The SQL Injection and Signature Evasion

Protecting Web Sites Against SQL Injection

SQL injection is one of the most common attack strategies employed by attackers to steal identity and other sensitive information from Web sites. By inserting unauthorized database commands into a vulnerable Web site, an attacker may gain unrestricted access to the entire contents of a backend database.

Network firewalls, IPS, and even some dedicated Web application firewall technologies attempt to identify SQL injection via traditional signature-based protections. Signature protections attempt to identify and block SQL injection-related text patterns within Web traffic flows. Unfortunately, real world experience has proven that reliance upon signatures alone is not enough to defeat SQL Injection.

This paper provides a detailed description of the SQL attack process by taking the reader though a hypothetical attack on a healthcare Web site. The paper then demonstrates a range of SQL injection evasion techniques that are commonly employed to circumvent traditional signature-based protections provided by network firewalls and intrusion prevention systems. The paper concludes that reliance upon signature protections alone to defeat SQL injection is not practical.

iMPERVA®

SQL Injection

SQL injection attacks expose sensitive database information by taking advantage of input validation vulnerabilities in Web site user interface software. In theory, Web sites validate all input, including character length and type, prior to sending queries to a backend database. However, if input validation is not carried out properly for each and every input (of which there may be thousands), an attacker may manipulate elements in a Web request to alter subsequent queries sent to a back-end database. The results of these unauthorized queries are then displayed as part of the HTML response generated by the Web site.

Consider the following simple example of a SQL Injection attack on a healthcare Web site.

> A module in the healthcare site lists Social Security Numbers (SSNs) of family members according to gender. The module is invoked with the following URL.
>
> http://www.superhealth.com/show_members.asp?gender=m
>
> The normal query subsequently sent to the database by the Web application looks like the following.
>
> ```
> select SSN, NAME from PATIENTS where FAMILY = XXX and gender = 'm'
> ```
>
> In this query, XXX represents the family identifier extracted from the database upon login. If the module is susceptible to SQL injection on the "gender" parameter, then the attacker may manipulate it by "injecting" additional characters as follows.
>
> http://www.superhealth.com/show_members.asp?gender=m' or 1=1 or '1'='1
>
> This URL effectively sends the following unauthorized query to the database.
>
> ```
> select SSN, NAME from PATIENTS where FAMILY = XXX and gender = 'm'
> or 1=1 or '1'='1'
> ```
>
> This query retrieves identification information for all patients in the entire database. The information is then displayed to the attacker by the Web application.

Variations on this technique can display social security numbers even if the original query does not address a table containing this social security numbers. The ability of SQL injection to achieve bulk retrieval of thousands of user names at once makes it one of the most dangerous identity theft threats.

Defending against SQL Injection with Signatures

Network firewalls, Intrusion Prevention Systems (IPS), and even some dedicated Web application firewalls attempt to defend against SQL injection using only signature protections. Signature mechanisms inspect network traffic flows looking for text strings or "signatures" that match known attacks. Certain strings are common to SQL injection and are therefore used as attack detection signatures. For example, the **"or 1=1"** string applied in the example above is a classic SQL injection string[1] and is therefore commonly applied as a signature. Most signature-based security products would easily identify this attack using this signature.

[1] The **"or 1=1"** string is applied as a SQL ***where*** override. It extends a SQL query from a single database record to include an entire column.

Unfortunately, today's attackers are well aware of signature detection technologies and are not deterred by initial failure of the classic "or 1=1" string. A range of evasion techniques are commonly applied to circumvent signature-based security. The remainder of this document describes a few of those techniques with a series of examples.

Note – Many examples described in this paper apply to MS-SQL Server. A few apply to MySQL and Oracle. The reader must not conclude that one database or another is more or less vulnerable based upon the ratio of examples herein. Many examples could be constructed for any database. The basics concepts of these techniques, however, remain the same for all databases.

Recognizing Signature Protections

Upon initial failure of SQL injection attempts without evasion, the attacker can assume that signature protection is in place. To confirm this assumption, a series of tests are run. The first step is to identify a Web site location where an arbitrary string that is unlikely to trigger a signature can be inserted without invoking a server error. Testing for arbitrary string insertion eliminates cases in which non-signature security mechanisms are the source of the problem. For instance, inserting an alphabetic string instead of an integer into a numerical parameter may trigger a type mismatch error on 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.

- Since signature protection operates on all Web pages in the site, any free input field on any page suffices. Examples include search fields and form submissions.

- If no free input fields are presented, any string format parameters can be examined.

- If no string fields exist in the system, a new parameter can be added to a request and is likely to be ignored by the application. Note that some application security products would block such a request. For example:

  …&id=43&testparam=ARBITRARY

- When 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

- When SQL Injection has already been detected and no signature exists on the word AND, a suspicious pattern may be place inside a string literal. For example:

  …&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.

