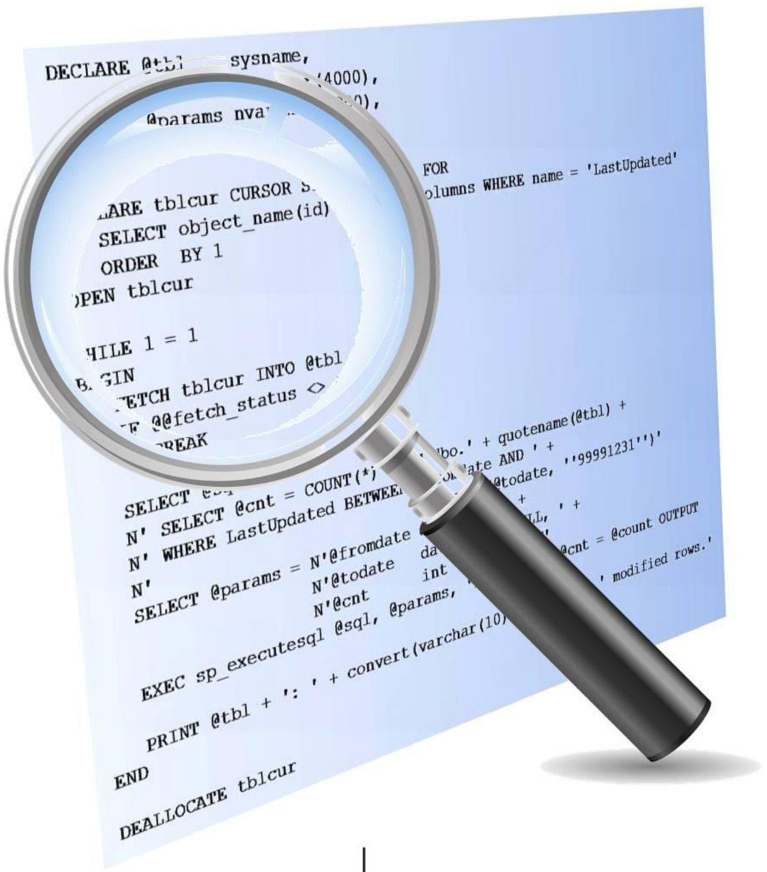
SQL Injection Attacks:

Detection in a
Web Application Environment



DB NETWORKS

Table of Contents

1 Foreword

It has been over a decade since the original research paper on SQL Injection was published. Over the years the SQL Injection threat has grown to the point where now we are seeing weaponized SQL Injection attacks perpetrated by state actors. Organizations are continually being breached via SQL Injection attacks that slip seamlessly through the firewall over port 80 (HTTP) or 443 (SSL) to soft internal networks and vulnerable databases.

Detecting SQL statements injected into a Web application has proven extremely challenging. There are several tacks organizations can take – prevention, remediation, and mitigation. When implementing prevention and remediation efforts, the enterprise strives to develop secure code and/or encrypt confidential data stored in the database. However, these are not always available options. For example, in some cases the applications source code may have been developed by a third party and not be available for modification. Additionally, fixing deployed code requires significant resources and time. Therefore, rewriting an existing operational application would need to be prioritized ahead of projects driving new business. Similarly, efforts to encrypt confidential data stored in the database can take even longer and require more resources. Given today’s compressed development cycles and the limited number of developers with security domain experience, even getting the code rewrite project off the ground is often a daunting task.

An often suggested approach to identifying SQL injection threats is the use of a Web Application Firewall (WAF). A WAF operates in front of the Web server and monitors the traffic into and out of the Web servers. A WAF attempts to identify threat patterns (see Figure 1). While this can be effective in detecting certain classes of attacks against Web applications, it has proven extremely limited in detecting SQL injection attacks.

