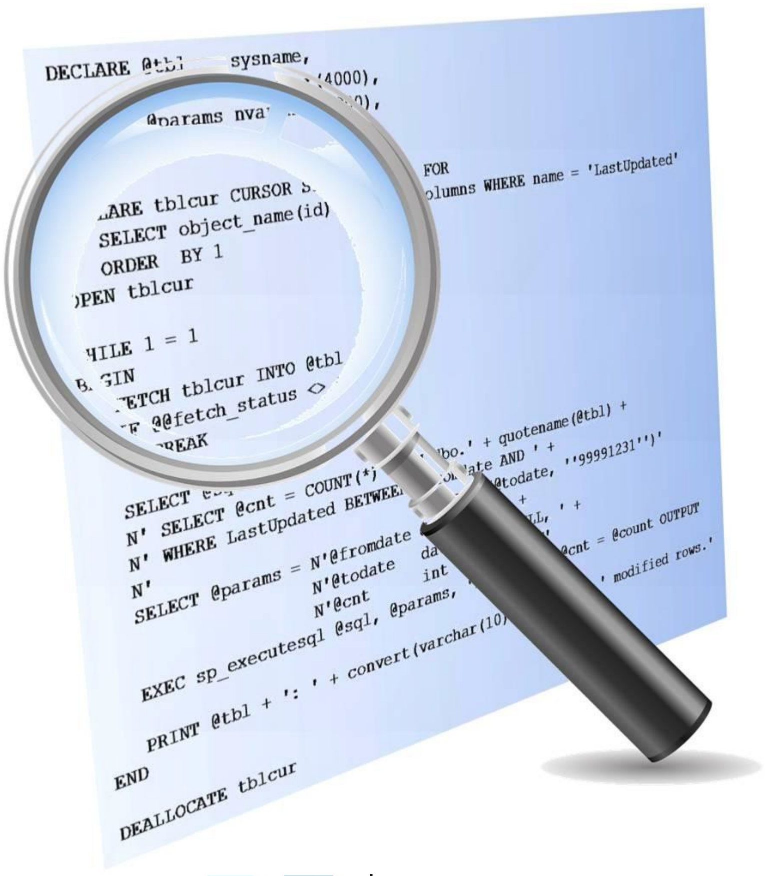
SQL Injection Defense: There are no Silver Bullets



NETWORKS®

www.dbnetworks.com

Table of Contents

Foreword

The SQL injection threat consistently ranks among the top cyber attack vectors and has haunted organizations for nearly two decades. Over that time the threat has become more widespread and evolved to be far more potent. According to Neira Jones, the former head of payment security for Barclaycard, the vast majority of database attacks worldwide are still due to SQL injection somewhere along the attack chain.

With each breach where SQL injection was the weapon of choice, another organization becomes keenly aware of the threat's potency - the hard way. Following each major breach it's common for information security pundits to march out their favorite single silver bullet solution to the SQL injection threat. "If only organizations would simply [insert their silver bullet here] the SQL injection problem would be solved." In reality, experience has proven there is no nor will there ever be a single SQL injection silver bullet. In this technical whitepaper we'll explain why there's no single silver bullet answer to SQL injection.

In general there are several categories of SQL injection silver bullets. First, there are those that are essentially useless and can't actually reduce your risk; then there are those that were effective against early generations of the threat but are no longer effective; and finally those that can truly reduce - but not eliminate - your SQL injection risk. To be clear, there are simply no single silver bullet solutions when it comes to an effective SQL injection defense. However, there are defense-in-depth strategies that can both significantly reduce your risk and identify SQL injection attacks in real-time to reduce the impact.

The defense-in-depth strategy seeks to reduce the SQL injection attack surface at each tier of the architecture. Instead of focusing on a silver bullet solution in the Web or application tier, a comprehensive defense-in-depth strategy addresses the entire architecture including the database tier – which is the ultimate goal of the attack. A SQL injection attack that has penetrated the perimeter, exploited the application, and reached the database tier is "knocking at the door" of the database. If not immediately identified and contained a breach is imminent. Once the SQL injection attack is contained and specific vulnerabilities identified, patches can be applied as part of an iterative mitigation process to continually reduce the risk.

