# Diesel: Applying Privilege Separation to Database Access

*Adrienne Porter Felt*
*Matthew Finifter*
*Joel Weinberger*
*David Wagner*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-149
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-149.html

December 8, 2010

# Diesel: Applying Privilege Separation to Database Access

A. Porter Felt Matthew Finifter Joel Weinberger David Wagner

December 8, 2010

**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 design and 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.

## 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 module, the damage that can result is limited to the set of operations the module can perform. This means that a bug in database-facing code (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 will be limited from performing unnecessary operations even if user-based access controls allow the user to perform those actions through another module.

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 [18]. 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 [10]; 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 data separation could be realized using existing database access control, as well as 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 method or 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 a policy 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 [18]. 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 could be repurposed to represent program modules. This is analogous to the Android security model [4], in which Linux users are repurposed to represent applications. This allows permissions to be assigned on a per-module basis, and it works if it is assumed 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 that we wish to data separate. Any modification to a user's permissions would require $n$ changes to a database user's permissions, and any modification to a module's permissions would require $m$ such changes. Additionally, this may be infeasible in an organization in which the application developer and the database administrator are not the same person.

Using database access control 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 from the table.

## 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 can continue to be used as normal. 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

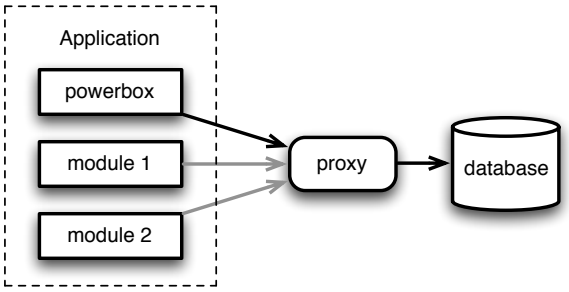



Figure 1: Diesel's architecture. Each module has its own connection object. The powerbox's connection has no data separation policy; it is a regular connection to the database. The powerbox set restrictive policies on the connections for modules 1 and 2. The proxy multiplexes the connections, so only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

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 to be verified 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 with the normal database API, without any extra accommodations for data separation. The powerbox is the only part of the application that needs to be changed, 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 this is mitigated by running the proxy on the same machine as the application. 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.

