. However, we can find today a lot of tools that are able to extract the whole information of a database in a completely automated way, taking advantage of different variations of the same technique, without needing special privileges.

For example, if the web application shows the error messages of the database manager system, it can be possible to extract the information by forcing the application to fail using intentional data type conversions that can reveal a lot of information about the internal structure of the database, and also about the data contained in it [2][3].

In cases where the application does not show any descriptive error message, there are also other techniques for inferring the information blindly (Blind SQL injection) [4][5][6][7], based in the variations of the response or of the behavior of the web application (variations in the returned HTML code, redirection to other pages, delays and so on).

The main disadvantage of the application of these types of techniques is the large amount of requests and time needed for their execution, leaving also a big trace in the log files of the server that could alert the server or web administrators if they notice a considerable increment of accesses, a performance degradation or a high number of errors produced by unknown reasons for them.

The techniques described in this paper take advantage of one of the features of Microsoft SQL Server since its 2000 version (8.0) which has been improved in the 2005 version and is maintained in the latest version 2008, allowing an easier and powerful exploitation. We are talking about the FOR XML clause included in the SELECT statement syntax.

Fundamentals: The FOR XML clause of the SELECT statement

Microsoft SQL Server 2000 introduces a new clause into the SELECT statement syntax to specify that the results of a query will be returned in XML format. We are talking about the FOR XML clause, which can be appended to the end of a query followed by some mandatory and optional arguments.

Here is an example of the results returned by the execution of the next queries in a SQL Server 2005 against a hypothetical sample database holding some simple data:

- `SELECT * FROM Numbers`

Number	Name
1	One
2	Two
3	Three
4	Four

- `SELECT * FROM Numbers FOR XML RAW`

XML_F52E2B61-18A1-11d1-B105-00805F49916B
<row Number="1" Name="One" /><row Number="2" Name="Two" /><row Number="3" Name="Three" /><row Number="4" Name="Four" />

We can see that the second query returns essentially the same data returned by the first one, except that the first is returning a record set while the second one is returning a single value containing the same information but in XML format. The latter also includes the column names in the XML attributes, which will allow us to extract the table structure directly (as for now, we will have to infer the data type of each column looking into the data itself, but later we will discuss how to extract this information by other ways).

Suppose that there is an application that shows the textual representation of a digit using this query (vulnerable to SQL injection):

- `"SELECT Name FROM Numbers WHERE Number=" + Params("num")`

What is described in the following sections is basically the explanation of what would happen if a string like this were injected in our sample application:

- `-1 UNION SELECT (SELECT * FROM Table FOR XML RAW)`

Microsoft SQL Server 2005 and 2008

If we refer to the product documentation, we will find a set of arguments that can be applied to the FOR XML clause. This is the syntax that appears in the documentation [8]:

```
[ FOR { BROWSE | <XML> } ]
<XML> ::=
XML
{
    { RAW [ ( 'ElementName' ) ] | AUTO }
    [
        <CommonDirectives>
        [ , { XMLDATA | XMLSCHEMA [ ( 'TargetNameSpaceURI' ) ] } ]
        [ , ELEMENTS [ XSINIL | ABSENT ]
    ]
  | EXPLICIT
    [
        <CommonDirectives>
        [ , XMLDATA ]
    ]
  | PATH [ ( 'ElementName' ) ]
    [
        <CommonDirectives>
        [ , ELEMENTS [ XSINIL | ABSENT ] ]
    ]
}

<CommonDirectives> ::=
[ , BINARY BASE64 ]
[ , TYPE ]
[ , ROOT [ ( 'RootName' ) ] ]
```


First we have the BROWSE option, which is completely unrelated to the objective we are looking for, so we will ignore it. We will focus into the other possible option (that is, XML), which also allows the use of a set of arguments. It is mandatory to specify at least one of the following arguments: RAW, AUTO, EXPLICIT or PATH:

- **RAW:** Shows the information converting each row of the result set in an XML element in the form *<row />*. Optionally we can also specify another element name that may reduce some bytes of the response if we use only a single character.
- **AUTO:** Shows the information in a hierarchical way for each table of the query. As we will not use more than one table for each query, this option will not help us at all.
- **EXPLICIT:** Allows to specify the desired structure of the generated XML tree, but it uses a complex syntax and does not support any benefit.
- **PATH:** Provides other ways of customization of the XML tree, simplifying the EXPLICIT model. It will generate an element for each row and column by default, so it will need more data length to represent the same information compared with the RAW mode.

