

SQL Injection Signature Evasion Whitepaper

The Wrong Solution to the Right Problem

One of the most dangerous and common Web application attacks is SQL injection. By manually inserting unauthorized SQL queries into a vulnerable Web page, an attacker may gain unrestricted access to the entire contents of a backend database. In response to the increasing frequency of this attack, security vendors (Web application firewall, network firewall, and IDS/IPS vendors) have begun to claim SQL injection protection. However, most of these protections rely on traditional signature detection techniques. They inspect Web traffic to identify SQL injection-related text patterns.

While many have been led to believe that signatures are sufficient for SQL Injection protection, research by the Imperva Application Defense Center (ADC) has shown that reliance upon signature protections alone is not effective against real-life attacks. This paper demonstrates various SQL injection signature evasion techniques. It then shows that these evasion techniques represent only a fraction of the all possible evasion techniques. Finally, the paper concludes that reliance upon signature protections alone is not a practical defense against SQL injections attacks. A reasonably sized signature database does not provide reliable protection while a comprehensive signature database results in excessive management overhead, dramatic performance limitations, and false positives.



Introduction

In this paper we follow the attempts of a theoretical hacker named Autolytus to gain access to valuable database information via its Internet connected Web application¹. The application is protected by a network firewall and a signature-based Intrusion Prevention System (IPS) – known to Autolytus as his archenemy, Signatorious.

Autolytus knows nothing about the target Web application, the backend database, or their security protections. However, based upon recent experience, he is confident in his success. One of his favorite techniques is SQL injection². This technique allows him not only to view sensitive database data, but to make changes to data as well. Autolytus has mastered this technique so well that he is now able to compromise the database even without the benefit of error messages from the Web server.³

Note – Many examples shown in this paper apply to MS SQL Server, some to MySQL, and some to Oracle. The reader is advised against concluding that any particular database is more or less vulnerable based upon the ratio of examples herein. Many examples could be constructed for databases from any vendor. The basics concepts underlying these techniques are the same for all database products.

Recognizing Signature Protection

Autolytus, initially unaware that Signatorious is protecting the Web application, attempts SQL injection without evasion. He is initially blocked, but he perseveres by repeating his attempts with a variety of slight variations. After noticing that all attempts fail, he quickly guesses that his archenemy, Signatorious is blocking SQL keywords.

Autolytus decides to run a series of tests to verify his guess. His first step is to find a free-text input field that will accept arbitrary strings (i.e. non-signatures) without generating an error. Typical free-text input fields include search inputs, form submissions, etc.⁴ After submitting a few arbitrary strings without generating an error, Autolytus attempts to insert SQL injection strings. If the site is protected with signatures, for example, the following strings will be blocked.

- **UNION SELECT**
- **OR 1=1**
- **EXEC SP_ (OR EXEC XP_)**

If these specific SQL injection signature strings are blocked and arbitrary strings are accepted, then signature protections must be in place.

Having confirmed that signature protection is in place, the next step is to enumerate the SQL injection signature list. This involves a methodical trial and error process. One by one, he tries the SQL injection strings that he normally needs to carry out an attack. Those that do not cause an error are noted for later

¹ In Greek mythology Autolytus was an accomplished thief and trickster. He was a son of the god Hermes who gave him the power of invisibility. In ancient times, Autolytus stole by using his powers of invisibility to sneak past fortress main gates and pretending to be a mere visitor once inside. In modern times, Autolytus has found that stealing valuable database information via an Internet connected Web application is just as much fun and involves substantially less physical risk.

² This paper assumes a familiarity with the SQL language and SQL injection strategies in general.

³ This paper assumes that the protected Web server hides error messages. It is recommended that the reader become familiar with the techniques that enable SQL Injection without error messages. For more information see 'Blindfolded SQL Injection', published by Imperva's Application Defense Center.

⁴ Any free input field suffices because the signature protections inspect all inputs discriminately. If no free input fields exist, string-format parameters may be modified.

use in attacks. Those that are blocked are broken down into components until he identifies the exact format of the signature. This may sound like a *Sisyphian* project, but it normally does not take too long to identify the specific signatures that disturb an attack.

Common Evasion Techniques

After recognizing Signatorious, Autolycus plans his next move. He now knows which of his normal SQL injection strategies are monitored by Signatorious. He goes home and digs up a book titled 'The Oldest Tricks in the World'. "I have fooled Signatorious in the past", he thinks to himself, "Maybe there are some old tricks he still hasn't learned..."