# 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.

```
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.

## 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 nearly identical, differing 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 which operations are allowed on parts of the database. The operational permissions available in our prototype are SELECT, UPDATE, DELETE, and INSERT. Developers can grant privileges for entire tables or table subsets. Table subsets are implemented 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 (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. The last step is accomplished 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. A policy object can be reused to create additional restricted connections.

**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 the 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). Since the `pareDown` method itself does not have the proper credentials to create a new database connection, it instead makes *proxy credentials* to authenticate the new connection to the proxy. It accomplishes this by generating a random username and sending that to the proxy over the original connection (i.e., the one being pared down). The `pareDown` method then requests (through the database API) a new connection using the randomly generated username. When the proxy recognizes the relationship between the two connections on the basis of their shared secret (the random username), it accepts the incoming connection. After obtaining a duplicate connection in this manner, the `pareDown` method calls `applyPolicy` to add the additional policy to the new connection, and then returns the new connection.

## 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 [16] to handle multiplexing.

**Policy Library Commands** When a policy library applies a policy to a restricted connection, it sends a series of policy setting commands to the proxy. Each restricted connection has its own policy state associated with it. The commands define a policy or start policy enforcement, and they are issued over the restricted connection as if they were normal SQL commands. The database server never sees any of these policy commands. Instead, an appropriate response (confirmation or error) is sent back to the application directly from the proxy as a result set. The policy commands accepted by the proxy are the following:

- `SET POLICY <table/label name> <permission list>`. This command adds an entry to the proxy's map of table/label names and permissions. If the connection already has a policy, any new policy set on it must be a subset of the original policy or else the proxy will reject it.
- `SET SUBSET <label name> QUERY_START <query>`.
  This command tells the proxy to create a database view. If the connection is the pared-down version of another connection, it must have inherited SELECT privileges from the parent for all of the tables referenced in the view definition.
- `START`. This tells the proxy to start policy enforcement. After this command has been issued, all statements issued over the connection are checked against the connection's permissions table.
- `DUPLICATE <username>`. This command is for paring down connections. If a connection attempt is later made with the supplied username, then the new connection is a pared-down version of the connection over which the `DUPLICATE` command was sent. The pared down connection inherits its initial policy from the connection that is duplicated.

**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 [16] 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 in the following ways:

- **Default database.** A database server can have multiple databases, and each database is a separate namespace for table names. If a database name is not included in a query when referencing a table, the default database is used. The proxy stores the default database for each restricted connection and rewrites table names to always be fully qualified, such that there is never a need for the database to actually use the default database.
- **Auto-commit.** If auto-commit is on, each statement is its own transaction. If auto-commit is off, an explicit COMMIT must occur for a transaction's statements to persist in the database. Each connection to the database may have a different auto-commit setting. The proxy can optimistically assume that auto-commit will be on for every restricted connection. If one restricted connection turns auto-commit off, the proxy can spawn a new connection to the database for that restricted connection. (This extra connection would be necessary anyway in an application that requires all modules but one to use auto-commit.)
