Hacking the DBMS to Prevent Injection Attacks

Ibéria Medeiros[1] Miguel Beatriz[2] Nuno Neves[3] Miguel Correia[2]
[1]INESC-ID, Faculdade de Ciências, Universidade de Lisboa, Portugal
[2]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
[3]LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
ibemed@gmail.com, miguel.beatriz@tecnico.ulisboa.pt, nuno@di.fc.ul.pt,
miguel.p.correia@tecnico.ulisboa.pt

ABSTRACT

After more than a decade of research, web application security continues to be a challenge and the backend database the most appetizing target. The paper proposes preventing injection attacks against the database management system (DBMS) behind web applications by embedding protections in the DBMS itself. The motivation is twofold. First, the approach of embedding protections in operating systems and applications running on top of them has been effective to protect these applications. Second, there is a semantic mismatch between how SQL queries are believed to be executed by the DBMS and how they are actually executed, leading to subtle vulnerabilities in protection mechanisms. The approach – SEPTIC – was implemented in MySQL and evaluated experimentally with web applications written in PHP and Java/Spring. In the evaluation SEPTIC has shown neither false negatives nor false positives, on the contrary of alternative approaches, causing also a low performance overhead in the order of 2.2%.

Keywords

web applications; injection attacks; DBMS self-protection; security; software security.

1. INTRODUCTION

Web applications are an important component of today's economy, with major players such as Google, Facebook and Yahoo. However, after more than a decade of research, web application security continues to be a challenge. For example, recently SQL injection (SQLI) attacks have allegedly victimized 12 million Drupal sites [4], SQLI attacks were considered an important threat against critical infrastructures [16], and stored cross-site scripting (XSS) was used to inject malicious code in servers running Wordpress [29].

The mechanisms most commonly used to protect web applications from malicious inputs are web application firewalls (WAFs), sanitization/validation functions, and prepared statements in the application source code. The first two mechanisms, respectively, inspect web application inputs and block and sanitize those that are considered malicious/dangerous, whereas the third bounds inputs to placeholders in the query. Other anti-SQLI mechanisms have been presented in the literature, but barely adopted. Some of these mechanisms monitor SQL queries and block them if they deviate from certain query models, but the queries are inspected without full knowledge about how the server-side scripting language and the DBMS process them [6, 7, 13, 34, 20]. In all these cases, administrators and programmers make assumptions about how the server-side language and the DBMS work and interact, which sometimes are simplistic, others blatantly wrong. For example, programmers often assume that PHP function `mysql_real_escape_string` always effectively sanitizes inputs and prevents SQLI attacks, which is not true, or they may ignore that data may be unsanitized when inserted in the DBMS leading to second-order SQLI vulnerabilities.

We argue that such simplistic or wrong assumptions are caused by a *semantic mismatch* between how an SQL query is expected to run and what actually occurs when it is executed. This mismatch leads to unexpected vulnerabilities in the sense that mechanisms such as those mentioned above can become ineffective, resulting in false negatives (attacks not detected). To avoid this problem, these attacks could be handled 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.

Today operating systems are much more secure than years ago due to the deployment of automatic protection mechanisms in themselves, in core libraries (e.g., .NET and glibc), and in compilers. For example, address space layout randomization, data execution prevention, or canaries/stack cookies are widely deployed in Windows and Linux [15, 19]. These mechanisms block a large range of attacks irrespectively of the programmer following secure programming practices or not. Clearly, something similar would be desirable for web applications. The DBMS is an interesting location to add these protections as it is a common target for attacks.

We propose modifying – "hacking" – DBMSs to detect and block attacks in runtime without programmer intervention. We call this approach *SElf-Protecting daTabases preventIng attaCks* (SEPTIC). In this paper, we focus on the two main categories of attacks related with databases: *SQL injection* attacks, which continue to be among those with highest risk [37] and for which new variants continue to appear [27], and *stored injection attacks*, which also involve SQL queries. For

CODASPY'16, March 09-11, 2016, New Orleans, LA, USA

DOI: http://dx.doi.org/10.1145/2857705.2857723

SQLI, we propose detecting attacks essentially by comparing queries with query models, taking to its full potential an idea that has been previously used only outside of the DBMS [6, 7, 13, 34] and circumventing the semantic mismatch problem. For stored injection, we propose having plugins to deal with specific attacks before data is inserted in the database.

We demonstrate the concept with a popular deployment scenario: MySQL, probably the most popular open-source DBMS [30], and PHP, the language more used in web applications (more than 77%) [17]. We also explore Java/Spring, the second most employed programming language. We evaluate SEPTIC experimentally to assess its effectiveness to block attacks, including a set of novel SQLI attacks presented recently [27]. SEPTIC is compared with a set of alternative solutions, including the ModSecurity WAF and recent anti-SQLI mechanisms proposed in the literature, with SEPTIC showing neither false negatives nor false positives, on the contrary of the others. We also evaluate the impact of our approach on the performance of web applications using BenchLab [8]. The overhead was very low, around 2.2%.

The main contribution of this paper is a mechanism that comes out of the box with the DBMS to detect and block injection attacks against the DBMS inside the DBMS itself. Moreover, by being placed inside the DBMS, the mechanism is able to mitigate the semantic mismatch problem and handle sophisticated SQL injection and stored injection attacks.

2. DBMS INJECTION ATTACKS

We define *semantic mismatch* as the distance between how programmers assume SQL queries are executed by the DBMS and how queries are effectively executed. This mismatch often leads to mistakes in the implementation of protections in the source code of web applications, letting these applications vulnerable to SQL injection and other attacks involving the DBMS. The semantic mismatch is subjective in the sense that it depends on the programmer, but some mistakes are usual. A common way to try to prevent SQLI consists in sanitizing user inputs before they are used in SQL queries. For instance, in PHP `mysql_real_escape_string` precedes special characters like the prime or the double prime with a backslash, transforming these delimiters into normal characters. However, sanitization functions do not behave as envisioned when the special characters are represented differently from expected. This problem has lead us to use the term semantic mismatch to refer to the gap between how the SQL queries that take these sanitized inputs are believed to be executed by the programmer, and how they are actually processed by the DBMS.

We identified several DBMS injection attacks in the literature, including a variety of cases related to semantic mismatch [9, 11, 12, 22, 27, 31]. Table 1 organizes these attacks in classes. The first three columns identify the classes, whereas the fourth and fifth state what PHP sanitization functions and the DBMS do to the example malicious inputs in the sixth column.

As mentioned in the introduction, we consider two main classes of attacks: *SQL injection* and *stored injection* (first column). These classes are divided in sub-classes for common designations of attacks targeted at DBMSs (A to C). Obfuscation attacks (class A) are the most obvious cases of semantic mismatch. Classes S.1 and S.2 classify attacks in terms of the way they affect the syntactic structure of the SQL query. Class S.1 is composed of attacks that modify this structure. Class S.2 is composed of attacks that modify the query but mimic its original structure.

Class A, obfuscation, contains five subclasses. Consider the code excerpt in Fig. 1 that shows a login script that checks if the credentials the user provides (username, password) exist in the database.[1] The user inputs are sanitized by the `mysql_real_escape_string` function (lines 1-2) before they are inserted in the query (line 3) and submitted to the DBMS (line 4). If an attacker injects the `admin'--` string as username (line 1), the `$user` variable receives this string sanitized, with the prime character preceded by a backslash. The user `admin\'--` does not exist in the database so this SQLI attack is not successful.

```
1 $user = mysql_real_escape_string($_POST['username']);
2 $pass = mysql_real_escape_string($_POST['password']);
3 $query = "SELECT * FROM users WHERE username='$user'
      AND password='$pass'";
4 $result = mysql_query($query);
```

Figure 1: Script vulnerable to SQLI with encoded characters.

