Diesel: Applying Privilege Separation to Database Access

Adrienne Porter Felt
UC Berkeley
apf@cs.berkeley.edu

Matthew Finifter
UC Berkeley
finifter@cs.berkeley.edu

Joel Weinberger
UC Berkeley
jww@cs.berkeley.edu

David Wagner
UC Berkeley
daw@cs.berkeley.edu

ABSTRACT

Database-backed applications typically grant complete database access to every part of the application. In this scenario, a flaw in one module can expose data that the module never uses for legitimate purposes. Drawing parallels to traditional privilege separation, we argue that database data should be subject to limitations such that each section of code receives access to only the data it needs. We call this *data separation*. Data separation defends against SQL-based errors including buggy queries and SQL injection attacks and facilitates code review, since a module's policy makes the extent of its database access explicit to programmers and code reviewers. We construct a system called Diesel, which implements data separation by intercepting database queries and applying modules' restrictions to the queries. We evaluate Diesel on three widely-used applications: Drupal, JForum, and WordPress.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—Security, integrity, and protection; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Security, Reliability

Keywords

data separation, privilege separation

1. INTRODUCTION

The principle of least privilege states that each principal should receive the privileges needed to perform its intended task and nothing more. Following this principle limits the scope of a bug or malicious attack and is commonly regarded as a good security and software engineering practice [26]. A *privilege-separated* application applies the principle of least privilege internally, decomposing the program into modules so that each module receives only the privileges it needs [24]. This provides error containment: a bug can leak only the privileges of the module that contains it, even if the application as a whole is highly privileged.

We propose applying privilege separation to data access within database-backed applications, which we refer to as *data separation*. With data separation, each module receives access to only the data needed for its intended task. A data-separated module receives a *restricted connection* instead of a regular database connection, and a policy limits the set of operations allowed over the module's restricted connection. A software developer can limit each module to the data required by that module's functionality. An application-side data separation framework provides a policy enforcement mechanism. We value data separation for the same reasons we value traditional privilege separation:

- **Additional line of defense for bugs.** If a bug is present in a database-facing module, damage is limited to the set of operations the module can perform. This means that a bug (e.g., a SQL injection vulnerability) can read or corrupt only the parts of the database that are accessible to that module.
- **Simpler code review.** Data separation aids code review. The reviewer can determine the potential impact of a bug in any given module, which makes it possible to devote extra attention to modules whose failures could endanger the integrity or confidentiality of critical data.

Traditional database user access control and data separation are complementary. Database access control limits human users' privileges, whereas data separation limits the data accessible to code modules. Data separation mitigates attacks in which a user is tricked into attacking her own data. For example, a cross-site request forgery attack could prompt a user's browser to submit a form with a SQL injection attack that deletes the user's data. A second-order SQL injection attack [22] or a combination cross-site scripting and SQL injection attack could similarly damage a user's data without her knowledge. User-based database access control would allow these attacks, since the user issuing the query has the required privileges. However, data separation limits the extent of these attacks to the data used by the vulnerable module. Similarly, data separation can enhance reliability. For example, a calendar display module cannot accidentally edit billing tables. Modules cannot perform unnecessary operations even if user-based access controls allow the user to perform those actions through another module.

AsiaCCS '11 Hong Kong

In practice, database-provided access control is often not used at all. Most web applications connect to a database with the same database user for all human users. This common practice is due to connection pooling, large numbers of human users, and a lack of database support for row-based database permissions. Data is particularly endangered in this scenario. If the application's logic is wrong or vulnerable, the entire database is at risk of exposure. Data separation can limit the extent of such an error in application logic: a buggy calendar module might leak all users' calendars, but data separation can prevent it from also leaking the billing and administrative tables.