- **Named cursors.** A named cursor points to a particular point in the result set of a database query, and its scope is a single connection to the database. The proxy can prepend any cursor name with the unique identifier of the restricted connection in order to create a cursor namespace.
- **Buffered result sets.** Query results can be fetched in their entirety, or they can remain on the database server for the application to fetch parts of the results only when necessary. This difference is exemplified by the `mysqli_store_result` and `mysqli_use_result` functions (respectively) in the PHP mysqli API. When a result set is being used (but not stored) by the client, no other query may run over the same connection until the result set is closed. As with the handling of auto-commit, the proxy can open an extra connection to the database when necessary. This will happen if two or more restricted connections simultaneously use a result set without fetching it.

Our prototype handles default databases in the manner described. For all other connection-wide state, our prototype currently 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 [17]. 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 [18]. In particular, capability systems give us a way to know for sure that we have

| | **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.

| | **number of queries** |
|---|---|
| JForum | 275 |
| Brilliant Gallery (Drupal) | 6 |
| WP-Forum (WordPress) | 123 |

Figure 4: The number of lines of code that issue queries to the database in JForum and the two plugins. All of the variables used to create these queries would need to be audited to check their correctness.

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 [15, 13, 27, 18], 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. [11] 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 [8]. It is architected as distinct modules with separate functionality, 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.

In our data-separated version of JForum, the connection pooling class acts as the powerbox. It initializes the proxy with multiple database connections and then populates its pool with restricted connections with allow-all policies. When the JForum servlet receives a new page request, it asks the connection pool for a connection to give to the currently running module. The connection pooling class chooses a connection from the pool, pares it down with the policy for the module, and gives the new restricted connection to the servlet. The module then uses the restricted connection to service all of its database requests.

In order to implement this, we altered the connection pool's `getConnection` method to take the name of the currently running module as a parameter, look up the policy, and pare down restricted connections. The `getConnection` method is used in other places (e.g., installation) where we do not want to distribute a restricted connection; we special case these situations so that a full-privilege connection is returned. Figure 3 shows how many lines of code in JForum had to be edited for the implementation.

We wrote policies for modules based on the Data Access Objects (DAOs) that they use. Each JForum module uses a set of DAOs to access the database instead of issuing database statements directly. We define a policy for each DAO, and each module's policy is the sum of that module's DAO policies. For example, the private message module uses `PrivateMessageDAO` and `UserDAO`, so the private message module's policy is the combination of those two modules' privileges. Since each DAO is reused by many modules, this reduces the number of policies. However, this might result in more permissions than a module actually needs because some modules don't use the full functionality of each DAO they utilize. Our policies are therefore slightly coarser than required, but this approach still greatly reduces the data privileges given to modules. We wrote policies for all of the modules except for those used solely for administrative actions.

As an example of the usefulness of data separation, consider the `posts` module. It is the most privilege-hungry module, requiring full access to the database tables for forum topics, posts, and supplemental functionality such as user votes. Despite its broad privileges, we were still able to restrict its access to sensitive tables such as 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 and their online bug-tracking software was offline when we attempted to access it, so we were unable to test our modified implementation on real vulnerabilities.