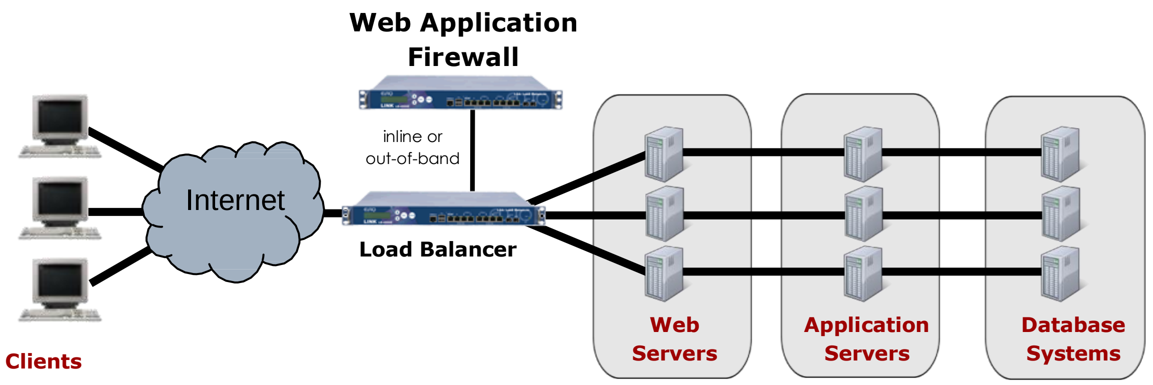


Figure 1 - Network Placement of a Web Application Firewall

This shouldn't be taken to infer a WAF isn't a useful component within a Web security environment. To the contrary, WAFs offer reasonable protection from header injection and XSS attacks. A WAF should always be considered as part of a Web security "defense in depth" strategy. However, with regards to SQL injection prevention specifically, organizations are turning to truly an effective countermeasure, adaptive database security appliances based on behavioral analysis. To better understand why WAFs have largely proven ineffective against SQL Injection threats, simply conduct a Web search for the phrase "WAF bypass". You'll find thousands articles, including tutorials offering step-by-step instructions as to how to defeat a WAF.

In addition to WAFs, recently Database Activity Monitoring (DAM) vendors are beginning to promote their products capability with regards to the SQL injection threat. DAMs were originally developed to monitor the activities of internal privileged database users such as the database administrator. SQL injection isn't an inside threat, it's a cyber security threat. So in this regard a DAM is not a useful countermeasure against a SQL injection attack. However, some DAMs include extrusion detection mechanisms that look at data leaving the database for specific data patterns such as social security numbers, credit card numbers, or a high volume of records. To avoid detection, attackers will often attempt to extract small amounts of data over a very long period of time in an attempt to "stay under the radar".

2 Background

2.1 Web Application Environment

Before we begin a discussion on the approaches to effectively detect and prevent SQL injection attacks, let’s first explore the Web application environment. Web application information is presented to the Web server by the user's client in the form of URL’s, cookies and form inputs (POSTs and GETs). These inputs drive both the logic of the application and the queries those applications dynamically create and send to a database to extract relevant data.

Unfortunately, too often applications do not adequately validate user input and are thus potentially susceptible to SQL injection attacks. Attackers capitalize on these flaws to attempt to cause the backend database to do something different than what the application (and the organization) intended. This can include divulging sensitive information, destroying information, or executing a database denial of service (DoS) attack that limits others' use of the application.

2.2 SQL Injection Attack Overview

SQL injection attacks are typically initiated by manipulating the data input on a Web form such that fragments of SQL instructions are passed to the Web application. The Web application then combines these rogue SQL fragments with the proper SQL dynamically generated by the application, thus creating what is seen as a valid SQL request. These new, unanticipated requests cause the database to perform the task the attacker intends.

To clarify, consider the following simple example. Assume we have an application with a Web page containing a simple login form with input fields for username and password. With these credentials the user can get a list of all credit card accounts they hold with a bank. Further assume that the bank’s application was built with no consideration of SQL injection attacks.

In this case, it is reasonable to assume the application merely takes the input the user types and places it directly into an SQL query constructed to retrieve that user's information. In PHP that query string might look something like this:

```
$query = “select accountName, accountNumber from
creditCardAccounts where username='”.$_POST[“username”].”'
and password='”.$_POST[“password”].”'”
```

Normally this would work properly as a user entered their credentials, say johnSmith and myPassword, and formed the query:

```
$query = "select accountName, accountNumber from
creditCardAccounts where username='johnSmith' and
password='myPassword'
```

This query would return one or more accounts linked to Mr. Smith.

