Securing Web Applications with Predicate Access Control

Zhaomo Yang[1(✉)] and Kirill Levchenko[2]

[1] University of California, San Diego, USA
zhy001@cs.ucsd.edu
[2] University of California, San Diego, USA
klevchenko@cs.ucsd.edu

Abstract. Web application security is an increasingly important concern as we entrust these applications to handle sensitive user data. Security vulnerabilities in these applications are quite common, however, allowing malicious users to steal other application users' data. A more reliable mechanism for enforcing application security policies is needed. Most applications rely on a database to store user data, making it a natural point to introduce additional access controls. Unfortunately, existing database access control mechanisms are too coarse-grained to express an application security policy. In this paper we propose and implement a fine-grained access control mechanism for controlling access to user data. Application access control policy is expressed using row-level access predicates, which allow an application's access control policy to be extended to the database. These predicates are expressed using the SQL syntax familiar to developers, minimizing the developer effort necessary to take advantage of this mechanism. We implement our predicate access control system in the PostgreSQL 9.2 DBMS and evaluate our system by developing an access control policy for the Drupal 7 and Spree Commerce. Our mechanism protected Drupal and Spree against five known security vulnerabilities.

1 Introduction

We depend on Web applications to handle more and more of our private and sensitive data. However, unauthorized data accesses are still very common today. A modern Web application consists of three distinct parts: client-side code running in the browser, server-side code running directly or in an application server, and a database often running on a separate server. Since the client side is completely under the user's control, in a typical application code that checks data accesses will be intermingled with code implementing the server-side functionality, split across multiple components of the application's server-side code. All checking code combined together consists of the application's security policy. Because there is not a centralized policy, developers may forget it when adding a new file or editing an existing file, thus introduce data access vulnerabilities.

G. Livraga and S. Zhu (Eds.): DBSec 2017, LNCS 10359, pp. 541–554, 2017.
DOI: 10.1007/978-3-319-61176-1_30

Most Web applications rely on a database to store and query user data. As the custodian of this data, the database management system (DBMS) seems a natural place to centralize those portions of security mechanism and policy that control access to user data. Unfortunately, the SQL access control mechanism is too coarse, providing only column-level access control. Moreover, a DBMS has no notion of application users, and so cannot protect one application user's data from another's. Such Web applications, therefore, do not use the SQL access control mechanism; instead, the database becomes a convenient data structure which the application uses to manage its data. From the DBMS point of view, there is only one database user, the application, which has complete access to all tables in the database.

In this work, we set out to design the most developer-friendly application user access control mechanism possible. We believe that it is simple, intuitive, and compatible with most applications' user protection boundaries. Our mechanism is implemented in the DBMS, where it can guarantee that the access control policy is enforced *even if the attacker gains direct access to the database*. The application developer specifies the access control policy to be enforced by the DBMS row-level predicates, which are checked on every query. These predicates are expressed using the familiar SQL syntax of WHERE clauses, attached to SQL GRANT statements.

In addition to specifying the desired policy, the security mechanism needs a way to tell the DBMS, at run time, on which application user's behalf the application is currently acting, in order for the DBMS to apply the policy correctly. We do this by allowing the developer to specify an authentication function that provides a means for the DBMS to authenticate a user.

We implemented our system as an extension to the PostgreSQL 9.2 database management system. The application policy, consisting of authentication functions and GRANT–WHERE clauses, is compiled by a separate tool into SQL statements accepted by the modified DBMS. Besides, we created security policies for several modules of the Drupal content management system and the Spree e-commerce platform. We appeal to the reader's own judgement and experience as a programmer and security practitioner to judge whether our means of expressing security policy is more clear and direct than the state of the art. The security policies provide protection against at least five known vulnerabilities in Drupal and Spree server-side code (Table 2).

The rest of the paper is organized as follows. Section 2, next, provides the necessary technical background for the rest of the paper. Section 3 provides a toy application example we use to illustrate our mechanism. Section 4 describes the application interface to the mechanism, that is, how the developer specifies the desired policy. Section 5 describes our implementation. Section 6 evaluates our system using Drupal and Spree. Section 7 describes related work. Section 8 concludes the paper.