On the contrary, this sanitization is ineffective if the input uses URL encoding [5], leading to an attack of class A.1. Suppose the attacker inserts the username URL-encoded: `%61%64%6D%69%6E%27%2D%2D%20`. `mysql_real_escape_string` function does not sanitize the input because it does not recognize `%27` as a prime. However, MySQL receives that string as part of a query and decodes it, so the query executed is `SELECT * FROM users WHERE username='admin'-- ' AND password='foo'`. The attack is therefore effective because this query is equivalent to `SELECT * FROM users WHERE username='admin'` (no password has to be provided). This is also an attack of class S.1 as the structure of the query is modified (the part that checks the password disappears). The other subclasses of class A involve similar techniques. In class A.2 the attacker encodes some characters in Unicode, e.g., the prime as `U+02BC`. In A.3 decoding involves calling dynamically a function (e.g., the prime is encoded as `char(39)`). Class A.4 attacks use spaces and equivalent strings to manipulate queries (e.g., concealing a space with a comment like `/**/`) [9]. A.5 attacks abuse the fact that numeric fields do not require values to be enclosed with primes, so a tautology similar to the example we gave for A.1 can be caused without these characters, fooling sanitization functions like `mysql_real_escape_string`.

Stored procedures that take user inputs may be exploited similarly to queries constructed in the application code (class B). These inputs may modify or mimic the syntactic structure of the query, leading to attacks of classes S.1 or S.2. Blind SQLI attacks (class C) aim to extract information from the database by observing how the application responds to different inputs, so they also fall in classes S.1 or S.2.

Class D attacks – stored injection – are characterized by being executed in two steps: the first involves doing an SQL query that inserts attacker data in the database (`INSERT`, `UPDATE`); the second uses this data to complete the attack. The specific attack depends on the data inserted in the database and how it is used in the second step. In a second order SQLI attack (class D.1) the data inserted is a string specially crafted to be inserted in a second SQL query executed in the second step. This second query is the attack

[1] All examples included in the paper were tested with Apache 2.2.15, PHP 5.5.9 and MySQL 5.7.4

Class		Class name	PHP sanit. func.	DBMS	Example malicious input
SQL injection	**A**	Obfuscation			
	A.1	- Encoded characters	do nothing	decodes and executes	%27, 0x027
	A.2	- Unicode characters	do nothing	translates and executes	U+0027, U+02BC
	A.3	- Dynamic SQL	do nothing	completes and executes	char(39)
	A.4	- Space character evasion	do nothing	removes and executes	char(39)/**/OR/**/1=1--
	A.5	- Numeric fields	do nothing	interprets and executes	0 OR 1=1--
	B	Stored procedures	sanitize	executes	admin' OR 1=1
	C	Blind SQLI	sanitize	executes	admin' OR 1=1
sted. inj.	**D**	Stored injection code			
	D.1	- Second order SQLI	–	executes	any of the above
	D.2	- Stored XSS	–	–	<script>alert('XSS')</script>
	D.3	- Stored RCI, RFI, LFI	–	–	malicious.php
	D.4	- Stored OSCI	–	–	; cat /etc/passwd
	S.1	Syntax structure	sanitize	executes	admin' OR 1=1
	S.2	Syntax mimicry	sanitize	executes	admin' AND 1=1--

RCI: Remote Code Injection; RFI:Remote File Inclusion; LFI: Local File Inclusion; OSCI: OS Command Injection

Table 1: Classes of attacks against DBMSs.

itself, which may fall in classes S.1 or S.2. This is another case of semantic mismatch as the sanitization created by functions like `mysql_real_escape_string` is removed by the DBMS when the string is inserted in the database (first step of the attack). A stored XSS (class D.2) involves inserting a browser script (typically JavaScript) in the database in the first step, then returning it to one or more users in the second step. In class D.3 the data inserted in the database can be a malicious PHP script or an URL of a website containing such a script, resulting on local or remote file inclusion, or on remote code injection. In class D.4 attacks the data that is inserted is an operating system command, which is executed in the second step.

3. THE SEPTIC APPROACH

This section presents the SEPTIC approach. The idea consists in having a module inside the DBMS that processes every query it receives in order to detect attacks against the DBMS. We designate both the approach and this module by SEPTIC. This approach circumvents the semantic mismatch problem as detection is executed near the end of the data flow entering the DBMS, just before it executes the query.

3.1 SEPTIC overview

This section presents an overview of the approach. Fig. 2(a) represents the architecture of a web application, including the DBMS and SEPTIC. This module is placed inside the DBMS, after the parsing and validation of the queries. There may be also hooks inside the server-side language engine (Section 3.3).

In runtime SEPTIC works basically the following way:

1. *Server-side application code:* requests the execution of a query Q;

2. *Server-side language engine:* receives Q and sends it to the DBMS; it may add an identifier (ID) to Q;

3. *DBMS:* receives, parses, validates, and executes Q; between validation and execution, SEPTIC detects and possibly blocks an incoming attack.

Fig. 2(b) provides more details on the operation of SEPTIC. The figure should be read starting from the gray arrow at the top/left. Dashed arrows and dashed processes represent alternative paths.

When a web application is started, SEPTIC has to undergo some training before it enters in normal execution. Training is typically done by putting SEPTIC in *training mode* and running the application for some time without attacks (Section 3.5). Training results in a set of query models (QM) stored in SEPTIC.

In normal execution, for every query SEPTIC receives, it extracts the query ID and the query structure (QS). If no ID is provided, SEPTIC generates one (Section 3.3). SEPTIC detects attacks first by comparing the query structure (QS) with the query model(s) stored for that ID. If there is no match, an SQLI attack was detected. Otherwise, SEPTIC uses a set of *plugins* to discover stored injection attacks. If no attack is detected the query is executed.

The action taken when an attack is detected depends on the mode SEPTIC is running. In *prevention mode*, SEPTIC aborts the attacks, i.e., it drops the queries and the DBMS stops the query processing. In *detection mode*, queries are executed, not dropped. In both modes of operation, SEPTIC logs information about the attacks detected.

In summary, SEPTIC runs in three modes, one for training (*training mode*) and two for normal operation (*prevention mode* and *detection mode*).

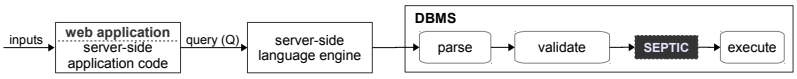
The following sections present the approach in detail.

3.2 Query structures and query models

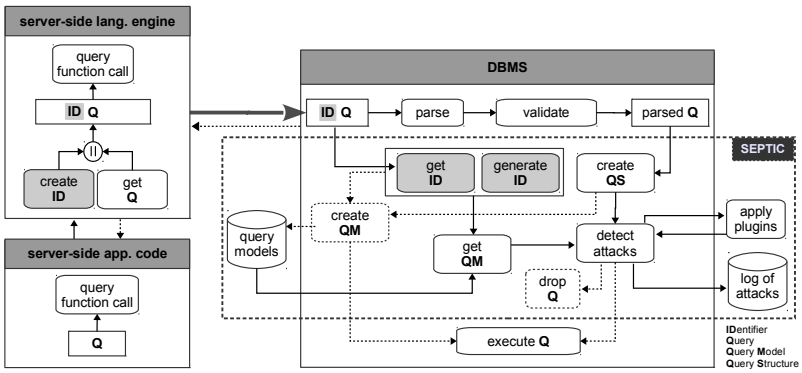
As explained in the previous section, in prevention and detection modes SEPTIC finds out if a query is an attack by comparing the *query structure* with the *query model(s)* associated to the query's ID.