Surprisingly enough, many of the new generation signature detection products fail in the same ways that older products failed in the past. Therefore, Autolycus' first move is to try some of the old tricks before proceeding to advanced signature evasion techniques. A few of these old tricks are presented below.

Encodings

Various encoding tricks have proved themselves useful throughout the history of computer attacks. The reasons for this are many. Some products simply fail to do the right implementation or sufficiently understand the application level protocol. Others are aware of the problem, yet the performance requirements limit what they can do in real time. Ultimately, a variety of encoding techniques, such as URL Encoding, UTF-8, etc. may prove successful.

White Spaces Diversity

Many of the signatures used to prevent SQL Injection attacks are a sequence of two or more expressions, separated by a white space. The reason for this is simple, a single word signatures such as SELECT, would generate an avalanche of false positives. The expression UNION SELECT, however, is almost unique to the SQL world, making it a better signature. This, however, carries the potential for white space vulnerability. If the signature is not carefully defined, all that Autolycus must do to avoid detection while preserving the integrity of his attack is to replace the single space between words with two spaces between the words (or alternatively three spaces, or a space with a tab or some other combination.)

IP Fragmentation and TCP Segmentation

Although less likely, some of the signature detection mechanisms, (usually those that focus on the network rather than those that actually parse the application protocol), may still be vulnerable to fragmentation of lower level network protocols.

Advanced Evasion Techniques

Autolycus is now tired. He tried every trick in his book, yet was unsuccessful. He will not give up on this. Signatorious is not that smart. He only knows what he was shown. "There must be a way to fool this guard" he thinks as he falls asleep. In the morning he decides to try again. He realizes that old books are not the solution and he decides to focus upon creative ideas of his own. "If I can come up with a good idea of my own, the guard will never catch me", he thinks to himself. And he is right.

We review here several techniques researched in Imperva's Application Defense Center that have proven successful in avoiding many common signature-based security products.

The 'OR 1=1' Signature

Among the most common groups of SQL injection signatures are those designed to detect the famous OR 1=1 attack. These signatures are usually built as a regular expression, aimed at catching as many

possible variations of the attack. Sadly (or luckily, for Autolytus), many of them can be tricked by using sophisticated matches. For example, an unusual string can be used.

```
OR 'Unusual' = 'Unusual'
```

Yet, with some of the better systems, this might not be enough. Autolytus therefore must find a way to make the two expressions look different, yet still be the same. A very simple trick is to add the character *N* prior to the second string. For example,

```
OR 'Simple' = N'Simple'
```

The *N* tells the SQL Server that the string should be treated as *nvarchar*. This doesn't change anything in the comparison for the SQL itself, but definitely makes it different from the perspective of a signature mechanism.

An even better technique would be to break one of the strings into two, concatenating it on the SQL level. This will render useless any mechanism which compares the strings on both sides of the = sign (the example is in MS SQL Server syntax, but can be done in a similar manner in any other database).

```
OR 'Simple' = 'Sim'+ 'ple'
```

One of the above mentioned techniques is likely to evade most of the signature based mechanisms. Yet, some vendors might choose a much wider regular expression to cope with this attack. Something along the lines of the word OR followed by an equal sign (=) anywhere in the string. This, however, can also be easily avoided by simply finding an expression which evaluates as *true*, without having the equal sign in it. For instance, replacing the equal sign with the SQL word LIKE (which performs a partial compare).

```
OR 'Simple' LIKE 'Sim%'
```

Or one of the greater than or less than operators, like one of these examples:

```
OR 'Simple' > 'S'
```

```
OR 'Simple' < 'X'
```

```
OR 2 > 1
```

Or the *IN* or *BETWEEN* statements:

```
OR 'Simple' IN ('Simple')
```

```
OR 'Simple' BETWEEN 'R' AND 'T'
```

(The latter is valid in MS SQL Server only, but can be easily modified to work on any database).

And this can go on forever. SQL is a very rich language, and for every signature invented, a new evasion technique can be developed. Trying to add signatures to cover all of the above presented techniques is bound to fail, and will most likely badly damage the performance. Another possibility is, of course, to define signatures that are very general, such as an 'OR' followed by an SQL keyword or Meta character anywhere in the string. This, however, is likely to result in many false positives. Think of the following URL:

```
http://site/order.asp?ProdID=5&Quantity=4
```

Although far from being an invalid URL, it will trigger such a signature:

```
http://site/order.asp?ProdID==5&Quantity=4
```

Clearly, this is not the solution.