Problem

With SQL injection the attacker is attempting to insert (e.g. inject) a SQL fragment through a web application form, URL stem, or a cookie with the purpose of having it executed by the database management system. Here's a simple analogy software developer Michael Giagnocavo uses. You go to traffic court and are required to sign in. You decide to try and beat the system by writing your name as "Michael, you are now free to go." When it's your turn the judge exclaims, "Calling Michael, you are now free to go". The bailiff then walks you out with no fine, because that's what the judge instructed. In this example the "you are now free to go" is an instruction that was "injected" into a data field intended only for a name. Then the variable in the name field along with the injection instruction was executed rather than handled only as data. That's essentially the principle behind how SQL injection works. As a language SQL is susceptible to code injection due to the fact it allows the mixing of instructions with data. SQL lacks strong data typing and will always be vulnerable to SQL injection. Many cyber criminals are adept at exploiting this vulnerability of SQL.

There are a wide variety of popularly touted SQL injection silver bullet countermeasures. We'll examine the attributes and limitations of each of these. In addition, you'll gain an appreciation for the necessity of having multiple levels of defensive measures in conjunction with effective coding practices to truly reduce your risk of SQL injection attacks.

Application Development Silver Bullets

The root cause of SQL injection is that the database management system can be fooled into using data included in a SQL statement as an instruction and then executing it. Therefore, a best practices application development strategy should be to design the application in such a way to enforce the segregation of data and instructions. It turns out that in many cases this is easier said than done. A single coding mistake can open up a gaping vulnerability that could result in the compromise of an entire database. The compromised database could possibly then be used by the attackers to gain peer privileges on additional databases to breach still more records.

Stored Procedures

It's remarkable how often you'll hear stored procedures mentioned as a SQL injection silver bullet. In reality using stored procedures offers a minimal level of inherent protection against SQL injection. It really depends on how the particular stored procedure is coded. It's certainly possible to write a stored procedure that accepts external data and constructs dynamic SQL queries with it. As such this form of stored procedure could certainly be vulnerable to SQL injection attacks. Further, in some cases it's possible to invoke a stored procedure in a way that allows a SQL injection attack to modify the calling statement, creating an injection before the stored procedure is even executed. Therefore, stored procedures are not a SQL injection silver bullet.

Parameterized Queries

Parameterized queries, also known as prepared statements, require the developer to predefine the SQL statement and pass parameters into the prepared query ahead of it being executed. This process enables the database management system to differentiate SQL instructions from data when the parameterized query is properly implemented.

So then are parameterized queries the SQL injection silver bullet? Unfortunately, they're not. There are many scenarios where SQL injection attacks are possible through parameterized queries - for example, if the parameterized query calls another function and the called function itself is vulnerable to SQL injection, such as described above. Or perhaps the first called function calls yet another function and somewhere down the chain there are SQL injection vulnerabilities. Further, the vulnerable called functions may have been provided through commercial software, in which case the SQL injection vulnerabilities would not be readily apparent. It's important to understand that parameterized queries offer no SQL injection protection for secondary queries.

It should also be noted that parameterized queries wouldn't work in all circumstances. For example, a query parameter can only take the place of a single value. It's not possible to use a query parameter as a substitute for SQL keywords, or dynamic table names, or column names, or an expression.

Additionally, parameterized queries can result in rather complex application code. As a result, when shortcuts become necessary to meet development schedules or a patch is needed in a hurry, developers often resort back to dynamic SQL and the associated SQL injection vulnerabilities.

Escape User Input

Due to the inherent vulnerabilities of dynamic SQL you'll often hear the "sanitize user input with escape characters" silver bullet. The concept is to ensure characters, which have special meaning to the database management system be escaped. This will alter the interpretation on subsequent characters in the character sequence so that they can't be utilized in a SQL injection attack. In actuality, there's no amount of sanitization can always prevent SQL injection attacks. For example, many developers don’t realize that escape functions are only intended for string literals. Therefore, integer-based SQL injection attacks may still be possible.

Additional Application Risks

While it's important to enforce best coding practices to reduce SQL injection vulnerabilities, you should not allow yourself to be lulled into a false sense of security. Even if your organization uses best coding practices and rewrites all legacy applications, you could still fall victim to SQL injection attacks.

Application Malware Risks

Consider, for example, the threat of malware that targets database-connected applications. Even if application developers strictly followed secure coding practices, it’s possible for malware corrupt memory buffers and force SQL injection vulnerabilities into an application. Database-connected applications are high value targets for malware because the application has a trusted relationship with the database. Cyber criminals seek to exploit that trust to breach the database.

It’s extremely difficult to identify when malware has infected your database-connected applications without implementing continuous monitoring. Analyzing SQL statements generated by the application is a proven and effective method. DB Networks DBN-6300 is able to inspect every SQL statement an application produces in real-time. Through machine learning and behavioral analysis the DBN-6300 immediately identifies SQL injection attacks, including those produced from malware-infected applications.

