You shall not pass: Mitigating SQL Injection Attacks on Legacy Web Applications

Rasoul Jahanshahi
rasoulj@bu.edu
Boston University

Adam Doupé
doupe@asu.edu
Arizona State University

Manuel Egele
megele@bu.edu
Boston University

ABSTRACT

SQL injection (SQLi) attacks pose a significant threat to the security of web applications. Existing approaches do not support object-oriented programming that renders these approaches unable to protect the real-world web apps such as Wordpress, Joomla, or Drupal against SQLi attacks.

We propose a novel hybrid static-dynamic analysis for PHP web applications that limits each PHP function for accessing the database. Our tool, SQLBlock, reduces the attack surface of the vulnerable PHP functions in a web application to a set of query descriptors that demonstrate the benign functionality of the PHP function.

We implement SQLBlock as a plugin for MySQL and PHP. Our approach does not require any modification to the web app. We evaluate SQLBlock on 11 SQLi vulnerabilities in Wordpress, Joomla, Drupal, Magento, and their plugins. We demonstrate that SQLBlock successfully prevents all 11 SQLi exploits with negligible performance overhead (i.e., a maximum of 3% on a heavily-loaded web server).

KEYWORDS

Network Security; Database; SQL Injection; Web Application

ACM Reference Format:
Rasoul Jahanshahi, Adam Doupé, and Manuel Egele. 2020. You shall not pass: Mitigating SQL Injection Attacks on Legacy Web Applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS '20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3320269.3384760

1 INTRODUCTION

The growing number of users for services such as social networks, news, online stores, and financial services makes these services a tempting source of sensitive information for attackers. Symantec's recent report [7], shows an increase of 56% in web attacks from 2017 to 2018. Moreover, according to Akamai [1], 65.1% of web attacks were SQLi attacks. SQLi is a type of code injection attack, where an attacker aims to execute arbitrary SQL queries on a database. In 2018, the number of SQLi vulnerabilities discovered in the top four most popular web apps (i.e., Wordpress, Joomla, Drupal, and Magento) increased by 267% compared to the prior year.

There has been a great deal of research into identifying SQLi vulnerabilities and defending against SQLi attacks on web apps. Proposed approaches used various techniques such as static analysis [10, 11, 20, 22, 35], dynamic analysis [3, 6, 15, 21, 24, 25, 37], or a mix of static-dynamic analysis [5, 17, 28]. While static analysis approaches can be promising, static analysis cannot determine whether input sanitization is performed correctly or not [34]. If the sanitization function does not properly sanitize user-input, SQLi attacks can still happen. Moreover, to the best of our knowledge, prior static analysis approaches for finding SQLi vulnerabilities in PHP web apps do not support Object-oriented programming (OOP) code. Such shortcomings in static analyses leave SQLi vulnerabilities undetected in web apps such as Wordpress, Joomla, and Drupal that more than 40% of active websites use [29].

Prior dynamic analyses use taint analysis [15, 18] and comparison of query parse trees [3, 6, 25, 26, 36, 37] for detecting SQLi attacks on web apps. Such dynamic analyses follow an incomplete definition of SQLi attacks where a SQLi attack always alters the syntactic structure of an SQL query. Ray et al. [31] show that this incomplete definition in CANDID [3], SQLCheck [36], WASP [15], and SQLPrevent [37] does not prevent specific SQLi attacks and also blocks benign requests. Other dynamic approaches have attempted to create a profile of the executed SQL queries and enforce the profile at runtime [25, 26]. The profiles are a mapping between the parse tree of the benign issued SQL queries and the PHP functions that issued the queries. The profiles created by such approaches are too coarse-grained. Particularly, modern and complex web apps such as Drupal and Joomla define database APIs that perform all database operations. Database APIs create SQL queries using the principle of the encapsulation that allows the local functions to issue an SQL query to the database without passing the SQL query as an argument. In such cases, existing approaches map SQL queries to functions in the database APIs instead of mapping to the function that uses database API for communicating with the database. Hence, the prior approaches create a coarse-grained mapping that can allow an attacker to perform mimicry SQLi attacks.

Specifically, Mereidos et al. in SEPTIC [25] propose an approach to block SQLi attacks inside the database. During training mode, SEPTIC records a profile that maps the parse trees of benign issued SQL queries to an identifier. SEPTIC generates the identifier in `mysql` and `mysqli`; two database extensions in PHP for communicating with a MySQL database. The identifier is inferred from the PHP call-stack that issued a call to one of the methods in the `mysql` or `mysqli` API for executing an SQL query on the database (e.g., `mysql_query`). The identifier is a sequence of functions in the

ASIA CCS '20, October 5–9, 2020, Taipei, Taiwan

ACM ISBN 978-1-4503-6750-9/20/10…$15.00
https://doi.org/10.1145/3320269.3384760

PHP call-stack that pass the SQL query as an argument. In enforcement mode, SEPTIC checks the parse tree of the SQL query against the profile it obtained during training mode. The design of SEPTIC leaves two unsolved challenges. (i) The strict comparison of the SQL query's parse tree against the profile leads SEPTIC to reject a range of dynamic yet benign SQL queries, thus causing false positives. (ii) The coarse-grained mapping of SEPTIC's profile allows an attacker to perform mimicry SQLi attacks successfully. The approach for creating the identifier in SEPTIC does not consider the fact that web apps do not necessarily pass the SQL query as an argument. As a result, SEPTIC assigns the SQL queries to a small set of functions in database API as an identifier.

We evaluated SEPTIC's protection model against the Drupalgeddon vulnerability in Drupal [8]. Database API in drupal uses the encapsulation concept that means Drupal's functions do not pass the SQL query as an argument to the database API. Hence, SEPTIC maps all issued SQL queries to the same sequence of functions in the database API instead of the function that interacts with the database through the database API. During training mode, SEPTIC creates its profile by mapping all the received SQL queries to a single identifier. This mapping in the profile means that any function that communicates with the database in Drupal can issue all the SQL queries in the SEPTIC's profile. For instance, an attacker can exploit the Drupalgeddon vulnerability in the presence of the SEPTIC and use the login functionality to issue an SQL query for creating an admin user.

Considering the challenges and open problems with existing defenses against SQLi attacks for PHP web apps, we propose a novel hybrid static-dynamic analysis and its implementation SQLBlock to defend OOP web apps against SQLi attacks. SQLBlock consists of four steps for defending web apps against SQLi attacks. In the first step, SQLBlock collects benign inputs through unit tests or benign browsing of web apps and creates a mapping between the issued SQL query and the function that issued the query. The static analysis is necessary to determine the database API precisely and subsequently identify the PHP function that uses this API to communicate with the database correctly. In the next step, SQLBlock creates a profile based on the issued query from each function in the web app during the training mode. The profile in SQLBlock is a mapping between the function that issues the SQL query and a query descriptor that describes the benign functionality of the SQL query. In the last step, SQLBlock enforces the profile inside the database to prevent the execution of any SQL query that does not match the profile at the runtime. We evaluate our system on a total of 11 known SQLi vulnerabilities of the top four most popular real-world web apps Wordpress, Drupal, Joomla, Magento, and their plugins. SQLBlock defends against all SQLi exploits, while SEPTIC can only defend against four SQLi attacks in our dataset.

In summary, we make the following contributions:

- We recognize that the object-oriented programming paradigm poses challenges for existing systems that lead to false positives and reduced protection against SQLi attacks. We propose a novel system to statically and precisely identify database API in a PHP web app, and dynamically restrict the SQL queries that MySQL executes based on the PHP function that composes the SQL query.
- We present a prototype implementation called SQLBlock as a MySQL plugin. It can be used with minimal modifications to MySQL for defending against more types of SQLi attacks against PHP web apps than prior work. (more details in § 3)
- We evaluate SQLBlock for its security and performance characteristics on four popular PHP web apps and seven plugins. SQLBlock protects the database and the web app against 11 previously known SQLi vulnerabilities in our evaluation dataset with an acceptable performance overhead (<3%).

We will open source our implementation of SQLBlock, including the testing and evaluation dataset. Our dataset includes 11 vulnerable PHP web app and plugins, as well as automated Selenium scripts recorded from human interactions with each web app.

