SQL Injection Signatures Evasion

An overview of why SQL Injection signature protection is just not enough
(Or: How Autolycus got past the fortress guards)

April 2004

Written by
Ofer Maor, *Application Defense Center Manager*
Amichai Shulman, *Chief Technology Officer*

IMPERVA™
APPLICATION DEFENSE CENTER

Table of Contents

Abstract **3**

Introduction **4**

Recognizing Signature Protection **5**

Common Evasion Techniques **7**

Different Encodings 7

White Spaces Diversity 7

TCP Fragmentation 7

Advanced Evasion Techniques **8**

The 'OR 1=1' Signature 8

Evading Signatures with White Spaces 10

Evading Any String Pattern 12

Conclusion **14**

References **16**

Abstract

In recent years, Web application security has become a focal center for security experts. Application attacks are constantly on the rise, posing new risks for the organization. One of the most dangerous and most common attack techniques is SQL Injection[1], which usually allows the hacker to obtain full access to the organization's Database.

With the rise in SQL Injection attacks, security vendors have begun to provide security measures to protect against SQL Injection. The first ones to claim such protection have been the various Web Application Firewall vendors, followed by most IDS/IPS vendors.

Most of this protection, however is Signature based. This is obviously the case with common IDS/IPS vendors, as they come from the network security world, and revolve around signature-based protection. However, most of the Web Application Firewalls base their SQL Injection protection on signatures as well. This is due to the fact that they inspect HTTP traffic only, and are able to look for attack patterns only within HTTP traffic. Moreover, it has lately become a common belief that signatures are indeed sufficient for SQL Injection protection. This belief has been backed up by an article recently published[2], describing, allegedly, a thorough guide for building SQL Injection signatures, in Snort™-like format.

The research done at Imperva's Application Defense Center shows, however, that providing protection against SQL Injection using signatures only is not enough. This paper demonstrates various techniques that can be used to evade SQL Injection signatures, including advanced techniques that were developed during the research.

The paper further demonstrates why these techniques are actually just the tip of the iceberg of different evasion techniques, due to the richness of the SQL language. Eventually, the conclusion that the research leads to is that providing protection against SQL Injection using only signatures is simply not practical. A reasonably sized signature database will never be complete, while an attempt to create a complete comprehensive signature database, even if theoretically possible, will yield an amount of signatures that is impossible to handle while maintaining a reasonable performance requirement, and is likely to generate too many false positives.

[1] The paper assumes that the reader has basic understanding of SQL Injection. Although this is not a mandatory requirement, it is advisable that the reader will be familiar with the basic concepts of SQL Injection.

[2] The article, named 'Detection of SQL Injection and Cross-site Scripting Attacks', written by K.K. Mookhey and Nilesh Burghate was published on March 2004 in SecurityFocus™.

Introduction

In this paper, we will look at a theoretical hacker, named *Autolycus*. In Greek mythology, Autolycus was an accomplished thief and trickster. He was a son of the god Hermes, who gave him the power of invisibility, a trait required for evading detection.

In ancient times, Autolycus spent his days sneaking into fortresses, through the main gate, pretending to be no other than a mere visitor. Once in the fortress, Autolycus headed towards the treasury and stole its entire contents. In modern times Autolycus found out that breaking into a system through the exposed Web application and stealing the entire contents of the database is as much fun, and involves substantially less physical risk.

One of Autolycus's most favorite techniques is called SQL injection. This technique allows him not only to steal the entire contents of databases but to make arbitrary changes to data as well. Autolycus has mastered this technique so well that by now he is now able to reach the treasury even when blindfolded[3].

When Autolycus first approaches our fortress he knows, as usual, nothing or very little about it. He is, however, determined as ever, to get through the main gate, sneak into the treasury, and steal what he thinks should be his. Based on his recent experiences, Autolycus believes today will not be any different, and proceeds through the main gate.

What Autolycus does not know at this point, however, is that a new old guard, known by the name of Signatorious, has been posted to our fortress. Signatorious has been around since the 'old days', when network security was the main issue. Back then Signatorious was posted to the external walls. Now, he is asked to examine all the passersby going through the main gate. Although unable to see Autolycus (he is invisible, after all), Signatorious has learned the pattern of common SQL Injection attacks, and can now identify and stop Autolycus, when attempting such attacks.