Now let's consider someone with a devious intent. This person attempts to access the account information of one or more of the bank's customers. To accomplish this they enter the following credential into the form:

```
' or 1=1 -- and anyThingAtAll
```

When this SQL fragment is inserted into the SQL query by the application it becomes:

```
$query = "select accountName, accountNumber from
creditCardAccounts where username='' or 1=1 -- and
password= anyThingAtAll
```

The injection of the term, ' or 1=1 --, accomplishes two things. First, it causes the first term in the SQL statement to be true for all rows of the query; second, the -- causes the remainder of the statement to be treated as a comment and, therefore, ignored during run time. The result is that all the credit cards in the database, up to the limit the Web page will list, are returned and the attacker has stolen the valuable information they were seeking.

It should be noted that this very simple example is just one of literally an infinite number of variations that could be used to accomplish the same attack and there are many other ways to exploit a vulnerable application. We will discuss more of these attacks as we delve into the efficacy of various attack mitigation techniques.

2.3 Applications Vulnerable to SQL Injection

There are a number of factors that conspire to make securely written applications a rarity. First, many applications were written at a time when Web security was not a major consideration. This is especially true of SQL injection. While recently SQL injection is being discussed at security conferences and other settings, the attack frequency of SQL injection only five or so years ago were low enough that most developers were simply not aware.

In addition, the application may have been initially written as an internal application with a lower security threshold and subsequently exposed to the Web without considering the security ramifications. Even applications being written and deployed today often inadequately address security concerns. IBM Managed Security Services group distinguishes SQL injection attacks as one of the continued top attacks experienced by client networks. IBM also noted a dramatic and sustained rise in SQL injection-based traffic due, in large part, to a consistent effort from the Asia Pacific region. The alerts came from all industry sectors, with a bias toward banking and finance targets.[1] According to Neira Jones, head of payment security for Barclays, 97% of data breaches worldwide are due to SQL injection somewhere along the line. [2]

Interestingly, modern environments and development approaches create a subtle vulnerability. With the advent of Web 2.0 there has been a shift in how developers treat user input. In these applications, input is rarely provided by a simple form that directly transmits the information into the Web server for processing. In many cases, the JavaScript portion of the application performs input validation so the feedback to the user is handled more smoothly. This often creates the sense that the application is protected because of this very specific input validation; therefore, the validation on the server side is often largely neglected. Unfortunately, attackers won't use the application to inject their input into the server component of the application. Rather, they leverage intermediate applications to capture the client-side input and allow them to manipulate it. Since the majority of the input validation is bypassed, the attacker can simply enter the SQL fragments needed to change the behavior of the database to accomplish their intent.

[1] IBM Internet Security Systems^{TM} X-Force® Trend and Risk Report, March 2013
[2] Techworld, "Barclays: 97 percent of data breaches still due to SQL injection", January 2012

3 The challenge with detection

3.1 Effective Security

The goal of any security technology is to provide a robust threat detection and avoidance mechanism that requires little or no setup, configuration, or tuning. Further, if that technology relies on learning or training to determine what is normal or to improve its ability to detect threats, those learning periods must be very short and well-defined. This is necessary to expedite installation and minimize the risk of attacks contaminating the learned dataset. Keep in mind the longer the learning period, the more likely an attack will occur and the larger the dataset you need to review to ensure that an attack has not occurred. Finally, given that few Web applications remain static, effective protection must be easy to maintain in the face of on-going changes to the Web application.

3.2 Types of attacks

Previously a simple attack on a vulnerable application was described to illustrate how a SQL Injection attack can occur. This general class of attacks is known as a Tautological attack. Tautologies are statements composed of simpler substatements in such a way that makes the overall statement true, regardless if simpler substatements are true or false. For example, the statement, "Either it will rain tomorrow or it will not rain tomorrow" is a tautology because the statement will be true regardless if it rains.

The complexity of detecting SQL injection can best be understood through a variety of examples demonstrating the various SQL injection attack classifications. This list is not exhaustive, but rather provides a sample of the most common SQL injection techniques seen in real deployments.