2 BACKGROUND

In this section, we provide an overview of object-oriented programming in PHP and the PHP extensions that are used to communicate with MySQL databases. Afterward, we discuss MySQL and its plugin architecture. Understanding the OOP model in PHP is necessary for our static analysis. Besides that, the knowledge of database extensions in PHP for communicating with MySQL and the power of MySQL plugins shapes the implementation of SQLBlock. We then discuss different types of SQLi attacks that impact the profile created in step ③ of SQLBlock.

2.1 PHP

PHP is an open-source server-side scripting language. According to W3Techs [30], 79.1% of all websites use PHP as their server-side language. PHP supports binary extensions called *plugins* that provide PHP with additional features such as cryptographic algorithms, mail transfer, or database communications.

Database API in PHP provides an interface for communicating with a database. Database API can be database-specific such as MySQL and SQLite, or a general interface such as PHP Data Objects (PDO) for accessing various databases. The *mysqli* extension provides functionality to access MySQL databases in PHP scripts. Compared to *mysql*, which is another PHP extension for accessing MySQL, *mysqli* provides three additional features: support for prepared statements, multiple statement queries, and transactions. PHP web apps tend to use `mysqli` due to aforementioned additional capabilities.

PDO is an abstraction layer that provides a consistent API for accessing databases regardless of the database type. This feature allows a PHP script to use the same piece of PHP code to connect to different types of databases and issue queries. Although PDO delivers a clean and simple API for accessing the database, it only provides generic query-building functionality. For instance, PDO neither supports multiple SQL queries in one string, asynchronous queries, nor automatic cleanup with persistent connections.

PHP supports the Object-Oriented Programming model, which introduces three new concepts for developing PHP web apps: inheritance, polymorphism, and encapsulation. Inheritance and polymorphism let developers extend the functionality of classes or implement an interface in more than one way. Encapsulation bundles data and methods into a single unit. Hence, OOP allows developers to create modular programs and extend the functionality of PHP

database extension. Additionally, PHP provides dynamic features, such as creating objects from dynamic strings. `new` is the keyword for creating objects from a class in PHP. The argument for the `new` keyword, can be a class name or a string that represents the name of the class. An example is shown in Figure 2, line 22, where the value of `getDriver()` defines the class that should be instantiated.

Besides the object-oriented design of database APIs, PHP web apps also implement database procedures. Database procedures handle instantiating objects from the database API and return an object from the database API or a sub-type of the database API. Throughout this paper, we call the database API and procedures as the database access layer. The database access layer in the web app handles the communication of the web app's modules with the database. SQLBlock determines the database access layer in PHP web apps by reasoning about the source code of the PHP web app statically with respect to the OOP implementation of the web apps.

2.2 MySQL

MySQL is an open-source database management system. As of August 2019, according to Datanyze [12], MySQL is used in 46.03% of the deployed websites on the Internet. MySQL supports a plugin API that enables developers to extend the functionality of MySQL. MySQL Plugins can implement user authentication, query rewriting components, or new parsers for additional keywords and capabilities. MySQL plugins have access to different data structures, depending on their role. Of particular interest to this paper is the query rewrite plugin, which can examine and modify a query when MySQL receives the query before execution.

Query rewrite plugin has access to the parse tree of the SQL query that MySQL received. Each node in the parse tree based on its type contains information regarding the element it represents from the SQL query. For instance, the function node (e.g., `IN`, `<`) contains information regarding the number of arguments passed to the SQL function. SQLBlock uses the information that each node contains during its training and enforcement. Postparse plugins also have access to the information regarding the type of the SQL query (e.g., SELECT, INSERT) and the name of the table that the SQL query needs to access. SQLBlock uses the information above to create and enforce the query descriptors for each received SQL query.

2.3 SQL Injection attacks

SQL injection (SQLi) is a code injection attack in which an attacker is able to control a SQL query to execute malicious SQL statements to manipulate the database. SQLi attacks are classified into eight categories [11, 16]:

(1) ***Tautologies:*** The attacker injects a piece of code into the conditional clause (i.e., where clause) in a SQL query such that the SQL query always evaluates to true [16]. The goal of this attack varies from bypassing authentication to extracting data depending on how the returned data is used in the application.
(2) ***Illegal/Logically incorrect Queries:*** By leveraging this vulnerability, an attacker can modify the SQL query to cause syntax, type conversion, or logical errors [16]. If the web app's error page shows the database error, the attacker can learn information about the back-end database. This vulnerability can be a stepping stone for further attacks that reveals the injectable parameters to the attacker.
(3) ***Union Query:*** In union query attacks, the attacker tricks the application to append data from the tables in the database for a given query [16]. An attacker adds one or more additional *SELECT* clause, which start with the keyword *UNION*, that leads to merging results from other tables in the database to the result of the original SQL query. The goal of such an attack is to extract data from additional tables in the database.
(4) ***Piggy-backed Query:*** Piggy-backed query enables attackers to append at least one additional query to the original query. Therefore the database receives multiple queries in one string for execution [16]. The attacker does not intend to modify the original query but to add additional queries. Using the piggy-backed query, an attacker can insert, extract, or modify data as well as execute remote commands as well as extract data from the database. The success of the attack depends on if the database allows the execution of multiple queries from a single string.
(5) ***Stored procedures:*** Stored procedures are a group of SQL queries that encapsulate a repetitive task. Stored procedures also allow interaction with the operating system [16], which can be invoked by another application, command line, or another stored procedure. While a database has a set of default stored procedures, the SQL queries in a stored procedure can be vulnerable similar to SQL queries outside the stored procedure.
(6) ***Inference:*** In this type of attack, the application and the database are prevented from returning feedback and error messages; therefore, the attacker cannot verify whether the injection was successful or not [16]. In the inference attacks, the attacker tries to extract data based on answers to true/-false questions about the data already stored in the database.
(7) ***Alternate Encoding:*** In order to evade detection, the attackers use different encoding methods to send their payload to the database. Each layer of the application deploys various approaches for handling encodings [16]. The difference between handling escape characters can help an attacker to evade the application layer and execute an alternate encoded string on the database layer.
(8) ***Second order injections:*** One common misconception is that the data already stored in the database is safe to extract [11]. In a second order attack, an attacker sends his crafted SQL query to the database to store his attack payload in the database. The malicious payload stays dormant in the database until the database returns it as a result of another query, and the malicious payload is insecurely used to create another SQL query.

3 RELATED WORK

In this section, we review the relevant literature on defending web apps against SQLi attacks. We also compare SQLBlock with five existing approaches and explain why prior systems are not sufficient for PHP web apps that utilize OOP to communicate with databases.

Our comparison based on the SQLi attack type is presented in Table 1. For each SQLi attack type in Table 1, ● means the tool can defend against the type of attack, ○ means the tool is ineffective, and ◑ means that the tool can partially defend the web app against SQLi attack. Partially defending means that either the tool can only defend web apps that do not use OOP for implementing the communication with the database, or the definition of SQLi attacks in the tool is incomplete. The last column of Table 1 shows the number of SQLi exploits from our dataset in Table 3 that each tool can prevent.

Static Analysis: Several proposed approaches focus on detecting injection vulnerabilities statically in the source code of web applications [10, 11, 18, 20, 38]. Dahse et al. [10] proposed RIPS, an inter- and intra-procedural data flow analysis for detecting XSS and SQLi vulnerabilities in web apps. Pixy [20] implements a flow-sensitive data flow analysis to find XSS and SQLi vulnerabilities in web apps. WebSSARI [18] uses taint analysis to track untrusted user-inputs to detect command injection vulnerabilities. Dahse et al. [11] implement a context-sensitive taint analysis to analyze read and write operations to the memory locations in webserver for finding the second-order injections. Wassermann et al. [38] proposed a static analysis for detecting the injection vulnerabilities in web apps. A major drawback of prior analyzes are the inability to detect SQLi vulnerabilities in web apps such as Wordpress, Joomla, and Drupal that utilize OOP for communicating with databases.