We consider that SEPTIC receives the parse tree of every query represented as a *list of stacks* data structure. Each stack of the list represents a clause of the query (e.g., `SELECT`, `FROM`, `WHERE`), and each of its nodes contains data about the query element, such as category (e.g., field, function, operator), data type (e.g., integer, string), and data.

The *query structure* (QS) of a query is constructed by creating a single stack with the content of all the stacks in the list of stacks of a query. Fig. 3 depicts a generic query structure, showing from bottom to top the clauses and their elements. Each *node* (a row) represents an element of the query. Each node is composed by the element type (category) and the element data: ⟨ELEM_TYPE, ELEM_DATA⟩. The single exception is the alternative format ⟨DATA_TYPE, DATA⟩ that represents an input inserted in the query and its (primitive) data type (DATA_TYPE). A part of the query is considered to be an input if its type is primitive (e.g., a string or an integer) or if it is compared to something in a predicate. For the clauses with conditional expressions (e.g., `WHERE`) the elements are inserted in QS by doing post-order traversal of the parse tree of the query (i.e., the left child is



(a) Main modules of a web application backed by a DBMS with SEPTIC.



(b) SEPTIC approach data flows.

Figure 2: Architecture and data flows of a web application and SEPTIC (optional components in gray).

elem_type	elem_data
...	...
elem_type	elem_data
clause_name	elem_data
(...)	(...)
elem_type	elem_data
...	...
elem_type	elem_data
clause_name	elem_data

Figure 3: A generic query structure.

visited and inserted in the stack first, then the right child, and so on until the root).

As mentioned in the previous section, in training mode SEPTIC creates query models. Specifically, it creates a *query model* (QM) whenever the DBMS processes a query, but stores it only if that model is not yet stored for that query ID. The query model is created based on the query structure of the query. The process consists simply in substituting DATA by a special value $\perp$ in all $\langle$DATA_TYPE, DATA$\rangle$ nodes to denote that these fields shall not be compared during attack detection (Section 3.4). All the other nodes are identical in QM and QS.

Take as example the query `SELECT name FROM users WHERE user='alice' AND pass='foo'`. Fig. 4 represents its (a) parse tree, (b) structure (QS), and (c) model (QM). In Fig. 4(b) and (c) the gray items at the bottom have data about the `SELECT` and `FROM` clauses, whereas the rest are about the `WHERE` clause. In Fig. 4(b) the inputs are represented in bold and in Fig. 4(c) they have the special value $\perp$ as explained. In the left-hand column, each item of the query takes a category (field, data type, condition operator, etc.), whereas the right-hand column has the query's keywords, variables and primitive data type. Primitive data types (real, integer, decimal and string) also take a category, such as STRING_ITEM (e.g., in the third row).

3.3 Query identifiers

Each query received by the DBMS has to be verified against one or more query models. *Query identifiers* (IDs) are used to match queries to their models. More specifically, each query is assigned an ID and for each ID the training mode creates a set of one or more models. The SEPTIC module matches a query to a set of models. From the point of view of the module, IDs are opaque, i.e., their structure is not relevant.

SEPTIC can use three kinds of IDs, depending on where they are generated: in the server-side language engine (SSLE), in the DBMS, and outside both the SSLE and the DBMS. We explain them below.

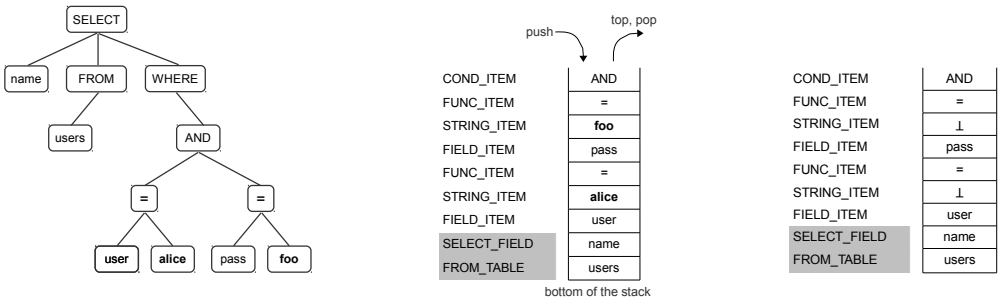
3.3.1 SSLE-generated IDs

The SSLE is arguably the best place to generate the IDs, because this can let the application administrator oblivious to the existence of IDs. Fig. 2(b) shows how this would work generically (SSLE in the left-hand side).

Ideally, every query issued by an application should have a unique ID (Section 3.4) and the SSLE can provide this in many cases. For instance, in the example of Fig. 1 there should be a unique ID for the query constructed in line 3 and issued in line 4. In training mode a model with this ID would be constructed and in prevention/detection modes any query issued there would have the same ID. This would allow comparing the queries against the model without confusion with queries issued elsewhere in the application source code. The SSLE can create this ID when it sees a call to function `mysql_query`. The ID may contain data such as file / line number in which the query is issued. However, this is not enough because many applications have a single function that calls the DBMS with different queries. This function is called from several places in the application, but the file and line number that calls the DBMS is always the same.

We consider the ID format to be a sequence of *file:line* pairs separated by the character |, one pair per each function entered while the query is being composed. Specifically, the first pair corresponds to the line where the DBMS is called and the rest to lines where the query is passed as argument to some function. *file* contains the complete path to allow distinguishing even queries from different applications using the same DBMS.

Assume that the code sample of Fig. 1 is in file `login.php`. The query is created in the same function that calls the func-



(a) Query parse tree (b) Query structure (c) Query model

Figure 4: Representation of a query as parse tree, structure (QS) and model (QM).

tion `mysql_query`, so the ID is simply *login.php:4*, meaning that the DBMS is called in line 4 of file `login.php`. Consider a second example in which line 4 is substituted by `$result = my_db_query($query)`, that function `my_db_query` is defined in file `my_db.php`, and that that function calls the DBMS using `mysql_query` in line 10 of that file. In this example, the ID is *my_db.php:10 | login.php:4*. This ID format is not guaranteed to generate unique IDs in all situations, but we observed no cases in which it did not. In these examples we show the filename without the full pathname for readability.

3.3.2 DBMS-generated IDs

Whenever the SEPTIC module in the DBMS receives a query without ID (e.g., because the SSLE does not generate IDs), it generates an ID automatically (Fig 2(b), gray boxes inside SEPTIC). The DBMS is unaware of what kind of client calls it (e.g., if it is an SSLE), much less about the web application source code. Therefore this ID has a different format. Similarly to SSLE-generated IDs, the application administrator can be oblivious to DBMS-generated IDs.

The ID format is the SQL command (typically `SELECT`) followed by the number of nodes of the query structure. For the example of Fig. 1 that has the query structure of Fig. 4(b) the ID would be *select_9*.