We envision our system being particularly useful in the following scenarios:

- **Capability-secure programs.** Capability systems provide a platform in which it is possible to limit and verify which parts of a program have access to which resources [19]. Data separation is a way for capability-secure programs to interact securely with a database.
- **Web applications.** Web applications typically use connection pooling [28], wherein the server establishes a fixed number of connections and reuses them between instances of the web application. All of the connections are associated with the same user, which represents the web application and not the client-side human user. With our data separation framework, connections from the connection pool can be dynamically restricted based on the identity of the currently running module and/or the logged-in user.
- **Secure extensibility.** Third-party program extensions are usually difficult or impossible to review as thoroughly as the core program. It is therefore desirable to restrict the privileges of the potentially buggy third-party code. In the case of web applications, data separation can help protect against vulnerable extensions. For example, one Drupal plugin had a vulnerability that could be exploited to obtain the administrator password of the Drupal-powered web site [12]; data separation could have limited the impact of this vulnerability.

This paper's primary contribution is the principle of data separation. We also design, implement, and evaluate a prototype data separation framework named Diesel. We apply data separation to three applications.

2. DESIGN

Data separation is a design pattern for limiting the database rights of buggy application modules. We discuss how existing database access controls can implement data separation and the limitations of this approach. Our prototype, Diesel, supports data separation with an application-side, proxy-based framework.

2.1 Data Separation

As in standard privilege separation, we define application modules as logically related units of code (e.g., a class) [24]. With data separation, modules receive *restricted connections* — database connections that can access only subsets of the database, according to their policies. Each module can have any number of restricted connections, each of which has its own policy. A *data separation framework* provides a policy-setting API and an enforcement mechanism.

The developer creates a small, trusted module known as the *powerbox* to manage restricted connections. Within the powerbox, the developer defines policies, associates them with connection objects, and distributes the resulting restricted connection objects to the appropriate modules. Policies restrict access to a subset of the database as a list of whole tables or table subsets (using database views). The developer specifies the permissible operations for each table or sub-table. Data separation benefits from incremental deployability; developers can focus first on modules with the largest attack surface and then gradually restrict the database access of other modules.

A module can create a *pared down* (i.e., further-restricted) version of its restricted connection to share with another module or a less privileged sub-module. This feature is especially important for capability-style programming, in which paring down a capability is a common programming pattern [19]. It also aids incremental deployment.

The data separation framework's policy enforcement mechanism must be resilient to SQL-based attacks that attempt to circumvent a connection's policy by sending SQL commands specifically formed to confuse the security mechanisms. We assume the developer is willing to use our system and not actively trying to subvert it; our threat model does not include malicious application code. Arbitrary untrusted code may do many malicious things that we cannot reasonably prevent with a tool of this scope. If such security guarantees are desired, we suggest the use of a capability-secure language (see Section 4.1).

2.2 Repurposing User-Based Access Control

In some instances, it may be possible to realize data separation with existing user-based database access control mechanisms. If an application does not make use of database users to represent human users or roles, database users can be used to represent program modules. This is analogous to the Android security model [6], where the operating system repurposes Linux users to represent applications. This allows permission assignment on a per-module basis, and it works if one assumes that there is only one human user.

This approach may work in a simple scenario, but there are a few shortcomings that render this approach not generally applicable. Consider an application that has m users and n modules. Enforcing data separation with existing database access control mechanisms would require creating nm users, and modifying a user's permissions or a module's permissions would be unwieldy. Additionally, this approach may be infeasible in an organization in which the application developer and the database administrator are not the same person.

Using database users for data separation also does not support dynamically paring down restricted connections. In order to pare down a connection, a module would need to create a new database user for its submodule. This operation typically requires the INSERT privilege for the database's user table, and every module would need this privilege. While this may be acceptable, removing the created user (e.g., after the module terminates) requires DELETE privileges on the same table. Giving every module DELETE privileges on the user table would allow any module to remove any other module's privileges or delete the `root` user.

2.3 Prototype Framework

We propose an application-side data separation framework, Diesel, that operates orthogonally to database access control. Database support is not necessary, so database access control use can continue normally. Diesel has two components: a policy-setting library and a proxy that interposes on connections in order to enforce policies. When the powerbox makes a restricted connection, it connects to a local proxy instead of the database (see Figure 1). The policy-setting library sends a policy to the proxy as part of the restricted connection initialization process. The powerbox then distributes the restricted connection to a module. Statements made over restricted connections are received by the proxy, which checks them against the associated policies. Permitted statements are forwarded to the database server over a single database connection (or over one of many identical connections for load balancing). The powerbox may retain a powerful connection for itself by refraining from setting a policy on its own connection.