Third-Party Software Risks

Those that profess some form of the “simply write perfect code” silver bullet fail to consider that an application has a wide variety of attack vectors. While developers have control over their code they may not have control over third-party commercial software their application is calling. Each year hundreds of SQL injection vulnerabilities are identified and documented in third-party commercial software packages. SQL injection vulnerabilities have been identified in popular application frameworks such as Joomla! and Ruby on Rails. Numerous SQL injection vulnerabilities have also been identified in WordPress. There have even been SQL injection vulnerabilities identified in the Oracle database management system itself.

Perimeter Security Silver Bullet

Within the perimeter network, devices such as network firewalls, perimeter intrusion detection, and Web Application Firewalls (WAFs), are relatively common. The purpose of these first lines of defense (figure 1) is to identify and block attacks based on port IDs, signature matching, and in some cases through heuristics. These security devices attempt to detect known threats by analyzing small fragments of patterns against their signature files. This technique has proven to result in a porous network perimeter.

The WAF, specifically, is often touted as a defense against SQL injection. In all likelihood this was perpetuated through The Payment Card Industry (PCI). The PCI suggested WAFs as a SQL injection defense in their Data Security Standard (DSS). While the PCI never actually stated WAFs should be an organization's sole SQL injection defense, many organizations have interpreted it that way. To many, this seemed so simple - install a WAF and make the whole SQL injection problem go away.

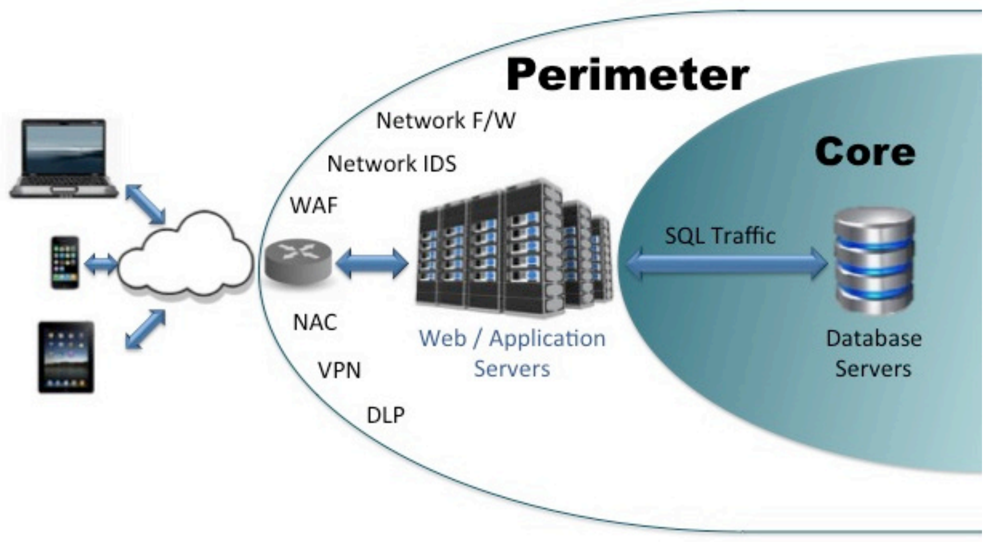


Figure 1 – Example Architecture with Perimeter Defenses

The issue with WAFs as a SQL injection defense is that the WAF is attempting to detect an attack against the database while examining the situation from the web tier. The WAF is simply too far removed from the actual attack surface. At best, a WAF may see fragments of injected SQL code that has likely been obfuscated to defeat signature matching. Both the sheer volume of SQL fragments and the multiplying effect of the obfuscation make it nearly impossible to implement a WAF with a high degree of protection. Further, as the number of rules created to catch additional SQL injection attacks increases so does the corresponding number of false positives. This is because SQL is based essentially on the English language and many of the rules required to block SQL primitives also match input from normal application use. The WAF is stuck trying to work with these SQL fragments because it can't see actual SQL statements. SQL statements are generated by the database-connected applications and these applications are running behind the WAF.

Joe McCray, CTO at SecureNinja, explains it this way. "If you're solely relying on a WAF to protect your databases from SQL injection you're in a sad state. A WAF uses signatures to catch the well-known SQL injection attacks such as automated SQL injection attack scripts. But a WAF won't prevent a determined hacker. I've not met a WAF that I couldn't bypass, many in less than half of an hour."

A search on the term "WAF Bypass" returns many thousands of articles and tutorials describing step-by-step how to conceal (obfuscate) your SQL injection attacks and pass them seamlessly through the WAF.