3.3.3 IDs generated outside the DBMS and the SSLE

In the previous two kinds of IDs the web application administrator is left aside from the process of assigning IDs to queries. If for some reason these kinds of IDs are not desirable, the administrator can define his own IDs. These IDs can have any format, e.g., a sequential number or the same format used in SSLE-generated IDs. They can be added to the queries in a few ways: (1) they may be appended to the query when it is defined or when the DBMS is called; or (2) a wrapper may be inserted between the applications code and the DBMS.

3.4 Attack detection

This section explains how SEPTIC detects attacks by dividing the categories of Table 1 in two groups that are detected differently: SQL injection and stored injection. The former contains the classes A to C and D.1, whereas the latter contains class D (except D.1). Class D.1 is also a form of stored injection, but it more convenient to detect these attacks using the approach to discover SQLI.

3.4.1 SQLI detection

SEPTIC detects SQLI attacks by verifying if queries fall in classes S.1 and S.2. We say that attack classes S.1 and S.2 are *primordial for SQL injection* because any SQLI attack falls in one of these two categories. The rationale is that if an SQLI attack neither modifies the query structure (class S.1) nor modifies the query mimicking the structure (class S.2), then it must leave the query unmodified, but then it is not an SQL injection attack.

SEPTIC detects SQLI attacks by comparing each query with the query models for the query's ID *structurally* (for class S.1) and *syntactically* (for class S.2). An attack is flagged if there are differences between the query and all the models for its ID.

Given a query Q with a certain ID and its query structure QS, detection involves iterating all the models QM_i stored for ID. For every QM_i there are two steps: (1) *Structural verification* – if the number of items in QS is different from the number of items in QM_i, then Q does not match the model QM_i and detection for QM_i ends. (2) *Syntactical verification* – if the data type of any of the items of QS is different from the type of any of the items of QM_i (except primitive types), then Q does not match the model QM_i and detection for QM_i ends. Items are compared starting at the top and going down the QS and QM stacks as represented in Figures 4(b)–(c). Primitive data types (real, integer, decimal and string) are an exception because DBMSs implicitly make type-casting between them (e.g., integer to string), so these types are considered equivalent. This process is iterated for all query models QM_i stored for ID. If Q matches one of the models, there is no attack; otherwise there is an attack. The action taken depends on the mode in which SEPTIC is running: in prevention mode the query processing is aborted; in detection mode the query is executed.

As mentioned in Section 3.3, IDs should be unique, so that a single query model QM would be stored for each ID during training. From that point of view *DBMS-generated IDs* are the worst option as they do not ensure uniqueness, except in applications with a very small number of queries. *SSLE-generated IDs* tend to be unique and *IDs generated outside the DBMS and the SSLE* may be created unique.

3.4.2 Stored injection detection

Stored injection attacks have two steps. In the first, malicious data is inserted in the database; in the second that data is taken from the database and used. For example, for stored XSS (D.2) the data includes a script to be executed at the victims' browsers; in the first step it is stored in the database; in the second step that script is taken from the database and sent to a browser. These attacks cannot be detected as SQLI because they do not work by modifying

queries. Therefore, we employ a different solution based on the idea of detecting the presence of malicious data.

SEPTIC detects the presence of malicious data in queries that insert data in the database (first step of the attacks). To do this detection, SEPTIC contains a set of *plugins*, typically one for each type of attack. The plugins analyze the queries searching for code that might be executed by browsers (JavaScript, VB Script), by an operating system (shell script, OS commands, binary files) or by server-side scripts (php). Since running the plugins may introduce some overhead, the mechanism is applied in two steps: (1) *Filtering* – searches for suspicious strings such as: <, >, `href`, and *javacsript* attributes (D.2); protocol keywords (e.g., `http`) and extensions of executable or script files (e.g., *exe, php*) (D.3); special characters (e.g., *;* and |) (D.4). If none is found, detection ends. (2) *Testing* – consists in passing the input to the proper plugin for inspection. For example, if the filtering phase finds the `href` string, the data is provided to a plugin that detects stored XSS attacks. This plugin inserts the input in a simple HTML page with the three main tags (<html>, <head>, <body>), and then calls an HTML parser to determine if other tags appear in the page indicating the presence of a script.

3.5 Training

As explained in Section 3.1, whenever an application is put to run, SEPTIC has to be subjected to training. This is necessary for SEPTIC to create the models of the queries for SQLI detection (Section 3.4.1). There are two methods to do training: *training phase* and *incremental*.

The first method – *training phase* – involves putting SEPTIC in training mode and executing all queries of the web application with correct inputs (i.e., inputs that are not attacks). For every query a model is created and stored, unless the same model is already stored for the same ID. If there is already a model (or more) associated to that ID and the model created is different, then ID becomes associated to two (or more) models. After this is done, SEPTIC can be put in prevention or detection mode and no further intervention from the administrator is needed. The execution of all queries can be achieved in two fashions: (1) using the unit tests of the application; or (2) with the assistance of an external module, called *septic_training*. This module is a web client that works as a crawler. For each web page, it searches for HTML forms and extracts information about the submission method, action, variables and values. Then, it issues HTTP requests for all forms, causing the SQL queries to be sent to the DBMS. These queries can contain user inputs generated by the training module, can be static, or can depend on the results of other queries.

In the second training method – *incremental* – SEPTIC runs in prevention or detection mode all the time, without the need to switch modes and run an explicit training phase. This is very convenient and efficient as long as no attacks happen before the models are created. In both modes, for every query SEPTIC obtains the query structure (QS), gets the set of QMs associated to the query ID, and compares QS with every QM in the set, as explained in Section 3.4.1. From the point of view of training, the relevant case is when there is no QM associated with the ID. In this situation, SEPTIC behaves as if it was in the training phase and creates and stores the query model. The administrator is notified and should confirm that the model was built with a correct query, as it did not appear previously. This verification, however, is not critical for two reasons: (1) it is highly unlikely that the first query with a certain ID in a web application is malicious (attackers take time to find the application and to learn how it works); (2) in the unlikely case of the model being built with a malicious query, this will become conspicuous as correct queries will start being detected as attacks, which will call attention.

In case there are modifications to the application code we envisage two cases. If the changes are not significant, SEPTIC can continue in detection or prevention mode, building new QMs incrementally (incremental method). If the application code suffers many changes, SEPTIC can be put in training mode (training phase method) and all QMs of the application are rebuilt. In this case, the existent QMs are substituted for the new ones. However, in both cases the administrator can opt for either training method. An interesting case is if a query changes from line x to line y in the new version of application with SSLE-generated IDs. This is not problematic if the training phase method is used, as all QMs are rebuilt. In the incremental method two unlikely scenarios may happen: (1) the QM of the query of line y is created and associated to a IDy not in use or to an existent IDy; (2) the IDx (query from line x) receives a new QM, if the line x has now a different query. In both cases the old QMs stored for IDx and IDy are checked for the queries that come with those IDs with the new version of the application. Even if SEPTIC checks that they do not match the old QMs, they match the new QMs, so SEPTIC does not flag an attack (no false positives). False negatives (attacks not detected) are possible as a wrong QM will be associated to an ID, but this is unlikely to happen for two reasons: the two scenarios above are unlikely as a query would have to move to the same line of another; an attack against one of the queries would have to match the QM of the other query.

4. IMPLEMENTATION IN MYSQL AND LANGUAGE RUNTIMES

This section explains how SEPTIC was implemented in MySQL and the creation of identifiers implemented in two contexts: for PHP applications by modifying the PHP runtime (Zend engine); and for web applications implemented in the Spring framework in Java, using aspect oriented programming and a pair of alternatives. The first solution involves a few modifications to the engine's source code, whereas the second does not. Table 2 summarizes the changes made to those software packages.