Dynamic Analysis: Dynamic approaches track user-inputs [3, 6, 36, 37], or build a profile of benign SQL queries [24–26, 34] to prevent SQLi attacks on web apps. SQLPrevent [37] analyzes generated queries for the existence of HTTP request parameters and raises an alert when an HTTP request parameter modifies the syntax structure of a query. SQLGuard [6] proposed a dynamic approach for comparing the parse tree of issued queries at runtime before and after the inclusion of user inputs. SQLGuard needs to modify the source code of the web app. WASP [15] proposes a taint analysis to detect SQLi attacks on web apps. CANDID [3] records a set of benign SQL queries that the web app can issue by instrumenting the web app's source code and dynamically executing the SQL statements with benign inputs. CANDID, SQLGaurd, WASP, and SQLPrevent assume that if the input does not change the syntax structure of a SQL query, then a SQLi attack has not occurred. Such an assumption can leave the web app vulnerable to SQLi attacks and also blocks benign generated queries [31]. Unlike CANDID, SQLGaurd, WASP, and SQLPrevent, SQLBlock does not detect SQLi attacks based on the modification to the syntax structure of the SQL query. SQLBlock generates a set of query descriptors for benign queries that each PHP function issues to the database. SQLBlock allows functions in the web app to issue queries, as long as the query matches its query descriptors. Beyond this, SQLBlock does not need to modify the source code of the web app for its operation.

Tool	Taut.	Illegal/Incorrect	Union	Piggy-back	Stored proc.	Infer.	Alt. encoding	Second order inj.	flagged as attack
SQLrand [5]	●	○	●	●	○	○	○	○	0
SQLCheck [36]	◑	◑	◑	◑	○	◑	◑	○	0
Merlo et. al. [26]	◑	○	◑	◑	◑	◑	◑	○	0
SEPTIC [25]	◑	○	◑	◑	◑	◑	◑	◑	4
DIGLOSSIA [34]	●	●	●	●	○	●	○	○	5
SQLBlock	●	○	●	●	●	●	●	●	11

Table 1: Comparison of SQLBlock with other techniques with respect to SQLi attack type. SQLBlock provides the most effective protection.

SQLCheck [36] tracks user-inputs to SQL queries and flags a SQL query as an attack if user-input modifies the syntactic structure of the SQL query. This incomplete definition of SQLi attacks prevents SQLCheck from defending against tautology, inference, stored-procedure, and alternate encoding attacks. These four attacks do not necessarily modify the syntax structure of a SQL query. Considering this weaknesses, SQLCheck cannot protect web apps against any of the vulnerabilities in our dataset mentioned in Table 3.

Diglossia [34] proposed a dual parser as an extension to the PHP interpreter. Diglossia maps the query without user-inputs to a shadow query, and then it checks whether the parse tree of actual query and the shadow query are isomorphic or not. If both parse trees are isomorphic and the code in the shadow query is not tainted with user-inputs, Diglossia passes the query to the back-end database. Diglossia is unable to defend against Second-order injection since Diglossia only checks queries with user-inputs. Moreover, Diglossia cannot detect alternate-encoding and stored-procedure attacks since these attacks do not modify the parse tree of the SQL query [25]. As shown in Table 1, SQLBlock defends web apps against more variants of SQLi attacks than Diglossia.

SEPTIC [25] creates a profile for each issued query during the training phase and enforces this profile to protect web apps against SQLi attacks. During the training, SEPTIC creates a query model that includes all the nodes in the parse tree of a SQL query. The profile in SEPTIC is a mapping between the query model and an ID. The ID is the sequence of functions that pass the query as an argument. During the enforcement, SEPTIC uses this sequence of functions as an identifier and finds the appropriate query model in the profile. If the issued query matches the query model in the profile, SEPTIC allows the database to execute the query. Enforcing a profile based on the exact model of the generated queries that includes the name of table columns and number of SQL functions prevents web apps from generating dynamic yet benign SQL queries, which causes false positives in SEPTIC. For instance, assume there is a webpage for searching for published music albums and users can search based on the name of an album, an artist's name, or the released year. If SEPTIC is trained with SQL queries that only includes the album's name or the released year, it rejects any SQL queries from a user that searches using the artist's name. SQLBlock solves this problem by creating query descriptors for SQL queries. Query descriptors generalize the benign SQL queries, which allows the web app to produce a range of dynamic queries.

Furthermore, to create an identifier for each issued query in the profile, SEPTIC uses the information in the PHP call-stack that issued the call to methods from *mysql* or *mysqli*. SEPTIC checks the sequence of functions in the PHP call-stack for the presence of SQL

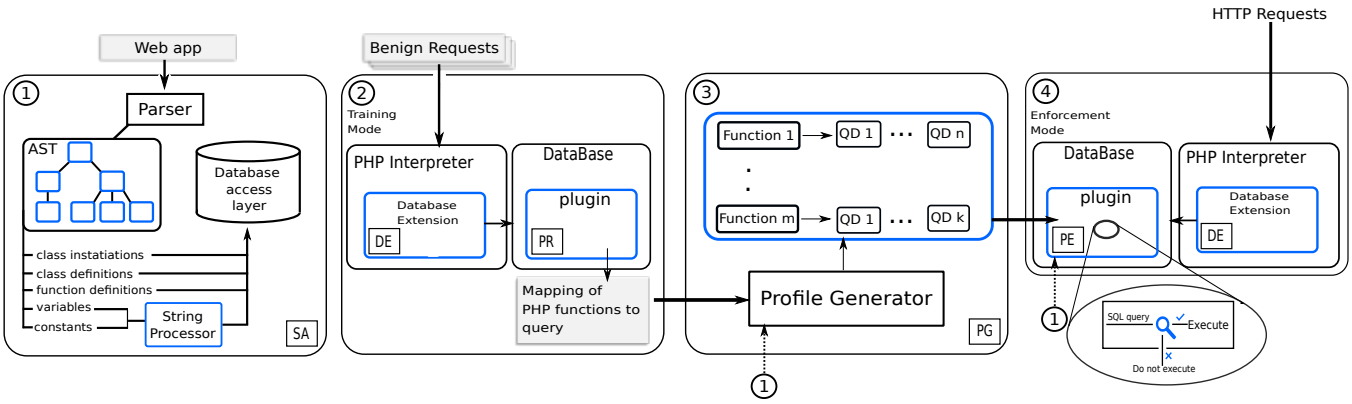

Figure 1: SQLBlock extracts the database access layer of the web app, builds a mapping between the function and the SQL queries it issues, creates a profile for each function in the web app and enforces the profile using a MySQL plugin.

query in function's arguments. Since OOP web apps do not pass the SQL query as an argument, SEPTIC cannot generate a correct identifier for SQL queries. Instead, it creates the same identifier for all the issued queries in the OOP web app. Consequently, an attacker can use a vulnerable function in the web app to issue any query from the profile. Considering the coarse-grained mapping that SEPTIC builds for the web apps that use OOP, SEPTIC can defend against only four variants of SQLi attacks in our dataset. All 4 SQLi attacks that SEPTIC can defend against reside in Wordpress. Wordpress does not use the encapsulation concept in its database API, and its modules provide SQL queries as function arguments; consequently, SEPTIC can correctly create its mapping. SQLBlock overcomes this problem by utilizing a static analysis that identifies the database API in the web app, which helps SQLBlock to correctly determine the function that interacts with the database.

Merlo et al. [26] proposed a two step approach. First, it intercepts every function call to `mysql_query` and records a profile for benign issued SQL queries. The profile is a mapping between the issued SQL query and the PHP function that calls the function `mysql_query`. During enforcement, [26] looks for the received SQL query in its mapping profile and if the query does not syntactically match with any recorded query for the PHP function, [26] blocks the query. The proposed approach in [26] maps all the SQL queries to the internal functions in the database API instead of the appropriate function that uses the database API for communicating with the database. Besides that, enforcing a strict comparison of the parse tree limits the functionality of the web app for generating dynamic SQL queries. Table 1 shows that the proposed approach in [26] cannot protect web apps against any of the SQLi attacks in our dataset.

Hybrid Analysis: Amnesia [17] builds a model of benign queries in Java web apps statically. At runtime, Amnesia checks the queries passed to the database against the built model. Amnesia highly depends on the benign queries that are built during the static analysis, which leads to a high number of false positives when applied to programs that generate SQL queries dynamically. SQLRand [5] proposed a randomization technique for randomizing queries in web apps. SQLRand randomizes the SQL queries in the web app and uses an intermediary proxy for de-randomizing before sending the queries to the database. Since web apps generate SQL queries dynamically, randomizing the queries using SQLRand is a challenging task. Using an intermediate proxy introduces overwhelming overhead to web app performance [16, 36]. Besides that, since there is one static key that modifies SQL keywords, the knowledge of new SQL keywords can compromise the security of SQLRand [6].