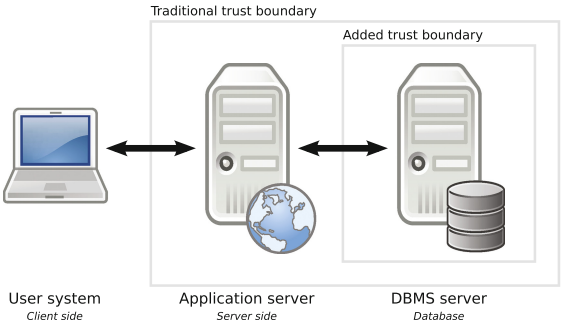

Fig. 1. Structure of a modern Web application.

2 Background

2.1 Modern Web Application Structure

A typical modern Web application consists of three parts, illustrated in Fig. 1: client-side code, server-side code, and a database.

Client-side. The client side of a Web application forms its user interface. It interacts with the server side via HTTP requests. This part of the application is in the difficult position of being trusted neither by user nor by the rest of the application itself.

Server-side. The server side of an application contains the major logic of the application. Each client interacts with a distinct instance of the server, and this interaction between a client and an instance of the server is termed a *session*. Applications may associate a set of access privileges with a session, which delineate what a client is allowed to do. The server side also interacts with a central data store, typically an SQL database. User and application data in the database is subject to access controls, and these are often implemented in the server side of the application as well.

Database. Many Web applications rely on an SQL database to manage application data. Web applications interact with the database by issuing SQL queries, either directly or via a framework (e.g., Ruby on Rails). Each server side instance, corresponding to a session, opens a connection to the database. In performing its function, the application server side may issue queries to the database or modify data in the database. In this sense, the server side *mediates* user access to the database.

Neither the operating system nor the DBMS has any notion of application users, and so can offer the application no help in implementing its access control policy; the server side of the application must implement all necessary access controls. A database has its own notion of users, managed by a database administrator, distinct from OS users and application users. An application connects

to the database as *a single database user*. Because of a single database user representing the application, the application database user must have the union of all access permissions necessary to carry out every application function.

2.2 Current SQL Access Controls

Our mechanism extends existing SQL access controls. SQL access controls work at the granularity of columns. A database user may be granted any combination of permissions to issue SELECT, UPDATE, DELETE, and INSERT statements affecting a set of columns. Syntactically, this is accomplished using the GRANT statement. For example, to grant user "Alice" SELECT and UPDATE privileges on table `Table 1`, the table owner would issue:

```
GRANT SELECT, UPDATE ON Table1 TO Alice;
```

After this statement is executed, the user "Alice" will be able to SELECT (read) and UPDATE (modify) values in table `Table 1`.

2.3 Threat Model

In this work, we assume that an attacker can co-opt the *server side* into letting him send arbitrary queries to the DBMS. Based on this threat model, the trusted computing base consists of the DBMS server and its operating system, the DBMS itself, and any user-defined functions installed in the application database when the database was first created. This model is assumed in Roichman *et al.* [9] and Oracle's VPD, and is stronger than the model of Nemesis [5], GuardRails [3], Scuta [12] and Diesel [6].

3 Toy Application: Gradebook

To make things concrete, we use a toy example application to illustrate both the traditional application security mechanism and our fine-grained mechanism. The Gradebook application is a simple Web application for students and instructors in a course. It allows students to check their own grades and allows instructors to view all grades, enter new grades and update old grades. Figure 2 shows the tables used by the application. The Gradebook application consists of two tables, `Users` and `Grades`, shown in Fig. 2. The `Users` table stores user names, a password salt, and a password hash. In addition, the `instr` column stores a boolean flag indicating if the user is an instructor or a student. The `Grades` table stores student grades for each assignment.