The implementation of query identifiers has to be compatible with all the components we have been discussing: application source code, SSLE, and DBMS. Specifically, it is important that having SEPTIC in the DBMS or generating IDs in the SSLE does not require modifications to the other components. The solution is to place the identifiers inside DMBS comments. SEPTIC assumes that the first comment in a query is the ID. We place the comment at the beginning of the query, before the query proper.

4.1 Protecting MySQL

We implemented SEPTIC – i.e., the center and right-hand side of Fig. 2(b) – in MySQL 5.7.4. We modified a single file of the package (`sql_parser.cc`) and added a new header file (SEPTIC detector) and a configuration file (SEPTIC setup),

Software	sfm	sfc	loc	sa
MySQL 5.7.4				
- `sql_parser.cc`	1	–	14	–
- SEPTIC detector	–	1	1570	plugins
- SEPTIC setup	–	1	15	–
- *septic_training*	–	1	380	–
Zend engine / PHP 5.5.9				
- mysql extension	1	–	6	–
- mysqli extension	2	–	21	–
- SEPTIC identifier	–	1	249	–
Spring 4.0.5 / Java				
- `JdbcTemplate.java`	1	–	16	–
- SEPTIC identifier	–	1	–	–

sfm: source file modified loc: lines of code
sfc: source file created sa: software added

Table 2: Summary of modifications to software packages.

plus the plugins, which are external to the DBMS (e.g., for stored XSS the plugin is essentially the *jsoup* library [18]). The *septic_training* module is not only external but also runs separately from the DBMS.

The lines added to the `sql_parser.cc` file were inserted in function `mysql_parse`, and just before the call to the function `mysql_execute_command` that executes the query. These lines call the SEPTIC detector with an input corresponding to the query parsed and validated by MySQL. The module performs the previously described operations: builds the query structure (QS); compares QS with its query model (QM); logs que query and the ID if an attack is detected; and optionally drops the query.

SEPTIC is configured using a few switches. The first allows putting SEPTIC in training mode, detection mode (logs attacks), or prevention mode (logs and blocks attacks). The other two allow enabling and disabling respectively the detection of SQLI and stored injection attacks. The values for these switches are defined in a configuration file (SEPTIC setup) that is read by MySQL whenever it is started or restarted. A typical routine consists in setting the first switch to training mode and the other two to *on*, starting the DBMS and the web server, running the *septic_training* module, modifying the first switch to prevention or detection mode, and restarting the DBMS and the application server.

4.2 Inserting identifiers in Zend

In Section 3.3 we discussed three kinds of IDs. We implemented the first kind – *SSLE-generated IDs* – for the PHP language, with the Zend engine as SSLE. As explained in that section, those IDs can be formed of pairs *file:line* separated by |, so the comments we consider in this section have the format `/* file:line | file:line | ... | file:line */`.

Table 2 shows the two Zend engine extensions to which we added a few lines of code to create and insert query IDs. Extensions are used in Zend to group related functions. The table shows also the new header file that we developed for the same purpose (SEPTIC identifier).

The identifiers have to be inserted when the DBMS is called, so we modified in Zend the 11 functions used for this purpose (e.g., `mysql_query`, `mysqli::real_query`). Specifically, the ID is inserted in these functions just before the line that passes the query to the DBMS. This involved modifying three files: `php_mysql.c`, `mysqli_api.c` and `mysqli_nonapi.c`.

When a PHP program is executed, Zend keeps in a stack data about every function call. This stack contains data about the functions called, such as function name, full pathname of the file and line of code where the function was called. This stack allows backtracking the query until a function that does not contain it as argument. This provides the places where the query has been composed and/or passed and allows obtaining query IDs in the format above.

4.3 Inserting identifiers in Spring / Java

We implemented the third kind of IDs explained in Section 3.3 – *IDs generated outside the DBMS and the SSLE* – in Spring / Java. Spring is a framework aimed at simplifying the implementation of enterprise applications in the Java programming language [1]. It allows building Java web applications using the Model-View-Controller (MVC) model. In Spring applications connect to the DBMS via a JDBC driver.

We used three different forms to insert the IDs to show that there are different ways of doing it. The first form consists in inserting the ID directly in the query in the source code of the application. Before the query is issued a comment with the ID is inserted. This is a very simple solution that has the inconvenient of requiring modifications to the source code. The second form uses a *wrapper* to catch the query request before it is sent to JDBC and MySQL, and insert the ID in a comment prefixing the query (e.g., the file and line data). Using a wrapper avoids the need to modify the source code of the application, except for the substitution of the calls to JDBC by calls to the wrapper.

The third form is the most interesting as it does not involve modifications to the application source code. We used *Spring AOP*, an implementation of Aspect-Oriented Programming for Spring, essentially to create a wrapper without modifying the applications' source code [32]. Spring AOP allows the programmer to create *aspects* for the application. These aspects allow intercepting method calls from the application, to insert code that is executed before the methods. These operations are performed without the programmer making changes to the application source code. On the contrary, the programmer develops new files with the *aspects* and their *point cuts*, where the point cuts are the application methods that will be intercepted. We used aspects for intercepting in runtime calls to JDBC, inserting the query ID in the query and proceeding with the query request to MySQL.

5. EXPERIMENTAL EVALUATION

The objective of the experimental evaluation was to answer the following questions: (1) Is SEPTIC able to detect and block attacks against code samples? (2) Is it more efficient than other tools in the literature? (3) Does it solve the semantic mismatch problem better than other tools? (4) How does it perform in terms of false positives and false negatives? (5) Is SEPTIC able to detect and block attacks against real (open source) software? (6) Is the performance overhead acceptable? The evaluation was carried out with the implementation of SEPTIC in MySQL and PHP/Zend.

5.1 Attack detection

This section presents the evaluation of SEPTIC in terms of its ability to detect attacks – questions (1) to (5).

5.1.1 Detection with code samples

To answer questions (1) to (4), we evaluated SEPTIC with: (1) a set of (small) code samples that perform attacks of all classes in Table 1 (17 for the semantic mismatch problem, 7 for other SQLI attacks, 5 for stored injection); (2) 23 code samples from the *sqlmap* project [33], unrelated

	Case	Attack/code
1	SELECT balance FROM acct WHERE password='' OR 1=1 -- '	Yes
2	SELECT balance FROM acct WHERE pin= exit()	Yes
3	...WHERE flag=1000>GLOBAL	Yes
4	SELECT * FROM properties WHERE filename='f.e'	No
5	...pin=exit()	Yes
6	...pin=aaaa()	Yes
7	SELECT * FROM t WHERE flag=TRUE	No
8	...pin=aaaa()	Yes
9	SELECT * FROM t WHERE password=password	Yes
10	CREATE TABLE t (name CHAR(40))	No
11	SELECT * FROM t WHERE name='x'	No

Table 3: Code (attacks) and non-code (non-attacks) cases defined by Ray and Ligatti [27]. Although these authors consider case 10 code/attack we disagree because the input is an integer, which is the type expected by the *char* function.

Figure 5: Placement of the protections considered in the experimental evaluation: SEPTIC, anti-SQLI tools, and WAF.

with semantic mismatch; (3) 11 samples with the code and non-code injection cases defined in [27] (Table 3).

