

A Study of Database Protection Techniques

Lakshmi Prasad¹, Bibin Varghese², Smita C Thomas³

¹(P G Scholar, Department of Computer Science and Engineering, Mount Zion College of Engineering, Kadammanitta

Email: lakshmiprd24@gmail.com)

²(Assistant Professor, Department of CSE, Mount Zion College Of Engineering, Kadammanitta)

³(Research Scholar, Vels University, Chennai

Email: smitabejoy@gmail.com)

Abstract:

Database management systems are important to businesses and organizations because they provide a highly efficient method for handling multiple types of data. Some of the data that are easily managed with this type of system include: employee records, student information, payroll, accounting, project management, inventory and library books. These systems are built to be extremely versatile. A database is a key tool for businesses that can cause serious headaches if breached. There are some serious threats because of hackers done various attempts to steal the data in the database. Various attacks like Sql injection and Stored Injection containing Cross site scripting may change the information in the databases which decreases the truthfulness of the database. This paper, proposes SEPTIC, a mechanism for DBMS attack prevention, which can also assist on the identification of the vulnerabilities in the applications and SQLrand which applies the concept of instruction-set randomization to SQL, creating instances of the language that are unpredictable to the attacker. Queries injected by the attacker will be caught and terminated by the database parser.

Keywords —SEPTIC; Sql injection; Stored Injection; Cross site scripting; SQLrand

I. INTRODUCTION

Since the Usage and application of internet increases, communications and computer network technology has been rapid development, especially the emergence of the Internet, makes the computer used in government, business, business, education, health care and other areas of society at an unprecedented rate, which are

profound impact on people's economic, work and live. Network brings you convenience while brings more and more malicious attackers. They target the network database; make the database information security under serious threat. The SQL attack is one of common attacks, the tool of the SQL attack is SQL statements. Attackers towards programming vulnerability of application developers, submit wellconstructed SQL

statement to the server to achieve the goal of attacking.

SQL injection attacks (SQLI), for example, continue to rise in number and severity [2]. Commonly used defences are validation functions, web application firewalls (WAFs), and prepared statements. The first two inspect web application inputs and sanitize those that are considered dangerous, whereas the third bounds inputs to placeholders in the SQL queries. Other anti-SQLI mechanisms have been developed but less adopted. Some of these monitor and block SQL queries that deviate from specific models, but the inspection is made without full knowledge about how they are processed by the DBMS. In all these cases, developers and system administrators make assumptions about how the server-side scripting language and the DBMS work and interact, which sometimes are simplistic, whereas in others are blatantly wrong.

For example, programmers usually assume that the PHP function `mysql_real_escape_string` always effectively sanitizes inputs and prevents SQLI attacks, which is not true. Also, they often assume that values retrieved from a database do not need to be validated before being inserted in a query, leading to second-order injection vulnerabilities. This is visible when, for instance, the code `admin' - -` is sanitized by escaping the prime character before sending it to the database, but the DBMS unsanitizes it before actually storing it. Later, the code is retrieved from the database and used unsanitized in some query, carrying out the attack.

Such simplistic/wrong assumptions seem to be caused by a semantic mismatch between how an SQL query is expected to run and what actually occurs when it is executed (e.g., the programmer expects it to be sanitized but the DBMS unsanitizes it). This mismatch may lead to vulnerabilities, as the protection mechanisms may be ineffective (e.g., they may miss some attacks). To avoid this problem, SQLI attacks could be handled inside, after the server-side code processes the inputs and the DBMS validates the queries, reducing the amount of assumptions

that are made. The mismatch and this solution are not restricted to web applications, meaning that the same problem can be present in other business applications. In fact, injection attacks are a generic form of attack, transversal to all applications that use a database as backend.

The paper propose a similar idea for applications backed by databases. Here propose to block injection attacks inside the DBMS at runtime. The approach is *Self-Protection databases from attacks* (SEPTIC). The DBMS is an interesting location to add protections against such attacks because it has an unambiguous knowledge about what will be considered as clauses, predicates, and expressions of an SQL statement. No mechanism that actuates outside of the DBMS has such knowledge.