Evading Signatures with White Spaces

As discussed previously, the simple evasion technique of changing the length or type of a white space is properly handled by many signature protection solutions today. Autolytus has responded with a new and improved technique. The new technique takes advantage of vendor specific SQL parsing decisions and creates a valid SQL statement without using spaces or by inserting arbitrary characters between them. The techniques presented here differ from one database to another, yet share the same principles.

The basic technique, which operates on databases that perform a rather loose (and more user-friendly) parsing of the SQL syntax, is to simply drop the spaces. With Microsoft SQL Server, for instance, spaces between SQL keywords and number or string literals can be completely omitted, allowing an easy evasion of signatures such as 'OR '. Instead of typing

```
...OrigText' OR 'Simple' = 'Simple',
```

which is the basic attack, Autolytus can simply type

```
...OrigText'OR'Simple'='Simple'.
```

This new string is accepted by the database and functions properly but has no spaces. It therefore completely evades any white-space-based signature. Unfortunately for Autolytus, this will not work for injections such as 'UNION SELECT' since SQL requires a separation between the two keywords. The solution is to separate them with something other than a white space. A good example of this technique is presented by the C-like comment syntax available in most database servers⁵.

A comment syntax that is supported by most database servers uses /* to start a comment and */ to end it. This means that a valid SQL statement can look like the following.

```
SELECT *
FROM tblProducts /* List of Prods */
WHERE ProdID = 5
```

This can be used by SQL injection code as follows.

```
...&ProdID=2 UNION /**/ SELECT name ...
```

Any signature attempting to detect a UNION followed by any amount of white spaces, followed by a SELECT, will fail to detect this signature. Moreover, in most cases the /**/ can actually replace any of the spaces (in the above example it was in addition to the spaces), allowing evasion of more sensitive signatures such as 'SELECT ' or 'INSERT ' (an SQL keyword followed by a single space), which have been noted to be used by some SQL signature protection mechanisms. The previous example then would appear like this:

```
...&ProdID=2/**/UNION/**/SELECT/**/name ...6
```

This technique can also be used in Oracle SQL to evade OR 1=1 signatures. Although Oracle will not allow omission of the white spaces, it allows replacement with a comment enabling the following exploit.

```
...OrigText'/**/OR/**/'Simple'='Simple'
```

⁵ This was tested on MS SQL, MySQL and Oracle.

⁶ This works for MS SQL and Oracle, but not MySQL. In MySQL, at least one space has to be present. It can be present after the comment, thus allowing evasion of signatures looking for a space immediately after the keyword.

This technique can be even further exploited, providing even better evasion (especially in the cases of advanced application firewalls that check the signatures on the parameter value only) in cases where two separate parameters are inserted into the SQL statement. Imagine a login page, where the following requests:

```
http://site/login.asp?User=X&Pass=Y
```

Generates the following query:

```
SELECT * FROM Users
WHERE User='X' AND Pass='Y'
```

In this case, the beginning of the comment can be injected into one parameter, whereas the termination of the comment can be injected into the other.

```
...login.asp?User=X'OR'1'/*&Pass=Y*/='1
```

Resulting in the following query, which easily logs Autolytus in:

```
SELECT * FROM Users
WHERE User='X'OR'1'/* AND Pass='*/='1'
```

As with the techniques described for the 'OR 1=1' signature evasion, there is no real solution here. It is of course possible to add /* and */ to the signature list. However, a new trick is likely to be devised shortly after. Alternatively, the actual keywords, such as SELECT and INSERT can be placed as a signature, but as with the OR keyword, this will result in many false positives in real world applications, providing no real solution. (Imagine a 'Contact Us' form in an ecommerce site, where the customer has typed '...I have **selected** the product, but then had a problem...'. This would trigger a signature on the word SELECT, without any attack taking place.)

Evading Any String Pattern

Despite the example showing why standalone keywords are likely to generate false positives, some stricter sites may choose to apply such signatures, while limiting the functionality so that no free text will be entered by the users (for instance, the main portion of a banking application does not have to allow free texts from the user). In this case, Autolytus will need other techniques, which allow breaking strings in the middle.