Restricted connections do not map one-to-one to network database connections. Instead, restricted connections within an application are derived from a single shared database connection. Applying data separation to an application does not increase the number of network connections to the database. Like full-fledged database connections, restricted connections are associated with a database user. SQL commands are subject to both the data separation policy restrictions (as enforced by the application-side data separation framework) and database user access rights (as enforced by the database). All restricted connections derived from the same database connection share the same database user.

Our proxy-based architecture is designed for reuse across different programming languages and frameworks. The policy enforcement mechanism in the proxy can be used for any application (regardless of language), and it needs verification only once. The policy-setting library is language-specific, but it is trivial; all of the complexity is in the proxy's enforcement mechanism. Additionally, this architecture requires very little refactoring to begin using data separation. Modules can issue SQL commands through the normal database API, without any extra accommodations for data separation. The powerbox is the only part of the application that requires modification, in order to define policies and distribute restricted connections. Section 4 describes our experience in refactoring three existing applications in different languages (PHP and Java) to work with our prototype.

This architecture incurs overhead when it proxies packets (e.g., when it passes a result set from the database to the application) and examines SQL statements. Some small cost is also added by having multiple connections open to the proxy on the application side, but running the proxy on the same machine as the application mitigates this cost. A remote proxy would add network overhead because it would require multiple TCP connections across the network to the proxy; with a local proxy, the cost is low because the connections are local TCP connections or pipes.

Alternate Architectures.

Other application-side policy enforcement mechanisms are possible as alternatives to our proxy-based prototype. One possibility is to modify the database API (e.g., the Java JDBC driver) to accept policies and interpose on queries. Another option is to wrap the database API with new procedures that perform policy enforcement. Either way, the entire framework would reside in a local library and there would be no need for a proxy. Both of these options remove the overhead of a proxy but require a new implementation of the policy enforcement code for each language and database API. Depending on the language, these alternative architectures could require application refactoring or a custom interpreter build. Despite these extra development costs, an implementation without a proxy might be preferable for a performance-critical application. We chose a proxy-based architecture for our prototype so that we could make it available for use with existing applications in multiple languages.

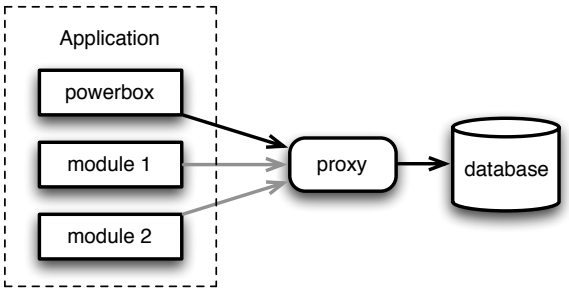

Figure 1: Diesel's architecture. Each module has its own connection object. The powerbox's connection has no data separation policy; it is a regular connection to the database. The powerbox set restrictive policies on the connections for modules 1 and 2. The proxy multiplexes the connections, so only one connection goes to the database.

3. IMPLEMENTATION

We provide Diesel policy libraries for Java, PHP, and Python applications (Section 3.1). The policy library is the interface between the data separation framework and the developer. The proxy (Section 3.2) is responsible for policy enforcement and represents the majority of the complexity of our system.

3.1 Policy Library

Developers interact with Diesel through a policy library. The policy library is an API for managing restricted connections and their policies. Each language and database API needs its own implementation of the policy library, and we built three: Java JDBC, PHP mysqli/mysql, and Python MySQLdb. The libraries are small (fewer than 170 lines of code each) and differ only by the semantics of the implementation language. Using a policy library (e.g., the Java library in Figure 2), a developer can define a policy, use policies to restrict connections, and pare down connections.

Defining a Policy.

Policies specify what operations can be performed on the database. The operations available in our prototype are SELECT, UPDATE, DELETE, and INSERT. Developers can grant privileges for entire tables or table subsets. We implement table subsets using database views. A view can be thought of as a "virtual table": it is the result of a SELECT query run over tables or other views. A developer provides the `defineTableSubset` method with the desired view name

```
public class DieselPolicy {
    public void grantTablePermissions(String name, Permission... p) {...}
    public void defineTableSubset(String name, String queryDef, Permission... p) {...}
    public void applyPolicy(Connection conn) {...}
    public Connection pareDown(Connection conn) {...}
}
```

Figure 2: The `DieselPolicy` class from the Java JDBC policy library. Slightly simplified for display.

