Preventing SQL Injections in Online Applications: Study, Recommendations and Java Solution Prototype Based on the SQL DOM

Etienne Janot, Pavol Zavarsky

Concordia University College of Alberta, Department of Information Systems Security,
7128 Ada Boulevard, Edmonton, AB, T5B 4E4, Canada
etienne@waziboo.com, pavol.zavarsky@concordia.ab.ca

Abstract. SQL Injection Attacks are a relatively recent threat to the confidentiality, integrity and availability of online applications and their technical infrastructure, accounting for nearly a fourth of web vulnerabilities [1]. In this paper based on a master thesis [2], and numerous references therein, we present our study on the prevention of SQL Injections: overview of proposed approaches and existing solutions, and recommendations on preventive coding techniques for Java-powered web applications and other environments. Then, we review McClure's SQL DOM [3] approach for the prevention of SQL Injections in object-oriented applications. We also present our solution for Java-based online applications, SQLDOM4J, which is freely based on the SQL DOM but attempts to address some of our criticisms toward it, and evaluate its performance.

Keywords: Java, Prevention, SQL, SQLDOM4J, SQL Injection, Web Security.

1 Introduction

Online data theft has recently become a very serious issue, and recent cases have been widely publicized over concerns for the confidentiality of personally identifiable information (PII). As a consequence, database extrusion prevention (DBEP) products have been rising lately. As early as 2002, the CSI & FBI survey reported that more than 50% of online databases experience a security breach each year [4]. As a matter of fact, Injection Flaws – and particularly SQL Injections – appear among the OWASP's Top Ten most critical web applications vulnerabilities list [5].

Application-level vulnerabilities, which are believed to account for 70% to 90% of overall flaws, are now the main focus of attackers and researchers. Online applications (websites and services) are especially at risk due to their universal exposure and their extensive use of the firewall-friendly HTTP protocol. Moreover, database security is too often overlooked in favor of web and application server security, resulting in backend databases being a major target for attackers which are able to use them as easy entry points to organizations' networks.

Protecting online applications (e.g. websites) and web services against SQL Injection Attacks has thus become a major concern for organizations, which face threats that can go far beyond the expected reach of the public web or application server. While several effective prevention methods have been developed, ensuring full protection against SQL Injections remains an issue on a practical level. This paper will therefore discuss the difficulties that challenge the implementation of a comprehensive SQL Injection protection solution before giving a critical overview of some of the major research proposals and main types of commercial solutions. The paper will then outline some simple mitigation recommendations that apply to the Java environment.

This paper also presents our solution, the SQLDOM4J, which targets Java environments. Freely based on McClure's SQL DOM, it enables developers to construct and execute safe SQL statements easily. The main concepts behind our SQL Injection prevention strategy are strong typing and the separation of control and data channels within SQL statements. These are implemented by leveraging both the strongly-typed nature of OO applications and JDBC's pre-compiled type-binded statement interface, `PreparedStatement`. This paper will show that the use of the SQLDOM4J API to build and execute database queries effectively protects applications against SQL Injection Attacks.

1.1 SQL Injection Attacks

SQL Injection Attacks (SQLIA's) [6, 7] are carried out by submitting maliciously crafted inputs to database-driven applications, such as interactive web sites. These inputs are then used by applications to build dynamic SQL queries, and have the potential to alter the semantic structure of the query, due to the lack of separation of control and data in SQL. The numerous SQLIA techniques used by attackers are based on the many statement structure combinations offered by SQL, and sometimes also take advantage of additional features in specific DBMS implementations, particularly Microsoft's SQL Server. They pursue different goals at various levels, from allowing other techniques to be used (SQLIA escalation) to actually extracting database data. The resulting threats are various and range from system fingerprinting to Denial-of-Service (DoS) and theft of confidential information. SQL Injections Attacks thus threaten the confidentiality, integrity and availability of databases' data and structure, of their hosting systems and of their dependant applications, and as such greatly require the attention of application developers and the deployment of effective prevention solutions.