Luckily, Autolytus still does not need to do a lot of research, as the previously mentioned techniques can, with some modification, be used for the purpose of this as well. The first technique goes back to the comments. With MySQL, the comments would not work as a replacement for a space. However, comments can be used in MySQL to break words in the middle, for instance:

```
...UN/**/ION/**/ SE/**/LECT/**/ ...
```

Another very promising prospect relies on the string concatenation evasion technique that was demonstrated earlier. Most databases allow the user to execute an SQL query through one or more statements (built in operations, stored procedures, etc.), that receive the SQL query as a string.

All Autolytus therefore needs to do, is find a way to build an SQL Injection exploit that will allow the execution of such a string. Once this exploit was created (either with no evasion techniques at all, because no one thought of this attack, or using previously mentioned techniques), all the other patterns can be evaded simply by using string concatenation in the middle of the suspicious string.

A simple example is demonstrated with MS SQL's built in EXEC command. This command can also be used as a function, receiving any SQL statement as a string, which can be naturally concatenated:

```
...; EXEC('INS'+ 'ERT INTO...')
```

Since the word INSERT was split into two parts, no signature mechanism is able to detect it. The SQL, however, rebuilds the string, allowing it to execute as planned.

As with our other examples, this is not a singular example. A similar attack on MS SQL can be done with a stored procedure named SP_EXECUTESQL. This is a new version of the outdated (yet still functioning) SP_SQLEXEC procedure. Both of them will receive a string containing an SQL query and will execute it. Naturally, this problem is not limited to MS SQL. Other databases suffer from the same problem. With Oracle, the syntax 'EXECUTE IMMEDIATE' can be used to execute a string which can, of course, be concatenated.

Another interesting twist on this attack, in MS SQL, can be based on a hexadecimal encoding of the string to be executed⁷. This way, the string 'SELECT' can be represented by the hexadecimal number 0x73656c656374, which will not be detected by any signature protection mechanism. This, combined with the loose-syntax nature of SQL, allows the execution of many signature defined statements.

Another good example in MS SQL relates to the OPENROWSET statement. This technique is an old, known technique but most signature based products fail to look for it. Again, since OPENROWSET receives a string parameter, it is possible to concatenate into it the required query without it being detected by the signature mechanism.

One may argue that the number of statements that can be used for such a technique is limited within each database. Yet although this might be true to some extent, it is likely that while building a database of signatures some of them will be forgotten.

An excellent example for that is provided by MS SQL, which contains *unlisted* stored procedures that can be used for execution of SQL queries. Microsoft's implementation of prepared statements in MS SQL Server is actually done using several internal, unlisted, stored procedures. When running a prepared statement, a stored procedure named *sp_prepare* runs first, preparing the statement, and then a stored procedure named *sp_execute* is run in order to execute the query. With these procedures not appearing in any listing of SQL Server, they are likely to be missing from any SQL Injection signature database.

Obviously, similar undocumented procedures and functions may exist in other databases, exposing them to future attacks that evade existing signatures.

Conclusion

The next morning, Autolykus is home again. "I did it again" he thinks to himself. With a satisfied smile he sits down to write his new book 'New Guards – New Tricks'. Sadly, he will choose to distribute that book only to his friends inside the thieves' guild, preventing the new guards from learning anything from it.

Three days later, Signatorious learns that he was outsmarted by Autolykus. His valuable database information has been posted on the Internet. Frustrated by defeat and desperate to prove his worth, he frantically stops anyone he imagines to be even remotely suspicious. He throws out many legitimate users. His boss then believes that Signatorious is unable to fulfill his task. Poor Signatorious is relieved of duty.

At this point, we believe the conclusion of this paper is clear. Signature protection against SQL Injection is simply not enough. Although this paper demonstrates only a few evasion techniques, some or even all are likely to defeat most of today's signature protection mechanisms.

Sadly, the trivial solution of adding ever more signatures to a signature-based security product is not a real solution. It can help thwart some techniques shown, but other can and will be developed. This is due to the richness of the SQL language as implemented by each of the database vendors. The richness of

⁷ This technique was described in a paper named '(more) Advanced SQL Injection' by Chris Anley which was published in 2002.

the language means that many different statements can be sent to the server, all resulting in the same basic operation.

Trying to provide full security for such a rich language must take one of two approaches. The first approach is to attempt accurate detection of all possible dangerous SQL statements. For a single database type, this would include all sufficiently accurate combinations of SQL keywords (such as INSERT... INTO, UNION SELECT), all stored procedures or functions (these usually have distinctive names, allowing them to appear as they are in a signature), and any other relevant SQL related syntax.

