Protecting PostgreSQL Against SQL Injection Attack

An EnterpriseDB White Paper

For DBAs, Application Developers, and Enterprise Architects
March 2010

EnterpriseDB®
The Enterprise PostgreSQL Company

www.enterprisedb.com

Table of Contents

Introduction

A number of recent high profile SQL injection attacks against prominent organizations have reminded data management professionals how quickly a serious data breach can happen and the consequences that come with being a victim. For example, in November 2010, the British Royal Navy website was compromised with a SQL injection attack that cost the organization its list of usernames and passwords. In December 2010, thieves used a SQL injection attack against the CitySights New York tour company to steal over 100,000 bankcard numbers.

In 2010, the Verizon Business Payment Card Industry Compliance Report stated that SQL injection attacks were the number two cause of payment card breaches (the number one cause was backdoors). And according to Cisco systems, SQL injection attacks comprise 37% of all events recorded by Cisco Remote Operations Services.

Because a database professional's number one job is to protect the data, it's important that everyone takes the threat of SQL injection attacks seriously and do what is possible to prevent them. This paper discusses the methods of SQL injection attacks, describes general ways to prevent them, and provides a look at EnterpriseDB's SQL/Protect module that provides built-in protection for PostgreSQL servers against SQL injection attacks.

What is a SQL Injection Attack?

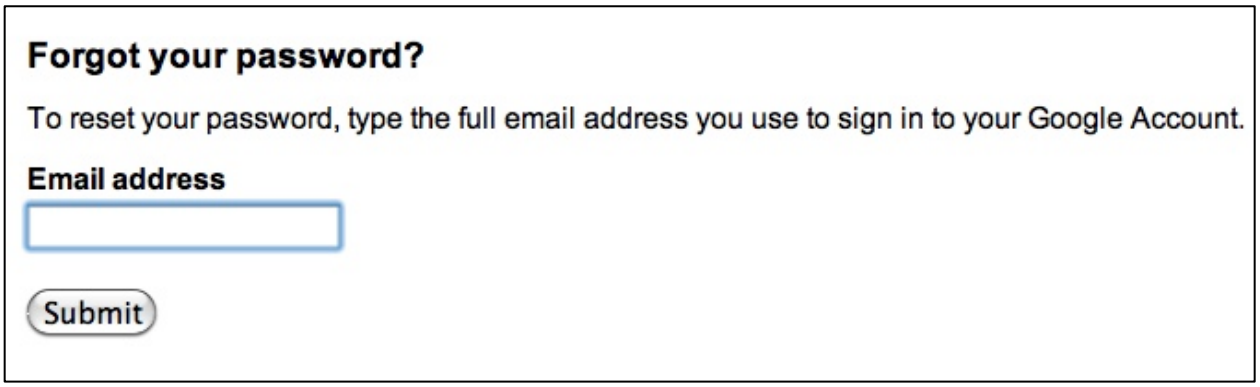
SQL injection refers to the act of a data pirate or vandal inserting a Structured Query Language (SQL) statement through some open door (e.g. a field on a PHP application/web form), which is run in an unauthorized fashion on a database server. A SQL injection attack is basically un-sanitized, unintended SQL-based user input with the goal being either the acquisition of information from the database or the destruction of data.

Because most every Web-based/online application has a relational database back end, a SQL injection attack is an easy way for thieves and vandals to gain access and compromise a database since the online application itself provides their window of opportunity.

An Example of a SQL Injection Attack

A SQL injection attack is typically launched through a web form that asks for user input, such as a login name and password, an email address, or other very common information. By entering in what amounts to a custom SQL `WHERE` clause, an unauthorized user can potentially see table data or have DML or DDL commands executed.