For now we can see that the RAW option is the only interesting one, because the other options are designed to have more control over the output format in which the XML tree will be generated, but in our case the first option is the most comfortable, having in mind that in principle we do not know nothing about the structure of the tables.

The behavior of the RAW option can be altered by some common options of the FOR XML mode and by specific ones also:

- **BINARY BASE64:** Specifies that the query will return the binary data encoded in BASE64 format. This is not the default option in the RAW mode, so the query will fail if we find any BINARY data type column (containing images, for example) if this option is not explicitly specified, so we will consider it necessary for this reason for now.
- **TYPE:** Specifies that the query returns results as "xml" type (new in SQL Server 2005) instead of returning it as string. As our objective is to convert a table into text, the "xml" data type does not offer any advantage.
- **ROOT:** Specifies if a root element will be added to the returned results. By default it appends the *<root>* element, but we can change this name optionally. This option may be useful to locate where the data result starts and where it ends. It does not matter if we are inspecting the results manually, but it could make the things easier for a tool designed to automate the process.
- **XMLDATA:** Specifies that a XDR schema will be added to the results at the beginning. It could be useful to determine the data types, but it is not supported when there is a binary column, so its use is not recommended in general.
- **XMLSCHEMA:** This option will return the structure of the returned data, including the data type in native format, the length and not null constraints, and so on. It is an interesting option, because although in case of returning no data, it allows to obtain the whole table structure, including the data types, column names and other constraints. It is also compatible with the ROOT option and binary data.
- **ELEMENTS:** Specifies that the columns are returned as XML elements (by default are returned as attributes). This option is not useful because the result will be larger, unless we are interested in determining the presence of the null values, in which case we will need also the XSINIL argument (otherwise, the tags corresponding to these fields will be simply omitted).

The latest 2008 version does not change anything compared with the 2005 version regarding the FOR XML clause, on the contrary, it is marked as obsolete and its use is not recommended in favor of XSD schemas. In any case, this does not cause problems nowadays, because it will have to pass many years to complete the migration of existing applications to the next version above the 2008, if Microsoft finally decides to remove this functionality on the next version (planned for 2010), as is noted in the product documentation.

General extraction procedure using the FOR XML clause

To achieve the dump of the whole database through an SQL injection vulnerable parameter, in general we can use the technique described below:

1. Adjust the injection to redirect the response
2. Get the table names from the catalog
3. Get the structure and contents of each table

Adjust the injection to redirect the response

The first step is to get an injection that will allow us to manipulate the results of any query in order to append the desired content into it. Probably we will have to use an existing query and start inferring its internal structure (number of parameters and data types). There are several documented techniques to achieve this purpose, depending on whether the application displays errors, if there is some kind of filtering, if we can execute some special commands, etc.

Once we have determined this structure, in the general case we will be able to append more rows using a union query with the same number and types of parameters. If we have adequate privileges, maybe we could update or insert some row of an existing table, create a new one, dump the data to a file in the disk, and so on.

What is described in this section is not something new. It is only a short summary of some known techniques as a prerequisite of the general extraction procedure described in this paper, so we will discuss them mainly for that reason.

Determining the number of columns or parameters of the query

If the application shows details about the errors produced in the database, then we are facing the simplest case, because we can easily infer the structure of a query, finding also the column names and the involved tables using the HAVING 1=1 and GROUP BY technique [3][9].

- `HAVING 1=1` → Raises an error with a text that contains the first column name
- `GROUP BY column1 HAVING 1=1` → Error with the second column name
- `GROUP BY column1, column2 HAVING 1=1` → Error with the third column name
- `...`
- `GROUP BY column1, column2, column3, … , columnN HAVING 1=1` → No errors

If the application does not show information about errors, then we must apply blind SQL injection techniques, using for example the one which uses the ORDER BY clause followed by the number of columns:

- `ORDER BY 1` → No errors
- `ORDER BY 2` → No errors
- `...`
- `ORDER BY N` → No errors
- `ORDER BY N+1` → Fails

Another option is to use the UNION statement followed by the same number of values (normally nulls) as exists in the original query [5][9]. If we are using this technique, appending a false condition (like WHERE 0=1) may help to simplify the process, because the injected row of null values will not interfere in the results.