This approach is flawed. Even if it were possible to covert *all* possible attacks and evasion techniques, the effort would require hundreds of signatures for each database type. Many of them are very complex and based upon regular expressions. Supporting several hundred signatures for each database type easily adds up to over one-thousand signatures for those organization using databases from multiple vendors. These are on top of the already existing signatures for other attacks. The performance price (throughput and latency) and operational support cost is simply unacceptable due to the large number of signatures.

The second approach is to generate a very few generic signatures. With MSSQL Server such a policy can include the following keywords (with their matching different encodings of course).

```
SELECT, INSERT, CREATE, DELETE, FROM, WHERE, OR, AND, LIKE, EXEC, SP_,  
XP_, SQL, ROWSET, OPEN, BEGIN, END, DECLARE
```

And also some Meta Characters (and their encodings), such as...

```
; -- + ' ( ) = > < @
```

This, however, can only work on a specifically built application running in a lab. In the real world, this minimized set of signatures is bound to block more users than hackers because it results in so many false positives. False positives may be acceptable in some IDS systems, but they are definitely not acceptable in an IPS or a Web application firewall, which block users.

In conclusion, although SQL Injection signatures may provide a certain level of security, the capable hacker can easily defeat them. Using SQL Injection signatures is simply not an effective mechanism for protecting against SQL injection attacks. Any attempt to create a signature database which prevents all SQL Injection attacks is bound to fail for one of two reasons – too many false positives or too many signatures.

Preventing SQL Injection with the SecureSphere Dynamic Profiling Firewall

As the discussion above illustrates, signature protection alone is effective only against the most basic SQL injection attacks. Against advanced attacks, signatures do not provide reliable defense.

One possible approach to detecting these more complex attacks is to combine signature detection with additional detection techniques. Such a combination approach makes it possible to validate any individual attack indicator (such as a signature match) with one or more pieces of corroborating evidence. For example, a security manager (with the full time task of tracking security alerts) who notices a SQL injection signature alert might look for corresponding anomalies in his database log files. If he finds unusual database activity that corresponds with the observed signature, he can be sure that an attack is in progress. The identification of two or more independent SQL injection indicators virtually eliminates the risk of false positives. The security manager would then be able to confidently block the attack without the problem of false positives.

The SecureSphere Dynamic Profiling Firewall employs this approach of combining multiple detection technologies – only without the need for a full time security manager. It combines an advanced signature-based intrusion prevention system (IPS) with Imperva's Dynamic Profiling technology and automatically correlates security events from each technology. This correlation technique achieves a

degree of accuracy that cannot be matched using signature protections alone⁸. The following examples illustrate SecureSphere's unique security capabilities and their ability to reliably defeat even the most advanced SQL injection attack.

Simple SQL Injection Defense – OR 1=1

As previously discussed, the OR 1=1 string is often appended to Web parameters in an attempt to override a SQL WHERE statement and access all entries in a given table. When *OR 1=1* is applied without evasion, it is a relatively simple attack to defeat - the standard signature is a strong indicator of an SQL injection and is highly accurate. Therefore, SecureSphere's advanced IPS is all that is necessary to reliably detect this basic form of the attack⁹ and SecureSphere's Instant Attack Validation immediately blocks the attack. This example illustrates a basic attack in which signatures provide a reasonable solution. In the next example, we'll look an advanced signature evasion attack in which signatures are ineffective.

SecureSphere IPS

SecureSphere's integrated IPS includes Imperva's HTTP and SQL dictionaries that are uniquely designed to detect advanced Web and database attacks such as SQL injection. SecureSphere's IPS implementation defeats all common signature evasion schemes including encoding, IP fragmentation, TCP segmentation, and white space diversity.

Instant Attack Validation

SecureSphere's Instant Attack Validation is an enforcement mechanism which blocks attacks based on a single, highly accurate attack detection event.

Advanced SQL Injection Defense – Or 1=1 Signature Evasion

In previous sections, we learned that attackers may replace the simple OR 1=1 string with less predictable strings and take advantage of concatenation to effectively evade signature protection. The following sequence illustrates the process SecureSphere uses to defeat such an attack without the use of signatures.