3.2.1 Tautologies

This attack works by inserting an "always true" fragment into a WHERE clause of the SQL statement. This is often used in combination with the insertion of a double dash -- to cause the remainder of a statement to be ignored, ensuring extraction of largest amount of data. Tautological injections can include techniques to further mask SQL expression fragments, such as the following:

```
' or 'simple' like 'sim%' --
' or 'simple' like 'sim' || 'ple' --
```

The || in the example is used to concatenate strings, when evaluated the text 'sim' || 'ple' becomes 'simple'.

3.2.2 Union Query

This attack exploits a vulnerable parameter by injecting a statement of the form:

```
foo'UNION SELECT <rest of injected query>
```

The attacker can insert any appropriate query to retrieve information from a table different from the one that was the target of the original statement. The database returns a dataset that is the union of the results of the original first query and the results of the injected second query.

3.2.3 Illegal/Logically Incorrect Queries

Attackers use this approach to gather important information about the type of database and its structure. Attacks of this nature are often used in the initial reconnaissance phase of the attack to gather critical knowledge used in subsequent attacks. Returned error pages that are not filtered can be very instructive. Even if the application sanitizes error messages, the fact that an error is returned or not returned can reveal vulnerable or injectable parameters. Syntax errors identify injectable parameters; type errors help decipher data types of certain columns; logical errors, if returned to the user, can reveal table or column names.

The specific attacks within this class are largely the same as those used in a Tautological attack. The difference is that these are intended to determine how the system responds to different attacks by looking at the response to a normal input, an input with a logically true statement appended (typical tautological attack), an input with a logically false statement appended (to catch the response to failure) and an invalid statement to see how the system responds to bad SQL. This will often allow the attacker to see if an attack got through to the database even if the application does not allow the output from that statement to be displayed.

There are a myriad of examples. In fact, the attacker may initially use a bot to detect a vulnerable web site and then recursively use this class of attack forensically to learn application and database specifics. It should be pointed out that WAFs are unable to detect such attacks if the injections fall outside of the signatures used by the WAF.

3.2.4 Stored Procedure Attacks

These attacks attempt to execute database stored procedures. The attacker initially determines the database type (typically through illegal/logically incorrect queries) and then uses that knowledge to determine what stored procedures might exist. Contrary to popular belief, using stored procedures does not make the database invulnerable to SQL injection attacks. Stored procedures can be vulnerable to privilege escalation, buffer overflows, and even provide administrative access to the operating system.

3.2.5 Alternate Encoding Obfuscation

In this case, text is encoded to avoid detection by defensive coding practices. It can also be very difficult to generate rules for a WAF to detect encoded input. Encodings, in fact, can be used in combination with other attack classifications. Since databases parse comments out of an SQL statement prior to processing it, comments are often used in the middle of an attack to hide the attack’s pattern.

Scanning and detection techniques, including those used in WAFs, have not been effective against alternate encodings or comment based obfuscation because all possible encodings must be considered.

Note that these attacks may have no SQL keywords embedded as plain text, though it could run arbitrary SQL.

3.2.6 Combination Attacks

Many attack vectors may be employed in combination:

- Learn information useful in generating additional successful injections (illegal/logically incorrect)
- Gain access to systems other than the initial database accessed by the application (stored procedures)
- Evade detection by masking intent of injection (alternate encoding)

3.3 Detection at the Web Tier

3.3.1 Detecting SQL Injection Challenges

Given the large variation in the form or pattern of SQL attacks, it can be very challenging to detect such attacks in front of the Web server. At this network location the WAF is attempting to identify a possible fragment of SQL in the input stream of a Web application.

Why is it so difficult to detect SQL injections at the Web tier? Recall, the WAF is not inspecting the SQL requests actually being sent to the database. Rather, the WAF has URL's, cookies and form inputs (POSTs and GETs) which it inspects. While inspecting each set of input values a WAF must consider the wide range of acceptable input against what is considered unacceptable for each input field on each form.