4 SYSTEM OVERVIEW

In this section we explain how SQLBlock records benign SQL queries and limits the access of functions in a web app to the database. Figure 1 shows an overview of how SQLBlock defends web apps against SQLi attacks. Specifically, SQLBlock records a profile by observing benign issued queries by a web app. SQLBlock then enforces the profile from inside the database for every query that the web app sends to the database.

In step ①, SQLBlock performs a static analysis over the web app to identify the database procedures that are used across the web app's scripts. This analysis is done once per web app and SQLBlock uses this information during training and enforcement of the profile.

In step ②, SQLBlock is in the training mode and records the benign issued SQL queries by the web app. SQLBlock can use benign browsing traces or the web app's unit tests in its training. SQLBlock creates a mapping between the benign SQL queries that MySQL receives and the functions in the web app that used the database access layer to issue the query to the database.

In Step ③, SQLBlock leverages the information from the first two steps to assemble a trusted database-access profile. The profile is a set of allowed tables, SQL functions, and type of SQL queries that each function in the web app can issue. At the end of the third step, SQLBlock acquires the necessary information to protect the web app from SQLi attacks.

In step ④, SQLBlock protects the running web app against unauthorized database access by filtering access to the database according to the trusted profile generated in step ③. The modified database extension (e.g., PDO in PHP) appends the execution information (i.e., call-stack) at the end of each SQL query as a comment before sending it to MySQL. Prior to the execution of each SQL

query, SQLBlock extracts the appended execution information from the SQL query and identifies the function that communicate with the database using the database access layer. SQLBlock checks the query against the profile that corresponds to the function that issued the query. Finally, if the SQL query matches the profile, MySQL executes the query and returns the results.

4.1 Static Analysis of Web apps

The web app database access layer provides a unified interface to interact with different databases. In step ①, SQLBlock identifies the database access layer by statically analyzing the web app. To this end, SQLBlock creates a class dependency graph (CDG). The CDG is a directed graph $CDG = (V, E)$, where the vertices (V) are classes and interfaces in the web app. An edge $e_{1,2} \in E$ is drawn between $v_1 \in V$ and $v_2 \in V$ if v_1 extends class v_2, implements interface v_2 .

After creating the CDG, SQLBlock extracts the list of classes and interfaces in the web app, which extends database APIs (e.g., PDO in PHP). To do so, we manually identify database extension classes (e.g., `mysqli` in PHP). Afterwards, SQLBlock iterates over the vertices of the CDG and checks whether a vertex is connected to the database API. If a vertex is connected to the database API, SQLBlock adds it to the list of database access layer. SQLBlock also adds classes into database access layer that their methods initialize an instannce from database API in PHP (e.g., `mysqli_init`). At the end of this iteration, SQLBlock possess a list of all classes and interfaces in the web app that extends the database API.

Besides the object-oriented design of database APIs in web apps, operations on databases (e.g., `SELECT` operation) also have procedures [13]. Database procedures handle creation of objects from database API and setting correct parameters for modules in the web app. A database procedure returns an object from a sub-type of a database API. SQLBlock analyzes the body of the functions and procedures in the web app for the returned objects. If the returned object is from a sub-type of a database API in the web app, then SQLBlock considers it as a database procedure. At the end of this step, SQLBlock extracts information regarding the database API as well as database procedures. This step is necessary for SQLBlock to find the function that used the database access layer for communicating with the database during training and enforcing of the profile.

Figure 2b shows a snippet of PHP code from a class that extends the database API `mysqli`. There is also a database procedure called the `executeQuery` in Figure 2b that return an object from `DatabaseConnectionmysqli` that is a subclass of `mysqli`. Figure 2a shows another snippet of code implementing a function called `get_public_info` that uses `executeQuery` to retrieve data from the database. SQLBlock identifies `DatabaseConnectionmysqli` as a subclass of `mysqli` and `executeQuery` as a database procedure.

```php
$id = $_GET["id"]
function get_public_info{
include dirname(__FILE__)."/db/database.php";
$users = executeQuery("public_info", $id);
...
}
get_public_info();
```

(a) `get_public_info.php`

```php
class DatabaseConnectionmysqli
        extends mysqli {
  private $query;
  function __construct(){
      parent::__construct("localhost","admin","admin","mysqldb");
  }
  public function setQuery( $query ){
      $this->query = $query;
      ...
  }
  public function execute(){
      return parent::query($this->query);
  }
  public function multi_execute(){
      $result = parent::multi_query($this->query);
      ...
  }
}
public function executeQuery( $tbl, $arg ) {
      $query = "SELECT * FROM ".$tbl." WHERE id > ".$arg;
      $classname = "DatabaseConnection".$this->getDriver();
      return new $classname()->setQuery($query)->multi_execute();
}
```

(b) `/db/database.php`

Figure 2: Illustrative PHP code snippets demonstrating dynamic inputs to *new* keyword

4.2 Collecting information regarding database access in the web app

In step ②, we train SQLBlock using benign traces or unit tests to learn benign SQL queries. Step ② consists of two components that work together to create a mapping between the received SQL query in MySQL and the function that composed the SQL query. The first component, appends the execution information at the end of each SQL query before sending it to the database. The execution information includes the call-stack in the web app that led to sending a SQL query to the database using database extensions (e.g., `PDO` or `mysqli` in PHP).

The second part, a MySQL plugin, intercepts the execution of the incoming SQL queries to MySQL. When MySQL receives a SQL query through benign traces or unit tests, SQLBlock records the SQL query that MySQL receives including the execution information appended to the SQL query. Since SQLBlock has access to the parse tree of the SQL query, SQLBlock traverses the parse tree and records information regarding the type of nodes in the parse tree of the SQL query. SQLBlock also logs the list of tables that the SQL query accesses, as well as the type of operation (e.g, `SELECT` operation) in the SQL query.

4.3 Creating the profile

SQLBlock in step ③, leverages the access logs collected from benign SQL queries in step ② and generates a profile that defines the access to the database for each function in the web app that interacted with the database. Particularly, the profile contains a set of query descriptors for each function in the web app. A query descriptor comprises four components. Each component specifies a different aspect of the database access, that we explain below.

- ***Operation:*** denotes the type of operation in the SQL query. The operation can be `SELECT, INSERT, UPDATE, DELETE`, etc. [9]. The profile records the type of operation in each SQL query. Enforcing the operation type removes the possibility of a SQLi attack performing a different operation. For instance, when the profile only specifies a `SELECT` operation, the SQL query cannot perform an `INSERT` SQL query.
- ***Table:*** determines the tables that the SQL query can operate on. Restricting the tables used in a SQL query prevents an attacker from executing a SQL query on a different table.
- ***Logical Operator:*** indicates the logical operators [9] used in the SQL query. Logical operators limit the ability of an attacker to use a tautology attack in a SQL query for extracting data from a table.
- ***SQL function:*** determines the list of functions that the query uses. The component also records the type of arguments that are passed to each function. The list of functions restricts the attacker to use only the functions that are recorded during the training. This limits the attacker's capability to use alternate encoding and stored-procedures attacks against the database.

At the end of step ③, SQLBlock acquired a set of query descriptors for each function in the web app that issued a SQL query based on the training data that was obtained in step ②.

4.4 Protecting the Web app

In the last step, SQLBlock is in enforcement mode and uses the profile created in step ③ to restrict access to the database for each function in the web app. When the database receives a SQL query, SQLBlock extracts information regarding the type of operation, table accesses, and parse tree of the received SQL query. Subsequently, SQLBlock extracts the function that issued the SQL query from the execution information appended to the incoming SQL query. Afterwards, SQLBlock looks up in the profile and retrieves query descriptors associated with the function that composed and issued the SQL query. For each query descriptor associated with function, SQLBlock compares each component of the query descriptor with the obtained information from the received SQL query. First, SQLBlock checks whether the type of operation in the received SQL query and in the query descriptor is the same or not. Second, SQLBlock examines the list of tables in the received SQL query. The list of table in the received SQL query must be a subset of the list of tables in the query descriptor. For the logical operators, SQLBlock checks whether the logical operators in the SQL query that MySQL received is subset of the logical operators in the query descriptor. Finally, SQLBlock inspects the functions used in the received SQL query as well as the type of arguments. The functions and the type of arguments must be in the recorded query descriptor. SQLBlock takes a conservative approach and allows the database to execute the SQL query only if all four components of a query descriptor associated with the function authorize the SQL query.