Below is a basic example of SQL Injection. Instead of submitting the credentials [`a_login`] and [`a_password`] in a website authentication form, the attacker enters [`' OR 1=1 --`] and [ ]. As a consequence, the following expected SQL query:

```
SELECT * FROM users WHERE login='a_login'
AND pwd='a_password'
```

Becomes:

```
SELECT * FROM user WHERE login='' OR 1=1 -'AND pwd=' '
```

First, the attacker "gets out" of the text field by starting his input with a single quote. By doing this, he closes the SQL Where clause on the `login` field, hence enabling the injection of SQL control code right into the query. Of course, no login is blank, so the first expression will always evaluate to false. In order to circumvent this problem, the attacker inserts the expression `OR 1=1`, which will always evaluate to true – this is called a tautology. Next, the `--` (double dash) operator marks the start of comments, prompting the SQL parser to ignore the Where clause on the `pwd` field. The resulting meaning of the altered SQL query is therefore equivalent to "select all users". Thus, the application logic controlling user authentication will authorize the attacker and, in the worst scenario, the application might even return an error message containing the data returned by the DBMS, i.e. the list of user credentials.

Please refer to our full paper [2] for an in-depth study of SQL Injection attacks, mediums (vectors) and techniques.

1.2 Prevention Implementation Difficulties

Developers are faced with multiple challenges when attempting to effectively secure online applications (especially web sites). Adequately addressing these issues depends on the state of the application at the time, the developer's priorities and ultimately on the approach that will be chosen. Here are some of the major issues that arise when implementing SQL Injection Attack protection:

Entry points of multiple data input channels (GET and POST data, cookies, user-agent headers, etc.) have to be identified for most protection schemes to be effective. In some large web sites applications this can prove too difficult and architects might opt for other protection approaches which work e.g. at the database interface level.

Segmented queries (queries built across several modules) are typical of web applications and of distributed services: queries are not entirely constructed at one specific location in the application but rather follow the application logic flow, i.e. they are progressively defined (selected columns, criteria) as the application reacts to users' actions and inputs. Hence, they are difficult to trace and sanitize at once. Moreover, should query sanitization or validation be done progressively or at finalization? Solutions based on tainting have the potential to address this issue.

White-list filtering is highly context-dependant and can rarely be implemented in strict and widespread manner without reducing usability, as valid characters vary considerably with multiple contexts.

Evasion techniques, as for all types of attacks subject to detection-based protection, are constantly evolving and have proved very effective. This greatly hinders the efficacy of black-list mechanisms, which are still largely used as adequate white-listing is hard to implement. As for pure HTTP attacks, inputs should be normalized before applying detection, however normalizing is itself a difficult task. Moreover, black-listing cannot offer full protection, even combined with normalization.

Maintaining input validation rules up to date with evolving database schemas (data types, column lengths) can also be challenging, opening the path for e.g. truncation attacks. Solutions which bring the database structure closer to the application (and therefore directly accessible to the developer) should resolve this issue.

Lastly, many protection schemes introduce significant overheads such as developer training, code rewriting, infrastructure modification or performance decrease.

2 Existing Methods and Solutions

Many detection and prevention schemes [8] have been developed proposed to address SQL Injections. These solutions typically follow different approaches as they are targeted at various system and language environments and implement protection at various levels of the architecture (network, server, application). Some approaches are applicable to other types of injection attacks such as Cross-Site-Scripting (XSS) or XPath Injection. On another level, some are specific to a particular environment (e.g. web applications) or language, while others are implementation-independent (e.g. reverse-proxies). Due to paper length restrictions, this section will be limited to a brief description of major proposals and main types of available solutions, pointing out their most important advantages or drawbacks. Please refer to [2] for a more comprehensive and in-depth analysis.

2.1 Proposed Approaches

In accordance with the scope of this paper, only approaches focusing on prevention methods will be mentioned – detection still being useful, but more adequate to an auditing, forensics or live response context.

Active input data encoding transforms string inputs into safe character sequences, using a 2-way encoding algorithm (e.g., Base64). While this is fail-proof for strings, it renders data stored in databases useless for SQL operations (as it is encoded), introduces storage overhead and must be combined with strong-typing.

Tainting [9] labels all input data as 'suspicious'. Individual filters valid the data (i.e., untaint it). SQL queries containing tainted data are then blocked before execution. Main drawbacks: accuracy relies on user-specified filters, and all data entry points must be identified.

Instruction-set randomization [10] encodes SQL keywords. A proxy decodes them, and blocks queries containing clear-text keywords. While this introduces a cryptographic processing overhead it is potentially effective, but for this the proxy needs to be able to recognize all keywords, including vendor-specific ones.