(a *label*), the SELECT statement that defines the view, and the permissions it wishes to associate with the view. For example, consider a policy that limits a module to two columns of a table `Users`. The developer would use `defineTableSubset` to map the label `Users` to a view defined by the query `SELECT name, email FROM Users`. When the module asks for the table `Users`, it will receive a view (labelled `Users`) that includes only names and e-mail addresses.

Restricting a Connection.

In order to create a restricted connection, the developer needs to: (1) create a policy, (2) create a connection to the proxy, and (3) apply the policy to the connection. We accomplish the last step by passing a connection object to a policy's `applyPolicy` method. The library sends the policy information over the connection to the proxy, followed by a `START` command. Policy enforcement begins when the `START` command is received, making the restricted connection ready for use by a module. At this point, the policy cannot be removed from the restricted connection; that is, no SQL query can escalate the connection's privileges.

Paring Down a Connection.

A module might need to further restrict a restricted connection. We provide a `pareDown` method that will return a new restricted connection object with additional restrictions added to it. The original connection can still be used as before, with its original policy. The new, pared down connection must have a subset of the original connection's privileges (or else the proxy will reject it).

3.2 Proxy

The proxy runs on the same machine as the application. When it is initialized, the proxy connects to the database server with the appropriate credentials (i.e., those that the application normally connects with) and begins listening for incoming connections. The proxy is responsible for accepting commands from the policy library, checking statements against connections' policies, and multiplexing restricted connections over the database connection(s).

We use MySQL Proxy 0.7.2 [20], an open source proxy for MySQL databases, with a plugin to support our framework. The core proxy informs our plugin of connection events like receipt of a new connection or packet. The plugin can insert packets into the queue, authenticate or refuse incoming connections, and relay, edit, or discard incoming packets. Our plugin includes code from the `funnel` plugin [17] to handle multiplexing.

Policy Enforcement.

The proxy inspects all statements from restricted connections. For a statement to pass through the proxy to the database, the restricted connection must have permission to perform the statement's operation on all tables listed in the statement. To enforce this, we extract the operation and table names from each statement. We use the MySQL Proxy tokenizer and our own parser written in Lua.

We do not need to fully parse SQL statements because they are highly structured. Table names can occur only in well-defined locations. For example, table names in a SELECT statement can appear only between a FROM token and one of ten end tokens. The resulting table reference list is also simple; any literal that does not appear between parentheses is a table name. Our parser also handles subqueries. MySQL allows for one level of subquerying, and subqueries must be SELECT statements. Only SELECT privileges are required for a subquery's tables; the outer statement type does not need to be considered.

Multiplexing Connections.

Restricted connections are typically multiplexed across a single database connection. If the application desires load balancing, the proxy can open connections to different (but identical) database servers and issue statements over the set of connections. If the application uses connection pooling, the proxy can open and maintain multiple connections to a single database server. The proxy then load balances across its real connection pool, and the application's connection pool maintains and distributes restricted connections; our JForum example in Section 4.2 illustrates this. We use code from the `funnel` plugin [17] to perform multiplexing and load balancing.

Restricted connections are not entirely isolated from one another because they share one underlying connection (or set of connections). MySQL database connections have state associated with them. Potential approaches to handling connection-wide state include virtualizing connection-wide state for each restricted connection, disabling all functionality that uses connection-wide state, or letting the powerbox set policies on connection-wide state. Virtualization makes a restricted connection a first class object, capable of everything a normal connection is capable of except when its actions exceed what its policy allows. With the latter two options, restricted connections are not first class objects. It is possible to virtualize state [5], but our current prototype allows only unrestricted connections to change settings that affect connection-wide state.

4. APPLICATIONS

In this section, we discuss the use of data separation in real applications. Our target use case is a program with a small powerbox and functionality that can be separated into relatively independent modules. Capability-secure applications are well-suited to this use case (Section 4.1). We present three web applications and discuss how data separation might be applied to them to improve their security. To demonstrate the benefits of data separation, we retrofit them with Diesel.