5 IMPLEMENTATION

In this section we elaborate on the implementation challenges that needed to be addressed build SQLBlock. First, we explain how SQLBlock statically analyzes PHP web apps to identify the database access layer. Afterwards, we describe how SQLBlock uses the MySQL plugin API to record the SQL queries that the database receives. We explain how SQLBlock creates a precise profile for each PHP function based on the SQL queries issued to the database. Finally, we describe SQLBlock's approach for using a MySQL plugin API to restrict database accesses.

5.1 Static Analysis of web apps

In step ①, SQLBlock analyzes the web app to determine the database API and database interfaces across the PHP scripts in a web app. SQLBlock performs a flow-insensitive analysis, which focuses on finding database API, interfaces, and procedures.

SQLBlock identifies all PHP files in the web app, using `libmagic`. We use php-parser [33] to parse each PHP script into an abstract syntax tree (AST). SQLBlock identifies classes, interfaces, and abstract definitions by scanning AST nodes that represent their corresponding definitions. SQLBlock examines interface and class definitions across the PHP web app to reason about the dependencies between classes and interfaces, During analysis, SQLBlock creates a class dependency graph (CDG) and draws an edge between interfaces and classes when: 1) An interface extends another interface. 2) A class implements an interface. 3) A class extends another class.

After creating the CDG, the static analyzer (SA) iterates over the nodes of the CDG to identify classes and interfaces that facilitate communication between the PHP web app and the database. To accomplish this, SQLBlock starts with the `PDO` and `mysqli` classes; two of the most popular database extensions in PHP. SQLBlock creates a list of classes and interfaces that share an edge with `PDO` or `mysqli` classes in the CDG. For example, after creating the CDG for the code in Figure 2b, SA identifies `DatabaseConnectionmysqli` as a subclass of `mysqli`.

SA must identify database procedures as well. SA decides whether a procedure is a database procedure or not by analyzing the type of object it returns. If a procedure returns an object from a subclass of the database API, SA marks that function as a database procedure. For determining the object type that a function returns, SA analyzes the AST node of the return statement. There are two cases that SA is interested to follow:

- **Instantiating an object using the `new` keyword:** If the function is instantiating an object using the `new` keyword in the return statement, SA analyzes the argument that is passed to the `new` keyword. If the argument is the name of a subclass of a database API, SA marks the function as a database procedure. If the argument is a variable, SA performs a lightweight static analysis as a limited form of constant folding over strings that compose the value. SA marks the function as a database procedure, if the resolved value is a subclass of database API.
- **Variable:** If the function returns a variable, SA iterates backward on the AST to the last assignment of the variable and checks whether the assignment is a class instantiation

```
1  SELECT * FROM public_info where id > 0 # mysqli::multi_query@DatabaseConnectionmysqli::multi_execute@executeQuery@get_public_info
2  FIELD@FUNC:>@2@FIELD@LITERAL # recorded info regarding the nodes in the SQL query
3  public_info@0 # recorded info regarding the table and operation type of the SQL query
```

Figure 3: recorded information regarding the execution of function `get_public_info`

or not. If it is a class instantiation, SA tries to resolve the type of instantiated object as described above.

As discussed in § 2.1, PHP web apps often use variables as an argument for creating objects from classes using the `new` keyword. During analysis, SA keeps track of arguments passed to `new` in PHP scripts using a string representation.

5.1.1 String Representation. SA encounters strings when handling variable assignments and constant definitions. Strings can be a mixture of literal components, function return values, and variables.

When SA iterates over an assignment node in the AST, it records a set of information from the assignment node in a hash table. SA keeps track of the name of the variable and the components on the right side of the assignment. SA also records the name of the function or the name of the class and method that the assignment statement occurs in. For example, in Figure 2b at Line 21, the function `executeQuery` has an assignment statement. The right side of the assignment concatenates a constant string and a return value from a function. SA records the name of the variable on the left side of the assignment as well as the value of the constant string and the return value from the function. SA also records the type of operation on the right side (as discussed next, it is a concatenation operation). SA implements common string operations to resolve the value of the assignment.

5.1.2 String Operations. SQLBlock manages frequent string-related operations.

Variables: The argument passed to `new` can contain variables defined in the script. SA keeps track of variable definition in the scope of script, class, or functions. When there is a variable assignment, SA creates an object for the variable and its value.

Concatenation: In PHP, strings can be constructed by joining multiple components with the `.` and `.=` operators. SA handles string concatenation by creating an object for concatenation and adds components that exist in the concatenation statement.

5.1.3 Identifying Database Procedures. To identify database procedures, SA iterates over the assignments and resolves the value of variables in the strings by looking for variables in the same class and function. If there is a variable without a value, SA represents the value as a regular expression `.*` wildcard. SA looks for a match between the generated regular expression and the list of database API subclasses. For example, in Figure 2b, line 21, SA cannot determine the return value of *$this->getDriver*. Instead, SA represents the value as a `.*` wildcard. SA searches the list of database API subclasses for a class that matches the regular expression `DatabaseConnection*`, and finds such a class named `DatabaseConnectionmysqli`. SA marks *executeQuery* as a database procedure.

At the end of this step, SA has a list of database access layer classes, interfaces, and procedures.

5.2 Profile Data Collection

This step trains SQLBlock to create a mapping between issued SQL queries and the web app's function that relied on the database access layer to issue the SQL query. The collected information in this step is necessary for generating query descriptors in step ③. As described in § 4, the information collected for each SQL query contains the operation, the access tables, the logical operators, the SQL functions that the query used, and the type of arguments in each SQL function.

5.2.1 Attaching a PHP call stack: When MySQL receives a SQL query, SQLBlock must infer which PHP function actually issued the SQL query. To achieve this, we modified the source code of the MySQL driver for the `PDO` and `mysqli` extensions. This modification appends the PHP call-stack at the end of the query as a comment before sending it to the database.

To access the PHP call-stack, we use the Zend framework's built-in function called `zend_fetch_debug_backtrace`. Zend keeps the information regarding the call-stack for the executing PHP script. This information includes the functions, class, their respective arguments, the file, and the line number that issued the call. The modified database extension (DE) extracts the PHP call stack and appends it as a comment to the end of the SQL query.

5.2.2 Extracting information from the parse tree: Recorder plugin (PR) acts as a post-parse MySQL plugin. PR has access to various information regarding the parsed SQL query in MySQL: the type of operation (e.g. `SELECT` operation, etc.), the name of the table, and the parse tree of the SQL query. MySQL provides a parse tree visitor function that PR uses to access the parse tree of SQL queries.

However, MySQL only allows plugins to access literal values of the query, such as user inputs in the parse tree. Because SQLBlock needs more information regarding the parsed SQL query, we modified the source code of MySQL-server so that the plugin can access non-Literal values as well. When MySQL invokes PR, PR records the SQL query that MySQL receives. Afterwards, PR iterates over the parse tree of the SQL query and records the type of each node. If the node represents a SQL function in the SQL query, PR also records the number of arguments used in the SQL function. The node that represents the SQL function in the SQL query also holds the number of arguments used in the SQL function. Afterwards, PR records the type of arguments passed to the SQL function as they appear in the parse tree of the SQL query. Lastly, PR logs the table and the type of operation for the SQL query that MySQL received. In MySQL the information regarding the type of

operation for a SQL query is shown as a number. Hence, PR logs the type of operation for a SQL query as an encoded number in the profile. Figure 3 shows the recorded information in the profile, when function `get_public_info` executes.

At the end of step ②, SQLBlock has detailed information on the received SQL queries for training.

5.3 Creating the Profile

In step ③, profile generator (PG) creates a profile for each PHP function in the web app that accesses the database. PG relies on the training data from step ② as input.