To authenticate a user, the Gradebook application issues the following query, which implements a salted hash:

```
SELECT user_id, instr
  FROM Users
 WHERE user_name = ?
   AND pass_hash = SHA1(pass_salt || ?);
```

Users	
user_id	INTEGER
instr	BOOLEAN
user_name	TEXT
pass_salt	TEXT
pass_hash	TEXT

Grades	
user_id	INTEGER
assignment	TEXT
score	INTEGER

Fig. 2. SQL database tables for the Gradebook toy application example.

Here the "?" stands for the user identifier saved by the application when it authenticated the user on login. (We show all queries as prepared statements, although our access control mechanism does not depend on their use.) To retrieve a student's grade, for example, the server side issues the query:

```
SELECT assignment, grade
  FROM Grades
 WHERE user_id = ?;
```

4 Application Interface

In this section we describe our access control mechanism. Our design was guided by two requirements:

- **Keep it simple.** Expressing a policy should be as simple and intuitive as possible.
- **Assume the worst.** The application should not be trusted; assume it is completely compromised.

First we introduce a straightforward concept called user-defined authentication function, which encapsulates the application's authentication mechanism. Then we extend the familiar SQL GRANT statement with a WHERE clause, as proposed by Chaudhuri *et al.* [4].

4.1 User Authentication Function

The first element of our mechanism is *authentication function*. This function is just like a normal user-defined function that returns table rows, but the results of the last evaluation of this function within a session are remembered in a per-session *authentication table* with the same name as the authentication function.

To illustrate, recall that the Gradebook application authenticates a user by querying a user table with the user name and password supplied by the user. If the user name and password match, the query returns the `user_id` and a flag indicating if the user is an instructor or not. (This information is used throughout the rest of the session.) To use our mechanism, the developer wraps this query in an authentication function:

```
CREATE AUTHENTICATION FUNCTION Auth(TEXT, TEXT)
RETURNS TABLE(user_id INTEGER, instr BOOLEAN)
AS $$
   SELECT user_id, instr
     FROM Users
    WHERE user_name = $1
      AND pass_hash = SHA1(pass_salt || $2);
$$ LANGUAGE SQL;
```

The authentication function `Auth` takes two text arguments, the user name and password of the user, represented by "`$1`" and "`$2`" in the function body. It returns a table row containing the `user_id` and `instr` flag of the row corresponding to the user, or no rows if the user name or password do not match. Note the similarity of the enclosed query to the authentication query given in Sect. 3. To authenticate a user, the application then issues the query:

```
SELECT user_id, instr FROM Auth($,$);
```

The result of the query is the same as in the corresponding query in Sect. 3, however because the query was issued through an authentication function, the result is now saved in an authentication table named `Auth`. This special table is available for use in access control predicates, as we will describe in the next section.

Note that some applications have more than one way to authenticate users. For example, applications which have a notion of session can authenticate a user using a session token stored in a cookie. To handle other authentication mechanisms, multiple instances of the authentication function taking different parameters can be defined.

4.2 Predicate Access Control

To protect individual rows in a table, the developer specifies a predicate defining which rows an application user may access. Syntactically, the predicate is expressed using a GRANT statement with a USING clause and WHERE clause. The USING clause lists the tables used in the predicate. The WHERE clause then specifies the predicate itself.

For example, in the Gradebook application, to express the policy that a student user may only see his own grades, the application developer would use the statement:

```
GRANT SELECT ON Grades TO Gradebook
USING Auth
WHERE Auth.user_id = Grades.user_id
   OR Auth.instr;
```

This statement grants a database user `Gradebook` permission to SELECT from the table Grades only those rows of the table where the `user_id` column is equal to the `user_id` of the currently authenticated user. The second term of the WHERE clause allows users with the `instr` flag to access all rows. All SELECT statements executed by the `Gradebook` database user will only see those rows of the `Grades` table that satisfy the predicate. The "`SELECT ... FROM Grades`" query from Sect. 3 is now, in effect:

```
SELECT assignment, grade
  FROM Grades, Auth
 WHERE user_id = ?
   AND (Auth.user_id = Grades.user_id
        OR Auth.instr);
```

The result of inner join above will return only those rows of the `Grades` table allowed by the policy.

