Demonstrating a Tool for Injection Attack Prevention in MySQL

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

***Abstract*—Despite the significant efforts put in building more secure web applications, cases of high impact breaches continue to appear. Vulnerabilities in web applications are often created due to inconsistencies in the way SQL queries are believed to be run and the way they are actually executed by a Database Management System (DBMS). This paper presents a demonstration of SEPTIC, a mechanism that detects and blocks injection attacks inside the DBMS. The demonstration considers a scenario of a non-trivial PHP web application, backed by a MySQL DBMS, which was modified to include SEPTIC. It presents how SEPTIC blocks injection attacks without compromising the application correctness and performance. In addition, SEPTIC is compared to alternative approaches, such as sanitizations carried out with standard functions provided language and a web application firewall.**

Keywords*—*Web applications; injection attacks; DBMS; software security; runtime protection; security.

I. Introduction

Despite all effort put in building more secure *web applications*, cases of high impact breaches continue to appear. In fact, the number of web application attacks increased by 26% in the first quarter of 2016 [21], 87% of which were *SQL injection attacks* (SQLI). SQLI attacks are performed with different aims, often for extracting information from databases. For example, recently, they may have been used to steal 11.5 million sensitive documents from a notorious law firm, causing much embarrassment to many people [8]. They have also been employed in multistage attacks to critical infrastructures, namely to alter the levels of chemicals added to drinking water in a water treatment facility [22].

Defending web applications from SQLI attacks has always been an important challenge. There are two approaches that are most common. In the first – *sanitization of user inputs* – developers insert in the code calls to sanitization functions provided by the language (e.g., function `mysql_real_escape_string` in PHP) or by third party libraries to process the inputs before they are included in a query that is susceptible to exploitation (e.g., a query executed by `mysql_query`). In the second – *use of protection components* – systems administrators install *web application firewalls* (WAFs) or *application delivery controllers* (ADCs) [23], [9] operating between the browser and the application, filtering all user inputs supplied to the application and blocking those that are considered suspicious, or *SQL proxies* or *database firewalls* [5], operating between the application and the Database Management System (DBMS), filtering the queries. There are also other alternatives, such as mechanisms that first analyse or modify the source code of the web applications, then block injections in runtime [2], [3], [6], [20], [10].

Although these solutions contribute to improve web security, there are many forms of injection attacks for which it is difficult to provide comprehensive defences. In particular, these flaws often arise due to a *semantic mismatch*, i.e., a gap between the way SQL queries are believed by developers to be processed and the way they are actually executed by databases, leading to subtle bugs. 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, meaning that the same problem can be present in other business applications. In fact, any class of applications that use a database as backend may be vulnerable to injection attacks.

In this paper, we present a demonstration of a novel mechanism that blocks injection attacks *inside the DBMS*, protecting any application that uses the database. By running the mechanism inside the DBMS, the defense is provided off-the-shelf (without requiring installation), which may lead to automatic protection similarly to what is currently available for binary programs based on techniques like address space layout randomization and canaries [17].

This mechanism is called *SElf-Protecting daTabases preventIng attaCks* (SEPTIC) [11]. It focuses on the main categories of attacks related with databases: SQLI attacks that continue to be among those with highest risk and for which new variants continue to appear; and stored injection attacks that also involve SQL queries. For SQLI, we identify the 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 [3], [6] and circumventing the semantic mismatch. For stored injection, we resort to plugins that are executed on the fly to deal with specific attacks before data is inserted in the database.

SEPTIC is demonstrated with a non-trivial web application backed by a database, which corresponds to a common deployment scenario. In particular, the demonstration is based based in the following elements: a web application programmed in PHP, the language most used in development of this sort of programs; the Apache web server, also one of the most commonly utilized web servers; and a MySQL

backend database, which is probably the most popular open-source DBMS. The demonstration illustrates how SEPTIC blocks injection attacks without compromising the application correctness and performance. The main objectives are:

1) To show example attacks that can take advantage of the semantic mismatch to circumvent protection mechanisms, such as those implemented in the application by resorting to PHP sanitization functions and protection components such as the popular ModSecurity WAF [23];
2) To present the operation of the SEPTIC mechanism implemented inside MySQL to prevent injection attacks, solving the semantic mismatch problem and blocking attacks that attempt to exploit it;
3) To show that SEPTIC is effective and more accurate than other commonly employed mechanisms, and that applications are no longer compromised when SEPTIC is in use.