PG reads the recorded information from step ②. As shown in Figure 3, the first line is the SQL query including the PHP call-stack. Using the list created in step ①, PG must infer which PHP used the database access layer to send the SQL query to the database. This is a difficult problem, because the last function on the call stack might be a helper function that issues all queries for the application (and, in fact, this is how modern real-world PHP applications such as Wordpress and Joomla are written). PG iterates over the stack of functions in the PHP call-stack and checks whether the function or the method was recognized as a database procedure or database API method in step ①. PG iterates over the stack starting from the last call in PHP call-stack until a function is not a database procedure or database API method. PG identifies this function as the function that created the database query.

As an example, the Line 1 in Figure 3 shows the SQL query that MySQL receives including the PHP call-stack. PG detects `mysqli` as a database extension in PHP and `DatabaseConnectionmysqli` as a class that extends `mysqli`. Then, PG visits the next function `executeQuery`, which was identified as a database procedure in step ①. The next function in the PHP call-stack is `get_public_info`. `get_public_info` is not in the list of database procedures from step ①, therefore PG identifies it as the PHP function that used database access layer to send the SQL query to the database. PG will then update `get_public_info`'s query descriptor.

Afterwards, PG iterates over the nodes of the SQL query's parse tree and extracts all the logical operators. If all the logical operators are the same, PG updates the *cond* with the respective value. If both logical operators (i.e, both *OR* and *AND*) are in the nodes of SQL query's parse tree, PG sets *cond* to *"Both"*. If there is no logical operators in the SQL query, PG sets *cond* to *"None"*. Based on Figure 3, PG specifies that `get_public_info` does not use any logical operators in its SQL query.

PG iterates over the list of nodes from the parsed tree of the SQL query and extracts the name of the used functions in the SQL query as well as their respective arguments. Since the number of arguments passed to the SQL function can be variable, PG does not record each argument's type. Instead, PG summarizes the types of arguments that a SQL function relies on. There are multiple types of functions in MySQL such as numeric, string, comparison, and date function. All of the aforementioned types of SQL functions except the comparison type either receive less or equal to two arguments or modifies the content of the first argument passed to the function. Comparison functions in MySQL (e.g., <, `IN`, etc.) compare a single argument to a variable sized argument array. Moreover, the single argument appears as the first argument in the SQL comparison functions. Owing to this, PG records the type of the first argument passed to a SQL function separately. If the argument is a table column, PG records it as a `FIELD` argument, otherwise PG records it as a `LITERAL` argument. Afterwards, PG iterates over the rest of the arguments passed to the SQL function. If the type of all the other arguments are the same type (i.e., `FIELD` or `LITERAL`), then PG records the value of the respective type in the profile. Otherwise PG sets the type as *var*. For instance, based on Figure 3, PG specifies that function `get_public_info` used function `">"`, that the first argument is a table column and the second argument is a `LITERAL`.

Lastly, PG reads the information about the name of the table and the type of SQL query. For instance, based on line 3 in Figure 3, PG deduces that function `get_public_info` accesses the table `public_info` using a `0`-type SQL query (i.e., `SELECT` SQL query).

At the end of step ③, PG has a set of query descriptors for each PHP function in the web app that issued a SQL query during training in step ②

5.4 Protecting the web app

In step ④, the enforcer plugin (PE) is on enforcement mode. PE uses the profile that was generated in step ③ and protects the database from queries that deviate from the profile. Similar to PG, PE is implemented as a postplugin, which gives it access to the parse tree of the received SQL query. PE also uses the same PHP database extensions as described in § 5.2.1. PE reads the profile for each PHP function and uses it to analyze the received queries.

After receiving a query, MySQL parses the SQL query and calls PE. PE locates the call-stack and extracts the PHP function that issued the query with the same approach described in § 5.3. Afterwards, PE finds the query descriptors in the profile associated with the PHP function. PE checks the query against all four components of each query descriptor found for the PHP function. For operation type, PE checks whether the received SQL query has the same operation type as it is recorded in the profile. PE also examines that the list of tables accessed for the received SQL query is a subset of table access listed in the query descriptor. The logical operators used in the received SQL query must be a subset of the logical operators in the query descriptor. Finally, the received SQL query can only use a subset of functions listed in the query descriptor. PE also checks whether the arguments passed to each function has the same type as it is recorded in the query descriptor. Only if the SQL query matches with all four components of at least one query descriptor in the profile, PE allows MySQL to execute the SQL query and return the results. Otherwise PE returns `False` to MySQL-server, aborting execution of the query and returning an error to the web app, thus preventing a potentially malicious attacker-controlled SQL query from executing.

6 EVALUATION

We assessed the ability of SQLBlock to prevent SQLi attacks on a set of popular PHP web apps. We also examined SQLBlock's false positive rate during the benign browsing of the web app. Additionally, we evaluated the performance overhead of step ③ for the benign browsing. For our evaluation, we answer the following research questions:

RQ1 How precise is SQLBlock's static analysis?

RQ2 Is SQLBlock effective against real world SQLi vulnerabilities in popular web apps?

RQ3 How practical is SQLBlock regarding performance overhead and false positives?

6.1 Evaluation Strategy

In our evaluation, we performed our static analysis once for each web app in Section 6.2. We evaluated the database access layer resolved by our static analysis in RQ1. Then we leveraged our database access layer to answer RQ2 and RQ3. We trained and built the profile for SQLBlock using the official unit tests of each web app once and used the generated profile for the experiments to answer RQ2 and RQ3. The official unit tests examine the correctness of functions in the web app by executing test-inputs and verifying their results. The advantage of unit tests over web crawlers is that there is no need for manual intervention of administrators, specifically for providing semantically correct inputs for each form in web apps. A web app's unit tests are specifically tailored to its implementation and therefore are likely to achieve higher code coverage. Figure 4

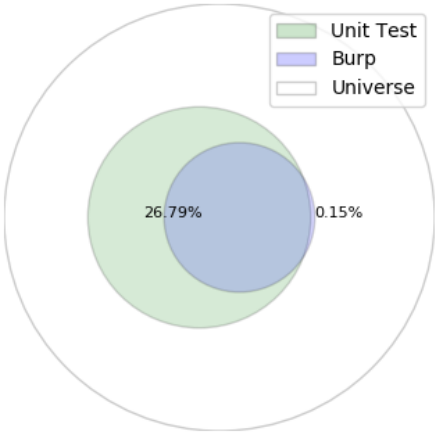

Figure 4: The line coverage for unit tests and Burp suite on Drupal 7.0

shows that Drupal's unit tests achieve higher line coverage compared to Burp suite and also covers almost all the lines that Burp Suite [23] covered. However, alternative approaches such as web crawlers can also be used for training SQLBlock.

6.2 Evaluation Dataset

We evaluated SQLBlock on the four most popular PHP web apps, Wordpress, Joomla, Drupal, and Magento. According to W3Techs, these web apps hold 70.5% of the market share among all existing content management systems (CMS) and power 38.4% of all the live websites on the Internet combined [30]. Administrators install plugins and additional components to customize the web app and extend its functionality. To reflect this behavior in our evaluation, we also evaluate SQLBlock on plugins. We installed four vulnerable Wordpress plugins called *Easy-Modal*, *Polls*, *Form-maker*, and *Autosuggest*. We also installed three vulnerable plugins in Joomla named *jsJobs*, *JE photo gallery*, and *QuickContact*. To assess the defensive capability of SQLBlock, we selected recent versions of the web apps and plugins that contain known SQLi vulnerabilities. We also considered the type of SQLi vulnerability in our dataset to include all types of SQLi exploits for a comprehensive evaluation. We collected a total of 11 SQLi vulnerabilities in different web apps and plugins.

6.3 Resolving The Database Access Layer (RQ1)

In step ①, SQLBlock scans the PHP web app to identify the database access layer that is used to communicate with the database. Step ① is a crucial step to identify the correct function in the PHP call-stack that relies on the database access layer for interacting with the database.

Table 2 presents the resolved database access layer statistics. The *resolved subclasses* column specifies the number of classes that extends the database API in PHP. The *resolved database procedures* column presents the number of functions that returns an object from a subclass of the database API. Since there is no ground truth for the database access layer in the web apps, we manually analyze the output of SA for true positives. Subclasses of database APIs in the PHP web apps also implement interfaces to facilitate actions such as iterating over elements in the object and counting elements. For instance, Drupal implements `Iterator` and `Countable` so that the PHP script can iterate over or count the number of records that the database returns to the PHP script. Since Drupal implements `Countable` and `Iterator` in the subclasses of database API, SA adds these two interfaces to the database access layer. As shown in Table 2, the only false positives we observed during our evaluation are caused by the `Iterator` and `Countable` interfaces. All the web apps in our dataset except for Wordpress, use encapsulation in their database API subclasses and database procedures that show the necessity of identifying the database access layer for creating a profile. Without identifying the database access layer, SQLBlock would operate similar to SEPTIC and map the received queries to a single identifier.