The USING clause specifies the authentication table to be used for this access control check. Ordinary tables may also be specified in the USING clause of the GRANT statement, which will result in them being added to the join. DELETE, UPDATE, and INSERT privileges may be granted in a similar way.

4.3 Composition

Grants are a monotonic operation: a GRANT statement can only increase the access privileges of the grantee. Multiple GRANT statements for the same privilege (e.g., SELECT) on the same table will result in the grantee having access to all rows satisfying at least one of the WHERE clauses of the grant. In other words, the WHERE clauses of the two GRANT statements are OR'ed together. Thus, the "`GRANT SELECT`" statement in Sect. 4.2 has the same effect as two separate grants of the SELECT privilege with predicates "`Auth.user_id = Grades.user_id`" and "`Auth.instr`." Traditional (unpredicated) GRANT statements can be mixed with predicated GRANT statements. An unpredicated grant is equivalent to predicated GRANT with WHERE clause `WHERE TRUE`.

4.4 Revocation, Ownership and De-authentication

The REVOKE statement, which revokes access privileges from a user, is unconditional. That is, the result of executing a revoke statement is that the named user will not have the specified access privilege to the named tables. Predicates are not preserved when privileges are revoked. Granting the revoked privileges to the same user again will not restore the predicate in place before the REVOKE.

Access privileges on a table or view can only be granted by its owner. If the application database user (`Gradebook` in the examples above) is to be treated as untrusted, then the protected tables must have a different owner. Otherwise, an attacker connected to the database as the application database user can restore to himself all privileges. All application tables and views should be owned by a user other than the database user used by the application to service application user requests.

To de-authenticate, the application can re-authenticate as another user or call the authentication function in a way that is guaranteed to return no rows. For the `Auth` function used in the Gradebook application, calling it with either user name or password NULL will result in no rows being returned. This clears the authentication table associated with the authentication function.

5 Implementation

We implemented the predicate access access control mechanism described above by extending the PostgreSQL DBMS. Our implementation consists of two parts: a minimally-modified PostgreSQL 9.2 DBMS and a separate tool to compile security policies into lower-level constructs available in PostgreSQL.

5.1 Architecture

We chose to prototype most of the mechanism as a separate policy compiler, rather than in the DBMS itself, for convenience. In the intended ultimate implementation, the GRANT–WHERE predicate access control mechanism would be part of the DBMS.

In our current implementation, all schema declarations, including CREATE TABLE, CREATE VIEW, and CREATE FUNCTION statements are sent directly to the database. All security policy statements, namely GRANT (with and without a WHERE clause), REVOKE, and CREATE AUTHENTICATION FUNCTION, are sent to our tool for compilation into rewriting rules, triggers, and function definitions understood by PostgreSQL. The compiled statements are then sent to the database.

Because the compiler composes predicates as described in Sect. 4.3 and then packages them to be installed at the start of each session, the security policy must be presented all at once to the compiler.

5.2 PostgreSQL

The PostgreSQL DMBS supports query rewriting via its rule system, and we use this facility to implement access control predicates. To avoid large, unwieldy rules covering every database user, we modified PostgreSQL to support session-local rules, allowing the appropriate set of rules to be installed for each database user at the start of their session.[1] We call such session-local rules *temporary rules*, by analogy to temporary tables which exist for the duration of a session and are visible only inside the session in which they are created.

Temporary rules are defined using the CREATE TEMPORARY RULE statement, which, except for the addition of the `TEMPORARY` keyword, is syntactically identical to an ordinary CREATE RULE statement. A temporary rule is visible and effective only inside the session in which it is created.

Temporary rules differ from ordinary rules in one other way. PostgreSQL applies ordinary rewriting rules recursively to each table in a query until no further rules apply. This means, in particular, that rewriting rules cannot be recursive (or the rewriting step would not terminate). However, temporary rules derived from access control predicates will result in rewritten queries referring

[1] Temporary rules are also necessary because the current app. user's information is cached in a temporary table and the rules which referring to this table must also be temporary. Thus, modifying PostgreSQL cannot be avoided completely.

to the same table. For this reason, temporary rules are applied once and before permanent rules are applied. This allows temporary rules to rewrite queries using the same table.