Query pre-modeling [11, 12, 13] validates queries' control structure against a pre-determined set of legal variations. Complex but promising, this approach would probably not support segmented queries.

IPS/IDS and application firewalls [21, 31, 32] are now specializing but suffer from their inherent limitations (e.g., evading signature-based detection is too easy) and lack contextual information. Anomaly-based detection works better but cannot provide guaranteed protection.

New query building paradigms [2, 3, 14] circumvent the danger of concatening strings to build SQL queries by introducing other construction means, e.g. an API. Particularly suited for OOP environments, they can be effective but introduce overheads and have a high chance of reducing querying expressivity.

2.2 Available Solutions

Protection products which can be used to prevent SQL Injections in online applications have been emerging these last 3 years, in the form of Web Application Firewalls (WAFs), HIPS solutions and more lately Database Extrusion Protection (DBEP) systems [2].

Snort-based solutions are used by many organizations who wish to leverage their IDS experience. They will catch some SQLIA's as Snort packages some database vulnerabilities, but it only features signature-based detection and thus can't offer serious protection [21].

Host-based IDS, such as Application Security's DbProtect [22], while not targeted at prevention, are mentioned as they play an important role: they provide advanced auditing, enabling organizations to comply with regulatory requirements. They can notably detect insider attacks – 70% of all database attacks [15].

WAFs [16] use SPI and Deep-Packet Inspection methods to inspect the data portion of HTTP traffic, blocking protocol violations and malicious content (cookies, POST, GET) and ship with enterprise-class features such as load balancing and HTTP acceleration. The most renowned and rewarded commercial WAF products are Imperva's SecureSphere Web Application Firewall [32] and F5 Networks' Big-IP Application Security Module [33]. These products are gaining a well-established position in the web and application security market, which is also influenced by expert organizations such as the OWASP Foundation and the WASC. They are quite effective as they use a combination of signatures, behaviour analysis and user policies.

ModSecurity [34] is a famous open source WAF for Apache with over 10,000 commercial deployments worldwide. According to a Forrester Research study published in 2006 [35], it provides the "best attack detection for web application threats". It runs as an Apache module either in reverse proxy or in embedded mode provides many features. Its strength notably relies in its high customization capability which enables users to for instance specify script-specific parameter filters. eEye Digital Security's SecureIIS [36] could perhaps be considered the commercial MS-oriented counterpart to ModSecurity. It runs as an ISAPI filter in order to integrate closely with the IIS, which allows it to inspect SSL traffic once it's decrypted. Its detection strategy relies both on behaviour analysis and on a base of known vulnerabilities. Its advantage apparently resides in its behaviour analysis-like CHAM (Common Hacking Attack Methods) technology which identifies generic attack methods and enables it to prevent unknown (zero-day) attacks [37]. It is said to have blocked all the new attacks conducted against IIS since its first deployment [36]. Nonetheless, it lacks important features such as auditing which is critical for organizations concerned with compliance issues.

DBEP or *Data Loss Prevention (DLP)* products [23] are new specialized protection systems which are placed right in front of database servers and integrate application firewall and IPS features. Specifically aimed at preventing data theft (extrusion), they go beyond compliance, risk reduction and attack detection by their use of fine-grained rules and event correlation methods. They can be deployed as in-line (IPS-like) or out-of-band (IDS-like) devices. Thus they specifically target database vulnerabilities and data protection. The two main front-runners in the yet small but quickly growing

DBEP market are Guardium's SQL Guard [24] and Imperva's SecureSphere Database Security Gateway [25]. Both products have received numerous awards as they offer comprehensive, effective and scalable solutions with features ranging from automatic reporting to dynamic user activity profiling (baselining). On the one hand, Imperva, which is also praised for its aforementioned WAF product, is appreciated for its central management, dynamic user profiling and vulnerability assessment features. On the other hand, Guardium, which poses as the most widely used DBEP product and has been named "leader across the board" by Forrester Research [38], is particularly appreciated for its full-automation capacities.

3 Preventive Coding Techniques

Here we briefly present eight recommended mitigation techniques [2, 17, 20] that can prove very effective against SQL Injections in online applications (e.g. web sites) and other environments.