Web app	Resolved subclasses (FP)	Resolved database procedure
Wordpress 4.7	1	-
Drupal 7.0	44 (2)	38
Joomla 3.7	30 (0)	-
Joomla 3.8	30 (0)	-
Mangeto 2.3.0	15 (0)	-

Table 2: Resolved database access layer

6.4 Defensive Capabilities (RQ2)

We assessed the defense capabilities of SQLBlock against 11 SQLi vulnerabilities listed in Table 3. We built and deployed five Docker containers that run a vulnerable version of a web app and a plugin. We exploit the vulnerabilities using exploits from Metasploit Framework [27], exploit-db [32], and sqlmap [4]. We consider an attack successful if an attacker can inject malicious SQL code into the generated query in the web app and the database executes the malicious SQL query.

For this evaluation we used the results of our static analysis in RQ1. We trained SQLBlock using the official unit tests of web apps in their respective repositories. After creating the profile, we configured SQLBlock in the enforcement mode and assess whether the exploits in exploit-db and Metasploit Framework are successful or not. Adversaries are not limited to use exploits in our evaluation and can craft their SQL queries to circumvent SQLBlock. To evaluate the potential of such attacks, we also used sqlmap [4] to generate various exploits for the vulnerabilities listed in Table 3.

In Table 3, we present the list of SQLi vulnerabilities that SQLBlock defends the web apps against. The second column in Table 3, represents the ID assigned to each vulnerability. We marked the SQLi vulnerabilities that reside in the core of web apps by Ⓒ. The third column shows the type of attacks we performed to exploit the respective vulnerability. SQLBlock protects the web apps against all 11 SQLi exploits in our dataset, while SEPTIC can only defend against four SQLi exploits that only reside in Wordpress plugins.

To evaluate the potential of circumventing SQLBlock, we also listed the available query descriptors for the SQL queries that the vulnerable PHP function in each web app or plugin can issue. For instance, any potential exploit against the first vulnerability in Table 3 is restricted to an `UPDATE` query exclusively on table `wp_em_modals` without further logical operators. Furthermore, the exploit can only use SQL functions `"="` and `"IN"`.

6.5 Performance (RQ3)

Performance/responsiveness is a crucial factor for web apps. Therefore, we evaluate SQLBlock's performance overhead. In SQLBlock, the first three steps can be performed offline. Steps ① and ③ are automatic and do not rely on help from the administrator. In step ②, the administrator must perform unit tests or create benign traffic in the web app to train SQLBlock. Step ④ is deployed as a MySQL plugin and a set of modified PHP database extensions to sandbox databases against malicious SQL queries. The MySQL server loads SQLBlock's protection plugin upon launch. SQLBlock loads the profile and waits for incoming SQL queries. We perform our experiments on a 4-core Intel Core i7-6700 with 4Gb of memory 2133Mhz DDR4 that runs Linux 4.9.0, with Nginx 1.13.0, PHP 7.1.20, and MySQL 5.7.

For the performance evaluation, we created a Docker [19] container that runs with a default configuration of PHP, Nginx, and MySQL containing the Drupal 7.0 web app. We measure the performance overhead of SQLBlock using ApacheBench [14], a tool for benchmarking HTTP web servers. We simulated a real-world scenario by increasing the level of concurrency in ApacheBench. The level of concurrency shows the number of open requests at a time. We measured the network response time of `index.html` in Drupal 7.0 that issues 26 queries to MySQL. For more precise results, we measured the response time for 10,000 requests at multiple levels of concurrency. Table 4 presents our results for the aforementioned scenario. The first column in Table 4 shows the level of concurrency for each test. The next two columns in Table 4 present the network response time for Drupal with/without SQLBlock. As shown in Table 4 SQLBlock incurs less than (2.5%) overhead to the network response time of the server. Based on the strong protections afforded by SQLBlock, we consider this overhead acceptable. Furthermore, SQLBlock is a prototype with no emphasis on performance optimization. Such optimizations likely could reduce the overhead even further.

We also measured the execution time of queries in MySQL. We modified the source code of MySQL to calculate the time it takes for MySQL to execute a SQL query. For this experiment, we used ApacheBench to send 10,000 requests to `index.html` in Drupal 7.0, which issued a total of 260,000 queries to MySQL. We measured the average execution time of issued queries for two different scenarios. The first scenario is MySQL without SQLBlock's plugin, and in the second scenario, we enabled SQLBlock's plugin in MySQL. The last two columns in Table 4 present the average execution time of all the received queries to MySQL. The performance overhead of SQLBlock in MySQL is less than 0.31 ms for each query.

	Server Response Time(ms)		MySQL Execution Time(ms)	
Concurrency	Unprotected	Protected	Unprotected	Protected
1	27.792	28.338 (1.96%)	0.150	0.23
4	11.644	11.813 (1.45%)	0.669	0.90
8	8.907	9.127 (2.46%)	0.732	1.02
16	8.885	9.084 (2.23%)	0.740	1.05
32	8.971	9.182 (2.35%)	0.747	1.02

Table 4: Response times for requests to Drupal index.php

6.5.1 False Positive Evaluation. We count an operation as a false positive if SQLBlock blocks a benign query to the database. For the false positive evaluation, we evaluated SQLBlock with Wordpress 4.7 and Drupal 7.0. For each web app we used the profile that we built in RQ2. Then, we configured SQLBlock in enforcement mode and replayed browsing traces collected by Selenium [2]. Our browsing traces explored the web app as a user and administrator with the goal of covering the web app as much as possible.

Based on Table 5, only 10.11% of the issued queries during benign browsing and the unit test had the same query structure. This legitimate difference in the query structure of issued queries renders prior approaches that build their profile based on query structure unable to distinguish benign SQL queries from malicious ones. For instance, SEPTIC has above 89% false positive on the same test for Drupal 7.0. SQLBlock allows a query to execute in MySQL as long as the query matches at least one of the query descriptors associated with the PHP function in the profile. In the false positive test for Drupal, SQLBlock did not block any query from the benign Selenium browsing. This shows that although the PHP functions during training and testing used different queries, the query descriptors were the same.

Table 5 shows that 82.57% of queries in our benign browsing were similar to queries recorded for SQLBlock's profile. Although the

Application	Vulnerability	SQLi Type	Available query descriptors
Wordpress 4.7	CVE-2017-12946	Taut., Infer., Alt. Encoding	(update, wp_em_modals, none, [(=,field,literal),(IN,field,literal)])
Wordpress 4.7	polls-widget 1.2.4	Taut., Infer., Alt. Encoding	(update, wp_polls, none, [(=,field,lietral)])
Wordpress 4.7	CVE-2019-10866	Infer.	(select, wp_formmaker_submits, and, [(=,field,literal)])
Wordpress 4.7	WPVDB-9188	Taut., Infer.	(select, wp_posts, and, [(=,field,literal)])
Drupal 7	CVE-2014-3704 Ⓒ	Taut., Union, Piggy-back, Stored Proc., Infer., Alt. Encoding	(select, users, and,[(=,field,literal)])
Joomla 3.7	CVE-2017-8917 Ⓒ	Union, Infer.,Alt. Encoding	(select, [users, languages, fields], both, [(=,field,literal),(=, field, field),(IN, field, literal)])
Joomla 3.8.3	com_jsjobs 1.2.5	Infer.	(select, js_jobs_fieldsordering, none, [(=, field, literal)])
Joomla 3.8.3	com_jephotogallery 1.1	Union, Infer.	(select, jephotogallery, none, [(=, field, literal)])
Joomla 3.8.3	CVE-2018-5983	Infer.	(select, jquickcontanct_captach, none, [(=, field, literal)])
Joomla 3.8.3	CVE-2018-17385 Ⓒ	Second order inj.	(select, template_styles, and, [(=, field, literal)])
magento 2.3.0	CVE-2019-7139 Ⓒ	Infer., Alt. Encoding	(select, catalog_product_frontend_action, and, [(>=, field, literal),(<=, field, literal)])