5.3 On CONNECT Trigger

Predicate access controls are enforced using temporary rules specific to the database user of each session. These rules must be put into place at the start of a session, before the user issues any queries. To implement this, we added a special kind of trigger function to PostgreSQL which is executed when a user connects. This trigger is specified using a CREATE ON CONNECT TRIGGER statement. To prevent a database administrator from accidentally locking herself out of the database, ON CONNECT triggers are disabled for the database superuser. In our implementation, only the database superuser can create an ON CONNECT trigger.

5.4 Policy Compiler

We implemented the security mechanism described in Sect. 4 as a standalone security policy compiler that compiles GRANT, REVOKE, and CREATE AUTHENTICATION FUNCTION statements into temporary rewriting rules that are put into place at the start of a session by an ON CONNECT trigger.

6 Evaluation

To evaluate our predicate access control mechanism, we developed security policies for two modules of Drupal 7.1, a popular open-source content management system written in PHP, and Spree 1.3.1, a popular open-source eCommerce application using Ruby on Rails. Policies are based on what we inferred to be the intended access control policies of these applications. A set of general security policies for these two different popular applications allowed us to exercise all parts of our prototype and evaluate its performance on two complex applications with large user bases. Due to the space limits, the security policies are not shown in the paper.

6.1 Expressiveness

The ability to express an application's intended access control policy is our main criterion for evaluation. Moreover, our access control mechanism allows a policy to be expressed in a declarative manner and in one place. Both Drupal and Spree expressed access control policy in their implementation programming language (Drupal in PHP and Spree in Ruby) at the point in the code program where the access control check took place. The resulting policy and mechanism were spread across multiple locations in code.

Drupal. Drupal uses role-based access control, storing the assigned permissions in the database. Both login and session authentication functions cache the current application user's permissions and user ID. Drupal manages sessions by storing session-to-user mapping in the database. We protected 14 key tables using 51 GRANT statements, each of which expresses a separate access condition.[2] Drupal's existing access control checks occur at 15 separate locations in the source code.

Spree. Spree access control is also role based. Our authentication functions cache the current application user's roles along with his user ID. By default, Spree manages session information using the Ruby on Rails `CookieStore` mechanism, which stores session information in a cryptographically-signed cookie. While it would be possible to implement cookie signature checking in a database user-defined function, we configured Spree to use the `ActiveRecordStore` mechanism, which stores session information on the server in the same way as Drupal. We protected 53 tables using 29 GRANT statements, each of which expresses a separate access control condition. Spree's existing access control checks are spread across 11 separate locations in the source code.

6.2 Security

Our security policies mitigated three known Drupal vulnerabilities and two known Spree vulnerabilities (see Table 2). By their nature, GRANT-based polices define what data a user is allowed to access rather than defining disallowed behavior; such polices are much more likely to be effective against future vulnerabilities than policies covering specific vulnerabilities.

6.3 Performance

To assess the overhead imposed by our prototype, we measured the performance of Drupal and Spree using Apache JMeter, a popular server performance testing tool. All the experiments use a Intel Core i7-3632QM 2.20 GHz laptop with 4 GB of RAM and all of the client side, the server side and database were running on the same machine. Table 1 shows the time to perform each benchmarked task with and without our security policy. We report the mean of 50 runs.

Drupal. For the Drupal tests, we populated the database with 2,000 users and 5,000 nodes (articles). Each view article task results in 114 queries issued to the database. Each edit article task generates in 257 queries and data modification statements.

Spree. For the Spree tests, we populated the database with 50 users, 100 items and 200 orders. The benchmark task simulates the entire process of a user adding an item to the cart and checking out. The task results in 2,987 queries and data modification statements issued to the database.

[2] Alternatively, we could combine several access control into one statement in a disjunction.

Table 1. Drupal and Spree performance with and without security policies (*Before* and *After* columns, respectively).