### 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 identity 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.

The architectures of Drupal and WordPress differ from our target application architecture. In the intended use case for data separation, we envision a trusted powerbox passing a restricted connection object to a less-trusted module in a capability-like fashion. Drupal and WordPress, however, are engineered in the opposite way. Plugins (less-trusted modules) pass the core (the trusted powerbox) queries, which the core then issues on behalf of the plugin. We therefore must take a stack inspection-based approach [31], where the core checks to see who the caller is and then issues the statement over the appropriate restricted connection.

In Drupal, all database access goes through the library function `_db_query`, which sends the input query to the database on behalf of its caller. We modify it to determine the identity of the caller and execute the query over the restricted connection associated with that caller. First, our modified `_db_query` function checks whether the caller is a function in a file with the file extension `module`. If so, the caller is a plugin and it will use a restricted connection for the request. If a restricted connection does not yet exist for the module, it will look for the policy file and create a new restricted connection by paring down the active database connection with the input policy. WordPress works identically; all database access goes through the `query` method of the global `wpdb` object, and we modify this method to perform stack inspection and apply a policy based on a configuration file. Figure 3 shows how many lines of code were edited or added to perform this refactoring. 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 [10]. 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 detailed in the Full Disclosure post [10] 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 [14]. 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 [14] is no longer a critical vulnerability; it can now only affect WP-Forum's database tables.

### 4.4 Manual Auditing

Instead of using data separation to prevent bugs, a reviewer could audit all of the SQL queries to ensure their correctness. In order to do this, the reviewer would need to find all of the places where SQL queries are issued and then check whether the statement is being properly constructed and prepared. This can be a non-trivial task, since query strings are often concatenated from multiple substrings; the reviewer would also need to check the origins of all of the substrings. For example, the query that makes WP-Forum vulnerable to attack is constructed in 31 lines from several substrings and user-supplied variables.

We searched through the source code of JForum, Brilliant Gallery (the Drupal plugin), and WP-Forum (the WordPress plugin) to find places where SQL queries are being issued. Figure 4 shows the results. For each one of these queries, a reviewer would need to check that the query is being properly prepared. We believe it is simpler and easier to write, for example, 6 lines of policy code than it is to check 123 SQL queries for correctness.

## 5 Performance

We quantify the overhead that Diesel imposes on the running time of database queries. We measure the following:

1. **Overhead on real queries.** How much overhead is imposed by Diesel on queries issued by real applications?
2. **Overhead vs. number of policy statements.** How much does Diesel's overhead increase as more policy statements are applied to a restricted connection?
3. **Overhead vs. result set size.** How does the overhead imposed by Diesel scale with the size of the result set returned by a query?
4. **Overhead for concurrent queries.** Does the proxy create a bottleneck for concurrent queries?

There may be other overheads associated with Diesel, such as the cost of opening restricted connections, but we believe this affects end-to-end performance less than the overhead of query execution.

### 5.1 Methodology

We used Python with the `MySQLdb` module to implement our performance tests. All tests were run on an Apple MacBook Pro with a 2.66 GHz Intel Core 2 Duo processor and 4 GB RAM, running Mac OS X. The MySQL query cache was disabled to ensure that each query was actually executed by MySQL. For all tests, we performed 110 trials. The first 10 trials were discarded in all tests in order to "warm" the proxy and the DBMS. We report the results of the remaining 100 trials. Our methodology is as follows:

**Overhead on real queries** We used 5 queries from JForum and 3 from Drupal. The 5 JForum queries consist of three SELECT queries and two INSERT queries, with policies applied to all of them. One Drupal query is the SELECT statement issued by the Brilliant Gallery module that had a policy applied to it. The other two Drupal queries (an UPDATE and a SELECT, respectively) were issued by the Drupal core, so no policy was applied to either of these. For each query, we took measurements of the same query executed 100 times in a row (a) without Diesel, (b) with Diesel, but no policy, and (c) with Diesel, and with the policy enforced. The latter was omitted for the 2 Drupal queries that had no associated policy. The actual queries used are listed in Appendix A.

**Overhead vs. number of policy statements** We measured how the Diesel overhead changed as we added more policy statements to the policy of a restricted connection. This corresponds to how many permissions a given statement is checked against. We measured the run time of a simple query (executed 100 times in a row) over the `world` sample database provided by MySQL [19], using the query `SELECT * FROM City;`. This sample database has 4079 records. We added increasing numbers of frivolous policy statements to the restricted connection used to issue the test SELECT query; each new policy statement gave access to an invented table. (As the proxy has no notion of which table names actually exist in the database, it cannot summarily ignore these restrictions.)

**Overhead vs. result set size** We added increasing numbers of randomly generated records to an otherwise empty table. For each number of records, we measured the time for a SELECT query to request all the records (100 times in a row) both with and without Diesel.

**Overhead for concurrent queries** Varying the number of concurrently executing threads from 1 to 20, we recorded the start time and end time of each of the 10 sequential queries executed by each thread. We calculated the average latency of all queries that both started and finished during the time period in which all threads were executing concurrently (i.e., between the latest start time of any thread's first query and the earliest end time of any thread's last query). The query used for this test was the same SELECT query as for the "Overhead vs. number of policy statements" test.

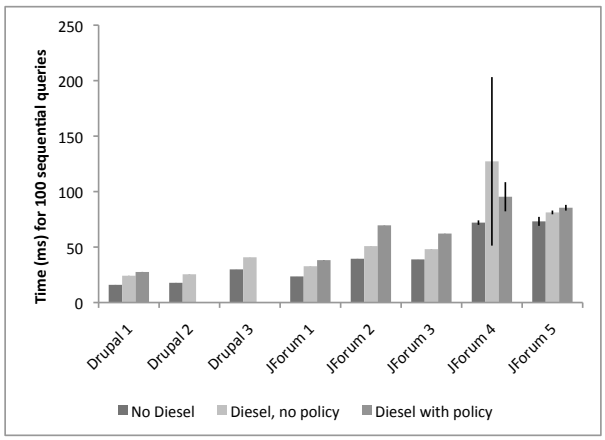
## 5.2 Results

Our performance results are presented in Figure 5. The error bars in all figures represent a 95% confidence interval. Where error bars are not visible, it is because of their proximity to the mean.

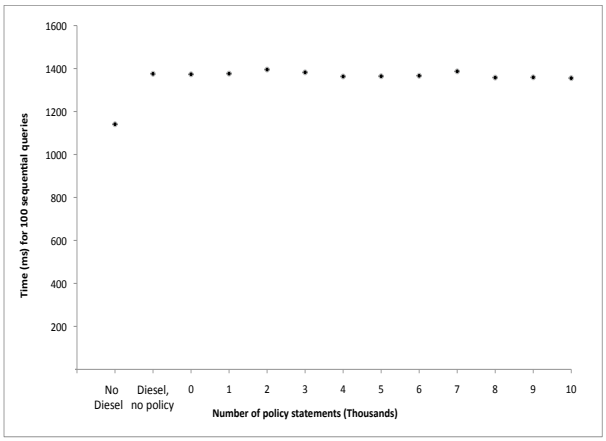
**Overhead on real queries** The overheads observed on the Drupal and JForum queries are presented in Figure 5(a). JForum queries 1-3 are SELECT statements, and 4 and 5 are INSERT statements. Drupal 1 and 3 are SELECTs, and Drupal 2 is an UPDATE statement. Note that the large confidence intervals for "JForum 4" are due to a single outlier.[1] If we exclude the anomalous trial, the mean execution times for "JForum 4" are the following: 72.1 ms (No Diesel), 88.6 ms (Diesel, no policy), and 88.8 ms (Diesel with policy). These are the numbers we consider in our following discussion. The median execution times for "JForum 4" are 70.4 ms, 80.3 ms, and 85.5 ms, respectively.

Figure 6 shows the slowdown for each of the queries. The proxy alone adds between 11% and 51.4%. The total overhead due to Diesel with enforcement ranges from 16.8% to 72.6%. Our system adds less overhead for our INSERTs than it does for our SELECTs.

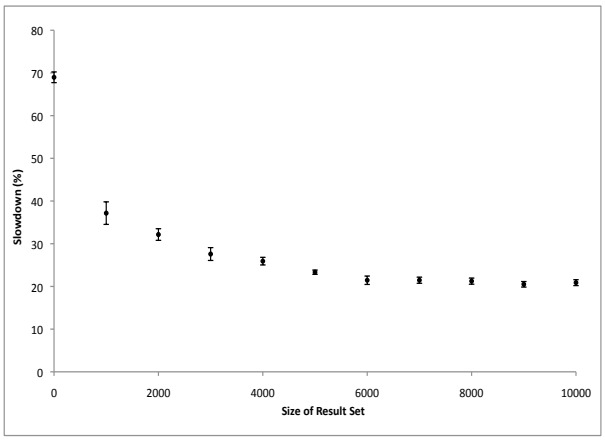
[1]This anomalous trial took 69 ms without Diesel, 3953 ms with Diesel, and 740 ms with Diesel plus policy enforcement. This outlier could have been caused by another process's resource consumption, index updates in the DBMS, or some other factor. We did not see any such large outliers in subsequent trials, although we do note that mild outliers (on the order of 2 times the median) were seen for inserts both with and without Diesel running. We suspect that these are due to database index reorganization, but we have not been able to confirm this.



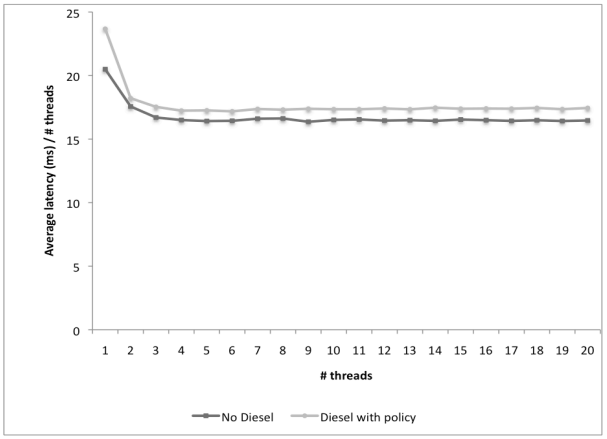

(a) Time for queries from Drupal and JForum




(b) Time vs. number of policy statements




(c) Slowdown vs. result set size




(d) Avg. time vs. number of concurrent threads

Figure 5: Performance results