Table 3: Exploits blocked by SQLBlock

rate of similar issued queries during training and testing of Wordpress is higher that Drupal, SQLBlock blocked 7 unique queries during the benign browsing, which corresponds to 5% of all issued queries. There are two main reasons for the false positives in Wordpress. The first reason is MySQL modifying the query based on the arguments passed to SQL function in the query. For instance, if the length of the array passed to the `IN` statement in a query is one, MySQL modifies the `IN` statement to an `equal (=)` statement. This modification in the query and subsequently in the parse tree of the query leads to false positives for SQLBlock since SQLBlock encounters a different function in enforcement than what is in the profile. The second reason is missing PHP functions in the profile. During the enforcement, SQLBlock blocks the SQL query if SQLBlock does not find any query descriptor for a PHP function that issued the SQL query. Six out of seven false positives in Wordpress was due to lack of query descriptors for the PHP function during benign browsing, which implies that covering all the functions that can issue a query during the training is an important factor for SQLBlock (Discussed further in § 7).

web app	Unit tests	Selenium	(Unit tests ∩ Selenium)	False Positive
Drupal	299961	336	34 (10.11%)	0
Wordpress	3099	132	109 (82.57%)	7

Table 5: Number of unique SQL queries during unit testing and Selenium browsing

6.6 Artifact Availability

SQLBlock implementation is open-source and available at https://www.github.com/BUseclab/SQLBlock. Additionally, we provide the five Docker containers that include a total of 11 vulnerable PHP web apps and plugins that we used in our evaluation. Our vulnerability dataset and the automated scripts were a significant part of our evaluation, and we think that it can be useful for future works in this area.

7 DISCUSSION AND LIMITATIONS

In this section, we discuss the limitations of the SQLBlock and possible future works in this area.

eval Function: PHP web apps use dynamic features implemented in PHP extensively, such as the `eval` function, which evaluates a string argument as a PHP code. Currently, SQLBlock does not handle function and class definitions inside `eval`. A web app can use `eval` for defining the database API or procedures dynamically and use it across the web app. This leads to generating a non-complete list of PHP database API and interfaces for a PHP web app in the step ①. In such cases, SQLBlock maps the query descriptors to a small set of PHP functions that can allow the attacker to execute a malicious query. In future work, the static analyzer in SQLBlock can be improved to handle the static PHP code passed to `eval`, to determine a more precise database access layer.

Incomplete coverage during training: PHP web apps generate dynamic queries based on user inputs. This approach makes it impossible to issue all possible queries to the database during the training phase. Dynamic analyses suffer from incomplete training phases, and SQLBlock is not an exception. Our Wordpress false positive test shows that the incomplete coverage of the issued queries leads to SQLBlock blocking benign queries.

8 CONCLUSION

We present SQLBlock, a hybrid dynamic-static technique, to restrict the PHP web app's access to the database. During the training step, SQLBlock infers issued SQL queries and their respective PHP callstacks. Using a lightweight static analysis, SQLBlock extracts a list of database API and procedures in the PHP web app. In the third step, SQLBlock creates a set of query descriptors for each PHP function in the PHP web app that issued a SQL query to the database. In the final step, SQLBlock acts as a MySQL plugin to restrict the interaction of the PHP web app and MySQL based on the generated query descriptors. SQLBlock can prevent SQLi attacks against 11 vulnerabilities in the top four most popular PHP web apps and seven plugins without any false positives for Drupal 7.0 and a low number of seven false positives for Wordpress benign browsing.

ACKNOWLEDGEMENTS

We thank our shepherd Giancarlo Pellegrino and the anonymous reviewers for their insightful comments and feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541.

REFERENCES

[1] Akamai. 2019. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet.
[2] Beans Erik Balde Samit, Barantsev Alexei. 2018. Selenium - Web Browser Automation. https://docs.seleniumhq.org/.
[3] Sruthi Bandhakavi, Prithvi Bisht, P Madhusudan, and VN Venkatakrishnan. 2007. CANDID: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security*. 12–24.

[4] G. Bernardo and S. Miroslav. 2019. sqlmap: automatic SQL injection tool. https://sqlmap.org.

[5] Stephen W Boyd and Angelos D Keromytis. 2004. SQLrand: Preventing SQL injection attacks. In *International Conference on Applied Cryptography and Network Security*. 292–302.

[6] Gregory Buehrer, Bruce W Weide, and Paolo AG Sivilotti. 2005. Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware*. 106–113.

[7] G. Cleary, M. Corpin, O. Cox, H. Lau, B. Nahorney, D. O'Brien, B. O'Gorman, J. Power, S. Wallace, P. Wood, and Wueest C. 2019. *Internet Security Threat Report*. Technical Report 24. Symantec Corporation.

[8] The MITRE Corp. 2014. CVE-2014-3704. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3704.

[9] Oracle Corporation. 2019. https://dev.mysql.com/doc/refman/8.0/en/.

[10] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis.. In *Proceedings of the Network and Distributed System Security Symposium*.

[11] Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium*. 989–1003.

[12] Dataanyze. 2019. MySQL Market Share and Competitor Report. https://www.datanyze.com/market-share/databases/mysql-market-share.

[13] Drupal. 2016. https://www.drupal.org/docs/7/api/database-api.

[14] Apache Software Foundation. 2018. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[15] W. Halfond, A. Orso, and P. Manolios. 2008. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering* 34 (2008), 65–81.

[16] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. 2006. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*. 13–15.

[17] William G. J. Halfond and Alessandro Orso. 2005. AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. 174–183.

[18] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th International Conference on World Wide Web*. 40–52.

[19] Docker Inc. 2018. Docker: Enterprise Container Platform.

[20] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*. 258–263.

[21] Anyi Liu, Yi Yuan, Duminda Wijesekera, and Angelos Stavrou. 2009. SQLProb: a proxy-based architecture towards preventing SQL injection attacks. In *Proceedings of the 2009 ACM symposium on Applied Computing*. 2054–2061.

[22] V Benjamin Livshits and Monica S Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis.. In *USENIX Security Symposium*. 18–18.

[23] PortSwigger Ltd. 2019. https://portswigger.net/burp.

[24] Ibéria Medeiros, Miguel Beatriz, Nuno Neves, and Miguel Correia. 2016. Hacking the DBMS to prevent injection attacks. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. 295–306.

[25] Ibéria Medeiros, Miguel Beatriz, Nuno Neves, and Miguel Correia. 2019. SEPTIC: Detecting Injection Attacks and Vulnerabilities Inside the DBMS. *IEEE Transactions on Reliability* (2019).

[26] Ettore Merlo, Dominic Letarte, and Giuliano Antoniol. 2007. Automated protection of php applications against SQL-injection attacks. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE, 191–202.

[27] Metasploit. 2019. metasploit. https://www.metasploit.com.

[28] Abbas Naderi-Afooshteh, Anh Nguyen-Tuong, Mandana Bagheri-Marzijarani, Jason D Hiser, and Jack W Davidson. 2015. Joza: Hybrid taint inference for defeating web application sql injection attacks. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 172–183.

[29] Q-Success. 2019. Usage Statistics and Market Share of Content Management Systems for Websites, August 2019. https://w3techs.com/technologies/overview/content_management/all.

[30] Q-Success. 2019. Usage Statistics and Market Share of PHP for Websites, August 2019. https://w3techs.com/technologies/details/pl-php/all/all.

[31] Donald Ray and Jay Ligatti. 2012. Defining code-injection attacks. In *Acm Sigplan Notices*. 179–190.

[32] Offensive Security. 2019. Exploit Database. https://exploit-db.com.

[33] Vadym Slizov. 2019. php-parser. https://github.com/z7zmey/php-parser.

[34] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2013. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 20th ACM SIGSAC conference on Computer*. 1181–1192.

[35] Sooel Son and Vitaly Shmatikov. 2011. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for ecurity*. 1–13.

[36] Zhendong Su and Gary Wassermann. 2006. The Essence of Command Injection Attacks in Web Applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 372–382.

[37] San-Tsai Sun and Konstantin Beznosov. 2008. Sqlprevent: Effective dynamic detection and prevention of sql injection attacks without access to the application source code.

[38] Gary Wassermann and Zhendong Su. 2007. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 32–41.