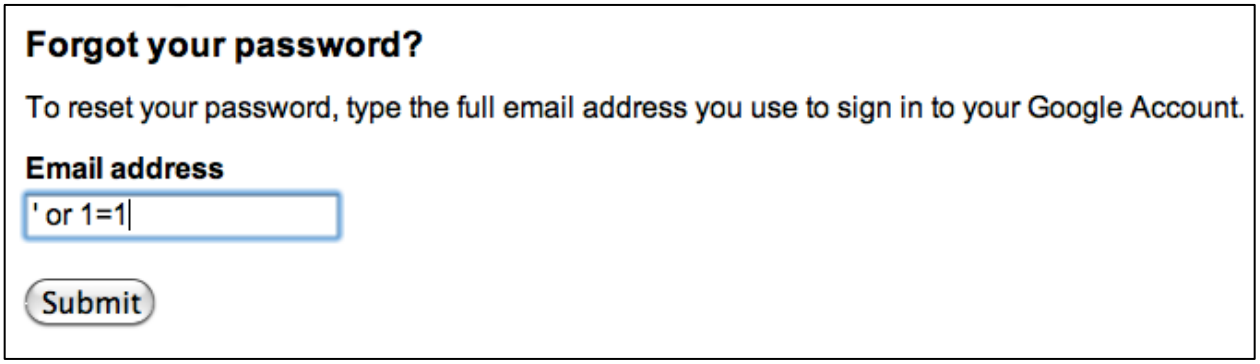
For example, a very common web form is one that asks a user to enter their email address so that user login information can be sent to them.



It's standard practice for an application to validate that an email address actually exists before it goes to the trouble of trying to email user data to someone, and this normally equates to performing a SQL query against a table that contains email addresses. A query like the following might be used:

```
SELECT USERID, PASSWORD
FROM   USERLOGIN
WHERE  EMAIL_ADDRESS = 'inputted email address';
```

Someone attempting a SQL injection attack might try entering a single quote character in the online form's email address control and typing in a tautology `WHERE` predicate:



Whereas an empty email address string would return no rows from PostgreSQL:

```
dev=# select * from userlogin where emailaddress = '';
 userid | password | emailaddress
--------+----------+--------------
(0 rows)
```

The end result of the tautology predicate above is that a query like the following is sent to the database:

```
SELECT USERNAME, PASSWORD
FROM   USERLOGIN
WHERE  EMAIL_ADDRESS = '' or 1=1;
```

Such a query would return all the rows back in the database's table and could potentially expose information to the SQL injection attacker:

```
dev=# select * from userlogin where emailaddress = '' or 1=1;

   userid    | password |     emailaddress
-------------+----------+----------------------
 johnsmith   | password | johnsmith@gmail.com
 janedoe     | password | janedoe@gmail.com
 timjohnson  | password | timjohnson@gmail.com
(3 rows)
```

Potentially even worse, if a data vandal learns the name of some of the underlying database tables, and security has not been set up properly in the database, they could enter a combination of single quotes and semi-colons (which terminate a SQL statement) so that a query set like the following is passed through:

```
SELECT USERNAME, PASSWORD
FROM   USERLOGIN
WHERE  EMAIL_ADDRESS = ''; DROP TABLE USER_INFO;
```

The end result, needless to say, is not something the DBA wants to have happen:

```
dev=# select * from userlogin where emailaddress = '';drop table
userlogin;

 userid | password | emailaddress
--------+----------+--------------
(0 rows)

DROP TABLE

dev=# select * from userlogin;
ERROR:  relation "userlogin" does not exist
LINE 1: select * from userlogin;
```

Protecting PostgreSQL Against SQL Injection

Protecting PostgreSQL from data pirates and vandals who attempt to use SQL injection attacks against a database doesn't have to be a difficult task. There are generic safeguards that can be implemented by the developer and DBA, which will close a number of potential openings.

However, in addition to generic best practice standards, another way of protecting PostgreSQL servers against SQL injection attacks is offered by EnterpriseDB in the form of its SQL/Protect module. SQL/Protect is a built-in SQL firewall that is both flexible and thorough, and guards against many different types of SQL injection attacks.

General Safeguards

A succinct list of standard defenses that a developer and/or a DBA can implement include the following:

- Sanitize the user input before it's sent to the database using either functions supplied by the database or application language, with the goal being to strip out any quotation marks, semi-colons, and the like.
- The use of prepared statements can make many SQL injection attacks fall flat, with the reason being there is no SQL parsing and the input is treated as just data coming in.
- Requests sent through web/online forms should be granted the most minimal security privileges possible, which helps remove the fear of any unauthorized DML or DDL requests being executed.