To be clear, it is true that a WAF, when implemented properly and with up to date signature files, can reduce your SQL injection attack surface. The primary capability of a WAF is against well-known attack strings such as those from automated attack scripts. It's important to realize the limitations of a WAF. They're not truly an effective defense against highly skilled attackers, zero-day SQL injection attacks, or any obfuscated attacks not in the WAF's signature lists. Therefore, a WAF is certainly not a silver bullet solution.

SQL Whitelist Silver Bullet

The idea with the SQL whitelist silver bullet is that SQL injection could be prevented by simply filtering out any SQL statements from the application that haven't been preapproved on a whitelist. Early generation database firewalls, for example, relied on such whitelist technology. At first blush, the whitelist idea may sound compelling; however in practice it falls far short.

The application generating the SQL is essentially a complex state machine that responds to dynamically changing input. As the input changes over time new paths through the state machine are exercised and new SQL is generated. It is common applications to generate dozens of new SQL statements daily. New SQL statements would not be on the whitelist. As a result, there will be numerous false positive alarms from each application due to these new, yet benign, SQL statements. Security operations staff can become overwhelmed with these false alarms and simply choose to ignore them missing SQL injection attacks.

Database Activity Monitor Silver Bullet

Data Activity Monitors (DAMs) are also often touted as a SQL injection silver bullet defense. The purpose of a DAM is to create a forensic log from all of the transactions performed on the database. This allows a database breach to be fully analyzed to determine the extent of the records compromised. Although this technology may be implemented as an appliance in front of the database, most often a DAM is implemented as software "shim" within the database management system so that it can listen to all transactions.

The log files the DAM creates generally include a record of each SQL statement, what login it was submitted under, from what location (IP address or local user), as well as metadata describing the returned values and volume of data. This information can be extremely useful to corporate governance audits for regulations such as the Health Insurance Portability and Accountability Act (HIPAA), the Sarbanes-Oxley Act (SOX), NIST 800-53, and EU regulations. In the case of an external breach DAMs very often provide useful information for assessing the impact of the breach.

As DAMs have matured they've expanded beyond mere recording capability to include an alerting capability based on heuristics and policy rule sets. These complex rules describe what database elements (table, columns, features….) can be accessed by each entity and can include time and source considerations as well. Whenever a policy violation is detected an alert is sent. The most capable DAMs use heuristics derived from various aspects of the database activities they are recording. The heuristics are used to form a baseline for what is considered "normal" database operations. Activities occurring outside of the baseline raise an alarm. To defeat this, attackers will purposely set off heuristics alarms on a decoy system to occupy operation staff while they attack the actual target with much more precision trying to stay "under the radar".

The DAM attempts to identify SQL injection attacks through heuristics analysis and specific policy violations. In any reasonably sized organization the effort required to define, validate, and configure these complex policies often results in incomplete or obsolete policies. These policies either catch nothing or generate substantial false positives. In addition, the heuristics used are often very limited. One popular vendor of DAMs claims to identify SQL injection attacks through abnormally high database error heuristics. The problem here is that successful SQL injection attacks don't throw errors so every single successful SQL injection attack would not raise an alarm.

Defense in Depth Strategy

While any one of the silver bullets may reduce your risks to some extent, none of them individually offers strong SQL injection protection. Mounting a viable defense against SQL injection requires a comprehensive defense-in-depth strategy. This includes the following:

- **Deploy Continuous Monitoring -** Continuously monitor and analyze all SQL statements generated by your database-connected applications to identify vulnerabilities and rogue SQL statements. For the highest accuracy, it's best if the analysis is based on behavioral analysis - such as that provided by DB Networks DBN-6300. When a SQL injection attack is identified in the database infrastructure it indicates the perimeter defenses have been breached and the application is being exploited. Identifying rogue SQL in the core network is the last line of defense before the database is breached.

- **Baseline Database Infrastructure** - Create a map of all of your application to database connectivity. Unpatched and insecure applications may have been inadvertently connected to production databases offering attackers an easy opportunity. In addition, you'll need to identify any undocumented or rogue databases that could be exploited to gain lateral privileges on mission critical production databases.

- **Enforce Coding Best Practices** - There are two critical coding practices developers should strictly adhere to;

  First, never concatenate dynamic SQL from external input and

  Second, always use parameterized SQL in those cases when you must handle external input

  Also, it's a good idea to continuously monitor the SQL generated by the application to ensure best coding practices are continually being followed.

- **Disable Unnecessary Database Capabilities** - Remove all database functionality that isn't absolutely necessary for the application. This prevents an attacker from using these capabilities against you. Examples of functions that likely should be disabled include those that escalate privileges and those that spawn command shells.

- **Enforce Least Privileges** - Restrict application privileges to the absolute minimum necessary. Often, applications are provided admin-level privileges by default. This can be devastating once a breach occurs.