1. SecureSphere's Dynamic Profiling Web Firewall automatically establishes a profile of minimum and maximum allowable lengths for each URL parameter. All incoming parameter lengths are then compared to these expected values. Thus, when the *OR 'Simple' = 'Sim'+ 'ple'* characters are appended to an existing parameter, the Web firewall detects the additional characters and issues a **Parameter Length** violation. However, a single **Parameter Length** violation is not sufficient to validate the attack. It's possible that a developer may have recently changed the parameter length. Therefore, SecureSphere does not use Instant Attack Validation to immediately block the user.
2. SecureSphere's Dynamic Profiling Database Firewall automatically builds a profile of all queries from each database user – including Web applications accessing the database. Thus, if the Web applications passes a new query containing *OR 'Simple' = 'Sim'+ 'ple'* to the back-end database, SecureSphere detects the unusual query and issues an **Unknown Query** violation. However, this single **Unknown Query** violation is not sufficient to validate an attack. It's possible that a developer may have added a new query. Therefore, SecureSphere does not use Instant Attack Validation to immediately block the user.

⁸ SecureSphere also includes network firewall and protocol compliance security technologies for protection against other attack vectors – i.e. worms, etc.

⁹ The OR 1=1 string also generates SecureSphere Web and database firewall violations. Although not strictly necessary since the OR 1=1 signature is a reliable attack indicator, SecureSphere correlates Or 1=1 signature violation with OR 1=1 Web and database profile violations providing further attack validation.

- SecureSphere's Correlated Attack Validation enforcement mechanism (playing the role of the full time security manager watching the alert screens) automatically correlates the **Unknown Query** with the **Parameter Length** violation. These two attack indicators have now been linked resulting in definitive attack validation. SecureSphere blocks the attack.

Dynamic Profiling

SecureSphere's Dynamic Profiling technology automatically examines live Web application and associated database traffic to create comprehensive models or "profiles" defining the structure and dynamics of the web application and the database. Dynamic Profiles are created for both Web (HTTP/XML/SOAP) and database (SQL¹⁰) traffic. Together, these profiles serve as the foundation for a positive security model for Web and database protection. By continuously comparing user interactions to the profile, SecureSphere can detect any unusual Web or database activity. As the Web and database changes over time, advanced learning algorithms automatically update the profiles.

Correlated Attack Validation

Correlated Attack Validation (CAV) is an enforcement mechanism that correlates security violations across multiple SecureSphere attack detection layers as well as over time. It enables SecureSphere to effectively identify and block SQL injections (even those employing evasion techniques) as well as other complex attacks in which single attack indicators does not definitively validate an attack. Using CAV SecureSphere can identify and block attacks with a much higher degree of accuracy than any competing product which can base blocking decisions on only a single violation.

For More Information

The preceding examples illustrate SecureSphere's ability to identify SQL Injection attacks, even those implementing sophisticated evasion techniques that would fool signature-based security products. SecureSphere does this because it uniquely combines a signature-based IPS with Imperva's patent-pending Dynamic Profiling and Correlated Attack Validation technologies. The examples only scratch the surface of SecureSphere's security capabilities. Any combination of signature, Web profile, or database profile violations may be applied individually or correlated together to reliably defeat SQL Injection evasion scenarios as well as a wide range of other sophisticated attacks. For more information see <http://www.imperva.com/products/securesphere/resources.asp>.

¹⁰ Dynamic Profiling supports MS SQL, Oracle, Sybase, and DB2 (including DB2 on the mainframe)

References

1. Basic SQL Injection Overview

Taken from Imperva's on-line glossary

http://www.imperva.com/application_defense_center/glossary/sql_injection.html

2. Detection of SQL Injection and Cross-site Scripting Attacks

By K. K. Mookhey and Nilesh Burghate, March 2004

<http://www.securityfocus.com/infocus/1768>

3. Blindfolded SQL Injection

By Ofer Maor and Amichai Shulman, September 2003

http://www.imperva.com/application_defense_center/white_papers/blind_sql_server_injection.html

4. (more) Advanced SQL Injection

By Chris Anley, June 2002

http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf

5. Manipulating Microsoft SQL Server Using SQL Injection

By Cesar Cerrudo, August 2002

http://www.appsecinc.com/presentations/Manipulating_SQL_Server_Using_SQL_Injection.pdf



US Headquarters

950 Tower Lane
Suite 1710
Foster City, CA 94404
Tel: (650) 345-9000
Fax: (650) 345-9004
www.imperva.com

International Headquarters

12 Hachilazon Street
Ramat-Gan 52522
Israel
Tel: +972-3-6120133
Fax: +972-3-7511133