

SQL Security - Clean Code, Secure Code

Introduction: The Structured Query Language, or SQL (pronounced “sequel” or “S-Q-L”) has become one of the driving forces behind the “data revolution,” and Web 2.0. “Web-based email, online shopping, forums and bulletin boards, corporate websites, and sports and news portals are all database-driven.” (Williams and Lane 2004) During its nascent development stages, “considerable thought was given to the human factors aspects of the SQL language,” which probably contributes to SQL’s ongoing prevalence in the world of databases. (Chamberlin, Gilbert et al. 1981) The human factors, however, continue to create and discover security risks in both modern and legacy SQL applications. This essay will focus on several SQL best practices that database developers can use to minimize security risks, and maximize database resiliency in the face of attack.

Best Practice: There are several SQL security best practices that have emerged from the enormous amount of security literature as consensus recommendations. To begin, by far the largest threat faced by a public-facing SQL database is a SQL injection attack. An injection attack occurs when an attacker purposely “enters a malformed SQL statement into the textbox that changes the nature of the query so that it can be used to break into, alter, or damage the back-end database.” (Litwin 2004) Some security experts go so far as to say “all input is evil,” but the majority of attacks can be defended by following these four simple principles:

1. Suppress Error Messages. (Litwin 2004)
2. Validate Input. (Grossman 2009)
3. Use Parameterized SQL Statements. (Grossman 2009)
4. Do Not Create Dynamic SQL with User-Supplied Data. (Grossman 2009)

The first principle, suppression of error messages, helps to prevent attackers from gaining useful information about your SQL database and/or servers. “Error messages that include in-

formation detailing why a database server failed are extremely useful in the identification and subsequent exploitation of SQL injection[s].” (Clarke 2009) Rather than allowing your database to display the default error message (which may include software version, server information, or even database-specific information), create an alternate error message that is displayed instead. This message can be simpler than the default message, or even more specific (in terms of what generated the error). The goal is to prevent “tipping your hand” to a potential attacker.

The second principle, validation of input, is “one of the most powerful controls [database developers] can use” to prevent attacks. “Input validation is the process of testing input received by the application for compliance against a standard defined within an application.” (Clarke 2009) For example, a 2 letter state abbreviation field has no need for non-numerical characters (including # and ‘). This sort of validation helps prevent both information probing and injection attacks. It also helps to prevent users from entering malformed data that is not an attack, producing cleaner and more useful output and generally enhances database performance.

The third principle, use of parameterized SQL statements, leverages SQL software’s ability to strip unnecessary characters out of input strings, preventing attackers from many of the input-based attacks we have already discussed. Sanitizing user input “prevents the execution of arbitrary SQL commands” and “prevents the database from being exposed.” (Harris 2009) Typically, this will include stripping the apostrophe (‘) and hash (#) keys, as they can be used to trigger an injection attack and are typically not needing to be stored in the database.

Finally, database developers should strive to not create dynamic SQL with user-supplied input. This is typically a more advanced attack prevention strategy, because dynamic SQL statements “allow a programmer or end user to create a SQL statement’s specifics at runtime, and pass that statement to the database.” (Stephens, Plew et al. 2008) By allowing users to essentially create new SQL statements, a malicious user could perform all sorts of attacks or exploitations on the database. Dynamic SQL statements are the digital equivalent of leaving your front door open, and unlocked. To prevent exploitation, database developers should ensure that the

only dynamic SQL statements they use are generated with data that is not supplied in any way by the user.

Assessment: There is an enormous amount of literature, both in-print and on-line, about the importance of, and strategies for, writing secure SQL code. Judging by sheer volume, it appears a safe judgement that these principles are clearly “best practices” that should be followed by database developers. Looking a bit beyond the sheer volume, however, and one begins to realize the enormous amounts of database code that are being used and re-used across organizations every day. What sort of security remediation is available for code that is already in existence (much less already in use)?

The volume of this code makes line-by-line review looking for vulnerabilities simply infeasible. Though there is clearly no lack of automated tools, they lag behind in discovering sophisticated or smaller security holes. They are, however, capable of catching the most glaring SQL code offenses.

Thus, the best practice must become somewhat of a hybrid, due to the amount of legacy code present in database applications world-wide:

- All newly written SQL code should conform to the above principles. Legacy code should be reviewed based on risk assessments of the database in question.

Each organization will have their own view of the data stored in their databases, and must therefore make their own decisions about cost/benefit for code analysis. If you have relatively few databases, most of which are public facing and contain sensitive data, then perhaps a line-by-line code review is warranted. If, on the other hand, your organization has many databases, none of them public facing, perhaps the equation simply does not balance out in-favor of reviewing legacy code. Maybe your organization would opt to employ an automated tool to catch some of the most obvious offenders, and then re-evaluate your risk assessment.

Regardless of how legacy code is evaluated, organizations and individuals should demand that all newly developed code adhere strictly to the above principles. One of the realities of SQL databases is the fact that one small vulnerability in an obscure portion of the database can ex-

pose the entire database, and its contents, in the hands of a capable attacker. Attackers often look for low-hanging fruit or “quick wins.” By employing these four principles, the majority of attack vectors are either minimized or eliminated, and attackers will quickly move on to an easier target.

Conclusion and Recommendations: As we have seen, writing secure SQL code is absolutely essential to any worthwhile database project. By implementing the best practices principles mentioned above, as well as staying current with security threats and techniques, database developers can not only ensure that their data stays secure, but that their users, and customers, and applications are all protected from the myriad of threats posed in today’s interconnected world.

word count: 1,146

Glossary

SQL: Structured Query Language.

SQL Injection Attack: An injection attack occurs when an attacker “enters a malformed SQL statement into the textbox that changes the nature of the query so that it can be used to break into, alter, or damage the back-end database.” (Litwin 2004)

Input Validation: Input validation is the process of testing input received by the application for compliance against a standard defined within an application. (Clarke 2009)

Parameterized SQL Statements: Parameterized SQL statements strip unnecessary characters out of input strings, preventing attackers from many of the input attacks we have already discussed. Sanitizing user input “prevents the execution of arbitrary SQL commands” and “prevents the database from being exposed.” (Harris 2009)

Dynamic SQL Statements: Dynamic SQL allows a programmer or end user to create a SQL statement’s specifics at runtime, and pass that statement to the database. The database then returns data into the program variables, which are bound at SQL runtime. (Stephens, Plew et al. 2008)

Legacy code: Legacy code is code that is already in use or existence. Typically, this will be found on older systems, or on re-purposed or modified databases. It is not unheard of for organizations to develop a single database application that is then copied and only slightly modified for other uses or purposes. If this “legacy code” contains errors, then it is likely that all applications including this code are equally vulnerable.

Sources Cited

- Chamberlin, D. D., A. M. Gilbert, et al. (1981). A history of system R and SQL/data system. Proceedings of the seventh international conference on Very Large Data Bases - Volume 7, Cannes, France, VLDB Endowment.
- Clarke, J. (2009). SQL Injection Attacks and Defense, Syngress.
- Grossman, J. (2009). "SQL Injection, Eye of the Storm." The Security Journal **26**(Winter 2009): 27.
- Harris, A. (2009). Pro IronPython, Springer.
- Litwin, P. (2004). "Stop SQL Injection Attacks Before They Stop You." MSDN Magazine **04**(09).
- Stephens, R., R. Plew, et al. (2008). Sams Teach Yourself SQL in 24 Hours, Sams Publishing.
- Williams, H. E. and D. J. Lane (2004). Web database applications with PHP and MySQL, O'Reilly Media, Inc.