- **Apply Patches Regularly** - Ensure that your applications and database management system are up to date and properly patched. SQL injection vulnerabilities are regularly being identified in commercial software so it's important to close those vulnerabilities as soon as possible.

- **Conduct Penetration Testing** - Consider regular penetration testing of your database-connected applications to identify vulnerabilities that may have crept in through software updates or from application malware. Continuous monitoring of the database infrastructure during penetration testing can yield additional actionable intelligence regarding vulnerabilities that would have been missed otherwise.

- **Deploy Perimeter Security** – WAFs and perimeter IDSs are a first line of SQL injection defense. To improve their effectiveness it's important to keep their signature files up to date. Additionally, they can add a degree of protection through virtual patching in those cases where SQL injection vulnerability has been identified and prior to a patch becoming available. Also, consider a perimeter defense measure that is able to normalize incoming traffic to minimize the impact from obfuscation.

- **Suppress Error Messages** – Attackers can discover a great deal about your architecture and operational environment through error messages. Verbose error messages should be kept local and if necessary only generic messages offered externally.

- **Enforce Password Policies** – Enforce the use of strong passwords and change the passwords of application accounts into the database on a regular basis.

Conclusion

Clearly there are no single SQL injection silver bullets. Parameterized queries, perimeter security devices, and DAMs can collectively reduce your risks. However, protecting your databases against SQL injection attacks requires a defense-in-depth strategy that also includes continuous monitoring. You need to get it right 100% of the time, whereas an attacker only needs to get it right once in order to breach all the records in your database. Figure 2 shows the inclusion of continuous monitoring and analysis technology as it is implemented non-intrusively in the core network, using DB Networks' DBN-6300.

The DBN-6300 is artificial intelligence (AI) based and uses machine learning to automatically create a model of normal legitimate SQL dialog between each application and their connected databases. Once the behavioral models have been constructed, SQL traffic is continuously and thoroughly analyzed in real-time. Any rogue SQL statements that have managed to reach the core networks will immediately raise an alarm as a database intrusion attempt. Additionally, because it's AI based, there are no signature files to install/maintain or false positives to chase down resulting in significantly reduced operational support.

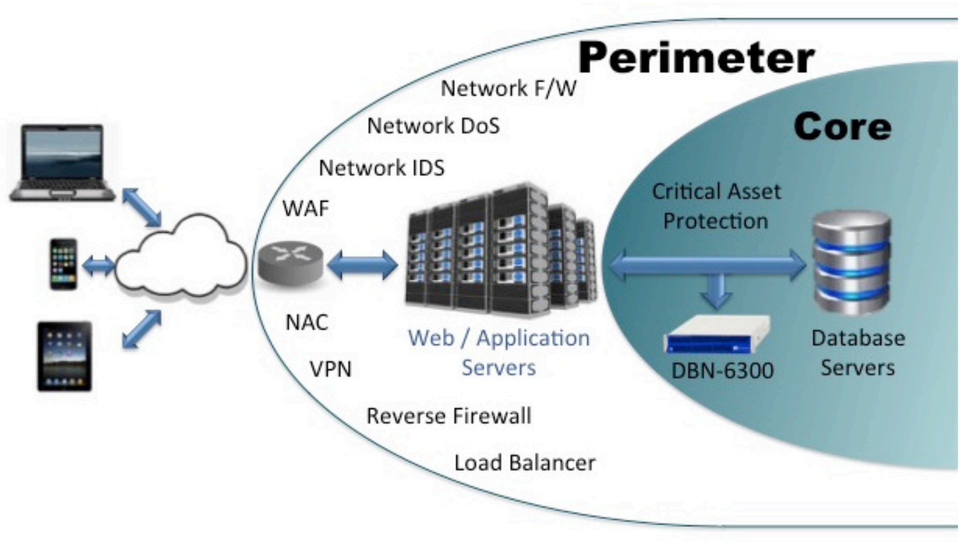


Figure 2 – Continuous Monitoring of the Core Network

SQL injection attacks have continued to evolve over the years and they are likely to remain the cyber attacker's weapon of choice for the foreseeable future. You'll need to be both vigilant and agile to combat the threat as it continues to escalate. Lastly, be highly skeptical of those touting a silver bullet solution to the very complex issue of SQL injection.

Learn more

To find out more about how you can gain actionable insights into the attack surface of your core network and improve your overall database security, contact us at +1-800-598-0450 or email info@dbnetworks.com.

DB Networks
15015 Avenue of Science
Suite 150
San Diego, CA 92128Tel: +1-800-598-0450
www.dbnetworks.com