Task	*Before*	*After*
Drupal view article	0.44 s	0.54 s
Drupal edit article	1.29 s	1.61 s
Spree buy item	18.40 s	24.12 s

Spree uses persistent connections more aggressively than Drupal, so Drupal creates database connections more frequently, which requires installing all the rules at the start of each session in the ON CONNECT trigger, which icnreases the performance penalty. Spree also issues more queries against tables that do not require row-level protection (e.g. `spree_countries`, which stores country information). On the other hand, Spree policies are more detailed, incurring a higher overhead. For example, the access control predicate for INSERT operations on the `spree_orders` table checks if the new row's total cost, order state and payment state are valid. In other words, our security policy encoded additional application constraints beyond plain access control.[3]

Although the performance overhead is not negligible, the absolute increase in query times is likely acceptable in many applications. Furthermore, these performance measurements are for an unoptimized prototype; an implementation integrated into the DBMS would likely perform better.

7 Related Work

Fine-grained and predicate access control has a long history, and it is not our intention to reinvent it. The aim of this work is to synthesize the most developer-friendly access control mechanism from the vast body of existing work, without sacrificing security guarantees.

7.1 Database Mechanisms

Rizvi *et al.* [8] present a predicate access control model using *authorization views.* Queries issued to the database must satisfy using authorization views granted to a user. Authorization views are defined using special authorization parameters, however how such parameters are communicated to the database is considered out of scope. Roichman *et al.* [9] also describe a similar solution using views parametrized by an authentication token. The application needs to be modified to store this token and transmit it back with each query.

[3] In effect, we are using the predicate access control to implement sophisticated foreign key and value constraints. A lower overhead, pure access control policy is also possible.

Table 2. Known vulnerabilities in Drupal 7.1 and Spree 1.3.1 which are mitigated by our policies.

Appl.	*Vulnerability*	*Description*
Drupal	CVE-2012-1590	User permissions are not checked properly for unpublished forum nodes, which allows remote authenticated users to obtain sensitive information such as the post title via the forum overview page
Drupal	CVE-2012-2153	Tag `node_access` is not added to queries thus queries are not rewritten properly and restrictions of content access module are ignored
Drupal	CVE-2011-2687	Proper tables are not joined when node access queries are rewritten thus access restrictions of content access module are ignored
Spree	*(No CVE ID)*[a]	By passing a crafted string to the API as the API token, a user may authenticate as a random user, potentially an administrator
Spree	CVE-2013-2506	Mass assignment is not performed safely, which allows remote authenticated users to assign any roles to themselves

[a] https://spreecommerce.com/blog/exploits-found-within-core-and-api

The Oracle DBMS introduced the Virtual Private Database (VPD) feature in release 8.1.5 to provide fine-grained access control [2]. Compared to our approach, the VPD mechanism is considerably more complex to use. The developer must supply a database function that returns a string containing a dynamically-generated WHERE clause fragment. User authentication information must be communicated via application contexts (key-value stores); the authentication function must explicitly store values in the context to be available by the dynamically-generated queries.

Row-level security introduced to PostgreSQL since version 9.5 allows the DB administrator to restrict which rows can be accessed or modified on a per-database user basis [1]. Although it provides a finer-grained access control compared to the standard privilege system, the DBMS is still not aware of application users, thus the per-application user access control cannot be achieved.

The GRANT-WHERE syntax we use was previously proposed by Chaudhuri *et al.* [4]. Like Rizvi *et al.* [8], they do not address the problem of authentication, assuming the availability of a function called `userId` conveyed securely by the database driver (e.g. via ODBC, JDBC, etc.), which supplies a user identifier. We are inspired by their clean and familiar syntax for expressing access control policy, which we pair with an intuitive way to authenticate a user to the application. There is no implementation of their system.