4.1 Capabilities

For threat models that include code injection attacks or malicious extension code, we suggest the use of a capability language such as Joe-E [18]. It is difficult to defend against malicious modules (e.g., third-party extensions) in non-capability systems. In a non-capability setting, there is no guarantee that a module cannot obtain a powerful connection even if it is intended to have only a restricted connection. For example, a malicious extension might be able to gain access to an unrestricted connection by accessing a global variable. This makes it difficult or impossible to make guarantees about the access that untrusted extensions have to the database in a non-capability setting.

In contrast, capability systems make it possible to limit and verify which parts of a program have access to which resources [19]. In particular, capability systems give us a way to know for sure that we have limited a module's database access. Capability-safe languages ensure that a module can access a resource (e.g., a database connection) only if the module has a reference to the resource. Thus, if a module has a reference only to a restricted connection, then we know that the module cannot circumvent its policy by accessing a different connection object. If all untrusted code is written in a capability-safe language, then the architecture can defend against malicious code.

Although capabilities have been extensively explored [16, 14, 27, 19], past research to our knowledge has not dealt with the problem of interacting with a database in a capability-friendly manner. Using one capability to represent the entire database violates the principle of least privilege, which capability systems are intended to support; data separation solves this problem. Consequently, database-facing capability-based programs can benefit from the use of data separation. For example, Krishnamurthy et al. [13] implemented a capability-secure web application framework for Joe-E, and they used it to build a webmail service. The capability-secure language enables them to verify an important security property: no user Mallory can access another user Alice's mailbox without knowing Alice's password. Each user's mailbox has its own file system directory on the server, and a capability to a directory provides access only to children of that directory. However, a database might be more appropriate than the file system; with data separation techniques, using a database in a capability setting becomes possible.

4.2 Retrofitting JForum

JForum is a Java message board system that runs several forums with more than 30,000 users each [10]. It is architected as a set of distinct modules, making it a good target for data separation. JForum is an example of a web application that uses connection pooling. We retrofit JForum 2.1.8 to work with Diesel. Figure 3 shows how many lines of code in JForum had to be edited for the implementation.

As an example of data separation, consider JForum's `posts` module. It is the most privileged module, requiring full access to the database tables for forum topics, posts, user votes, etc. Despite its broad privileges, we were still able to restrict its access to sensitive tables like the `jforum_users` table. Thus, we greatly limit the potential damage that a bug in the `posts` module could cause. Unfortunately, JForum does not release vulnerability reports, so we were unable to test our modified implementation on real vulnerabilities.

	modifications	**policy**
JForum	211	162
Drupal	41	1
WordPress	46	6

Figure 3: *Modifications* shows how many lines of code were added/altered. The *policy* column shows how many lines of policy code were used to data separate all JForum modules, one Drupal plugin, and one WordPress plugin.

4.3 Retrofitting Drupal and WordPress

Drupal and WordPress are popular open source content management systems written in PHP. In both platforms, the core program provides functionality for creating and administering websites, and third-party plugins provide additional specialized features. While the core platforms are large projects with security teams, many of the plugins are small projects with less rigorous security reviews. We retrofit Drupal and WordPress with Diesel to limit the database privileges of plugins. Drupal and WordPress could require plugin developers to package a configuration file with their software to identify its required database privileges. This would lessen the impact of a bug in a plugin. In many cases, a plugin may need access only to the tables it creates.

Figure 3 shows how many lines of code were edited or added to refactor Drupal and WordPress. As described below, we wrote a policy for one plugin for each platform; additional plugins can provide policy files that would be honored without any further code modifications.

Drupal Vulnerability.

A flaw in the Brilliant Gallery plugin for Drupal enables an attacker to retrieve the administrative password for the Drupal-powered website [12]. This piece of data is never used by the plugin, so there is no reason for the plugin to have access to it. We refactor the vulnerable versions of Drupal and Brilliant Gallery (versions 5.10 and 5.x-4.1, respectively) to use Diesel. We wrote a one-line policy file for the Brilliant Gallery module that specifies that it needs full access to the `brilliant_gallery_checklist` table, which is the only table it creates. It receives no other database access. With this policy in place, the SQL injection vulnerability post [12] affects only the `brilliant_gallery_checklist` table, so it is no longer a critical vulnerability that can expose administrative credentials.