The paper is organized as follows. Section II presents an overview of the SEPTIC mechanism, including its features and modules, detection examples and impact caused in the MySQL performance. Section III presents the scenario, which corresponds to an usual interaction between web applications and databases. This section also explains attacks that exploit flaws originating from the semantic mismatch. Section IV presents the phases of the demonstration based on the scenario. The paper ends with conclusions and discussion in Section V.

II. SEPTIC MECHANISM

This section presents the SEPTIC mechanism, giving a general overview of its functionality and features. There is a description of the four modules that compose it, together with the operation modes and actions. Some examples of attack detection are included and there is a discussion on the performance impact in MySQL.

A. Overview

SEPTIC operates inside of the DBMS to detect and block injection attacks at runtime. Normally, prior to the execution of a query transmitted by an application, the DBMS parses and validates the received information. SEPTIC runs right before the execution step, after all potential modifications have been applied to the queries. It analyzes the queries to determine if they are malicious, flagging them as attacks and then possibly stopping their processing.

SEPTIC detects SQLI and stored injection attacks. To find SQLI attacks, it compares the structure of the query currently being processed with a query model previously learned. If the structure does not match the model, then this indicates that the query was somehow changed, maybe to cause some (malicious) unexpected behavior. For stored injection attacks, SEPTIC applies plugins to check if the user inputs provided to `INSERT` and `UPDATE` commands are erroneous. In the current implementation there are plugins capable of discovering the following classes of attacks: stored (persistent) XSS, remote and local file inclusion (RFI and LFI), and OS and remote command execution (OSCI and RCE).

SEPTIC has two main modes of operation: (1) *training mode* (or learning mode), to learn the query models of the queries issued by the application; (2) *normal mode*, to find, block, and log attacks. The normal mode can be set up as being detection(-only) or prevention. In the detection mode, the attacks are not blocked but only logged; in prevention mode, they are both stopped and logged. The natural order of using SEPTIC is to first run in learning mode and then later on put it in normal mode.

SEPTIC is composed of 4 modules, as shown in Figure 1 and explained in more detail in Section II-C.

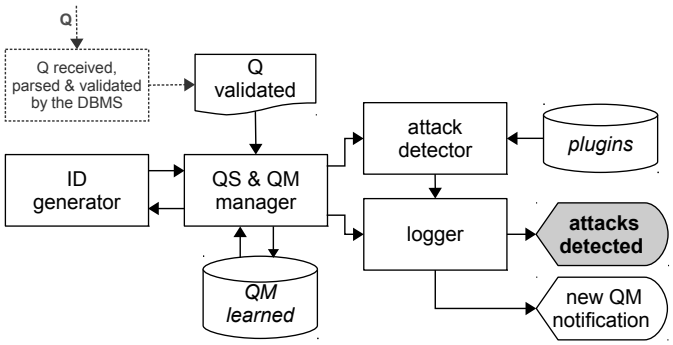

Fig. 1: Overview of the SEPTIC modules and data flow.

Before being executed in normal mode, SEPTIC has to be trained in order to learn the query models of the queries issued by application. In *training mode* only the *QS&QM manager* and *ID generator* modules are used. For each query, the first component receives a validated query (*Q validated*) by the DBMS and creates its model, whereas the second component produces its ID. Then, the query model and the ID are stored in the *QM learned* store.

When the mechanism is in *normal mode*, the *QS&QM manager* receives a validated query, extracts its query structure (QS), and requests a query identifier (ID) from the *ID generator*. Then, it obtains the query model (QM) associated to that ID. Next, the *attack detector* module looks for SQLI and stored injection attacks, by comparing the QS with the QM and by applying plugins, respectively. The attacks are registered by the *logger* module. In case SEPTIC does not know a QM for that ID, the *QS&QM manager* creates a QM, associates the QM with that ID, and stores it in the *QM learned* store. The *logger* also saves information that a new model was created.

SEPTIC was developed in C. In our prototype, a few lines of code were added to a single MySQL file, and the rest of the code was included in two new files that were linked with the rest of the DBMS.

B. Features

SEPTIC has a set of important features, which we present by comparing them with other mechanisms used to detect SQLI attacks. Notice that none of these mechanism operate inside the DBMS, but typically between the application and the DBMS.