(Note: many of the examples shown in this paper refer to MS SQL Server, while some others refer to MySQL or Oracle. The reader must not conclude that one database or another is more or less vulnerable to evasion techniques, based on the number of examples shown here. There are many more specific examples, in any database, whether listed here or not, to be found. The basics concepts of these techniques, however, remain the same).

[3] This paper assumes that the protected Web server hides its error messages. It is recommended that the reader should be familiar with the techniques that allow performing SQL Injection without detailed error messages, as they appear in the paper 'Blindfolded SQL Injection', published in September 2003 by Imperva's Application Defense Center.

Recognizing Signature Protection

Autolycus, unaware of the new guard, sticks to his old habits, and is quickly stopped and thrown out of the fortress. Luckily, Autolycus has nothing stopping him from coming back a minute after being thrown out, and he repeats his attempts. Having enough experience in this technique, Autolycus quickly understands that something new is holding him back, and he moves on to identifying what is it that blocks his way to the treasury. It will not be long before Autolycus recognizes his foe from the old times, Signaturious.

In modern days, Autolycus is a savvy hacker. He makes a point of reading all relevant mailing lists, being up to date with new security features and security products, and understanding how they work. After noticing that all of his attempts are blocked, unrelated to their actual content, Autolycus is likely to understand, very quickly, that signature protection is set on the Web server, and that his requests are failing not because his SQL Injection is wrong, but because the Web server or the security gateway in front of it have noticed some SQL keywords.

Guessing, however, is never sufficient, and Autolycus decides to run a series of tests to verify that this is indeed the case. Luckily, for this test, Autolycus doesn't really need to be able to do any kind of SQL Injection. The mechanisms which recognize these signatures can not tell whether a parameter is subsequently transferred to an SQL query or not. The identification of the signature-trigging strings can therefore be easily done on any page with no relation to its actual functionality in the system.

Autolycus, therefore, needs for his first step to simply identify a location in the application where an arbitrary string, that is unlikely to trigger a signature, can be inserted without invoking any server side error. Using such a plain string allows eliminating the cases where other security mechanisms are stopping Autolycus. (For instance, inserting a string instead of an integer inside a numerical parameter may trigger a type mismatch error in the server. When detailed errors are hidden, this error can not be differentiated from the error generated by the signature mechanism.).

The insertion of such an arbitrary string into the HTTP request can be done in a variety of ways:

- Any free input field in any of the pages in the application (remember – the signature detection operates on all the pages in the site), including search fields, form submissions, etc.

- If no free input fields are presented, any of the parameters that are in string format can be examined to check if free string insertion is available.

- Alternatively, if no string fields exist at all in the system, a new parameter can be simply added to the request, and is likely to be ignored by the application itself. (Notice that some of the application security products would block such an attempt.) For example:

  ```
  …&id=43&testparam=ARBITRARY
  ```

- In other cases, where SQL Injection has already been detected, and assuming no signature exists for the SQL comment characters (-- or /* */), a working simple injection can be built and the suspicious pattern placed inside a comment. For example:

  ```
  …&dbid=originalid' -- ARBITRARY
  ```

- Another technique, assuming SQL Injection has already been detected, and no signature exists on the word AND, is to place the suspicious pattern inside a string literal:

  ```
  …&dbid=originalid' AND 'ARBITRARY'='ARBITRARY'
  ```

Eventually, in almost every application, a location can be found where an arbitrary string can be inserted without causing any other error. Once it is located, the second stage of the testing can take place.

In the second stage, Autolycus tries to verify whether indeed signature blocking takes place. This is simply done by replacing the arbitrary string with a string that is likely to trigger the signature mechanism, such as:

- `UNION SELECT`
- `OR 1=1`
- `EXEC SP_` (or `EXEC XP_`)

Assuming the site is indeed protected, any, or at least most of these requests will yield an error, generated by the signature mechanism. Autolycus now knows that signature protection is in place, and can move to the final step, namely enumeration (at least partial) of the signature list.

Sadly, this part is not very glamorous. It involves a tedious, methodical process, of trial and error. Autolycus simply takes a list of the attacks he uses during the hack, and tries them out one by one. Those that do not cause an error are listed as safe. Those that are blocked by the server are broken down into components, until the exact string or regular expression is identified. This may sound like a sisyphic project, but it normally does not take too long to identify those signatures that actually disturb the attack.