Here address two categories of database attacks: *SQL injection attacks*, which continue to be among those with highest risk and for which new variants continue to appear; and *stored injection attacks*, including stored cross-site scripting, which also involve SQL queries. For SQLI, here propose to catch the attacks by comparing queries with query models, improving an idea that has been previously used only outside of the DBMS and by comparing queries with validated queries with a similarity method, improving detection accuracy. For stored injection, here employ plugins to deal with specific attacks before data are inserted in the database. SEPTIC relies on two new concepts. Before detecting attacks, the mechanism can be trained by forcing calls to all queries in an application. The result is a set of query models. However, as training may be incomplete and not cover all queries, here introduce the notion of putting in *quarantine* queries at runtime for which SEPTIC has no query model. The second concept, *aging*, deals with updates to query models after a new release of an application, something that is inevitable in real world software. Both concepts allow a reduction of the false negative (attacks not detected) and false positive (alerts for nonattacks) rates. Another technique which introduce the concept of instruction-set randomization for safeguarding systems against any type of code-injection attack,

by creating process-specific randomized instruction sets (e.g., machine instructions) of the system executing potentially vulnerable software. An attacker that does not know the key to the randomization algorithm will inject code that is invalid for that randomized processor (and process), causing a runtime exception.

Here applies the same technique to the problem of SQL injection attacks: create randomized instances of the SQL query language, by randomizing the template query inside the CGI script and the database parser. To allow for easy retrofitting of the solution to existing systems, introduce a de-randomizing proxy, which converts randomized queries to proper SQL queries for the database. Code injected by the rogue client evaluates to undefined keywords and expressions. When this is the outcome, then standard keywords (e.g., “or”) lose their significance, and attacks are frustrated before they can even commence. The performance overhead of our approach is minimal, adding up to 6.5ms to query processing time.

II. LITERATURE SURVEY

There is a vast corpus of research in web application security, so Here survey only related runtime protection mechanisms, which is the category in which SEPTIC fits. All the works Here describe have a point in common that makes them quite different from this paper: their focus is on *how to do detection or protection*. On the contrary, this paper is more concerned with an architectural problem: *how to do detection/protection inside the DBMS*, so that it runs out of the box when the DBMS is started. None of the related works does detection inside the DBMS.

Buehrer *et al.* present a similar scheme that manages to detect mimicry attacks by enriching the models (parse trees) with comment tokens. However, their scheme cannot deal with most attacks related with the semantic mismatch problem. SqlCheck is another scheme that compares parse trees to detect attacks. SqlCheck detects some of the attacks related with semantic mismatch, but not those involving encoding and

evasion. Again, both these mechanisms involve modifying the application code, unlike SEPTIC. DIGLOSSIA is a technique to detect SQLI attacks that was implemented as an extension of the PHP interpreter. The technique first obtains the query models by mapping all query statements’ characters to shadow characters except user inputs, and computes shadow values for all string user inputs. Second, for a query execution, it computes the query and verifies if the root nodes from the two parsed trees are equal. Like SEPTIC, DIGLOSSIA detects syntax structure and mimicry attacks but, unlike SEPTIC, it neither detects second-order SQLI once it only computes queries with user inputs, nor encoding and evasion space characters attacks as these attacks do not alter the parse tree root nodes before the malicious user inputs are processed by the DBMS. Although better than AMNESIA and CANDID, it does not deal with all semantic mismatch problems. Works based on anomaly intrusion detection systems also aim to detect SQLI attacks by comparing models with queries sent by web applications.

Valeuret *al.* present one of these works. The system also undergoes a training phase to create models (a set of profiles) of normal access to the database. In runtime, it detects deviations from that model. SQL-IDS is another system that compares queries against query specifications that define the query syntactic structure (a kind of model) implemented in the application. However, there is no information about how such specifications are created, despite the authors arguing that their source code does not need instrumentation.

SQLProb is a proxy-based system that also uses models previously extracted by a specific data collection phase. Afterward, the system evaluates the queries produced by applications, parsing them, and extracting their user inputs, then validates the inputs against the parse tree, resorting to an input repository. For web services, Laranjeiro *et al.* propose a similar approach to discover SQL and XPath injection attacks. In a first phase, their approach learns regular requests

by representing them into invariant statements (a kind of models), and later protects web services by matching incoming requests with those collected in the learning phase. Moreover, the approach uses heuristics to deal with incoming requests that the approach does not learn as invariant. All these systems, like the previously mentioned tools, are external to the DBMS, so they do not use our approach to deal with the semantic mismatch problem. Machine-learning approaches for detection SQLI have been emerging. idMAS-SQL is one of these works. SOFIA also uses machine learning to classify queries issued by applications, resorting to a clustering algorithm. The tool has a training phase to get the parse tree from legitimate queries and to create clusters with these trees. Afterward, in evaluating phase it classifies as attack the queries that do not fit any cluster. Dynamic taint analysis tracks the flow of user inputs in the application and verifies if they reach dangerous instructions. Xu *et al.* show how this technique can be used to detect SQLI and reflected XSS. They annotate the arguments from source functions and sensitive sinks as untrusted and instrument the source code to track the user inputs to verify if they reach the untrusted arguments of sensitive sinks (e.g., functions that send queries to the database). ARDILLA creates attack vectors that contain mutations of user inputs generated previously, and then deploys such vectors, tracking the inputs and verifying if they exploit SQLI and XSS vulnerabilities. A different but related idea is implemented by CSSE that protects PHP applications from SQLI, XSS, and OSCI by modifying the platform to distinguish between what is part of the program and what is external (input), defining checks to be performed to the latter (e.g., if the query structure becomes different due to inputs). WASP does something similar to block SQLI attacks. SEPTIC does not track inputs in the application, but runs in the DBMS.