- `UNION SELECT null WHERE 0=1` → Fails
- `UNION SELECT null, null WHERE 0=1` → Fails
- `UNION SELECT null, null, null WHERE 0=1` → Fails
- `...`
- `UNION SELECT null, null, null, null, … , null WHERE 0=1` → No errors

Determining the type of the obtained columns

Once we have obtained the number of columns returned by the chosen query, we will try to get the data type of each one, although not always will be needed to complete this step with total precision, neither to guess all the column types in most cases. It will be enough if we succeed in injecting one character in a text column, whenever it is processed by the database and returned by the server.

Again, the simplest case will be the one when the application shows error messages, because we can use functions like CAST() and CONVERT() to force a data type conversion and, in case of fail, the error message itself will reveal the type of the column (or the absence of this message will reveal that the type we tried is correct). These functions can be used either in a WHERE condition or in another subquery with the UNION clause.

If the application does not show errors, we will start injecting a union query with as many null values as columns we have found. Then we will continue trying to substitute the null values by different data types (usually it will be enough with integers and strings). We can also append a false condition before and/or after the union query, although we will not always obtain the right results when using this technique, at least it may help us with the process.

Anyway, we will have not only to adjust the values which make the query valid and do not fail in the DBMS, but the application that uses these data cannot fail either (mainly caused by the null values and other unexpected values), so it is always better to find a query as simple as possible with the purpose of avoiding troubles that are unrelated to the extraction technique described here.

Always having in mind our goal, if we achieve an injection able to visualize any character or string while we are following this procedure, then we can stop and continue with the next step.

Obtaining an adequate union query

Having a union query with the correct number and type of values in their columns without causing any error, sometimes we will also need to adjust some values manually in order to avoid errors in the application layer and in order to get the response with our query.

For example, if the application only shows the first result of the query, we can achieve to make the original query not returning any row by using a false condition just before the union (e.g. WHERE 0=1). If the application fails or does not return any results because we have introduced some parameter out of the expected range, then we will try with other values that could be more logical and suitable inside the application context, and so on.

Once we have this point solved, the only remaining step is to replace any of the text values by a subquery created as we want and then check that it returns the expected result. Whenever possible, we will try to find a field which contents are not being modified by the application (for instance, encoding it in HTML or limiting the number of characters), although even in the case of not finding the optimal option, we will usually be able to find workarounds to avoid these kind of restrictions affect the process.

Get the contents of a table

To get the whole content of a table, in general the structure of the injected query will have to look like this (being v_1-v_N the suitable values to make the application work correctly and return results):

- `1 AND 1=0 UNION SELECT` v_1`,` v_2`, … , (SELECT * FROM Table FOR XML RAW, BINARY BASE64), … ,` v_N

Depending on each specific case, is possible that it will be required some quotes to close the first value if it is a string, a double dash to ignore the ending part of the original query in the application, to replace the blank spaces by an empty comment (/**/), to include the schema names before the table names, and so on.

The returned response will be a large string containing as much XML elements as rows are returned by the injected query. Because a well-formed XML document must contain a single root element, we will have to add it manually (provided that we have not specified the ROOT option in the injection).

Finally, to reassemble all data we can use the DataSet class of the Microsoft .NET Framework [10], because it has methods to import the raw XML content that we have and bind it to any control as its data source, allowing to sort and filter the data easily. Being both products from the same manufacturer and being designed to simplify the developer tasks, we can assume that we will not find any incompatibility problems.

Get the schema of a table

To get the structure of a table we can use the XMLSCHEMA option of the same FOR XML clause, we can query the system catalog (e.g.: SYSOBJECTS), or we can infer it directly from the extracted data (which already includes the column names). Usually we can just apply the latter method, provided that we do not want to make a preliminary sampling in order to determine the interesting tables and columns to dump.