WordPress Vulnerability.

The WP-Forum plugin adds a forum to a WordPress-powered website. A flaw in this plugin allows an attacker to access the WordPress user account database, which includes administrative credentials [15]. We refactor WP-Forum 2.3 and WordPress 2.7 to use Diesel. We wrote a policy file for WP-Forum stating that it needs full access to the six tables it creates. Due to our default-deny policy, the restricted version of WP-Forum no longer has access to sensitive WordPress databases. The proof of concept SQL injection attack [15] is no longer a critical vulnerability; it can now only affect WP-Forum's database tables.

5. RELATED WORK

User-Based Access Control.

User-based database access control has been studied a great deal [7, 21, 25, 11]. This body of work aims to allow fine-grained access control for different database users. Our work allows fine-grained access control for different modules of a single program, which may be acting on behalf of one or many users.

CLAMP and Nemesis focus on isolating the users of a web application from one another. In CLAMP, database access rights of the application are limited based on the identity of the user currently logged in via SSL [23]. Strong separation is put in place to disallow one user from accessing another user's data. Nemesis similarly addresses the problem of authentication and access control attacks in web applications [3]. Instead of isolating users from one another, data separation aims to limit the access rights of modules within a program. This includes not only user-accessible data, but also data that is stored for the application's own purposes. Additionally, data separation is applicable to all applications that use databases, not just web applications, which are the focus of CLAMP and Nemesis. Our goals are complementary to those of CLAMP and Nemesis.

Redundant Authentication.

The goal of redundant authentication [1] is to reduce the damage that a compromised application server can do to the database. With redundant authentication, every database request issued by an application server must be accompanied by a time-stamped authentication token verifying the user's credentials. A proxy checks the credentials and forwards commands to the database only if the credentials belong to a recently logged-in user. Their attacker is more powerful than ours; we consider only the case of a secure application server running a buggy application. However, data separation offers finer-grained privileges than redundant authentication. With data separation, database access rights are restricted by both the identity of the user *and* the part of the application that issued the command. In addition to shielding the database, data separation also facilitates code review. From an implementation standpoint, data separation does not necessarily require a proxy; it can be implemented in a driver or with wrappers, which is preferable for performance-conscious applications.

SQL Injection Defenses.

SQL injection is a well-studied problem with many workable solutions [9, 29, 2, 8]. While use of our system will mitigate or eliminate the damage caused by many SQL injection vulnerabilities, we emphasize that this is not its only goal. Our goal is broader: to encourage each module of a program to be explicit about the database access it needs to get its job done. This will help programmers and code reviewers better understand the extent to which a given module affects and/or depends on particular sets of data in a database. Data separation protects against a large class of bugs, including but not limited to SQL injection vulnerabilities. For example, web application logic vulnerabilities [4] do not stem from input validation errors.

6. CONCLUSION

We propose *data separation*, the application of privilege separation to database access rights. Privilege separation is a design pattern in which code is separated into functionally independent modules, and each module is given only minimal privileges. In a database-facing application, this means that modules should receive only the database access rights they require. Restricting the database access rights of code mitigates the effects of SQL injection attacks and other bugs that could expose data that the code never needed to access in the first place.

We present a proxy-based architecture for enforcing data separation on the application side without support from the database. The proxy intercepts statements over restricted connections and checks the statements against a module's policy. We demonstrate the effectiveness of our prototype, Diesel, on three popular applications: JForum, Drupal, and WordPress. We added or modified only 211, 41, and 46 lines of code, respectively, to retrofit these applications with Diesel. We show how our modifications to Drupal and WordPress could have mitigated actual attacks on these systems.

