Diesel: Applying Privilege Separation to Database Access

Talk Notes, AsiaCCS 2011

Matthew Finifter
finifter@cs.berkeley.edu

March 23, 2011

Introduction

The problem I will be discussing today is that database-backed applications tend to be designed such that the entire database is accessible by any part of the application. I'll explain why this is important by relating this problem to previous work on privilege separation.

From previous work, we know that it makes sense to design, for example, an SSH daemon in such a way that its private key is accessible only to the program module that needs to access it. This is good for security because a flaw or vulnerability in a module that can't access the private key cannot possibly leak it.

But private keys are, of course, not the only things of value that we do not want leaked. User data stored in a database is often valuable, and should be protected where possible. We propose what we term *data separation*, the idea of separating access to database data amongst different program modules. In particular, we propose designing applications such that each program module has access to the smallest subset of database data that it needs in order to do its job.

This design pattern provides defense in depth for data stored in a database, just as privilege separation provides defense in depth for things like private keys, network access, and `passwd` files.

Talk outline

I've already introduced the problem we address. Next, I will discuss some of the concrete benefits of data separation as well as some specific scenarios in which we envision data separation being particularly useful. Then, I'll talk about the design of a system for data separation and our implementation of a data separation framework, which we have named Diesel. Finally, I'll discuss our experience using Diesel on several applications.

Benefits of data separation

The primary benefit of data separation is that it provides an additional line of defense against software defects, including both vulnerabilities and logic flaws. A SQL injection vulnerability in a module with access to very little data, such as a module that keeps track of users' last access time, is not a critical vulnerability. Furthermore, this means that a security review need not spend as much time and effort on every module. It can focus first on the modules that have access to critical data before considering those modules that have access to less critical data or no data at all. In short, if a module can't access data, it can't leak the data and the integrity of the data cannot be affected. Developers and security reviewers can quantify the worst case scenario in terms of how much data can be leaked or corrupted on a per-module basis.

Use cases for data separation

One use case for data separation is for software written using a capability-secure language, such as E, Joe-E, or Emily. Capability-secure languages make the privileges of each code module explicit by representing all powerful operations as capabilities that must be explicitly transferred from one module to another. With data separation, fine-grained subsets of data can be passed as capabilities from one module to another.

Another use case we envision is for web applications. It is common to find web applications that connect to a database server as exactly one user. That is, it is not the case that a separate database user is created for each user of the web site; rather, the web site itself is the database user. Our data separation framework allows the web application to reduce database privileges on a per-user or per-module basis, or a combination of the two.

A third use case we envision for data separation is secure extensibility. Consider a program designed to allow for third-party add-ons. We have personally seen several instances in which the security of the core program is held to a much higher standard than the security of a third-party add-on. With data separation, the author of an add-on can declare exactly the set of data needed by the add-on, and the core program can give the add-on database access to only that set of data. Users installing the add-on can know exactly what the worst-case scenario is should the add-on be found to be vulnerable, and our experience shows that this worst-case scenario is almost never compromise of the entire set of data.

Terminology

I'll talk now about some of the details of the design pattern of data separation.

I'd like to introduce the notion of *restricted connections*. A restricted connection is a connection to the database that is limited to a subset of data by a policy that has been applied to it. A *data separation framework* like the one we built provides a policy-setting API and a mechanism for enforcing policies on restricted connections.

Using restricted connections

A developer using a data separation framework can expect to follow a particular workflow. One small, powerful program module, which we call the powerbox, has a full unrestricted connection to the database. The developer defines policies and creates restricted connections with these policies before distributing the restricted connections to the program modules. A module that receives a restricted connection can use it just like a regular database connection. In other words, it does not need to be aware that it has received access to only a subset of the data in the database.

Paring down a connection

We borrow the concept of *paring down* a connection from the capability literature. A module can apply an additional policy to a restricted connection in order to create a new restricted connection with access to an even smaller subset of data. Using this mechanism, a module can apply a fine-grained policy when deciding on the database access to grant each sub-module. This hierarchical mechanism allows for fine-grained distribution of data amongst program modules and sub-modules, as well as incremental deployability for the data separation paradigm. An initial deployment of data separation might use a coarse-grained separation of data, while a second round of applying data separation might dig into the details of each module, paring down its restricted connection for every individual method call in the module.

Diesel Architecture (1)

We implemented a prototype data separation framework called Diesel. Diesel operates completely on the application side, without any support from the database management system. We use a proxy-based architecture in order to be language-neutral. The proxy accepts policy-setting commands, and it enforces policies on restricted connections.

Diesel Architecture (2)

Each connection from the application is a connection to the proxy, which is itself connected to the database. The powerbox is the only module with a full, unrestricted database connection. Other modules have restricted connections created and given to them by the powerbox.

Policies

A policy specifies the set of operations that can be performed on each table of the database. Additionally, a policy can specify privileges on subsets of a database table using database views, which are virtual tables defined by an arbitrary query. Our prototype supports policies that specify a subset of SELECT, INSERT, UPDATE, and DELETE privileges for each table or view. Once a policy has been set on a connection, it cannot be removed. This limits a module's restricted connection to the policy applied by whomever created it.

Proxy

The proxy is implemented as a plugin to MySQL proxy. It accepts policy-setting commands and maintains state for each restricted connection to keep track of the active policy for it. Non-policy-setting commands are checked against the active policy, and, if acceptable according to the policy, passed on to the database, from which a result set is returned to the proxy and then to the client.

The proxy checks statements against the active policy by parsing the statement to find the operations (like, for example, SELECT, or UPDATE) and the table names that are mentioned. If it sees a query that is disallowed by the policy, it simply does not forward the query to the database server, and it returns a null result set to the client.

Experience

Now I'll talk about our experience using data separation for some actual applications. We retrofitted three applications to realize security benefits from data separation.

The first one I'll discuss is JForum. JForum is a popular web application written in Java for building a forum. JForum was architected as distinct modules, so we data-separated it on a per-module basis. For example, the `posts` module, which handles users' posts to the forum, required a lot of privileges. In fact, it required access to more tables in the database than any other module. Despite this, it did not need access to the `jforum_users` table, which holds personal information about users of the forum. Due to the use of data separation, a flaw in this module cannot result in leakage or corruption of the users table.

Next, I'll discuss Drupal and Wordpress. Both of these are web applications written in PHP. Drupal is a content management system, and WordPress is a blogging platform. Both are built around an architecture that has a core that is extensible using third-party extensions.

The Drupal plugin called Brilliant Gallery had a critical SQL Injection vulnerability that could give an attacker complete database access. After retrofitting Drupal with Diesel in order to apply the data separation pattern, this was no longer the case. The Brilliant Gallery module needed access only to a single database

table that it created, so that's what we gave it. After our retrofitting, the same SQL injection vulnerability is no longer a critical vulnerability; it allows an attacker to read or change properties of a photo gallery, and nothing more.

Similarly, the WordPress plugin called WP-Forum had a vulnerability that allowed an attacker to gain administrative access to the WordPress-powered site. Again, what was previously a critical vulnerability can now do very little harm after being refactored to use data separation.

Conclusion

In conclusion, we have introduced the notion of data separation, which says that each program module should have access only to the data it needs to get its job done. We have implemented a prototype data separation framework called Diesel, which is language-neutral but specific to MySQL databases. We have evaluated our system by refactoring several applications to use Diesel and showing that the severity of vulnerabilities is significantly reduced when we do so. Data separation provides defense-in-depth for important data that we do not want to be leaked or altered. There are many more details in the paper than I was able to fit into this talk, so I encourage you to read the paper if you are interested. Thank you.