- *Server-side language independence* – SEPTIC requires minimal and optional support at server-side

language engine (SSLE) level to obtain the external identifiers (unlike [18], [14], [7], [25]);

- *No client configuration* – the DBMS client connectors do not need reconfiguration to use SEPTIC, as it is inside the DBMS;
- *Client diversity* – several DBMS clients of different types may be connected to a single DBMS server with SEPTIC;
- *No application source code modification* – the programmer does not need to make changes to the web application source code to use the mechanism (unlike [6], [2], [3], [20], [25]);
- *No application source code analysis* – SEPTIC does not need to do source code analysis to find the queries in the source code of the web application (unlike [6], [1]);
- *Learning* – SEPTIC is able to learn the query models in training mode or incrementally in normal mode (see Section II-E). Similarly to GreenSQL [5] and Percona Tools [12], it needs information on the queries that it will be monitored to be able to detect attacks. However, SEPTIC has two different ways of learning that information, whereas these systems have only one (during a training phase).

C. Modules

SEPTIC contains four main modules, which are presented in the following paragraphs:

1) QS&QM manager: MySQL parses and validates a query, storing the query elements in a stack data structure. SEPTIC receives this structure and creates another stack with that data, the *query structure* (QS). Each node of the stack represents a query element belonging to a category (e.g., field, function, operator) and information about it – data type (e.g., integer, string) and data (e.g., user inputs). Each node of the stack has one of the following formats: ⟨ELEM_TYPE, ELEM_DATA⟩ or ⟨DATA_TYPE, DATA⟩.

Figure 2(a) depicts as an example the QS for the query `SELECT * FROM tickets WHERE reservID = 'ID34FG' AND creditCard = 1234`. This query returns all data associated with a flight ticket, after an user provided the ticket reservation ID and the last four digits of the credit card number. The figure shows from bottom to top the SQL clauses and its elements. As we can observe, each element (a line in the figure) is represented either as ELEM_TYPE or as DATA_TYPE.

After the QS is built, the QS&QM manager requests from the ID generator module an identifier for the query (ID). Next, it searches the *QM learned* store for a query model (QM) with the same identifier. If the QM is found, the QS and the QM are sent to the *attack detector* module and the query processing continues from there. Otherwise, the QM of the query is built from the QS, and is associated with the previously created ID. Finally, it is stored in the QM Learned store.

To create the QM from the QS the following operation is performed: DATA information, in all ⟨DATA_TYPE, DATA⟩ nodes of the QS, are replaced by a special value ⊥. Figure 2(b) shows the QM for the query presented above, where we can observe these substitutions.

COND_ITEM	AND
FUNC_ITEM	=
INT_ITEM	**1234**
FIELD_ITEM	creditCard
FUNC_ITEM	=
STRING_ITEM	**ID34FG**
FIELD_ITEM	reservID
SELECT_FIELD	*
FROM_TABLE	tickets

(a) Query structure (QS)

COND_ITEM	AND
FUNC_ITEM	=
INT_ITEM	⊥
FIELD_ITEM	creditCard
FUNC_ITEM	=
STRING_ITEM	⊥
FIELD_ITEM	reservID
SELECT_FIELD	*
FROM_TABLE	tickets

(b) Query model (QM)

Fig. 2: QS and QM of a query.

2) ID generator: SEPTIC is first trained to learn the query models of the queries issued by the application, associating to each a unique ID. Later on, in normal mode, for each query received, it is necessary to compute a query identifier and look (in QM Learned store) if there is a query model with this ID.

The ID is formed by composing distinct types of information related to the query. We call identifiers to these information types. One of these identifiers may be (optionally) provided by the application or SSLE (e.g., PHP Zend) and the other is (mandatorily) created by SEPTIC. The first identifier can take an arbitrary value defined by the programmer. It is sent to MySQL inside a comment (i.e., `/* external identifier */`) that is concatenated with the query. The second identifier is produced by SEPTIC based on the QM in order to ensure uniqueness.

The ID generator receives a request from the QS&QM manager module. First, it determines if the query comes with an identifier (an external identifier), which is then retrieved. Second, it creates its own (internal) identifier. The ID is the concatenation of both identifiers (or just the internal identifier in case the other does not exist).