Getting the table names from the database catalog

Whenever possible, we will try to determine the table names from the internal catalog (from metadata) of the database itself, using predefined system views like SYSOBJECTS or INFORMATION_SCHEMA.TABLES, dumping its entire content with the same technique as was described before. Once we have got the table and schema names, we will only need to apply the same technique again for each table we want to dump.

This is a possible query which would return enough information about the existing tables in the database:

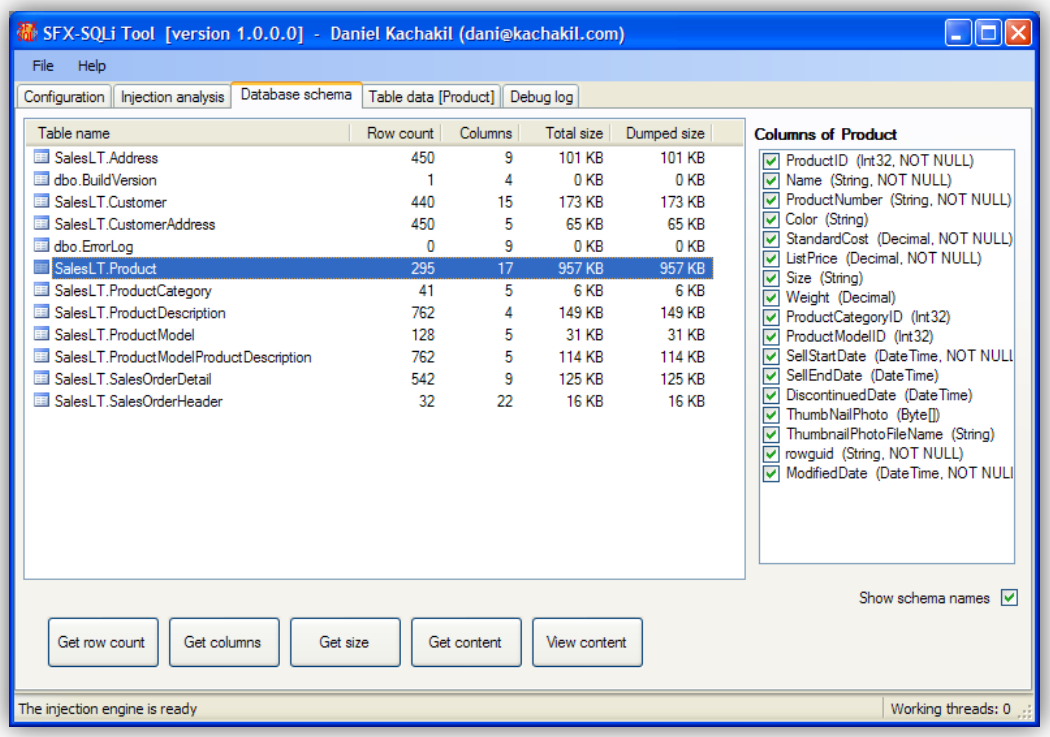
- `SELECT TABLE_SCHEMA, TABLE_NAME FROM INFORMATION_SCHEMA.TABLES WHERE Table_Type='BASE TABLE'`

Note that this method only needs a single request to the server for each table we want to extract its information (in the ideal case).

Automation: The SFX-SQLi Tool

As was discussed previously in this paper, this technique can be easily automated, because its general application is not an especially complex procedure. A tool which implements all methods described here has been created in order to demonstrate how this technique works and to show its power in practice. The SFX-SQLi tool is able to extract the whole structure and contents of a vulnerable web application database in an automated way (after the injection adjust), allowing to see and analyze which is happening all the time.

It is possible that we could find some table with a lot of rows or with large binary data, and the database manager server cannot process it in the maximum time assigned for its execution (or the timeout of the web server response, causing the failure of the method because of circumstances which are unconnected with the procedure). For this reason, among other features, the SFX-SQLi tool will optimize the process, reducing the size of the returned XML string (assigning alias to the columns) and dividing the query in rows subsets (using the ROW_NUMBER function) and in text blocks (using the SUBSTRING function) in order to minimize the server load, although at expense of increasing the number of requests.