7.2 Non-DBMS Mechanisms

Several proposals from the computer security community solve the problem by mediating the communication between application and database. The Nemesis [5] system relies on taint tracking in a specialized PHP interpreter to automatically infer when a user has authenticated to a database and then applies appropriate access controls. Taint tracking is also used by GuardRails, a source-to-source translator for Ruby on Rails programs, which together with developer annotations allows a security policy to be applied to program objects [3]. Both Nemesis and GuardRails cannot protect data in the database if the application is tricked into sending arbitrary queries to DB. The CLAMP [7] system solves the authentication and access control problem by extracting from the application the authentication mechanism and access control logic to create a "User Authenticator"(UA) and a "Query Restrictor"(QR). These two components are isolated from the application itself, and so, should the attacker gain control of the application, would not be corrupted. Building a UA and a QR consisting of authentication logic and data access logic respectively, however, may be too complex for developers.

The fine-grained access control we propose can be applied at the level of users, the number of which is not known *a priori*. Two systems, Scuta [12] and Diesel [6] provide protection at the level of application modules or components. This provides isolation between untrusted modules of an application. Scuta works by placing components into different access control rings, which are mapped to multiple database users with the necessary table-level access controls. Diesel, on the other hand, intercepts and rewrites queries sent to the database. The level of access is indicated to the rewriting proxy at the start of a session, in effect allowing an application to drop privileges before issuing queries on behalf of an untrusted module. Besides, Son *et al.* [10,11] attempt to find and repair vulnerabilities in applications using static analysis. This is a very promising approach, but is likely still too complex for developers to use.

8 Conclusion

In this paper we described a row-level access control mechanism implemented in the database aimed at Web applications. Our mechanism allows the application developer to express her application's security policy using familiar SQL syntax.

We implemented our system as an extension to PostgreSQL and developed a security policy for two core Drupal modules and the Spree e-commerce platform. Our prototype has acceptable performance overhead, yet provides protection against five known and future unknown vulnerabilities. Moreover, our mechanism allows the security policy for these systems to be expressed in a centralized manner.

References

1. PostgreSQL's row-level security. https://www.postgresql.org/docs/9.5/static/ddl-rowsecurity.html
2. Virtual Private Database. http://www.oracle.com/technetwork/database/security/index-088277.html
3. Burket, J., Mutchler, P., Weaver, M., Zaveri, M., Evans, D.: GuardRails: a data-centric web application security framework. In: Proceedings of the 2nd USENIX Conference on Web Application Development (2011)
4. Chaudhuri, S., Dutta, T., Sudarashan, S.: Fine grained authorization through predicated grants. In: Proceedings of the 23rd IEEE International Conference on Data Engineering (2007)
5. Dalton, M., Kozyrakis, C., Zeldovich, N.: Nemesis: preventing authentication and access control vulnerabilities in web applications. In: Proceedings of the 18th USENIX Security Symposium (2009)
6. Felt, A.P., Finifter, M., Weinberger, J., Wagner, D. : Diesel: applying privilege separation to database access. In: Proceedings of 6th ACM Symposium on Information, Computer and Communication Security (2011)
7. Parno, B., McCune, J., Wendlandt, D., Andersen, D., Perrig, A.: CLAMP: practical prevention of large-scale data leaks. In: Proceedings of the 30th IEEE Symposium on Security and Privacy (2009)
8. Rizvi, S., Mendelzon, A., Sudarshan, S., Roy, P.: Extending query rewriting techniques for fine-grained access control. In: Proceedings of the 2004 ACM SIGMOD international conference on Management of Data (2004)
9. Roichman, A., Gudes, E.: Fine-grained access control to web databases. In: Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (2007)
10. Son, S., McKinley, K.S., Shmatikov, V.: RoleCast: finding missing security checks when you do not know what checks are. In: Proceedings of the 2011 ACM Conference on Object Oriented Programing Systems Languages and Applications (2011)
11. Son, S., McKinley, K.S., Shmatikov, V., Up, F.M.: Repairing Access-Control Bugs in Web Applications. In Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (2013)
12. Tan, X., Du, W., Luo, T., Soundararaj, K.D.: SCUTA: a server-side access control system for web applications. In: Proceedings of the 17th ACM Symposium on Access Control Models and Technologies (2012)