3) Attack detector: This module is executed only during the normal mode of operation. It performs two kinds of attack discovery, namely SQLI and stored injection detection.

- SQLI detection – is implemented by comparing the query structure with the query model. The module executes an algorithm with two steps: (1) it verifies if the number of nodes of QS and QM are equal; (2) checks, for each node of QS, if its element is equal to the corresponding node in the QM. Step (2) is only carried out if step (1) does not fail. An attack is detected if any of these steps fails. In such case, the *logger* module is triggered. Otherwise, the query is delivered to MySQL to be executed.
- Stored injection detection – is performed for `INSERT` and `UPDATE` commands. The module executes two steps per query: (1) a lightweight checking of the user input is done to determine if it contains characters associated with malicious actions (e.g., `'<'` and `'>'` for stored XSS), which are then used to find out the

potential type of attack; (2) a more precise validation is run, tailored to confirm with higher certainty the attack. The second step is executed only if the first flags problems. If an attack is detected, the *logger* module is activated; otherwise, the query execution proceeds.

4) Logger: This module is used to register the events observed by SEPTIC, namely an attack being discovered or a new query reaches the database (in which case there is no query model). The module is activated by the attack detector and QS&QM manager modules.

An attack record contains the query received by MySQL, the query identifier, its query model, and the step of the algorithm that found the problem (these two last items are registered for a SQLI attack). For a new observed query, the logger registers the received query, the query model and its query identifier.

D. Attack detection examples

This section presents two detection examples to illustrate the process.

1) SQLI attack detection: Consider the query `SELECT * FROM tickets WHERE reservID = ? AND creditCard = ?`, based on the example introduced previously. It accepts two inputs represented by a question mark. The corresponding query model is shown in Figure 2(b). Consider a second-order SQLI attack: (1) a malicious user provides an input that leads the application to insert `concat(ID34FG,U+02BC-- )` in the database, i.e., `ID34FG'--` with the prime represented in Unicode as `U+02BC`; (2) later this data is retrieved from the database and inserted in the `reservID` field in the query above, resulting in the query `SELECT * FROM tickets WHERE reservID = concat(ID34FG,U+02BC-- ) AND creditCard = 0`; (3) MySQL parses and validates the query, decoding `U+02BC` into a prime, and the resulting query becomes `SELECT * FROM tickets WHERE reservID=ID34FG`. This attack modifies the structure of the query. Figure 3 presents the QS for this query. When the query is issued, SEPTIC compares the QS with the QM during structural verification (first step). This comparison shows that they do not match as the number of nodes is different, detecting the attack.

FUNC_ITEM	=
STRING_ITEM	**ID34FG**
FIELD_ITEM	reservID
SELECT_FIELD	*
FROM_TABLE	tickets

Fig. 3: QS of query `SELECT * FROM tickets WHERE reservID = ? AND creditCard = ?` with `ID34FG'--` as reservID.

As a second example consider a syntax mimicry attack, i.e., an attack that reproduces the structure of the original query. The attacks is against the query above and the malicious input `ID34FG' AND 1=1--` is inserted in the `reservID` field. The resulting query is `SELECT * FROM tickets WHERE reservID=ID34FG AND 1=1`. Figure 4 represents the query structure of this query. When the query is issued, SEPTIC compares QS with QM (Figure 4 with Figure 2(b)). First, it checks that they match, as the number of items of both structures is equal; then, it observes that the ⟨INT_ITEM, 1⟩ node from QS (fourth row in Figure 4) does not match with the ⟨FIELD_ITEM, CREDITCARD⟩ node from QM (fourth row in Figure 2(b)). The attack is flagged due to this difference.

COND_ITEM	AND
FUNC_ITEM	=
INT_ITEM	**1**
INT_ITEM	**1**
FUNC_ITEM	=
STRING_ITEM	**ID34FG**
FIELD_ITEM	reservID
SELECT_FIELD	*
FROM_TABLE	tickets

Fig. 4: QS of query with `ID34FG' AND 1=1` input as reservID.

2) Stored XSS attack detection: Consider a web application that registers new users. Consider also that a malicious user inserts as his first name `<script> alert(''Hello!'');</script>`, which is a JavaScript code. When SEPTIC receives the query, it does the filtering step (first step) and finds two characters associated with XSS, `'<'` and `'>'`. So, it calls the plugin that detects stored XSS attacks. This plugin inserts this input in a web page and calls an HTML parser. Then, it finds that the input contains a script and flags a stored XSS attack.