1. Always apply the "Least Privilege" rule: set up low-privileged database accounts for applications that access the DBMS.
2. Always validate user-supplied data – as well as any data obtained from a potentially unsafe source – on the server side. Client-side input validation can be useful (mostly for user experience) but cannot in any case be relied upon.
3. Do not return SQL error messages to users as they contain information useful for attackers, such as the query, details about the targeted tables or even their content. This can be easily prevented in Java using exception handling: simply catch all `SQLExceptions`.
4. Enforce data types for all inputs. Type-specifying regular expressions can be used to validate the data. Types can also be enforced via pre-compiled statements with binded variables (e.g., JDBC's `PreparedStatement` interface. Also check boundaries to prevent buffer overflows and truncation errors which could lead to a crash of the DBMS.
5. Encode text input fields likely to contain problematic characters into an alphanumeric version using a two-way function such as Base64.
6. Filter all input data via a 2-step process. First, apply white-list filtering at user input collection (e.g., web forms): allow only field-relevant characters, string formats and data types; restrict string length. Then, black-list filtering or escaping should be applied in the data access layer before generating SQL queries: escape SQL meta-characters and keywords/operators.
7. Validate dynamically-generated database object names (e.g. table names) with strict white-list filtering. For instance with Oracle, allow only alphanumeric characters, '_', '$' and '#'.
8. Avoid quoted/delimited identifiers as they significantly complicate all white-listing, black-listing and escaping efforts.

4 The SQL DOM

As our solution builds on this proposition, we will give a brief overview of the concept behind the SQL DOM [3], explain its mechanisms and discuss its strengths and weaknesses.

4.1 Overview

The concept behind the SQL DOM is fairly simple: instead of relying on developers to implement cumbersome defensive coding techniques while using strings to build dynamic SQL queries, force them to use a safe API which will take care of security.

An API generation tool (*sqldomgen*) analyses the database schema at compile time and writes code for a custom set of SQL query construction classes (which then integrate into the IDE and are directly called by developers to build SQL queries). The resulting DOM is a tree-like structure based on a generic template, mapping the possible variations of SQL queries according to tables and columns definition.

There are 3 main types of classes: SQL statements, table columns and where conditions. These classes have strong-typed methods mapping the data types in the database schema. This enables them to validate data types automatically. The constructors of column classes escape strings (i.e., replace each quote by a double quote) at runtime to sanitize them.

4.2 Strengths

McClure's approach holds important advantages for application layer-based prevention of SQL Injections.

Firstly, it is especially suited for OOP environments: native strong-typing and constructor-based automatic string sanitization excuse it from developing a complex SQL Injection detection scheme.

Also, the consistency of SQL strings is improved: object (tables and columns) names are generated by the database-bound API and types are automatically enforced. As McClure argues, this has the potential to substantially reduce testing and maintenance in database-driven applications.

Another main advantage is that the attack surface is reduced, as building queries is carried out without directly manipulating SQL keywords and operators.

4.3 Weaknesses and Limitations

The SQL DOM has inherent weaknesses and also some limitations.

1. As any new query development paradigm, it is bound to introduce overheads for developer training and code rewriting, as query-generating code needs to be rewritten.
2. Its full-object policy (at least one object is instantiated for each criteria) comes at a cost: performance, which could be reduced by using static classes.
3. Stored procedures remain unprotected even though they are very common.

4. Another limitation is that the SQL DOM does not execute queries (it only generates them), while this could improve database integration and perhaps further reduce the attack surface by hiding database connectivity.

5. McClure's paper neither precisely describes its string sanitization strategy (aside from a hint at quotes escaping) nor elaborates on exception handling, although it is central in OOP application security. How will the SQL DOM behave if a null value is passed on as a criterion?

5 The SQLDOM4J Solution

Our solution (full version at [2]) was freely adapted from the SQL DOM and aimed, through an overhauled architecture and reviewed design, at meeting security and functionality objectives. Items rated 'critical' are summed up here.

Firstly, type enforcement and string sanitizing should be automatic, transparent and preventive (before any interaction with database).

Secondly, SQL errors should never be directly returned to visitors, i.e. only managed SQLDOM4J exceptions should be raised, and easily identifiable.

Moreover, the API should be tamper-resistant.

Lastly, the database structure mapping should be precise and equivalent Java data types defined in an optimal way to match SQL types.

Additionally, the SQL DOM creates one class per table and for each class/table, one method per possible operation per column, making the API both inefficient and cumbersome. To simplify this, all database structure mapping information will be stored in a single repository class, whose members will be accesses statically to avoid unnecessary object duplication.