Experience with Diesel is encouraging, because it shows that data separation can have security benefits. However, the performance of Diesel leaves considerable room for improvement: our measurements show a significant performance overhead (up to 73%, depending upon the type of query), which may not be acceptable in many application domains. Much of this overhead is due to our use of an alpha release of a non-commercial proxy, and it may be possible to significantly improve performance by using an application-level, non-proxy-based architecture. We hope that these results will motivate further research on data separation and efficient support for data separation.

Acknowledgements

We thank Adrian Mettler and the anonymous reviewers for their helpful feedback on earlier drafts of this work. This work is partially supported by National Science Foundation grants CCF-0424422 and CNS-1018924.

References

[1] J. P. Boyer, R. Hasan, L. E. Olson, N. Borisov, C. A. Gunter, and D. Raila. Improving multi-tier security using redundant authentication. In *CSAW '07: Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 54–62, New York, NY, USA, 2007. ACM.

[2] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*, pages 106–113, New York, NY, USA, 2005. ACM.

[3] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, August 2009.

[4] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *USENIX Security*, 2010.

[5] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner. Diesel: Applying privilege separation to database access. Technical Report UCB/EECS-2010-149, EECS Department, University of California, Berkeley, Dec 2010.

[6] Google. Android Developers: Security and Permissions. `http://developer.android.com/guide/topics/security/security.html`.

[7] P. P. Griffiths and B. W. Wade. An authorization mechanism for a relational database system. *ACM Trans. Database Syst.*, 1(3):242–255, 1976.

[8] W. Halfond and A. Orso. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the IEEE and ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 174–183, Long Beach, CA, USA, November 2005.

[9] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, March 2006.

[10] JForum—Powering communities. `http://www.jforum.net`.

[11] G. Kabra, R. Ramamurthy, and S. Sudarshan. Redundancy and information leakage in fine-grained access control. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 133–144, New York, NY, USA, 2006. ACM.

[12] J. C. K. Keane. [Full-disclosure] Drupal Brilliant Gallery module SQL injection vulnerability. `http://www.derkeiler.com/Mailing-Lists/Full-Disclosure/2008-09/msg00506.html`.

[13] A. Krishnamurthy, A. Mettler, and D. Wagner. Fine-Grained Privilege Separation for Web Applications. In *WWW '10: Proceedings of the 19th international conference on World Wide Web*. ACM, 2010.

[14] C. R. Landau. Security in a secure capability-based system. *SIGOPS Oper. Syst. Rev.*, 23(4):2–4, 1989.

[15] J. G. Lara. Internet Security Auditors Alert: WP-Forum $\leq$ 2.3 SQL Injection vulnerabilities. `http://www.securityfocus.com/archive/1/archive/1/508504/100/0/threaded`, 2009.

[16] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[17] N. Loeve. Funnel - A Multiplexer Plugin for MySQL-Proxy. `https://lists.launchpad.net/mysql-proxy-discuss/msg00030.html`.

[18] A. Mettler, D. Wagner, and T. Close. Joe-E: A Security-Oriented Subset of Java. In *Proceedings of the 17th Annual Network and Distributed Systems Security Symposium (NDSS 2010)*, 2010.

[19] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[20] MySQL Proxy. `http://forge.mysql.com/wiki/MySQL_Proxy`.

[21] L. E. Olson, C. A. Gunter, and P. Madhusudan. A Formal Framework for Reflective Database Access Control Policies. In *CCS '08: Proceedings of the 15th ACM conference on Computer and Communications Security*, pages 289–298, New York, NY, USA, 2008. ACM.

[22] Oracle. Examples of Second Order SQL Injection Attack. `http://st-curriculum.oracle.com/tutorial/SQLInjection/html/lesson1/les0%1_tm_attacks2.htm`.

[23] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical prevention of large-scale data leaks. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.

[24] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[25] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 551–562, New York, NY, USA, 2004. ACM.

[26] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[27] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM Symposium on Operating Systems Principles*, pages 170–185, New York, NY, USA, 1999. ACM.

[28] Sun Microsystems, Inc. Connection pooling, 2008. `http://java.sun.com/developer/onlineTraining/Programming/JDCBook/conpool.html#pool`.

[29] S. Thomas and L. Williams. Using Automated Fix Generation to Secure SQL Statements. In *SESS '07: Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, page 9, Washington, DC, USA, 2007. IEEE Computer Society.