E. Operation modes and actions

As mentioned above, before performing the detection of attacks, SEPTIC has to learn query models of the queries that might be called in the application. Therefore, SEPTIC has two main operation modes: training mode and normal mode.

- Training mode – for each different query that SEPTIC receives, the *QS&QM manager* and *ID generator* modules work together to create the query model, the query identifier, and associate them. Then, the QS&QM manager stores the query model in the *QM Learned* store.
  There are several options to trigger the queries. This can be made with application unit tests, manually by the programmer/administrator, or by using the *septic_training* module. This module runs externally to SEPTIC and currently supports normal web applications. It works like a crawler, navigating in the application looking for forms, to then inject benign inputs that eventually are inserted in queries transmitted to MySQL.
- Normal mode – in which SEPTIC can operate in detection or prevention mode. The difference between

	Query model			Attack detection			Query	
	T	I	Log	SQLI	Stored Inj	Log	Drop	Exec
Training	x							x
Prevention		x	x	x	x	x	x	
Detention		x	x	x	x	x		x

T: training I: incremental

TABLE I: Operation modes and actions taken by SEPTIC.

these two modes resides in executing or not the query when an attack is found. In the former, the query is executed, whereas for the latter it is not, dropping the query and blocking the attack. In both modes the attack is logged.

For each query that SEPTIC receives, the *QS&QM manager* and *ID generator* modules work together to create the query structure and get the query identifier. Next, the *QS&QM manager* gets the query model identified by the query identifier. If a query model exists, then the *attack detector* is activated to perform the SQLI and stored injection detection algorithms described above, and *logger* acts if an attack is found. If a query model does not exist for that query identifier, the *QS&QM manager* creates the query model, stores it with the query identifier, and the *logger* registers this event. This action corresponds to an *incremental* training because the query models are learned and stored gradually. Later, the programmer/administrator will have to decide if the query model comes from a malicious or a benign query. In the latter case, it is saved together with the other QM already known.

Table I summarizes the operation modes and the actions taken by SEPTIC. The last two rows show the prevention and detection modes included in the normal operation mode. As one can observe in the last two columns for these modes, the difference between them is the execution or not of queries when attacks are flagged.

F. Performance impact in MySQL

We studied the impact of the SEPTIC in MySQL when setup in the four configurations of detection, i.e., turning on and off the detection of SQLI and stored injection attacks. We performed these evaluations using three real web applications, namely *PHP Address Book* [13], *refbase* [16], and *ZeroCMS* [26]. To automate the experiments, we resorted to the *BenchLab*, a benchmarking testbed for web applications [4].

The experimental environment was based on a network with six machines with identical characteristics (Intel Pentium 4 CPU 2.8 GHz (1-core and 1-thread) with 2 GB of RAM, 80 GB of hard disk SCSI, and 1 Gb ethernet card, running Linux Ubuntu 14.04). These machines belong to a computational cluster dedicated to large-scale experiments of distributed systems, the Quinta cluster [15]. The Quinta is comprised of 38 physical machines aggregated in four different clusters, each one composed by identical machines. Our experiments were realized in the R cluster, which contains eleven machines. From the six computers that we used from cluster R, two of them performed the server roles, whereas the other four were the clients. A server machine contained the MySQL DBMS with the SEPTIC mechanism installed, and the other machine had installed the Apache web server and the PHP Zend to run those three web applications. The Apache Tomcat was also necessary to run the BenchLab server. The four client machines had installed the Firefox web browser and the BenchLab client. They run workloads previously recorded and stored by the BenchLab server, i.e., a sequence of requests made to the web applications. These requests forced the web applications to execute of queries in MySQL.

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. For that purpose, we created several scenarios, varying the number of client machines (1 to 4) and browsers (1 to 5 browsers per machine). We also created three workloads from the web applications, one of each application. The ZeroCMS workload had 26 requests to the web application with queries of several types (SELECT, UPDATE, INSERT and DELETE) and downloading of web objects (e.g., images, css). The other two application workloads were similar but for PHP Address Book it had 12 requests, while for refbase it had 14 requests.