Now, the second stage of the test can take place. The attacker tries to verify whether signature protections are indeed in place. This is simply done by replacing the arbitrary string with a string that is likely to trigger the signature mechanism. For example, if the site is protected most of the following requests will yield an error.

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

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, the attacker tries the SQL injection strings that he normally needs to carry out an attack. Those that do not cause an error are listed as safe. Those that are blocked are broken down into components until the exact string or regular expression is identified. This may sound like a never-ending project, but it normally does not take too long to identify the specific signatures that may disturb an attack.

Basic Evasion Techniques

With a list of signatures use to protect the Web site in hand, most attackers apply basic evasion techniques before proceeding to more advanced techniques. A few of these basic techniques are presented below.

Encoding

Encoding tricks have proven useful throughout the history of computer attacks. The reasons for this are many. Some security products simply fail to decode properly. Others decode properly, but performance requirements limit what can be done in real time. One way or another, a variety of encoding techniques such as URL Encoding and UTF-8 are often used to hide attacks from the prying eyes of signature detection technologies.

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 unique to the SQL world making it a better signature. This, however, introduces the opportunity for white space evasion. If the signature is not carefully defined, the attacker may avoid detection while preserving the integrity of his attack by replacing the single space between words with two spaces, a space plus a tab, or a comment.

IP Fragmentation and TCP Segmentation

Another evasion technique seeks to hide an attack from signature mechanisms by dividing the string into multiple packet fragments. If the signature mechanism does not reassemble the packet fragments, it does not match the attack string to a signature since each packet, inspected individually, will only include part of the attack string.

Advanced Evasion Techniques

If none of the previous basic evasion techniques are successful, the attacker will move on to more advanced techniques. The advanced techniques presented below, can be applied to evade virtually any signature-based security device.

OR 1=1 Signature Evasion

One of the common SQL injection signature categories defend against on the classic “or 1=1” attack described above. Signatures are often built as a regular expression, aimed at catching as many possible variations of the “or 1=1” attack. Sadly (or luckily, for attackers), many can be tricked by using equivalents such as the following.

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

Yet, some of the better signature detection systems will still identify such a simple equivalent. Therefore, the attacker must find a way to make the two expressions look different to the signature device while retaining the same SQL meaning. A very simple trick is to add the character N prior to the second string as follows:

```
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 SQL comparison, but definitely makes it different for any signature driven mechanism.

An even better approach would break one of the strings into two, concatenating it at the SQL level. This will render useless any mechanism which compares the strings on both sides of the = sign.

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

One of the above mentioned techniques is likely to evade most any signature mechanism. Yet, some vendors might choose a much more general regular expression to cope with this attack. For example a signature that looks for the **"or"** word followed by an **"="** anywhere a message. Such a generic signature is likely to lead to false positives since some combination of **"or"** and **"="** is likely to legitimately occur within normal Web content and/or software. But even if it did not lead to false positives, it can also be easily evaded by simply finding an expression which evaluates as true, without including the equal sign. For instance, replacing the equal sign with the SQL word **"LIKE"** (a partial compare) achieves the desired result.

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

Alternatively, the attacker might choose to use **"<"**or **">"** operators.

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

Or, the attacker may apply **"IN"** or **"BETWEEN"** statements.

```
OR 'Simple' IN ('Simple')
OR 'Simple' BETWEEN 'R' AND 'T'
```

The opportunities go on and on. 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 damage performance. Another possibility is, of course, to define signatures that are extremely general, such as an **'OR'** followed anywhere by any SQL keyword or Meta character. This, however, results in many false positives. Consider the following URL.

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

Although far from being an invalid URL, it triggers a false positive alert for such a general signature. Clearly, this is not a solution.

Evading Signatures with White Spaces

As mentioned previously, signatures commonly include white spaces. Stings such as **"UNION SELECT"** or **'EXEC SP_'** provide relatively accurate signatures. Other signatures, aimed at neutralizing **"or 1=1"** false positives, may include strings such as 'OR ' (an OR followed by a white space).

In a previous section the basic technique of replacing the number or type of white spaces was discussed. Many modern signature mechanisms, however, have evolved to properly handle any combination of white spaces. As a result, a new technique has been developed to counter these newer signature mechanisms. The new technique takes advantage of vendor specific SQL parsing decisions to create valid SQL statements 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 fundamental idea behind the new technique, which operates on databases that perform a rather loose (and more user-friendly) SQL parsing, is to simply drop the white 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 '. For example,