We compare SEPTIC with a WAF and four anti-SQLI tools. Fig. 5 shows the place where the WAF and the anti-SQLI tools act and intercept, respectively, the user inputs sent in HTTP requests and the query sent by the web application. SEPTIC acts inside the DBMS. The WAF, ModSecurity 2.7.3.3 [35], was configured with the OWASP Core Rule Set 2.2.9. ModSecurity is the most adopted WAF worldwide, with a stable rule set developed by experienced security administrators. In fact, it has been argued that its ability to detect attacks is hard to exceed [23]. It detects SQLI and other types of attacks by inspecting HTTP requests. The anti-SQLI tools used were: CANDID [3], AMNESIA [13], DIGLOSSIA [31] and SQLrand [6]. The evaluation of these tools was made manually by analyzing the data in [27] and the papers that describe them. More information about them can be found in Section 6.

In the experiments, first with SEPTIC turned *off* we injected malicious user inputs created manually in the code samples to confirm the presence of the vulnerabilities. We also used the *sqlmap* tool to exploit the vulnerabilities from the first two groups of code samples. *sqlmap* is a tool widely used to perform SQLI attacks, both by security professionals and hackers. Second, with SEPTIC turned *on* and in training mode we injected benign inputs in the code samples for the mechanism to learn the queries and to get their models. Then, we run the same attacks from the first phase in detection mode and analyzed the results to determine if they were detected.

Table 4 shows the results of the evaluation. There were 63 tests executed (third column), 4 of which not attacks (the 4 non-attack cases in Table 3). SEPTIC (last column) correctly detected all 59 attacks (row 34) and correctly did not flag as attacks the 4 non-attack cases defined by Ray and Ligatti (row 11). SEPTIC had neither false negatives nor positives (rows 35–36) and correctly handled the semantic mismatch problem by detecting all attacks from classes A (rows 17–21), B (7), C (8–9), and D.2–D.4 (26–30).

The other tools can also detect the syntax structure 1st order (row 3), blind SQLI syntax structure (8), and sqlmap (12) attacks, all from class S.1, but not stored procedure (7) and stored injection attacks (26–30). The anti-SQLI tools, from the semantic mismatch attacks detected only the attack from class A.5 (row 21). ModSecurity detected this attack plus 1st order SQLI attacks with encoding and space evasion (A.1 and A.4, rows 17 and 19). Furthermore, ModSecurity cannot detect 2nd order SQLI attacks, because in the second step of these attacks the malicious input comes from the DBMS, not from outside. All tools other than SEPTIC had a few false positives (except DIGLOSSIA) and many false negatives (around 50% of the attacks). This is essentially justified by the non-detection of semantic mismatch attacks and the Ray and Ligatti code cases (row 10) where the injected code does not contain malicious characters recognized by the tools.

Globally ModSecurity and DIGLOSSIA had a similar performance (35 attacks detected). The latter was the best of the four anti-SQLI tools and the only one that detected the syntax mimicry 1st order attack (row 5). ModSecurity does not detect 2nd order attacks, because it just analyses queries reached by user inputs (rows 18 and 20). On the contrary, SQLrand and AMNESIA detect this type of attack. CANDID does not detect either of them. The false positive reported for ModSecurity was case 11 from [27], as the input contained the prime character that is considered malicious by this WAF.

The answer to the first four questions is positive. We conclude that the proposed approach to detected and block injection attacks inside the DBMS is effective because it uses the information given by the DBMS – that processes the queries – without the need of assumptions about how the queries are executed, which is the root of the semantic mismatch problem.

5.1.2 Detection with real software

We used SEPTIC with real web applications to verify if it detects attacks against them – question (5). We evaluated it with five open source PHP web applications: *ZeroCMS*, a content management system [39]; *WebChess*, an application to play chess online [36]; *measureit*, an energy metering application that stores and visualizes voltage and temperature data [21]; *PHP Address Book*, a web-based address/phone book [25]; and *refbase*, a web reference database [28].

Table 5 shows the detection results. The *ZeroCMS* version used contains three SQLI vulnerabilities that appeared in the Common Vulnerabilities and Exposures (CVE) [10] and the Open Source Vulnerability Database (OSVDB) [24]: CVE-2014-4194, CVE-2014-4034 and OSVDB ID 108025. Using *sqlmap*, we performed SQLI attacks to exploit these vulnerabilities and to verify if SEPTIC detected them. SEPTIC successfully detected the attacks and blocked them, protecting the vulnerable web application. Also, we performed attacks against a patched version of *ZeroCMS* and verified that the attacks were no longer successful or detected by SEPTIC.

With *WebChess* and *measureit*, we performed attacks manually and with *sqlmap*. SEPTIC blocked 13 different attacks against *WebChess* and one stored XSS against *measureit*. To confirm the detection, we repeated the attacks with SEPTIC in detection mode (instead of prevention mode), allowing attack detection but without blocking them, and we verified their impact. Also, we confirmed the vulnerabilities explored by these attacks by inspecting the source code with the assistance of identifiers registered in the log file. We recall that our approach detects in runtime attacks and registers

1	Type of attack	N. Tests	SQLrand	AMNESIA	CANDID	DIGLOSSIA	ModSecurity	SEPTIC
2	**SQLI without sanitization and semantic mismatch (S.1, S.2, B, C, D.1)**							
3	Syntax structure 1st order	1	Yes	Yes	Yes	Yes	Yes	Yes
4	Syntax structure 2nd order	1	Yes	Yes	No	No	No	Yes
5	Syntax mimicry 1st order	1	No	No	No	Yes	Yes	Yes
6	Syntax mimicry 2nd order	1	No	No	No	No	No	Yes
7	Stored procedure	1	No	No	No	No	No	Yes
8	Blind SQLI syntax structure	1	Yes	Yes	Yes	Yes	Yes	Yes
9	Blind SQLI syntax mimicry	1	No	No	No	Yes	Yes	Yes
10	Ray & Ligatti code	7	2	3	3	7	2	7
11	Ray & Ligatti non-code	4 (non-attacks)	2	1	2	0	1	0
12	sqlmap project	23	23	23	23	23	23	23
13	Flagged as attack	–	30	30	30	34	30	37
14	False positives	–	2	1	2	0	1	0
15	False negatives	–	9	8	9	3	8	0
16	**SQLI with sanitization and semantic mismatch (S.1, S.2, A.1–A.5, D.1)**							
17	Syntax structure 1st order	4	0	0	0	0	2	4
18	Syntax structure 2nd order	4	0	0	0	0	0	4
19	Syntax mimicry 1st order	4	0	0	0	0	2	4
20	Syntax mimicry 2nd order	4	0	0	0	0	0	4
21	Numeric fields	1	1	1	1	1	1	1
22	Flagged as attack	–	1	1	1	1	5	17
23	False positives	–	0	0	0	0	0	0
24	False negatives	–	16	16	16	16	12	0
25	**Stored injection (D.2–D.4)**							
26	Stored XSS	1	No	No	No	No	No	Yes
27	RFI	1	No	No	No	No	No	Yes
28	LFI	1	No	No	No	No	No	Yes
29	RCI	1	No	No	No	No	No	Yes
30	OSCI	1	No	No	No	No	No	Yes
31	Flagged as attack	–	0	0	0	0	0	5
32	False positives	–	0	0	0	0	0	0
33	False negatives	–	5	5	5	5	5	0
34	**Flagged as attack**	–	31	31	31	35	35	59
35	**False positives**	–	2	1	2	0	1	0
36	**False negatives**	–	30	29	30	24	25	0

Table 4: Detection of attacks with code samples.