Although many attacks use special characters that may not be expected in a typical Web form, two problems complicate detection. With no prior knowledge of the application it is impossible to know with certainty what characters are expected in any given field. Furthermore, in some cases the characters used, in fact, occur in normal input and blocking them at the character level is not possible. Consider the single quote often used to delimit a string. Unfortunately, this character appears in names such as *O'Brien* or in possessive expressions like *Steve's*; therefore, single quotes are valid in some input fields.

As a result larger patterns must be considered that are more demonstrative of an actual attack to bring the false positives down to a reasonable rate. And this is where the problem begins. The choice then becomes to either use a very general set of patterns (such as checking for a single quote or the word "like") to catch every conceivable attack but have extremely high false positives or to implement a more complicated pattern matching that reduces the false positive rate but misses the advanced SQL injection attacks.

Since there is a reasonable likelihood that general patterns exist in normal input, the WAF must then inspect all form input (in learning or training mode) for an extended period of time before it can determine which of these simple patterns can reliably be used to validate each form and each input field in the Web application. Considering the complexity, range and limited structure within the natural language used in forms, it can take a very long time to ensure that an adequate sample size has been gathered to confirm that selected detection patterns are not found in legitimate input. Complicating this further is the fact that some sections of an application are often used infrequently, extending even further the training time. An example would be routines that are only run at the end of the quarter or end of year. Add it all up and you can see this approach requires an extensive time period to ensure that the learning cycle has adequately considered all the variations of valid input for each field on each form of the Web application.

Alternatively, much more complex patterns that are clearly indicative of an attack can be used. Unfortunately, as we demonstrated in our discussion of the attack types, the number and variation of possible attacks is so large that it is not possible to effectively cover all attack patterns. Creating the initial pattern set, keeping up with the evolving attacks and verifying that they are sufficiently unique as to not show up in some fields is simply not operationally possible. Further, consider these applications are also changing and evolving over time, requiring further, time-consuming learning.

3.3.2 Web Tier Detection in Practice

So how are WAF's used in the real world? One way is to use a combination of approaches, each aimed at reducing the negative effects of the other approach. These negative effects include limited capability to detect a SQL injection versus a high number of false positives, complex configurations, and long training times. Specifically, a large set of patterns ranging from relatively simple to much more complex are used. Some patterns are configured to be applied to all input sources regardless of what is learned during training; some patterns are configured such that they will be removed, for a given input field, if they are contained within the training data. Some rules and patterns also attempt to classify the range of input by length and character set, for example, numerical fields.

The WAF is then placed into learning mode and allowed to learn until it is believed that a large enough set of each input field has been examined to reduce subsequent false positives. The resulting sets are then reviewed to determine if the learned set for some fields is considered too small, requiring additional learning time or manual manipulation. Other fields, whose default rule set have been reduced too far, are reviewed to determine what hand crafted rules can be constructed to increase the coverage.

This manual inspection process on top of the long learning cycle, while more effective than any one approach in isolation, is far from efficient. However, it still suffers the weaknesses of an administrator having to make decisions, configuring a significant number of rule/pattern sets for fields not effectively configured through training. This can be true even after a substantial learning period has been used.

This, in a nutshell, is why WAFs have been ineffective in curtailing SQL injection attacks. It's self evident, had WAFs actually been effective, the size and scope of SQL injection attacks would not be increasing year over year.

3.4 A better way – a Database Firewall

Thus far we have described the method of detecting SQL injection attacks at the Web tier interface. A more effective and efficient method is to analyze the actual SQL generated by the application and presented to the database. A Database Firewall monitors the network between the application servers and databases (see Figure 2). Why is this more effective and more efficient? The simple answer is that while the input into the Web tier has an enormous pattern set with very little structure associated with each input field, an application creates a comparatively small set of SQL statements (ignoring the literal values associated with those statements). In addition, the structure of SQL statement is conducive to structured analysis. Both of these factors make analysis more determinant than the rudimentary input pattern validation of a WAF. We will discuss how to deal with the variation of the literal values (the actual intended user input) below.