In its initial state, the SFX-SQLi tool lacks some features that can be even considered as essential, but we have to take into consideration that this tool has not been designed to be used as an auditing tool, but only as a proof of concept.

Application of the technique in other environments

Although all techniques described before are focused on web application based on Microsoft SQL Server 2005/2008, some ideas that have been presented in this paper are also applicable to other environments.

SQL Server 2000

The syntax of this SQL Server version is quite simpler [11] than the 2005, probably due to being the first version which introduces this clause and because the adoption of the XML standard was not very extended in those days:

```
 [ FOR XML { RAW | AUTO | EXPLICIT }
             [ , XMLDATA ]
             [ , ELEMENTS ]
             [ , BINARY BASE64 ]
]
```

In this version 2000 we can see that the limitations are more evident than the 2005 version, not only at this syntax level, but also in its execution. One of the biggest constraints in this version is the fact of not allowing the inclusion of the FOR XML clause in subqueries [12],

so that its only valid location is at the very end of the general query. Fortunately, we have the RAW option and the BINARY BASE64 argument, which will allow us to obtain the information in a similar way like we would do it in the 2005 version.

Extracting files using serialization

The fundamentals of the technique described in this paper are also applicable in other existing injection techniques. For example, instead of serializing a table in XML format, we can convert a file to its hexadecimal representation in order to obtain its entire content with a single request to the server:

- ```
  SELECT SYS.fn_VarBinToHexStr((SELECT c FROM OpenRowSet(BULK
  'c:\boot.ini', SINGLE_BLOB) AS T(c)))
  ```

Conclusions

In this paper we have described a new SQL injection technique that allows dumping a large amount of data in an extremely efficient way using XML-based serialization. Compared with the common techniques used by the best tools available nowadays, the difference is enormous (although being less specific, the scope of these techniques is often wider).

The procedure described here is almost fully automatable, allowing the development of tools to make the data extraction easier and efficient, as is also described in this paper.


  ```

# References


[1] **"NT Web Technology Vulnerabilities".** Rain Forest Puppy (1998)
http://www.wiretrip.net/rfp/txt/phrack54.txt

[2] **"Web Application Disassembly with ODBC Error Messages".** David Litchfield (2001)
http://www.nextgenss.com/papers/webappdis.doc

[3] **"Advanced SQL Injection in SQL Server Applications".** Chris Anley (2002)
http://www.ngssoftware.com/papers/advanced_sql_injection.pdf

[4] **"(more) Advanced SQL Injection".** Chris Anley (2002)
http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf

[5] **"Blindfolded SQL injection".** Ofer Maor and Amichai Shulman (2003)
http://www.imperva.com/docs/Blindfolded_SQL_Injection.pdf

[6] **"Blind SQL Injection".** Kevin Spett (2003)
http://www.net-security.org/dl/articles/Blind_SQLInjection.pdf

[7] **"Automating Blind SQL Exploitation".** Cameron Hotchkies (2004)
http://althing.cs.dartmouth.edu/secref/resources/defcon12/dc-12-Hotchkies.ppt
http://www.0x90.org/releases/absinthe/

[8] **"FOR Clause (Transact-SQL)".** Microsoft MSDN Library, SQL Server 2005/2008.
http://msdn.microsoft.com/en-us/library/ms173812(SQL.90).aspx
http://msdn.microsoft.com/en-us/library/ms173812.aspx

[9] **"SQL Injection Cheat Sheet: Union Injections".** Ferruh Mavituna.
http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/#UnionInjections

[10] **"DataSet (Clase)".** Microsoft MSDN Library, .NET Framework
http://msdn.microsoft.com/en-us/library/system.data.dataset.aspx

[11] **"SELECT (Transact-SQL): FOR Clause".** Microsoft MSDN Library, SQL Server 2000.
http://msdn.microsoft.com/en-us/library/aa259187(SQL.80).aspx#_for_clause

[12] **"Guidelines for Using the FOR XML Clause".** MSDN Library, SQL Server 2000.
http://msdn.microsoft.com/en-us/library/aa226520(SQL.80).aspx