Using SQL/Protect from EnterpriseDB

One of the best ways to ensure proper protection against SQL injection attacks is to implement a SQL firewall inside a database that automatically guards against any malicious attempts at accessing or damaging data. This becomes very easy for PostgreSQL servers when SQL/Protect from EnterpriseDB is utilized.

SQL/Protect provides a DBA-managed layer of security in addition to normal database security policies by screening incoming queries for common SQL injection profiles. In addition, SQL/Protect can also be taught to accept learned 'friendly' queries and reject unfamiliar data request patterns. Everything is controlled and monitored by the DBA through simple database commands.

The following sections walk through the process of getting started with SQL/Protect and explain how SQL/Protect establishes itself as a solid SQL firewall against SQL injection attacks.

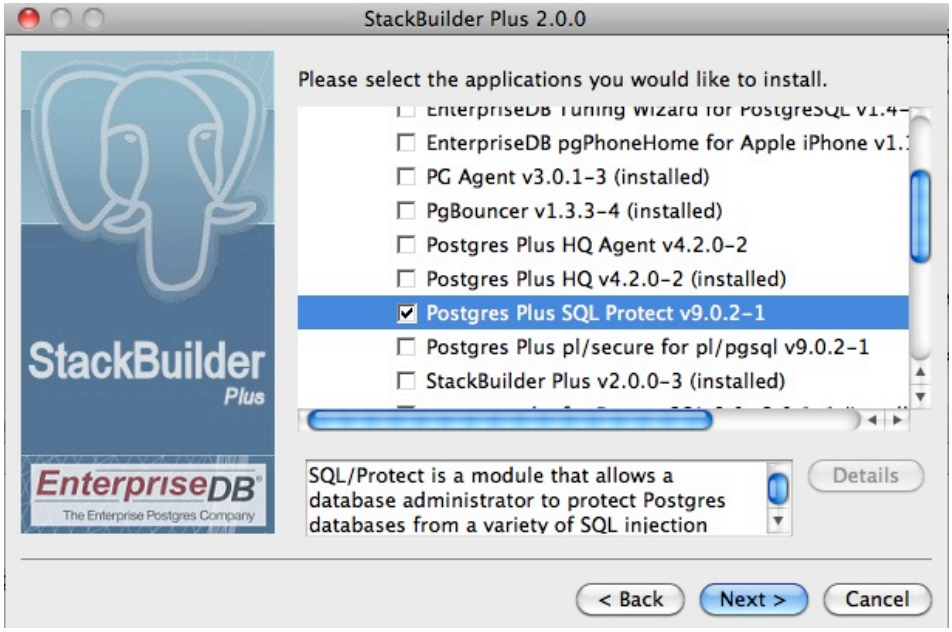
Getting Started with SQL/Protect

Getting started with SQL/Protect is easy. The first step is to download a version of either Postgres Plus Standard Server or Postgres Plus Advanced Server from the EnterpriseDB website and use the software's graphical installer, which installs the PostgreSQL server, all

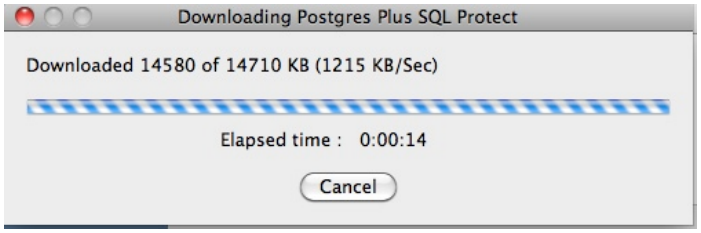
its dependencies, and a number of ancillary pieces of software (e.g. the GIS module for spatial development).

One piece of software installed is StackBuilder Plus, which provides an easy way to install optional community sets of software for PostgreSQL as well as EnterpriseDB supplied modules.

To install SQL/Protect, the DBA invokes StackBuilder Plus and selects the SQL/Protect option:



Once SQL/Protect is selected and the Next button is clicked, StackBuilder Plus will download the SQL/Protect software:



Once the download is complete, StackBuilder Plus will indicate it is finished, and then proceed to the installation panel for SQL/Protect:

The installation wizard will ask the DBA to log into their EnterpriseDB account, validate the directory the software will be installed into, and finally SQL/Protect will be installed.