Application	SQLI	Stored inj.	Registered
measureit	–	1	–
PHP Address Book	–	–	–
refbase	–	–	–
WebChess	13	–	–
ZeroCMS	3	–	CVE-2014-4194 CVE-2014-4034 OSVDB ID 108025
Total	16	1	3

Table 5: Detection of attacks in real applications.

the source code location of the vulnerabilities explored by attacks when they are detected. SEPTIC does not registered any attack against the *PHP Address Book* and *refbase* applications, meaning that these applications are secure against attacks injection. So these results allow us to answer affirmatively to question (5).

5.2 Performance overhead

To answer question (6), the performance overhead of SEPTIC was evaluated using BenchLab v2.2 [8] with the *PHP Address Book*, *refbase* and *ZeroCMS* applications. BenchLab is a testbed for web application benchmarking. It generates realistic workloads, then replays their traces using real web browsers, while measuring the application performance.

We have set up a network composed of six identical machines: Intel Pentium 4 CPU 2.8 GHz (1-core and 1-thread) with 2 GB of RAM, running Linux Ubuntu 14.04. Two machines played the role of servers: one run the MySQL DBMS with SEPTIC; the other an Apache web server with Zend to run the web applications, and Apache Tomcat to run the BenchLab server. The other four machines were used as client machines, running BenchLab clients and Firefox web browsers to replay workloads previously stored by the BenchLab server, i.e., to issue a sequence of requests to the web application being benchmarked. The BenchLab server has te role of managing the experiments.

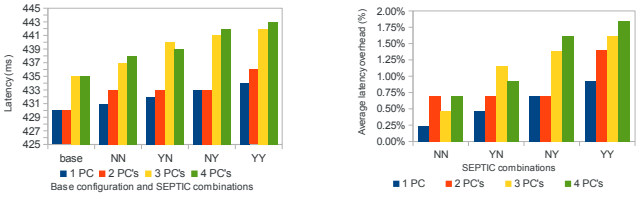
We evaluated SEPTIC with its four combinations of protections turned on and off (SQLI and stored injection on/off) and compared them with the original MySQL without SEPTIC installed (base). For that purpose, we created several scenarios, varying the number of client machines and browsers. The *ZeroCMS* trace was composed of 26 requests to the web application with queries of several types (`SELECT`, `UPDATE`, `INSERT` and `DELETE`). The traces for the other applications were similar but for *PHP Address Book* the trace had 12 requests, while for *refbase* it had 14 requests. All traces involved downloading images, cascading style sheets documents, and other web objects. Each browser executes the traces in a loop many times.

Table 6 summarizes the performance measurements. The main metric assessed was the *latency*, i.e., the time elapsed between the browser starts sending a request and finishes receiving the corresponding reply. For each configuration the table shows the *average latency* and the *average latency overhead* (i.e., the average latency divided by the latency obtained with MySQL without SEPTIC with the same configuration, multiplied by 100 to become percentage). These values are presented as a pair *(latency (ms), overhead (%))* and are shown in the 2nd to 6th columns of the table. The 1st column characterizes the scenario tested, varying the number of client machines (*PCs*) and browsers (*brws*). The latency obtained with MySQL without SEPTIC is shown in the 2nd column and the SEPTIC combinations in the next four. The last two columns show the number of times that each configuration was tested with a trace (*num exps*) and the total number of requests done in these executions (*total reqs*). Each configuration was tested with 5500 trace executions, in a total of 87,200 requests (last row of the table).

The first set of experiments evaluated the overhead of SEPTIC with the *refbase* application (rows 3–6). We run a single Firefox browser in each client machine but varied the number of these machines from 1 to 4. For each additional machine we increase the number of experiments (*num exps*) by 50. Fig. 6 represents graphically these results, showing the latency measurements (a) and the latency overhead of the different SEPTIC configurations (b). The most interesting conclusion taken from the figure is that the overhead of running SEPTIC is very low, always below 2%. Another in-

N. PCs & brws	Base	SEPTIC: SQL injection – stored injection				Num exps	Total reqs
		off–off	on–off	off–on	on–on		
refbase varying the number of PCs, one browser per PC							
1 PC	430, –	431, 0.23	432, 0.47	433, 0.70	434, 0.93	70	980
2 PCs	430, –	433, 0.70	433, 0.70	433, 0.70	436, 1.40	120	1680
3 PCs	435, –	437, 0.46	440, 1.15	441, 1.38	442, 1.61	170	2380
4 PCs	435, –	438, 0.69	439, 0.92	442, 1.61	443, 1.84	220	3080
refbase with four PCs and varying the number of browsers							
8 brws	504, –	506, 0.40	510, 1.19	513, 1.79	516, 2.38	420	5880
12 brws	530, –	532, 0.38	535, 0.94	539, 1.70	544, 2.64	620	8680
16 brws	540, –	541, 0.19	545, 0.93	550, 1.85	553, 2.41	820	11480
20 brws	570, –	573, 0.53	575, 0.88	581, 1.93	584, 2.46	1020	14280
PHP Address Book with four PCs							
20 brws	79, –	79.26, 0.33	79.50, 0.63	80.60, 2.03	81, 2.53	1020	12240
ZeroCMS with four PCs							
20 brws	239, –	240, 0.42	241, 0.84	243, 1.67	245, 2.51	1020	26520
Avg. overhead / Total		*0.41%*	*0.82%*	*1.65%*	*2.24%*	5500	87200

Table 6: Performance overhead of SEPTIC measured with Benchlab for three web applications: *PHP Address Book*, *refbase* and *ZeroCMS*. Latencies in ms, overheads in %.



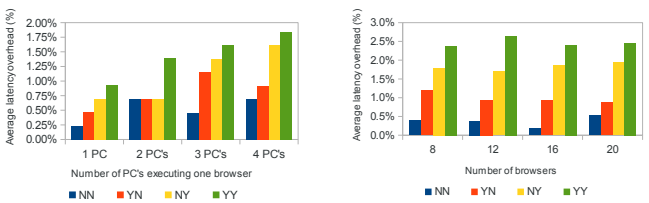
(a) Latency (b) Overhead

Figure 6: Latency and overhead with *refbase* varying the number of PCs, each one with a single browser.

teresting conclusion is that SQLI detection has less overhead than stored injection detection, as the values for configuration NY are just slightly higher than those for YN. Finally, the overhead tends to increase with the number of PCs and browsers generating traffic as the load increases.

The second set of experiments were again with *refbase*, this time with the number of client machines (PCs) set to 4 and varying the number of browsers (Table 6, rows 8–11). Fig. 7 shows how the overhead varies when going from 1 to 4 PCs with one browser each (a) then from 8 browsers (2 per PC) to 20 browsers (5 per PC). The figure allows extracting some of the same conclusions as the first set of experiments. However, they also show that increasing the number of browsers initially increases the overhead (Fig. 7(a)), then stabilizes (b), as neither the CPU at the PCs nor the bandwidth of the network were the performance bottleneck.

The third and fourth sets of experiments used the *PHP Address Book* and *ZeroCMS* web applications and 20 browsers in 4 PCs (Table 6, rows 13 and 15). Fig. 8 shows the overhead of these two applications and *refbase* with the same number of browsers and PCs. The overhead of all applications is similar for each SEPTIC configuration. This is interesting because the applications and their traces have quite different characteristics, which suggests that the overhead imposed by SEPTIC is independent of the server-side language and web application.