Common Evasion Techniques

After recognizing his archenemy Signaturious, Autolycus sits on the ground, next to the fortress, and plans his next move. He now knows which of his actions are monitored by Signatorious, yet this just builds up frustration. All of his normal moves are detected by this new guard. He goes back home, and digs up from his library a book titled 'The Oldest Tricks in the World'. "I was able to fool Signatorious in the old times", he thinks to himself, "Maybe there are some old tricks he still hasn't learned…"

Surprisingly enough, many of the new generation products fail where older generation products also failed in the past. It is therefore likely that the first move of Autolycus would be to try some of these old tricks, before proceeding to the advanced techniques. Some such common old tricks are presented below.

Different 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. One way or another, using a variety of encoding techniques, such as URL Encoding, UTF-8, etc. may prove useful.

White Spaces Diversity

Many of the signatures that are 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, such as SELECT, may generate a lot of false positives. The expression UNION SELECT, however, is quite unique to the SQL world, making it a good signature. This, however, carries the potential for the white spaces problem. If the signature is not accurately built, all Autolycus needs to do is replace a single space with two spaces, or a space with a tab, rendering the signature useless, and allowing the attack to take place.

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 thrown out of the fortress over and over again. A little voice in his head whispers "Let it go, there are so many other fortresses around". Autolycus gives in to the voice, and returns home. At night, however, he is restless. It can't be that he will 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, after dreams of the treasures found inside the fortress, he decides to try another move. He realizes that using old books from his library is not the solution, and he decides to sit down, and think of creative ideas of his own, and then to try again. "If I will come up with a good idea of my own, the guard will never catch me", he thinks to himself, and rightfully so.

We assume now that Autolycus has reached the point where common evasion techniques were not successful, and he moves on to the next step. We review here several techniques researched in Imperva's Application Defense Center that have proven successful in avoiding many common signature based protection mechanisms.

The 'OR 1=1' Signature

One of the common signatures used by such mechanisms is some sort of variant on the famous OR 1=1 attack. Signatures are usually built as a regular expression, aimed at catching as many possible variations of the attack. Sadly (or luckily, for Autolycus), many of them can be tricked by using sophisticated matches. For example, an unusual string can be used, such as:

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

Yet, with some better systems, this might not be enough. Autolycus 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, like this:

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

This character 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 for any signature driven 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 to use one of the < or > operators, like one of these examples:

```
OR 'Simple' > 'S'
OR 'Simple' < 'X'
OR 2 > 1
```

Or to use 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, anywhere in the string, by an SQL keyword or Meta character. This, however, is likely to result with 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 already mentioned before, common signatures nowadays include white spaces within them. Patterns like 'UNION SELECT' or 'EXEC SP_' (under MSSQL Server), provide relatively high accuracy for signatures. Other signatures, aimed at neutralizing the above described false positives may include signatures such as 'OR ' (an OR followed by a white space) or similar signatures.

In a previous section, a simple, common technique of replacing the number or type of white spaces was discussed. Many of the modern protection mechanisms, however, evolved past this stage, and can properly handle any combination of white spaces. As a result, a better technique is developed so that Autolycus can penetrate the site.

The 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 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), Autolycus can simply type:

```
…OrigText'OR'Simple'='Simple'
```

This works exactly the same way, but has no spaces in it whatsoever, completely evading any spaces based signature.

This, however, will not work for injections such as 'UNION SELECT', since there must be a separation between the two keywords. The solution is, therefore, to find a way 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 servers (this was tested on MS SQL, MySQL and Oracle).

Most readers are probably familiar with the double hyphen (--) comment syntax, that comments out everything up to a new line. However, another syntax for comments is supported by most database servers. This is the C-like syntax, using /* to start a comment and */ to end it. This means that a valid SQL statement can look like this:

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

Similarly, this can be taken into the injection code, generating an injection string 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 of 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 …
```

(The above example works in MS SQL and in Oracle, but not in MySQL. In MySQL, at least one space has to be present. It can be present, however, after the comment, which still allows evading the signature that holds the space right after the keyword).

This technique can also be used in SQL Injections to Oracle, to replace the OR 1=1 example showed above for MS SQL. Although Oracle will not allow omitting the white spaces, it does allow replacing them with a comment, which can lead to the following exploit:

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

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 Autolycus 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 **select**ed 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, Autolycus will need other techniques, which allow breaking strings in the middle.

Luckily, Autolycus 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 C-like comments. With MySQL, the C-like comments would not work as a replacement for a space. However, what was disturbing back there is now an advantage. The same comments can be used in MySQL to break words in the middle, for instance:

```
…UN/**/ION/**/ SE/**/LECT/**/ …
```

Another very promising prospect lies back in the string concatenation 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 Autolycus therefore needs to do, is find a way to build an exploit to the SQL Injection 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 executed[4]. 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 executing many signature defined statements.

Another good example, in MS SQL relates to the OPENROWSET statement. This technique was published in a white paper[5] over a year and a half ago, yet despite it being an old, known technique, most signature based products fail to look for it (for instance, the recently published article in SecurityFocus™, mentioned before, completely ignores this keyword). 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 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 obviously 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 evading existing signatures.

---

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

[5] The paper, 'Manipulating Microsoft SQL Server Using SQL Injection', written by Cesar Cerrudo, covers advanced techniques for exploitation of MS SQL Server, including the technique using OPENROWSET.

Conclusion

The next morning, Autolycus is home again. Next to him, lies the stolen treasure. "I did it again," he thinks to himself, as the sunlight reflects off the gold bars. 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 guards of the other fortresses to learn anything new from it.

Three days later, an angry man approaches the gates of the fortress. He claims he saw his mother's necklace, stored for safekeeping in the fortress treasury, on sale in the market. All the fortress' guards run down to the treasury, and find that indeed it was plundered. Signatorious then understands that he was outsmarted by Autolycus. Frustrated by his defeat, and in a desperate attempt to prove his worth, he starts to frantically stop anyone he imagines as suspicious, throwing out many innocent passersby out of the fortress. The chief of guards then understands that Signatorious is simply unable to fulfill his task, and relieves him from duty.

At this point, we believe the conclusion of this paper is clear to the reader. Signature protection against SQL Injection is simply not enough. Although this paper demonstrates only some of the variety of evasion techniques for avoiding SQL Injection signatures, some or even all of these techniques are likely to operate on most of today's signature protection mechanisms.

Sadly, the trivial solution of adding a few more signatures to your protection mechanism is not a real solution. It can help thwart some of the technique shown here, but other techniques can be developed. This is due to the richness of the SQL language implemented by many database vendors. The richness of the language means that many different statements can be sent to the server, all resulting in one specific operation.

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

That approach, however, has many disadvantages. Even if theoretically, covering *all* possible attacks and evasion techniques was possible, this would require building hundreds of signatures,

many of them fairly complex, regular expression based. Although providing reasonable good accuracy, this is simply not practical, performance wise.

Having several hundreds of signatures for every database type easily rises to over a thousand signatures in a diverse organization holding several different database types. These are on top of the already existing signatures for other attacks. The performance price (throughput and latency) is simply unacceptable for such a large number of signatures.

The second approach is to generate very few, very 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.

A fine example of this is provided by the previously mentioned 'Detection of SQL Injection and Cross-site Scripting Attacks' article. This article tried a reduced set of the second approach, using regular expressions seeking specific Meta characters, as well as regular expressions that allow the detection of specific key words, such as OR. Sadly, this is bound to result with many false positives. This amount of false positives may be acceptable in some IDS systems, but is definitely not acceptable in an IPS or a Web Application Firewall, which blocks any such request. Not many organizations will agree to block any user who studies in St. John's University, and wishes to fill in his personal details.

We are now at the point where the original claim has been proven. Using SQL Injection signatures is simply not a good enough mechanism to protect against SQL Injection attacks. Any attempt to create a signature base good enough to prevent all SQL Injection attacks is bound to fail for one of two reasons – too many false positives, or too many signatures, more than the mechanism can handle.

Although using SQL Injection signatures does, of course, provide a certain level of security (which pushes off many less capable hackers) it provides a false sense of security, as the more capable hacker would be able to penetrate through it.

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

Imperva, Inc.
12 Hachilazon st. Ramat-Gan **52522**, Israel
Tel: **+972 3 6120133** | Fax: **+972 3 7511133**
U.S. Toll Free: **1-866-592-1289**

1065 East Hillsdale Blvd., Suite 109
Foster City, CA 94404, USA
Tel: (650) 345-9000 | Fax: (650) 345-9004

info@imperva.com | www.imperva.com

IMPERVA™