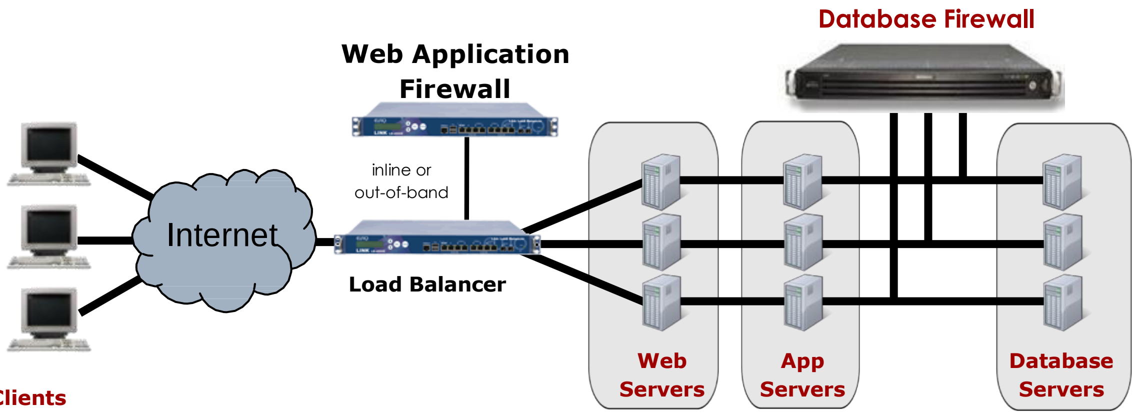


Figure 2 - Placement of Database Firewall

At the database interface, SQL statements can be processed in much the same way the database itself processes them – breaking them down into the statement structure and separating out the literals. Once this is accomplished, the very first appearance of any given input will generate the unique SQL statements associated with that input – as opposed to needing a large sample set to determine what patterns are not present.

As a result, the sample set for learning is immediately reduced from that required for a WAF to a much smaller set needed to train a device inspecting traffic between the application and database. Once a training set is operational, it can be used to analyze all subsequent SQL statements and any structural differences from the known set can be immediately flagged. By inspecting traffic at the interface to the database it is clear which commands are leveraging stored procedures and it is easy to analyze the strings passed to stored procedures to determine if any attacks are present. Several techniques can be applied in this analysis, such as observing the lack of delimiting special characters within literal strings.

Although analyzing the stream of SQL statements as described above provides a significant improvement over a WAF sitting at the Web tier, a true Database Firewall requires additional capabilities.

As pointed out during the discussion about training a WAF, many of the input fields within an application may not be exercised often during normal operations. Fortunately, most modern applications build their SQL from a set of logic that operates much like a code generator. This fact means that, using a relatively small sample set, it is possible to construct a behavioral model of how an application builds statements. An Adaptive Database Firewall can then use that behavioral model to analyze newly discovered SQL statements and assess their likelihood of being an attack.

SQL injection attacks must be constructed out of an existing statement in the application. This fact further simplifies the analysis of locating rogue SQL. If a new statement can be created wholly by inserting a string into the literal field of an existing statement, then it becomes highly suspect. Combining these concepts provides a means of assessing any new statement using algorithms that determine:

- Uniqueness relative to other statements previously seen
- Ability for the new statement to have been constructed from a previously known statement
- Likelihood that the statement could have been generated within the application itself

Although an Adaptive Database Firewall may use a number of other important algorithms for conducting SQL behavioral analysis, the three algorithms above demonstrate the core value of operating at the interface to the database. No other approach can achieve the accuracy provided with this architecture. Furthermore, no other solution can be deployed with as little configuration and as short a training interval.

4 Conclusions

The efficacy of a database security solution is measured by the robustness of its attack detection mechanisms, its ease of setup, configuration, tuning, and its ability to detect SQL injection attacks with low false positive rates. Using these metrics, a true Adaptive Database Firewall, based on behavioral analysis, is vastly superior to a WAF in identifying SQL injection attacks. This is true because an Adaptive Database Firewall can be trained quicker, has minimal false positives, and is capable of seeing through attack obfuscation techniques which slip easily through WAFs.

In the end, a multi-layer Web security strategy is the best solution, drawing on the strengths of all relevant technologies. Considering the seriousness of the SQL injection threat, an Adaptive Database Firewall should be a prominent element in every Database security solution.