Recently, Ahuja *et al.* [1] and Masri and Sleiman presented two works about prevention of SQLI attacks. The first presents a tool called SQLPIL that simply transforms SQL queries

created as strings into prepared statements, preventing SQLI in the source code. The second presents three new approaches to detect and prevent SQLI attacks based on rewriting queries, encoding queries, and adding assertions to the code.

Here propose three new directions to detect and prevent SQLIA, as an alternative solution. These are (i) Query Rewriting-based approach, (ii) Encoding-based approach, and (iii) Assertion-based approach. The main objective of first approach is to mitigate the effect of concatenation operation during query generation. The objective of the second approach is to mitigate the mal-effect of bad input on query semantics. The assertion-based technique is a state verification technique which can be performed either at application level or at database level. Here provide a comparative study among our proposed approaches based on security guaranty, usability and applicative point of view.

III. PROPOSED METHOD

SEPTIC Approach

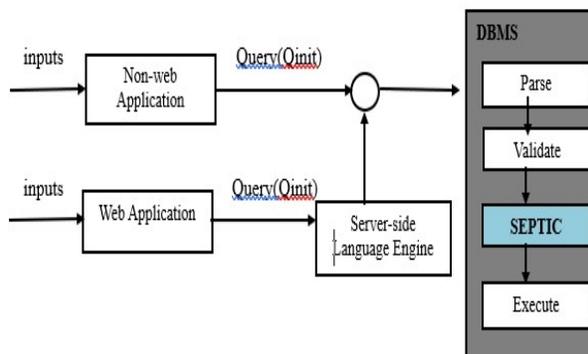
SEPTIC is implemented by a module inside the DBMS, allowing every query to be checked for attacks. The semantic mismatch problem is circumvented because queries are evaluated for detection purposes near the end of the DBMS data flow, just before the query is executed. As SEPTIC is inside the DBMS, it is independent from the application (e.g., from the application programming language) and the way it builds queries (e.g., dynamically). This lets SEPTIC analyse queries issued by any kind of application. However, with support from the application, SEPTIC can also contribute to the identification of the vulnerabilities that are being exploited.

Approach Overview

Figure 1 shows the architecture of a web and a non-Web application with a backend database. When the system starts, SEPTIC may undergo a training phase in order to obtain the query models for the application. Here designates administrator, the person or persons in charge of managing the

DBMS and SEPTIC (e.g., decides when the training mode ends). Later on, when SEPTIC is put in normal operation, it works basically in the following way.

- 1) An application requests the execution of a query. Optionally, the query instruction may contain an (external) identifier produced by the server-side language engine (SSLE) or the application.
- 2) The DBMS receives, parses, and validates the query. Before it executes the query, SEPTIC is called to retrieve its associated query model, which is used to detect and block a potential incoming attack. If an external identifier arrives with the query, it is extracted to get context information about the places in the source code of the application where the query was built. This information can be helpful to locate a vulnerability in case an attack is found.



SQLRAND Architecture

Injecting SQL code into a web application requires little effort by those who understand both the semantics of the SQL language and CGI scripts. Numerous applications take user input and feed it into a pre-defined query. The query is then handed to the database for execution. Unless developers properly design their application code to protect against unexpected data input by users, alteration to the database structure, corruption of data or revelation of private and confidential information may be granted inadvertently.