**Overhead vs. number of policy statements** The results from this test are presented in Figure 5(b). The overhead due to Diesel remains constant as the number of policy statements (i.e. the complexity of the policy) on a restricted connection increases.

**Overhead vs. result set size** We present this set of findings in Figure 5(c). The greatest slowdown occurs with small result sets. The slowdown trends toward around 20% as the result set size increases.

**Overhead for concurrent queries** Figure 5(d) presents the results from this test as the average latency divided by the number of threads vs. the number of concurrently executing threads. This allows us to see that as the number of threads increases, Diesel adds an approximately constant overhead for each thread. That is, the percent overhead due to Diesel remains constant (at approximately 6%) as the concurrency level increases. The average latency responds to increased concurrency similarly both with and without Diesel.

## 5.3 Discussion

Figures 5(a) and 6 show the slowdown for the queries from Drupal and JForum. A significant amount of Diesel's overhead (in some cases, nearly all of it) is due to the overhead introduced by MySQL Proxy, rather than our policy enforcement mechanism. This is evidenced by the differences in slowdown between (a) the

| | Proxy | Policy | Total |
|---|---|---|---|
| **Drupal 1** | 51.4% | 21.17% | 72.6% |
| **Drupal 2** | 43.1 | N/A | 43.1 |
| **Drupal 3** | 36.5 | N/A | 36.5 |
| **JForum 1** | 39.0% | 23.4% | 62.3% |
| **JForum 2** | 28.8 | 47.5 | 76.3 |
| **JForum 3** | 23.7 | 36.2 | 60.0 |
| **JForum 4** | 23.0 | 0.329 | 23.3 |
| **JForum 5** | 11.0 | 5.79 | 16.8 |

Figure 6: Slowdown percentages observed for queries from Drupal and JForum. The first column of numbers represents the average slowdown due to the proxy alone (i.e. no policy enforcement), calculated as $100 \cdot (t' - t)/t$, where $t$ is the mean time taken without Diesel, and $t'$ is the mean time taken with Diesel, but no policy enforcement. The next column represents the average *additional* slowdown due to policy enforcement, calculated as $100 \cdot (t'' - t')/t$, where $t$ and $t'$ are as before, and $t''$ is the mean time taken with Diesel with policy enforcement. The final column is the average total overhead, calculated as the sum of the previous two columns.

tests that use the proxy without any policy, and (b) those that use the proxy with a policy. Our prototype is based upon an alpha release of MySQL Proxy, which has considerable room for improvement. For example, this release is single-threaded. Performance-critical applications could forgo the proxy and instead take an application-level approach (Section 2.3). Another consideration is that we have not experimented with a non-local DBMS. We expect the relative slowdown due to Diesel to decrease in this case, perhaps substantially, due to the increased latency for each query.

We have not investigated how performance is affected by the number of tokens or table names in the query. We expect more complex queries to have slightly increased overhead, since the policy enforcement code has to parse the queries to identify table names. We do not expect to see a significant increase, however, since we use a simple parsing scheme that does not need to fully parse the entire statement (Section 3.2).

We also have not investigated the performance of label-based policies since they were not necessary for our applications. These label-based policies depend on database views, so their performance is heavily dependent upon MySQL's performance with views. We may be able to leverage materialized views (views that are written to disk) to speed up queries over restricted connections with label-based policies. The cost would be in disk space, and depending on typical workloads, the materialized views may need to be refreshed frequently. Parno et al. describe issues in using views to enforce security policies in their system, CLAMP [23]. They note that a bug in MySQL causes cached view-based queries to never hit in the cache, which negatively impacts the performance for these queries. Additionally, MySQL places some restrictions on updates to views, which can limit the set of expressible policies. Our prototype Diesel faces these limitations as well, but we do not view them as problems fundamental to the concept of data separation.

Our other tests (Figures 5(b), 5(c), and 5(d)) indicate that Diesel scales well with respect to the complexity of policies and the size of returned result sets, and that Diesel does not create a performance bottleneck for queries running concurrently. We note that the relative slowdown due to Diesel decreases as result set sizes grow larger. The queries with the largest result sets show a slowdown of about 20%. We stress that we are not claiming performance gains with larger result sets, but merely less severe relative slowdowns.

The end-to-end performance impact of Diesel on real applications is what really matters. This will depend on what fraction of processing is spent on database queries as opposed to other application tasks, and this can vary greatly. We are unable to meaningfully report this figure: our example applications spent so little time making database queries that the end-to-end performance overhead, as a fraction of total processing time, is likely to be too small for a meaningful comparison. In future work, it would be useful to examine applications that are more reliant on large numbers of queries in order to measure the end-to-end performance of those cases.