(a) Overhead (1 browser per PC) (b) Overhead (varying browsers)

Figure 7: Overhead with *refbase* with 4 PCs and varying the number browsers.

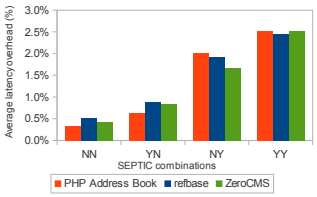


Figure 8: Overhead of SEPTIC with *PHP Address Book*, *refbase* and *ZeroCMS* applications using 20 browsers.

The average of the overheads varied between 0.82% and 2.24% (last row of the table). This seems to be a reasonable overhead, suggesting that SEPTIC is usable in real settings, answering positively question (6).

6. RELATED WORK

There is a vast corpus of research in web application security, so we survey only related runtime protection mechanisms, which is the category in which SEPTIC fits.

All the works we describe have a point in common that makes them quite different from our work: their focus is on *how to do detection or protection*. On the contrary, our work 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.

AMNESIA [13] and CANDID [3] are two of the first works about detecting SQLI by comparing the structure of an SQL query before and after the inclusion of inputs and before the DBMS processes the queries. Both use query models to represent the queries and do detection. AMNESIA creates models by analyzing the source code of the application and extracting the query structure. Then, AMNESIA instruments the source code with calls to a wrapper that compares queries with models and blocks attacks. CANDID also analyses the source code of the application to find database queries, then simulates their execution with benign strings to create the models. On the contrary, SEPTIC does not involve source code analysis or instrumentation. With SEPTIC we aim to make the DBMS protect itself, so both model creation and attack detection are performed inside the DBMS. Moreover, SEPTIC aims to handle the semantic mismatch problem, so it analyses queries just before they are executed, whereas AMNESIA and CANDID do it much earlier. These two tools also cannot detect attacks that do not change the structure of the query (syntax mimicry).

Buehrer et al. [7] 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 [34] 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 [31] 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.

Recently, Masri et al. [20] and Ahuja et al. [2] 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. However, these approaches are not even evaluated experimentally. Again, both works involve instrumenting and modifying the application code, unlike SEPTIC that works inside the DBMS.

Dynamic taint analysis tracks the flow of user inputs in the application and verifies it they reach dangerous instructions. Xu et al. [38] 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). 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 [26] (e.g., if the query structure becomes different due to inputs). WASP does something similar to block SQLI attacks [14]. SEPTIC does not track inputs in the application, but runs in the DBMS.

7. CONCLUSION

The paper explores a new form of protection from attacks against web application databases. It presents the idea of "hacking" the DBMSs to let it protected from SQLI and stored injection attacks. Moreover, by putting protection inside the DBMS, we show that it is possible to detect and block sophisticated attacks, including those related with the semantic mismatch problem. The mechanism was experimented both with synthetic code with vulnerabilities inserted on purpose and with open source PHP web applications. This evaluation suggests that the mechanism can detect and block the attacks it is programmed to handle, performing better that all other tools in the literature and the WAF most used in practice. The performance overhead evaluation shows an impact of around 2.2%, suggesting that our approach can be used in real systems.

Acknowledgments

We thank the anonymous reviewers and our shepherd Anna Squicciarini for their valuable comments. This work was partially supported by the EC through project FP7-607109 (SEGRID), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UID/CEC/50021/2013 (INESC-ID) and UID/CEC/00408/2013 (LaSIGE).

8. REFERENCES

[1] Spring framework, 2014. http://spring.io/.

[2] B. Ahuja, A. Jana, A. Swarnkar, and R. Halder. On preventing SQL injection attacks. *Advanced Computing and Systems for Security*, 395:49–64, 2015.

[3] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: preventing SQL injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 12–24, Oct. 2007.

[4] BBC Technology. Millions of websites hit by Drupal hack attack, Oct. 2014. http://www.bbc.com/news/technology-29846539.

[5] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. IETF Request for Comments: RFC 3986, Jan. 2005.

[6] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security Conference*, pages 292–302, 2004.

[7] G. T. Buehrer, B. W. Weide, and P. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, pages 106–113, Sept. 2005.

[8] E. Cecchet, V. Udayabhanu, T. Wood, and P. Shenoy. Benchlab: An open testbed for realistic benchmarking of web applications. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, 2011.

[9] J. Clarke. *SQL Injection Attacks and Defense*. Syngress, 2009.

[10] CVE. http://cve.mitre.org.

[11] A. Douglen. SQL smuggling or, the attack that wasn't there. Technical report, COMSEC Consulting, Information Security, 2007.

[12] M. Dowd, J. Mcdonald, and J. Schuh. *Art of Software Security Assessment*. Pearson Professional Education, 2006.

[13] W. Halfond and A. Orso. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183, Nov. 2005.

[14] W. Halfond, A. Orso, and P. Manolios. WASP: protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 34(1):65–81, 2008.

[15] M. Howard and D. LeBlanc. *Writing Secure Code for Windows Vista*. Microsoft Press, 1st edition, 2007.

[16] ICS-CERT. Incident response/vulnerability coordination in 2014. ICS-CERT Monitor, Set.-Feb. 2015.

[17] Imperva. Hacker intelligence initiative, monthly trend report #8. Apr. 2012.

[18] JSoup. http://jsoup.org.

[19] M. Koschany. Debian hardening, 2013. https://wiki.debian.org/ Hardening.

[20] W. Masri and S. Sleiman. SQLPIL: SQL injection prevention by input labeling. *Security and Communication Networks*, 8(15):2545–2560, 2015.

[21] Measureit. https://code.google.com/p/measureit/.

[22] I. Medeiros, N. F. Neves, and M. Correia. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the International World Wide Web Conference*, pages 63–74, Apr. 2014.

[23] G. Modelo-Howard, C. Gutierrezand, F. Arshad, S. Bagchi, and Y. Qi. Psigene: Webcrawling to generalize SQL injection signatures. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014.

[24] OSVDB. http://osvdb.org.

[25] PHP Address Book. http://php-addressbook.sourceforge.net.

[26] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, pages 124–145, 2005.

[27] D. Ray and J. Ligatti. Defining code-injection attacks. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–190, 2012.

[28] refbase. http://http://www.refbase.net.

[29] Search Security TechTarget. Wordpress vulnerable to stored XSS, Apr. 2015. http://searchsecurity.techtarget.com/news/4500245137/WordPress-vulnerable-to-stored-XSS-researchers-find.

[30] SolidIT. DB-Engines Ranking. http://db-engines.com/en/ranking, accessed Aug. 10th, 2015.

[31] S. Son, K. S. McKinley, and V. Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, pages 1181–1192, 2013.

[32] Spring. http://docs.spring.io/spring/docs/2.5.4/reference/aop.html.

[33] sqlmap. https://github.com/sqlmapproject/testenv/tree/master/mysql.

[34] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, Jan. 2006.

[35] Trustwave SpiderLabs. ModSecurity Open Source Web Application Firewall. http://www.modsecurity.org.

[36] WebChess. http://sourceforge.net/projects/webchess/.

[37] J. Williams and D. Wichers. OWASP Top 10: The ten most critical web application security risks. Technical report, OWASP Foundation, 2013.

[38] W. Xu, S. Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, 2005.

[39] ZeroCMS. Content management system built using PHP and MySQL. http://www.aas9.in/zerocms/.