5.1 Architecture & Design

The architecture of the SQLDOM4J has been kept deliberately simple in order to fulfill its security purpose. There are 3 class packages in our solution: `database`, `sql` and `exceptions`. The `database` package's `DB` class stores all database mappings (table names, column names and data types) as static enumerations. The `sql` package contains the SQL manipulation classes (e.g. `SelectQuery`) – those that developers use to build and execute queries, using `DB`'s enumeration members as constructor parameters. The `exceptions` package contains detailed but safe classes which all derive from the generic `SQLDOM4JException`, enabling developers to catch the API's errors easily. Please refer to the appendix for a detailed class diagram and further explanations.

As a result, the security model is simple. First, the API checks data types against its mappings, upon input value submission. Second, the query is pre-compiled by the DBMS-specific driver using JDBC's `PreparedStatement` interface with binded variables. Any error in either step will prevent the query's execution.

This design also aims at addressing the recommendations presented in section 3. Indeed, server-side validation (2), SQL error interception (3) and strong typing (4) are directly enforced, while text input encoding (5) and 2-step input validation (6) are not needed in our case, as dynamic input is injected through a separate protected data channel (binded variables) via the `PreparedStatement` interface. With the SQLDOM4J, object names (7) are not inputted by the user and are routinely validated. A low-privileged database account (1) should however still be provided to the API. Quoted delimiters (8) are not supported and thus avoided.

5.2 Example

Here is a data type violation example:

```
PreparedStatement ps =
    new SelectQuery(conn, DB.Table.MEMBERS)
         .select(DB.MEMBERS.ID,
                 DB.MEMBERS.LOGIN)
         .orderBy(DB.MEMBERS.ID, OrderBy.ASC)
         .whereEquals(DB.MEMBERS.AGE, 123456)
         .getPreparedStatement();
```

The problem here is that AGE is a smallint (i.e. short) and 123456 is out of its bounds. Here is the resulting exception:

```
SQLDOM4JColumnWrongTypeException SQLDOM4J error: a
specified comparison cannot be conducted as data types
do not match for the specified column.
```

6 Evaluation

Throughout this project we have privileged security over performance while still keeping in mind development- and efficiency-related objectives. We will first outline how the SQLDOM4J addresses common implementation difficulties. Then, we will provide a qualitative evaluation of the SQLDOM4J's accuracy followed by a quantitative evaluation of the API's performance overhead. Both these evaluations will be presented as comparisons with McClure's SQL DOM solution, which was implemented in C#. We will also identify possible future improvements.

6.1 Addressing Prevention Implementation Difficulties

The design of the SQLDOM4J addresses common implementation difficulties (section 1.2). Data input entry points do not need to be identified as protection is applied right before database interaction. Segmented queries are fully supported; their security is ensured as each query modification is validated by the API. White-filtering and blacklisting are unnecessary as dynamic inputs are specified using binded

variables (i.e. data channel). Evolving database schemas are not a difficulty here as the API reflects the changes to the database structure – a simple rebuild is required.

6.2 Accuracy

There are 2 possible attack vectors for SQL Injections: non-text inputs with no type checking, and string inputs with no proper sanitization. Hence, the 1st step in preventing SQLIA's is type enforcement, which eliminates risks on all non-text inputs. Our solution checks types dynamically at runtime, by 2 separate agents: the API (upon value submission and in accordance with the schema mappings) and the DBMS (at query finalization, before actual execution). The 2nd step is string sanitization, which is quasi-impossible to implement universally. Using the `PreparedStatement` interface to do this has the advantage of separating control and data channels and performing an accurate, vendor-specific sanitization – in a portable manner.

6.3 Performance

As our goal was to provide evaluation data that would be comparable with the test results provided by McClure's SQL DOM paper, we designed a similar series of tests which used the SQLDOM4J to build prepared statements and varied the number of `SELECT` columns from 1 to 13. As for the SQL DOM, the results of these tests are averaged over 10,000 executions. The machine used was a 1.86 GHz Pentium M Centrino workstation with 1GB of DDRAM2. For this test, both the classic `String`-based generation and the SQLDOM4J generation are shown using bars, while the relative overhead is shown using a blue line.