```
…OrigText' OR 'Simple' = 'Simple'
```

may be replaced by,

```
…OrigText'OR'Simple'='Simple'
```

The two represent completely equivalent SQL, but the second contains no white spaces. Any space-based signature is evaded. 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. For example, one common syntax uses a "/*" to start a comment and "*/" to end it. This means that a valid SQL statement may be constructed as follows.

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

This idea can be applied by 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 attack. Moreover, in most cases the **"/**/"** can replace the spaces allowing evasion of more sensitive signatures such as **"SELECT "** or **"INSERT "** (a SQL keyword followed by a single space), which have been noted to be used by some SQL signature protection mechanisms. The previous example would then appear as follows.

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

This technique can be used in Oracle SQL Injections as an OR 1=1 replacement. Although Oracle does not allow omission of white spaces, it does allow replacing them with a comment. This leads to the following exploit.

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

This technique is also exploited for evasion (especially for Web application firewalls that check the signatures on the parameter value only) when two separate parameters are inserted into the SQL statement. Imagine a login page with the following request.

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

This request then generates the following query.

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

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

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

This results in the following query, which easily logs the attacker into the Web site.

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

As with the previously described techniques for 'OR 1=1' evasion, there is no good signature-based solution here. SQL keywords such as **"SELECT"** and **"INSERT"** may be applied as signatures, but as with the **"OR"** keyword, the result is false positives. 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 triggers a signature match on the word "select", with no attack in progress. Adding more generic signatures increases the frequency of false positives attackers. At the same time, attackers have a never ending list of evasion alternatives to choose from.

Evading Any String Pattern

Although standalone keywords are likely to generate false positives, some sites may choose to apply such signatures while limiting site functionality so that no free user inputs are available. For instance, the main portion of a banking site may not allow free text user inputs. In this case, other techniques for breaking strings into parts are needed.

Again, many options are available. The first technique goes back to the C-like comments. Although C-like comments do not work as a replacement for a white space in MySQL, they can be used to break words into parts. For instance, the following represents valid MySQL syntax.

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

Another very promising prospect returns the discussion to string concatenation. Most databases allow the user to execute a SQL query through one or more statements, like built in operations or stored procedures, that receive a SQL query as a string. All that the attacker needs to do, therefore, is to build a

SQL injection that allows the execution of such a string. Once the exploit is created, all signatures can be evaded simply by using string concatenation within 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. The string can be concatenated as follows.

```
…; 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 will receive a string containing an SQL query and execute it. Naturally, this problem is not limited to MS SQL. Other databases suffer from the same problem.

An interesting twist on this attack, relies upon a hexadecimal encoding of the string to be executed[2]. 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 execution of many supposedly signature protected statements.

Another good MYSQL example, relates to the OPENROWSET statement. Since OPENROWSET receives a string parameter, a concatenated attack query may be inserted without being detected by a signature mechanism. This technique was published[3] years ago, yet most signature based products fail to detect it.

One may argue that the number of statements that can be used for such a technique is limited within each database. Although this is true to some extent, it is also true that consistent construction of signatures is not likely.

An excellent example is provided by MS SQL, which contains *unlisted* stored procedures 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 SQL Server listing, they are obviously likely to be missing from any SQL Injection signature database. Obviously, similar undocumented procedures and functions exist in other databases.

Signatures Alone are not Enough

Hopefully, at this point one conclusion is clear. Signatures are not effective against SQL Injection as a standalone solution. Any attempt to create a signature base for all SQL Injection attacks is bound to fail for one of two reasons – poor performance or false positives.

The inherent flexibility of the SQL language provides attackers with a never ending toolkit of evasion options. Even if coverage for all possible evasion techniques were possible, the task would require construction of several hundred complex, regular expression-based signatures for each database type. Although this would deliver reasonable accuracy, it is not practical from a performance perspective. Hundreds of signatures per database type results in over one thousand signatures for a diverse organization with several database types.

[2] Described in "(More) Advanced SQL Injection" by Chris Anley
[3] "Manipulating Microsoft SQL Server Using SQL Injection", Cesar Cerrudo

This is in addition to existing signatures for other attacks. The performance price in terms of throughput and latency is simply unacceptable for such a large number of signatures.

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

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

It would also include relevant Meta characters and their encodings.

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

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. Signatures are useful as an attack indicator, but as definitive attack detection technology – they need help.

Preventing SQL Injection with the SecureSphere Web Application Firewall

One approach to reliably identifying SQL injection is to look for multiple pieces of corroborating evidence. For example, a security manager with the task of tracking security alerts may notice a SQL injection signature alert from his intrusion detection system. He might then look for corresponding anomalies in his database log files. If he finds unusual database activity occurring in parallel with SQL related Web signatures, 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. He may now block the user with confidence. This is exactly the approach taken by the SecureSphere Web Application Firewall – 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. Security violations from each of these technology layers are automatically correlated to achieve a degree of accuracy that cannot be matched by using signature protections alone[4].