The evaluation started with one machine running one browser executing the refbase workload, next we gradually increased the number of machines (one by one) running one browser. On a second phase, we evaluated the same workload with four machines running two browsers each one (8 in the total), then we incremented to 12, 16 and 20 browsers. The last two batteries of experiments, we had all machines with all browsers running the other two workloads, respectively. On all experiments, each browser executed the workload in a loop many times, sending the requests one by one.

Figure 5 depicts the results of the experimental evaluation for the three web applications and 20 browsers in 4 machines. The figure shows how the average latency overhead varied from 0.5% to 2.2%, depending if SEPTIC was configured with both detections disabled (NN) or both enabled (YY). The figure also shows that the overhead of all applications is similar for each SEPTIC configuration. With SEPTIC set up to detect only SQLI (YN), the overhead was only 0.8%. These values suggest that it is feasible to run SEPTIC by default inside MySQL as there is a very limited impact on performance.

III. Application Scenario

We consider the *WaspMon* web application [24]. It is a real open source web application that can manage the energy consumption in devices (e.g., of a household or a factory). This sort of application is included in typical smart power grid scenarios. For serious forms of vulnerabilities, it can cause problems not only to the owner of the devices but in extreme cases could create power imbalances in the grid.

In our scenario, the web application is programmed in PHP and runs in an Apache web server with Zend and employs a

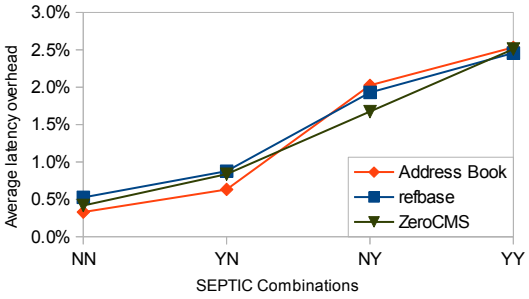

Fig. 5: Overhead of SEPTIC with the applications *PHP Address Book*, *refbase* and *ZeroCMS*.

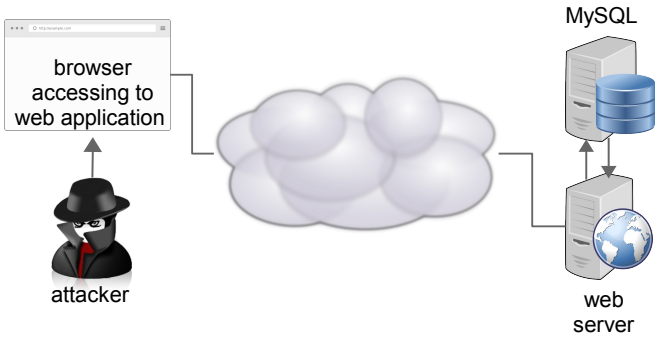

Fig. 6: Attack to the web application and database.

background database managed by the MySQL DBMS. The application allows the users to insert and retrieve data with forms in web pages. These forms lead to reads and writes in the background database by invoking SQL queries. The web application is accessed through a web browser and is supposed to be utilized by different users from distinct locations.

The main supported queries allow the insertion in the database of data collected from the devices and to later read this data for the user to check it. This supports for instance the tracking of the device history, and eventually take some action on it (e.g., disconnect the device, re-schedule the device for a new data collect).

The programmer was careful and used PHP sanitization functions (e.g., `mysql_real_escape_string`) to check all inputs before inserting them in queries. Therefore, this web application is apparently protected from the attacks we aim to demonstrate. In the demonstration we also consider as an alternative layer of protection the *ModSecurity* WAF [23] (version 2.9.1, configured with OWASP Core Rule Set (CRS) 3.0). ModSecurity is the most popular WAF, widely adopted by industry. It is integrated in the Apache web server and checks the requests incoming from the browsers to the web server before they reach the web application(s).

A. Attack scenario.

The attack scenario is depicted in Figure 6. The web application is accessed by an user through a browser from some place using a network connection. The web server receives HTTP/HTTPS requests and sends them to the application. The application uses the user inputs that come in the requests, includes them in queries, and requests the execution of queries to the MySQL DBMS. MySQL processes the queries and returns the results to the application, which forwards them to the browsers/users.

An attacker will scan the application, looking for entry points in forms, to later inject malicious inputs, i.e., making injection attacks in an attempt to compromise the application.