# 6 Related Work

**User-based access control** User-based database access control has been studied a great deal [5, 21, 25, 9]. 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. We view our goal as complementary to that of CLAMP or 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 [7, 29, 2, 6]. 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.

**Operating systems** Asbestos [30] and HiStar [32] are operating systems designed according to the principle of least privilege. These systems provide primitives for controlling information flow between programs. They enforce Mandatory Access Control (MAC) for the policies specified by an application. Flume [12] accomplishes similar goals, but it is implemented as a user-level reference monitor in Linux instead of as a completely new operating system. These systems control information flow. Data separation instead focuses on separating (within an application) access to the data residing in a DBMS. Our policies are constructed by specifying permissible operations for certain pieces of data. Data separation can be implemented using a new database API, a wrapped database API, or a proxy; it does not require low-level system primitives, a reference monitor, or operating system changes.

# 7 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. This prototype is named Diesel, and we demonstrate its use in 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.

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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`.

[11] 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.

[12] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 321–334, New York, NY, USA, 2007. ACM.

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

[14] 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.

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

[16] N. Loeve. funnel - a multiplexer plugin for mysql-proxy. `https://lists.launchpad.net/mysql-proxy-discuss/msg00030.html`.

[17] 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.

[18] 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.

[19] MySQL :: Setting Up the world Database. `http://dev.mysql.com/doc/world-setup/en/world-setup.html`.

[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/les01_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.

[30] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[31] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for Java. *SIGOPS Oper. Syst. Rev.*, 31(5):116–128, 1997.

[32] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI '06: Proceedings of the 7th symposium on Operating Systems Design and Implementation*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

# A Queries used in performance test

| Query ID | Query |
|---|---|
| Drupal 1 | SELECT qid FROM brilliant_gallery_checklist WHERE qid LIKE 'user-%' AND state='1' |
| Drupal 2 | UPDATE users SET access = 1250790538 WHERE uid = 1 |
| Drupal 3 | SELECT u.*, s.* FROM users u INNER JOIN sessions s ON u.uid = s.uid WHERE s.sid = 'cd00d4f59c32ef264d27178c43655305' |
| JForum 1 | SELECT forum_id, forum_name, categories_id, forum_desc, forum_order, forum_topics, forum_last_post_id, moderated FROM jforum_forums ORDER BY forum_order ASC |
| JForum 2 | SELECT p.post_id, topic_id, forum_id, p.user_id, post_time, poster_ip, enable_bbcode, p.attach, enable_html, enable_smilies, enable_sig, post_edit_time, post_edit_count, status, pt.post_subject, pt.post_text, username, p.need_moderate FROM jforum_posts p, jforum_posts_text pt, jforum_users u WHERE p.post_id = pt.post_id AND topic_id = 2 AND p.user_id = u.user_id AND p.need_moderate = 0 ORDER BY post_time ASC LIMIT 0, 15 |
| JForum 3 | SELECT pm.privmsgs_type, pm.privmsgs_id, pm.privmsgs_date, pm.privmsgs_subject, u.user_id, u.username FROM jforum_privmsgs pm, jforum_users u WHERE privmsgs_to_userid = 4 AND u.user_id = pm.privmsgs_from_userid AND ( pm.privmsgs_type = 1 OR pm.privmsgs_type = 0 OR privmsgs_type = 5) ORDER BY pm.privmsgs_date DESC |
| JForum 4 | INSERT INTO jforum_posts_text ( post_id, post_text, post_subject ) VALUES (<count>, <random-100>, <random-20>) |
| JForum 5 | INSERT INTO jforum_privmsgs_text ( privmsgs_id, privmsgs_text ) VALUES (<count>, <random-100>) |

Table 1: Queries from JForum and Drupal used for performance testing. <count> means that the value was incremented each time the query was executed, starting from 0. <random-n> means that an n-character random value was used.