Advanced IPS Identifies SQL Injection Characters

SecureSphere's advanced IPS includes advanced SQL injection signatures designed to detect any combination of characters related to SQL injection. SecureSphere's advanced SQL Injection signatures, along with other database attack signatures, are provided and updated weekly by Imperva's international security research organization, the Application Defense center.

The "or 1=1" string discussed in the healthcare example above is an obvious attack that SecureSphere IPS immediately blocks with an exact signature match. However, to deal with the range of evasion techniques described previously, SecureSphere takes a more sophisticated approach.

First, SecureSphere applies a reasonably-sized list of generic signatures that detect virtually any SQL injection attack. For example, suppose the attacker attempts to evade the **"or 1=1"** signature with an equivalent such as **"Unusual = Unusual"**. The threat is identified by a special SecureSphere IPS signature that looks for any combination of **"or"** and **"="** within URL parameters and a **Signature Violation Alert is issued**. However, since **"or"** and **"="** are common elements of legitimate parameters, a match on this

[4] SecureSphere also includes network firewall and protocol compliance security technologies for protection against other attack vectors such as zero-day Web worms and application layer DoS attacks.

signature cannot be blocked without occasional false positives. To clarify the nature of this alert, more corroborating evidence is needed. SecureSphere collects such evidence through Dynamic Profiling.

Dynamic Profiling Identifies Unusual Parameters

SecureSphere's Dynamic Profiling technology examines live traffic to automatically create a comprehensive model or "profile" of the site. Specific elements of the profile include dynamic URLs, http methods, cookies, parameter names, parameter lengths, and parameter types. The profile then serves as a positive-model security policy for the Web application. By continuously comparing user interactions to the profile, SecureSphere can detect any unusual Web or database activity. As the Web site changes over time, advanced learning algorithms automatically update the profiles to eliminate any need for manual tuning.

Figure 1 presents a SecureSphere Dynamic Profile of the "gender" parameter corresponding to the healthcare example above. The model identifies the parameter as a required parameter consisting of Latin characters with a maximum length of one character. The insertion of more than one character into the parameter (**OR Unusual = Unusual**) conflicts with the profile and a SecureSphere Parameter Length Violation Alert is triggered.

At this point in our example, SecureSphere has been able to detect two different security violations within the same Web request: an IPS violation and a Dynamic Profile violation. Even if the attacker has used evasion techniques (**OR Unusual = Unusual** instead of **OR 1=1**) to avoid outright blocking by the signature-based IPS, the Dynamic Profile violation may be used to validate the attack. All that is necessary is to link these events to the same user. SecureSphere's Correlated Attack Validation delivers that capability.

Correlated Attack Validation Confirms the Attack

SecureSphere's Correlated Attack Validation (CAV) correlates multiple events, such as profile violations, application signatures, number of occurrences and user name, to more accurately identify SQL injections. If a user triggers multiple violations that match an attack pattern, malicious intent is confirmed with high accuracy. In the example above, CAV correlates a SQL Injection signature violation (even a low accuracy signature such as **"or"** combined with **"="**) with the parameter length violation to validate that an attack is indeed in progress. By linking multiple violations to the same user, SecureSphere is able to accurately identify attacks even when sophisticated evasion techniques are used.

More Information

The examples above only scratch the surface of SecureSphere capabilities. Any combination of SecureSphere security technology layers (Dynamic Profile, application attack signatures, protocol validation, network firewall) may be applied individually or correlated to defeat a range of attack vectors without risk of false positives. For more information see http://www.imperva.com/products/securesphere/resources.asp.

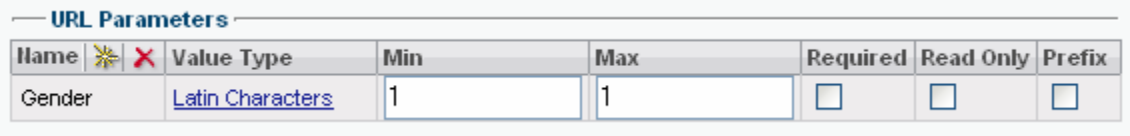

Figure 1: SecureSphere automatically builds a model of each Web application parameter

References

1. Basic SQL Injection Overview

Imperva's on-line glossary

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

2. Blindfolded SQL Injection

By Ofer Maor and Amichai Shulman

http://www.imperva.com/application_defense_center/papers/

3. (More) Advanced SQL Injection

By Chris Anley

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

4. Manipulating Microsoft SQL Server Using SQL Injection

By Cesar Cerrudo

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

US Headquarters
950 Tower Lane
Suite 1550
Foster City, CA 94404
Tel: (650) 345-9000
Fax: (650) 345-9004

International Headquarters
125 Menachem Begin Street
Tel Aviv 67010
Israel
Tel: +972-3-684-0100
Fax: +972-3-684-0200