We can consider two kinds of vulnerabilities in the application, depending on the sanitization (or not) of user inputs included in the queries. An attacker will easily exploit vulnerabilities associated to queries that use unsanitized user inputs, performing SQLI attacks and the first step of stored attacks (storing of the malicious inputs in the database). The second kind of vulnerabilities are more troublesome because the user inputs are sanitized, i.e., the application is supposedly secure. Also, the application can be behind of a protection mechanism such as ModSecurity. The attacker injects inputs in a way that circumvents these security measures. These cases are related with the exploitation of the *semantic mismatch problem*. These attacks are not easily detected, passing unnoticed by the protection mechanisms.

In the demonstration we consider only these cases of injection attacks – when protections are in place – since the first ones (unsanitized inputs) are commonly known. The activation of SEPTIC inside of the MySQL will block these attacks, thus defending the application, i.e., the application is safeguarded from the injection attacks.

IV. The Demonstration

The demonstration is based on the application scenario described in the previous section. The setup is presented in Figure 7. It involves two computers and one Ethernet switch. One of the computers represents the web and DBMS servers, containing two virtual machines, one per server. The other computer takes the role of a client. In this way, the computers represent the following entities: MySQL DBMS server, including the SEPTIC mechanism (1 virtual machine); Apache web server with ModSecurity, Zend engine, and the web application (1 virtual machine); a browser to access the web application and other tools to perform SQLI attacks, such as *sqlmap* tool [19] (probably, the most used tool for testing web applications against SQLI vulnerabilities and used by hackers and professionals) (1 machine). The displays from SEPTIC and ModSecurity are used to show the events related to these two protection mechanisms. For SEPTIC, we developed a register of events that logs all actions taken by the mechanism, such as query model creation, query processing, and attack detection. It was inserted in the *logger* module.

The demonstration has five phases. The first shows the exploitation of the semantic mismatch problem, although the application is protected with the sanitization functions it includes out-of-the-box. In the second phase, the protection of the application is enhanced with the ModSecurity WAF. The next three phases are dedicated to SEPTIC, from its training to attack detection.

A. Attacks with sanitization function protection

In the first phase there are no external protection mechanisms enabled. The application is only protected by the

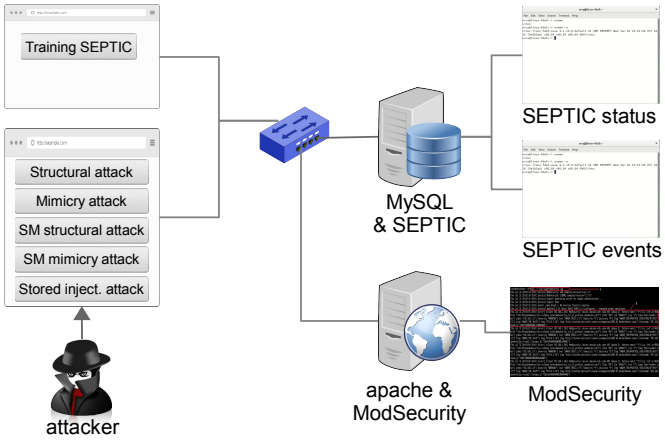

Fig. 7: The setup of the demonstration, including example displays.

sanitization functions from the PHP language. We illustrate that the application contains vulnerabilities – is attackable – even with its entry points sanitized.

The attacker uses the browser and/or the sqlmap tool to access the application, and inserts erroneous data that reaches the queries. Then these queries are sent to MySQL and are executed. The results of the attacks are showed in the browser and/or console, respectively, as the response of the database and application.

B. Attacks with additional ModSecurity protection

In the second phase, we activate ModSecurity to show if it can improve the protection of the application. After turning on ModSecurity and restarting the web server, the attacks performed in the previous phase are executed again. Some those that were previously successful are blocked by ModSecurity, whereas others are not, corresponding to ModSecurity false negatives. This is observed in two ways: the blocked attacks appear in the ModSecurity log, whereas the effect of those that pass the protection mechanism have their outcome shown in the browser. At the end of these experiments, ModSecurity is disabled and the web server restarted.

C. Training SEPTIC

In this third phase, SEPTIC is configured for training mode operation, the MySQL server is restarted to assume this configuration, and the SEPTIC status display notifies that SEPTIC operates now in that mode.