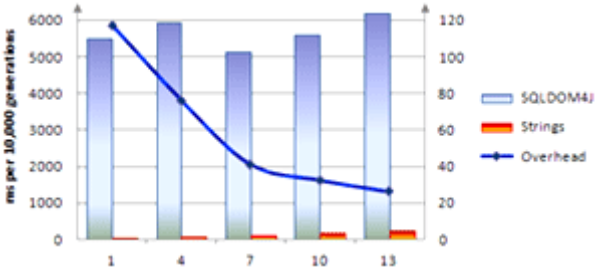

Fig. 1. Test: variation on the number of SELECT columns

The absolute overhead is important: 0.5 to 0.65ms (13 columns) per query. Yet, it is still below the millisecond barrier and doesn't exceed the query execution time (~0.7ms). The relative overhead decreases harshly with the number of specified columns (1 to 13): from 120x to 25-30x.
In comparison, the SQL DOM's absolute overhead was of approximately 0.13ms for 13 selected columns. Why such a difference? Apart from the very likely performance differences between our and the SQL DOM's test machines (unspecified in his paper),

the main design difference between the 2 solutions is that the SQLDOM4J uses the `PreparedStatement` interface instead of strings. Indeed, we measured that the second phase of the construction of an SQL query with our API, which corresponds to the DBMS-based preparation of the pre-compiled statement, accounted for half of the overhead introduced by the SQLDOM4J. We however believe that this overhead falls within the scope of a reasonable performance-accuracy trade-off, when accuracy is identified as security.

6.4 Future Improvements

We believe that the SQLDOM4J solution, which was developed in the scope of a master thesis research project, could significantly benefit from improvements to its accuracy and performance; here are the main ones that we have identified.

Column lengths (e.g. for `varchar` fields) could be stored in the `DB` class (as names and data types are) allowing the solution to perform bounds validation for input data and therefore increase its protection level and overall accuracy.

Whereas it is still confined under a 1 millisecond barrier (with our test systems), our solution's performance overhead is significant, especially when compared to the SQL DOM's. The main source of overhead is the use of the underlying DBMS-driven `PreparedStatement` interface and we believe that its use within our API could be limited while still offering a similar level of protection. For instance, accuracy tests have shown that type violations are successfully detected by the API. Hence, SQL queries which do not use variable text values could be built without the use of `PreparedStatement`s, whose added value mainly pertains to text field protection.

An interesting evolution for the SQLDOM4J solution would be to integrate the API with a Java Object-Relation Mapping (ORM) or persistence framework, such as Hibernate or Oracle TopLink [18]. These frameworks generally build strong-typed queries automatically, but may allow developers to specify custom SQL queries for increased flexibility, thus opening a window for SQL Injections. For instance, the constructor of Oracle TopLink's SQLCall [19] class accepts a simple Java string as argument and it is executed without verification. It could be modified to only accept queries built with the SQLDOM4J API (`SQLQuery` objects) in order to ensure their validity while still allowing developers to specify custom queries. Another possibility would be to integrate the SQLDOM4J with TopLink JPA, TopLink's open source implementation. The SQLCall class could then inherit from the SQLDOM4J's SQLQuery subclasses' safe query building features.

7 Related Work

Safe Query Objects [14] is an advanced Java solution which integrates with the JDO ORM [26]. Its native querying paradigm evolution has the inherent drawback of hindering querying expressivity as it defines a new querying language.

LINQ [27] is a .NET component introduced in Visual Studio 2008. It adds native querying features to C# and VB9 and enforces strong typing based on type inference. Unfortunately, the LINK to SQL provider is only available for SQL Server.

Quaere [28] is a LINQ-like open framework that adds a querying syntax reminiscent of SQL to Java applications. It performs queries on Collections and is able to query databases. As such, it is very practical; however it does not provide any protection against SQL Injection Attacks.

JEQUEL [29] is a Java solution similar to the SQLDOM4J, as it allows developers to specify queries using methods and maps the database structure. It supports prepared statements but does not currently enforce strong typing or validate queries.

8 Conclusion

In this paper we have tried to give a balanced overview of the protection methods applicable to the prevention of SQL Injections in online applications such as web sites and services. The academic field has developed promising strategies such as WebSSARI [9], while the industry is now producing specialized WAF and DBEP products which are rising and likely to become essential parts of comprehensive online data protection strategies. Yet, despite the effectiveness of these products, we believe that their use should not excuse developers from applying preventive coding techniques, as these hold true potential when implemented properly. Instead, they should be complimentary components in a global defense-in-depth strategy against SQL Injection Attacks.