Configuring SQL/Protect

Once SQL/Protect has been installed, there are a few parameter changes that need to be made to PostgreSQL's `postgresql.conf` configuration file. The DBA needs to edit two existing parameters to include the following variables:

```
shared_preload_libraries = '$libdir/sqlprotect'
custom_variable_classes = 'edb_sql_protect'
```

Once they parameters above have been modified, two new lines need to be added to the `postgresql.conf` file:

```
edb_sql_protect.enabled = on
edb_sql_protect.level = learn
```

The `edb_sql_protect.enabled` parameter controls whether SQL/Protect is actively monitoring the server for SQL injection attacks. The `edb_sql_protect.level` parameter manages the level of protection for the server, and will be covered in more detail in the section.

There are two optional parameters that can be set. The first is `edb_sql_protect.max_protected_roles`, which sets the maximum number of roles that

can be protected. If the parameter is omitted, the default setting is 64. The second parameter is `edb_sql_protect.max_protected_relations`, which sets the maximum number of relations (tables) that can be protected per role. If this parameter is explicitly set, the default setting is 1024.

Once the modifications have been made to the `postgresql.conf` file, the PostgreSQL server needs to either be stopped and restarted for the changes to take effect, or the DBA needs to issue a reload configuration command (`pg_ctl reload`) to have the configuration file reloaded.

Determining How to Protect the Server

The next step in using SQL/Protect is determining how to protect the PostgreSQL server against SQL injection attacks. The first decision revolves around which databases need protection.

For each database on the server that the DBA wants to protect, they need to run a quick SQL script (`sqlprotect.sql`, which can be found in the `share` directory of the Postgres Plus installation) that creates a number of objects needed to manage the metadata in SQL/Protect.

Once the database(s) to protect are chosen, a DBA then decides which security roles and/or user accounts will be monitored for SQL injection attack activity. Within PostgreSQL, roles (sometimes called groups) serve as containers for security privileges on underlying objects such as tables. SQL/Protect can use PostgreSQL's security roles as its object of focus in determining what security accounts to monitor for SQL injection attack attempts. Standard user accounts may also be monitored.

It's very simple for a DBA to monitor a security role with SQL/Protect. For example, suppose the DBA defines a user login account called `webuser` that is assigned to the security role of `stduser`. To have the role and underlying user's activities monitored with SQL/Protect, the DBA simply issues the following commands from within a SQL query session:

```
SET SEARCH_PATH=sqlprotect;
SELECT protect_role('stduser');
```

To find out what roles have been designated as being managed by SQL/Protect and their protection options, the DBA can query one of SQL/Protect's objects like this:

```
dev=# \x
Expanded display is on.
dev=# SELECT * FROM sqlprotect.list_protected_users;
-[ RECORD 1 ]------+--------
dbname             | dev
username           | stduser
protect_relations  | t
allow_utility_cmds | f
allow_tautology    | f
allow_empty_dml    | f
```

The above output shows the following:

- The role `stduser` within the `dev` database will be monitored for SQL injection attack activity
- The following activities for the role `stduser` will be either warned about or restricted, depending on the level of protection configured by the DBA:
    - Utility commands: DDL commands
    - Tautology statements: `WHERE` predicates such as `WHERE 1 = 1`
    - Empty DML commands: statements such as `DELETE FROM USERLOGIN`

To actually activate the monitoring of the `stduser` role, the DBA needs to select one of the modes used by SQL/Protect to check for SQL injection attacks. The modes are controlled by the `edb_sql_protect.level` parameter. There are currently three modes available:

1. **learn** – this mode tracks the activities of protected roles and records the relations used by the roles. This is used when initially configuring SQL/Protect so the expected behaviors of the protected applications are learned.
2. **passive** – this mode issues warnings if protected roles/user accounts are breaking the defined rules, but does not actually prevent a suspect SQL statement from executing. This also comes in handy when testing SQL/Protect test cases.
3. **active** – this mode prevents all monitored statement types (DDL, tautology, empty DML) for a protected role/user account from executing. In this mode, SQL/Protect has in essence become a SQL firewall for the database.