We illustrate the training of SEPTIC by inserting benign inputs in application forms in the browser. These inputs reach existing queries in the application and the queries are sent to MySQL. SEPTIC, for each new query received, creates its query model and stores it with a query identifier. SEPTIC events display logs with the addition of the query models for each new query. We also illustrate that for a query processed twice by SEPTIC, the query model is created and stored only once. This happens because SEPTIC only creates query models when it does not have stored any model with the produced query identifier. This is shown in the SEPTIC events display, which does not notify any query model addition the second time. All query models are in memory and are stored persistently. Also, after the query models are built, the queries are executed by MySQL as expected (with benign inputs).

D. SEPTIC protection

In this phase SEPTIC is set to normal mode, more concretely to prevention mode (blocks and logs attacks). Then, MySQL is restarted to assume the new configuration, the persistent query models are loaded and SEPTIC status display notifies this change.

Using the same injection attacks performed in the firsts two phases, we illustrate that SEPTIC detects and blocks all of them (i.e., no false negatives). The actions taken by SEPTIC are registered in the events register. It registers: query structure construction, query identifier generation, query model discovery, comparison of both structures, attacks detected, and the type of attack (SQLI or stored injection). For the SQLI attacks, it also logs if they are structural or syntactical, i.e., in which step of the SQLI detection algorithm discovered the attack.

When SEPTIC flags an attack, we observe that the attack is blocked, the query is dropped and its execution is stopped in MySQL. This action is visible in the browser. Moreover, we show that the injection of benign inputs does not break any step of the detection algorithms used by SEPTIC, meaning that queries are executed as expected and SEPTIC does not interfere with the normal processing inside of MySQL (i.e., no false positives).

E. ModSecurity versus SEPTIC

Finally, in this last phase we filter the results of both external protection mechanisms, looking for the attacks detected by both. We observe that ModSecurity does not protect the application from all injected attacks. For SEPTIC we observe that all attacks are detected and no false positives are reported.

V. Conclusions and Discussion

The demonstration described in this paper illustrates, in the first place, how injection attacks can compromise an application developed following secure coding best practices (sanitization of entry points before they reach a sensitive sink) and pass barriers of protection that are put before the user inputs reach the application (web application firewalls). These attacks have the aim of inserting or retrieving data to/from the application database. We consider a specific class of attacks that exploit semantic mismatch flaws, which appear due to a gap on the way SQL queries are believed to be run and the way they are actually executed by databases. For this purpose, we use a non-trivial PHP web application implementing entry point sanitization and ModSecurity.

In second place, the demonstration shows the protection of the web application using our own mechanism, SEPTIC [11]. This mechanism runs inside of the DBMS, which is MySQL in its first implementation. Running SEPTIC inside the DBMS allows it to handle queries just before execution. Therefore, adding SEPTIC to MySQL mitigates the semantic mismatch problem.

In the original paper about SEPTIC we report an experimental comparison between SEPTIC and several other protection mechanisms, showing that SEPTIC provides better protection [11]. Moreover, we also show that its overhead is low.

ACKNOWLEDGMENT

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).

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] GreenSQL. http://www.greensql.net.

[6] 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.

[7] 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.

[8] info security. SQL Injection Flaw Found in Mossack Fonseca CMS, Apr. 2016. http://www.infosecurity-magazine.com/news/sql-injection-flaw-mossack-fonseca/.

[9] A. Lerner, J. Skorupa, and D. Ciscato. Gartner Inc. Magic quadrant for application delivery controllers. 2016.

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

[11] I. Medeiros, M. Beatriz, N. F. Neves, and M. Correia. Hacking the DBMS to prevent injection attacks. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 295–306, Mar. 2016.

[12] Percona toolkit. https://www.percona.com.

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

[14] 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.

[15] Quinta. http://www.navigators.di.fc.ul.pt/wiki/Quinta.

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

[17] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 574–588, 2013.

[18] 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.

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

[20] 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.

[21] A. Technologies. Q1 2016 state of the internet / security report. June 2016.

[22] The Register. Water treatment plant hacked, chemical mix changed for tap supplies, Mar. 2016. http://www.theregister.co.uk/2016/03/24/water_utility_hacked/.

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

[24] WaspMon. https://github.com/azping/WaspMon.

[25] 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.

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