Then, after discussing the strengths and weaknesses of McClure's SQL DOM proposal, we have presented our SQLDOM4J solution. The main difference, aside from the architectural simplification, is the use of JDBC's `PreparedStatement` interface, which comes at an important performance cost while providing considerable security benefits, especially in complex web environments.

All interested readers can obtain a copy of [2] and the Java prototype solution of the SQLDOM4J from the authors upon request, or at http://waziboo.com/thesis.

9 Acknowledgements

The research was supported by the RISTEX program of the Japan Science and Technology Agency (JST).

10 License

References

1. Andrews, M.: Guest Editor's Introduction: The State of Web Security. IEEE Security and Privacy, 4, 4, 14--15 (2006)
2. Janot, E.: SQLDOM4J: Preventing SQL Injections in Object-Oriented Applications. Master thesis, Concordia University College of Alberta (2008), http://waziboo.com/thesis
3. McClure, R., Krüger, I.: SQL DOM: Compile Time Checking of Dynamic SQL Statements. In: 27th IEEE International Conference on Software Engineering, pp. 88--96. IEEE Press, New York (2005)
4. Power, R.: 2002 CSI/FBI Computer Crime and Security Survey. Computer Security Issues & Trends, 8, 1, 1--22 (2002)
5. OWASP Top Ten 2007, http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf
6. OWASP Foundation: SQL Injection, http://www.owasp.org/index.php/SQL_injection
7. Chapela, V.: Advanced SQL Injection, http://www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt
8. Halfond, W., Viegas, J., Orso, A.: A Classification of SQL-Injection Attacks and Countermeasures. In: IEEE International Symposium on Secure Software Engineering (2006)
9. Huang, Y., Yu, F., Hang, C., Tsai, C., Lee, D., Kuo, S.: Securing Web Application Code by Static Analysis and Runtime Protection. In: Di Nitto, E., Murphy, A.L. (eds.) 13th international conference on World Wide Web, pp. 40--52. ACM, New York (2004)
10. Boyd, S., Keromytis, A.: SQLrand: Preventing SQL Injection Attacks. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) ACNS 2004. LNCS, vol. 3089, pp. 292--304. Springer, Heidelberg (2004)
11. Buehrer, G., Weide, B.W., Sivilotti, P.A.: Using Parse Tree Validation to Prevent SQL Injection Attacks. In: Di Nitto, E., Murphy, A.L. (eds.) 5th International Workshop on Software Engineering and Middleware, pp. 106--113. ACM, New York (2005)
12. Halfond, W., Orso, A.: Preventing SQL Injection Attacks Using AMNESIA. In: Di Nitto, E., Murphy, A.L. (eds.) 28th ACM/IEEE International Conference on Software Engineering, pp. 795--798. ACM, New York (2006)
13. Su, Z., Wassermann, G.: The Essence of Command Injection Attacks in Web Applications. ACM SIGPLAN Notice 41, 1, 372--382
14. Cook, W., Rai, S.: Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In: Di Nitto, E., Murphy, A.L. (eds.) 27th ACM/IEEE International Conference on Software Engineering, pp. 97--106. ACM, New York (2005)
15. Cole, L.: AppSecInc to Launch Database Security Suite. Database Journal (2007), http://www.databasejournal.com/news/article.php/3657096
16. Ristic, I.: Web Application Firewalls: When Are They Useful?. OWASP AppSec Europe 2006, http://owasp.org/images/9/9c/OWASPAppSecEU2006_WAFs_WhenAreTheyUseful.ppt
17. OWASP Foundation: Preventing SQL Injection in Java, http://www.owasp.org/index.php/Preventing_SQL_Injection_in_Java
18. Oracle TopLink, http://oracle.com/technology/products/ias/toplink
19. Oracle Fusion Middleware Developer's Guide for Oracle TopLink – Using a SQLCall, http://oracle.com/technology/products/ias/toplink/doc/11110/devguide/qrybas.htm#CIHEBFID
20. Kost, S.: An Introduction to SQL Injection Attacks for Oracle Developers, http://www.net-security.org/dl/articles/IntegrigyIntrotoSQLInjectionAttacks.pdf