When SQL/Protect is set to the `learn` mode, feedback will be issued when a monitored role/user account executes a SQL statement for the first time against an object:

```
dev=> select emailaddress from userlogin where userid =
'fred';
NOTICE:  SQLPROTECT: Learned relation: 50305
 emailaddress
--------------
(0 rows)
```

Using SQL/Protect to Prevent SQL Injection Attacks

Once SQL/Protect has learned the typical SQL access patterns for a monitored role/user account, the DBA can then set the `edb_sql_protect.level` parameter to 'active' and then all monitored roles/user accounts will now be protected from SQL injection attacks:

```
dev=> select * from userlogin where userid = '' or 1=1;
ERROR:  SQLPROTECT: Illegal Query: tautology
dev=> delete from userlogin;
ERROR:  SQLPROTECT: Illegal Query: empty DML
dev=>
```

Monitoring SQL injection attack attempts that have occurred against a server is very easy. A DBA simply queries the `edb_sql_protect_stats` table in the `sqlprotect` schema to find out if any attempts have occurred against their database:

```
dev=# \x
Expanded display is on.
dev=# SELECT * FROM sqlprotect.edb_sql_protect_stats;
-[ RECORD 1 ]-------
username   | webuser
```

```
superusers | 0
relations  | 0
commands   | 0
tautology  | 1
dml        | 1
```

Conclusions

SQL injection attacks represent a very credible threat to databases and should be taken seriously by both the DBA who manages the database and the developer who creates the front-end application where many SQL injection attacks originate. While there are a number of general rules of thumb that should be followed to guard against SQL injection attacks, a thorough plan includes implementing a SQL firewall inside the database that monitors and guards against malicious attempts at accessing and modifying data. SQL/Protect from EnterpriseDB supplies just this type of functionality in an easy to use manner.

For more information on Postgres Plus Standard Server including downloads of the software, visit: http://www.enterprisedb.com/products-services-training/products/postgres-plus-standard-server. More information on Postgres Plus Advanced Server can be found at: http://www.enterprisedb.com/products-services-training/products/postgres-plus-advanced-server. Both downloads provide a trial of SQL/Protect.

Online documentation for SQL/Protect can be found at: http://www.enterprisedb.com/docs/en/9.0/sqlprotect/Table%20of%20Contents.htm.

About EnterpriseDB

EnterpriseDB is the enterprise PostgreSQL company, providing products and services worldwide that are based on and support PostgreSQL, the world's most advanced open source database. EnterpriseDB's Postgres Plus products are ideally suited for transaction-intensive applications requiring superior performance, massive scalability, and compatibility with proprietary database products. Postgres Plus products provide an economical open source alternative or complement to proprietary databases without sacrificing features or quality.

EnterpriseDB understands that adopting an open source database is not a trivial task. You have lots of questions needing answers, schedules and budgets to keep, and processes to follow. We have helped thousands of organizations like yours through the steps to investigate, evaluate, prove, develop, and deploy their open source solutions.

To make your work easier and faster we have special self-service sections on our website dedicated to assisting you in each of the steps. For working with any of these versions, EnterpriseDB has many free resources on the web site targeted at the various stages of open source adoption. Visit http://www.enterprisedb.com/solutions/stages/overview.do.

- **Getting started** – access to free downloads, installation guides, demos, starter tutorials, and more to help get familiar with the database.
- **Evaluations and pilots** – learn how Postgres has helped hundreds of Oracle users cut costs and MySQL users improve operations.
- **Development** – EnterpriseDB employs more Postgres experts, developers and community members and than any other company, and offers key application development resources.
- **Deployment** – information on how to scale a Postgres application, add Qualities of Service (QoS) like high availability or security, or get a health check.

If you would like to discuss training, consulting, or enterprise support options, please do not hesitate to contact EnterpriseDB directly. EnterpriseDB has offices in North America, Europe, and Asia. The company was founded in 2004 and is headquartered in Bedford, MA. For more information, please visit http://www.enterprisedb.com.

Sales Inquiries:

sales-us@enterprisedb.com (US)
sales-intl@enterprisedb.com (Intl)
+1-732-331-1315
1-877-377-4352

General Inquiries:

info@enterprisedb.com
info.asiapacific@enterprisedb.com (APAC)
info.emea@enterprisedb.com (EMEA)
+1-732-331-1300