For example, consider a login page of a CGI application that expects a user-name and the corresponding password. When the credentials are

submitted, they are inserted within a query template such as the following:

```
"select * from mysql.user where username=' " . $uid . "' and password=password(' " . $pwd . " ');"
```

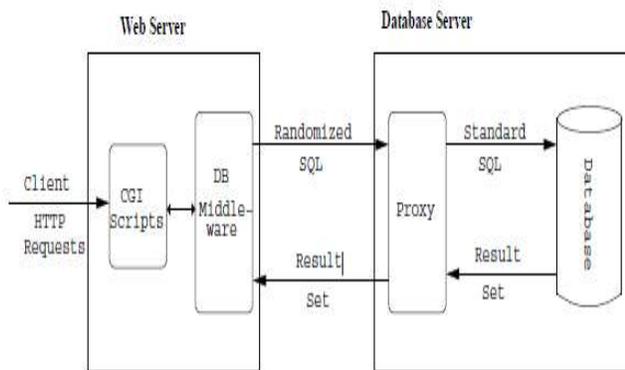
Instead of a valid user-name, the malicious user sets the \$uid variable to the string: ' or 1=1; - -', causing the CGI script to issue the following SQL query to the database:

```
"select * from mysql.user where username=' ' or 1=1; - -' and password=password('_any_text_');"
```

Notice that the single quotes balance the quotes in the pre-defined query, and the double hyphen comments out the remainder of the SQL query. Therefore, the password value is irrelevant and may be set to any character string. The result set of the query contains at least one record, since the "where" clause evaluates to true. If the application identifies a valid user by testing whether the result set is non-empty, the attacker can bypass the security check.

The solution extends the application of Instruction-Set Randomization to the SQL language: the SQL standard keywords are manipulated by appending a random integer to them, one that an attacker cannot easily guess. Therefore, any malicious user attempting an SQL injection attack would be thwarted, for the user input inserted into the "randomized" query would always be classified as a set of non-keywords, resulting in an invalid expression. Essentially, the structured query language has taken on new keywords that will not be recognized by the database's SQL interpreter. A difficult approach would be to modify the database's interpreter to accept the new set of keywords. However, attempting to change its behaviour would be a daunting task. Furthermore, a modified database would require all applications submitting SQL queries to conform to its new language. Although dedicating the database server for selected applications might be possible, the random key would not be varied among the SQL applications using it. Ideally, having the ability to vary the random SQL key, while maintaining one database system, grants a greater level of security, by making it difficult to subvert multiple applications by successfully attacking the

least protected one. The design consists of a proxy that sits between the client and database server (see Figure 2).



Note that the proxy may be on a separate machine, unlike the figure's depiction. By moving the de-randomization process outside the DBMS to the proxy, gains in flexibility, simplicity, and security. Multiple proxies using unique random keys to decode SQL commands can be listening for connections on behalf of the same database, yet allowing disparate SQL applications to communicate in their own "tongue." The interpreter is no longer bound to the internals of the DBMS. The proxy's primary obligation is to decipher the random SQL query and then forward the SQL command with the standard set of keywords to the database for computation. Another benefit of the proxy is the concealment of database errors which may unveil the random SQL keyword extension to the user. A typical attack consists of a simple injection of SQL, hoping that the error message will disclose a subset of the query or table information, which may be used to deduce intuitively hidden properties of the database. By stripping away the randomization tags in the proxy, we need not worry about the DBMS inadvertently exposing such information through error messages; the DBMS itself never sees the randomization tags. Thus, to ensure the security of the scheme, we only need to ensure that no messages generated by the proxy itself are ever sent to the DBMS or the front-end

server. Given that the proxy itself is fairly simple, it seems possible to secure it against attacks. In the event that the proxy is compromised, the database remains safe, assuming that other security measures are in place.

IV. CONCLUSION

This paper explored two forms of protection from attacks against web and business application databases. It presented the idea of catching attacks inside the DBMS, letting it be protected from SQLI and stored injection attacks. Moreover, by putting protection inside the DBMS, here showed that it is possible to detect and block sophisticated attacks, including those related with the semantic mismatch problem. As a second idea, it presented a form of identifying vulnerabilities in application code, when attacks were detected. This paper also presented SEPTIC, a mechanism implemented inside MySQL and SQLrand, a system for preventing SQL injection attacks against web servers. The main intuition is that by using a randomized SQL query language, specific to a particular CGI application, it is possible to detect and abort queries that include injected code. By using a proxy for the de-randomization process, we achieve portability and security gains: the same proxy can be used with various DBMS back-end, and it can ensure that no information that would expose the randomization process can leak from the database itself.

REFERENCES

- [1]. B. Ahuja, A. Jana, A. Swamkar, and R. Halder, "On preventing SQL injection attacks", *Adv. Comput. Syst. Secur.* vol. 395, pp. 49–64, 2015.
- [2]. S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "CANDID: Preventing SQL injection attacks using dynamic candidate evaluations", in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, pp. 12–24, Oct. 2007.
- [3]. Berman, V. Bourassa, and E. Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of the USENIX Technical Conference*, Jan. 1995.
- [4]. S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, pp. 105–120, Aug. 2003.