21. Kost, S., Kanter, J.: Evading Network-Based Oracle Database Intrusion Detection Systems, http://www.integrigy.com/security-resources/whitepapers/Integrigy_Evading_Oracle_IDS.pdf
22. Application Security Inc. DbProtect, http://www.appsecinc.com/products/dbprotect
23. Wiens, J.: Time To Take Action Against Data Loss, http://networkcomputing.com/channels/security/showArticle.jhtml?articleID=203103046
24. Guardium Products, http://www.guardium.com/products/products.html
25. Imperva SecureSphere Database Security Gateway, http://imperva.com/products/dsg.html
26. Java Data Objects, http://db.apache.org/jdo
27. The LINK Project, http://msdn2.microsoft.com/fr-fr/netframework/aa904594(en-us).aspx
28. Quaere, http://quaere.codehaus.org
29. JEQUEL, http://jequel.de
30. Mookhey, K. K., Burghate, N.: Detection of SQL Injection and Cross-site Scripting Attacks, http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-mookhey/old/bh-us-04-mookhey_whitepaper.pdf
31. Bachfeld, D.: Lethal injection, http://www.heise-security.co.uk/articles/75593/0
32. Imperva SecureSphere Web Application Firewall, http://imperva.com/products/waf.html
33. F5 BIG-IP Application Security Manager, http://www.f5.com/products/big-ip/product-modules/application-security-manager.html
34. ModSecurity, http://modsecurity.org
35. Gavin, M.: The Forrester Wave™: Web Application Firewalls, Q2 2006, http://www.forrester.com/Research/Document/Excerpt/0,7211,38766,00.html
36. SecureIIS Web Server Security, http://eeye.com/html/assets/pdf/datasheet_secureiis.pdf
37. eEye Digital Security: Windows 2000 IIS 5.0 Remote Buffer Overflow Vulnerability (Remote SYSTEM Level Access), http://research.eeye.com/html/advisories/published/AD20010501.html
38. Yuhanna, N.: The Forrester Wave™: Enterprise Database Auditing And Real-Time Protection, Q4 2007, http://www.guardium.com/files/resources/ForresterWaveEnterpriseDatabaseAuditingQ42007.pdf

Appendix: SQLDOM4J API Architecture

Below is a UML class diagram representing a sample concrete model of our prototype solution's architecture. This is a simplified view (not all classes and class members are shown) which displays both the API aspect of the solution and its internal components. For instance, SQL query construction classes are shown (`SelectQuery`, `UpdateQuery`, `InsertQuery`, `DeleteQuery`) as well as non-instantiable abstract classes (`SQLQuery`).

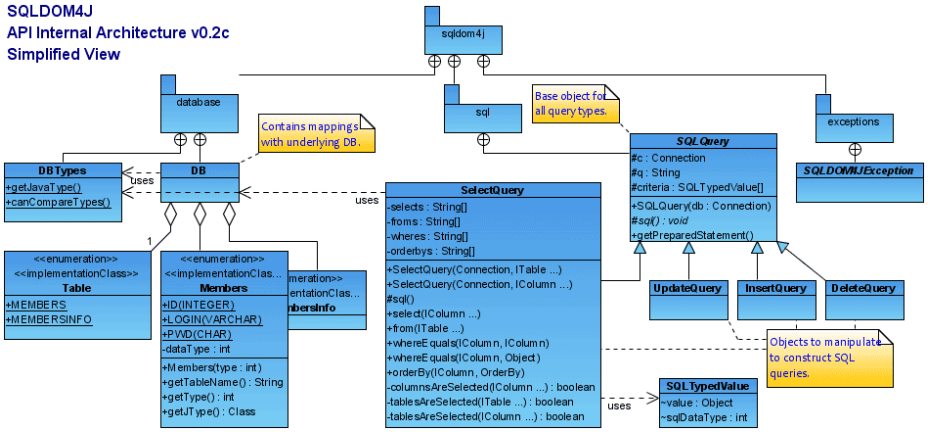

Fig. 2. SQLDOM4J sample API architecture (simplified view)

The database package contains static data type mappings (between SQL, JDBC and Java) and associated methods in the `DBTypes` class which is used internally, and the `DB` class which is accessed by developers to select tables and fields when they build queries. The `DB` class maps the underlying database structure (tables' and columns' names and data types) via static enumerations; the first (`DB.Table`) containing the table names and the subsequent containing each table's definition. In our case, the API was mapping a dummy database containing 2 tables ('Members' and 'MembersInfo'); the diagram shows the details for the 'Members' table mappings.

Exceptions are kept in a specific package and are not shown here except for the generic `SQLDOM4JException`.