Total security in a PostgreSQL database

Readiness for the first strike

Robert Bernier November 17, 2009

Database security is the single biggest concern with today's Web-based applications. Without control, you risk exposing sensitive information about your company or, worse yet, your valuable customers. In this article, learn about security measures you can take to protect your PostgreSQL database.

Introduction

There are lots of stories in the press about crackers accessing corporate databases. Gone are the days when prepubescent teens were the authors of most cracks. Today, data harvesting is big business and is accomplished by dedicated experts who work within a corporate infrastructure. It's not a question of *how* you can prevent the unauthorized access attempt — you can't — but, rather, how can you *reduce* the effect when it does happen.

Definitions

Hacker— A hacker explores, inquires, and discovers by understanding the technology on a level rarely duplicated in normal circumstances. To be called a hacker by your peers is a badge of honor not because you do something bad but because your expertise is unparalleled.

Cracker— A hacker with malicious intent, such as vandalism, credit card fraud, identity theft, piracy, or other types of illegal activity.

This article explores the challenges of protecting your PostgreSQL (also known as Postgres) database server. PostgreSQL is a powerful open source object-relational database system. It has a proven architecture with a reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux®, UNIX®, and Windows®. It is fully ACID-compliant, and has full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages).

Be sure to download the sample code listings used in this article.

The ideal administrator

In the grand tradition of UNIX, PostgreSQL was designed from the ground up to complement the OS it sits upon. Leveraging PostgreSQL to its maximum potential requires knowledge beyond the mindshare typically expected of the average database administrator (DBA).

In a nutshell, the ideal PostgreSQL DBA has the following background:

- Knowledgeable of relational theory and familiar with SQL '92, '99, and 2003, respectively.
- Knows how to read source code, preferably C, and can compile source code on Linux.
- Can system administrate and is comfortable with the system-V UNIX or Linux.
- Can maintain, if required, the various hardware items typically found in an IT shop. Understands the TCP OS layer, can subnet a network, tune firewalls, etc.

Many DBAs have only the skills to administrate, monitor, and tune the database itself. However, PostgreSQL is built to also rely upon the OS's utilities. It's a rare DBA who excels in all these disciplines, but having this knowledge enables the PostgreSQL DBA to accomplish more in less time than would otherwise be possible.

Review of access privileges

Knowing what a database role can do is paramount if you are to appreciate possible attack vectors. First, you have to control access to data by the granting and revoking of permissions.

Roles, and granting rights and privileges

Just how safe is an ordinary role with default rights and privileges? The user account can be created with one of the following commands:

- The SQL statement `CREATE USER`
- The SQL statement `CREATE ROLE`
- The Postgres command-line utility `createuser`

These three methods of creating user accounts behave dissimilarly, and result in drastically different default rights and privileges.

For an ordinary role, the typical user can:

- Access any database if the data cluster uses the default authentication policy as described in pg_hba.conf.
- Create objects in the PUBLIC schema of any database the user can access.
- Create session (temporary) objects in temporary sessions, such as schema pg_temp_?
- Alter runtime parameters.
- Create user-defined functions.
- Execute user-defined functions created by other users in the PUBLIC schema (as long as they interact only with objects that the user has been granted privileges to access).

It's important to know what the user is allowed to do, but it's equally important to understand the activities that the ordinary user cannot do by default. Ordinary users cannot:

- Create a database or a schema.
- Create other users.
- Access objects created by other users.
- Log in (applies only to the statement `CREATE ROLE`).

Superuser rights and privileges

Though an ordinary user can't execute the rights and privileges defined as superuser capabilities, the ordinary user can still cause quite a bit of grief with defaulted rights and privileges.

This section discusses the attack vectors that the ordinary user can manipulate.

Accessing objects

An extremely common, and unsafe, practice occurs when PostgreSQL is used as the back end to a Web server. The developer creates the ordinary user intending to only carry out those commands that manipulate the data using the commands `INSERT`, `UPDATE`, and `DELETE`. However, unauthorized actions are possible because the PUBLIC schema is open to all. The user can, for example, data mine those tables. It would even be possible to modify them by adding rules and triggers, saving the data in tables located in the PUBLIC schema, which can then be harvested.

Remember, a compromised user account can do anything it wants to the objects it owns.

Countering this threat is easy: Don't let the ordinary user account own or create anything. Listing 1 shows how to secure a table.

Listing 1. Securing a table

```
postgres=# SET SESSION AUTHORIZATION postgres;
SET
postgres=# CREATE ROLE user1 WITH LOGIN UNENCRYPTED PASSWORD '123';
CREATE ROLE
postgres=# CREATE SCHEMA user1 CREATE TABLE t1(i int);
CREATE SCHEMA
postgres=# INSERT INTO user1.t1 VALUES(1);
INSERT 0 1
postgres=# GRANT USAGE ON SCHEMA user1 TO user1;
GRANT
postgres=# SELECT I FROM user1.t1;
 i
---
 2
(1 row)

postgres=# SET SESSION AUTHORIZATION user1;
SET
postgres=> SELECT I FROM user1.t1;
ERROR:  permission denied for relation t1
postgres=> SET SESSION AUTHORIZATION postgres;
SET
postgres=# GRANT SELECT ON user1.t1 TO user1;
GRANT
postgres=# SET SESSION AUTHORIZATION user1;
SET
postgres=> SELECT I FROM user1.t1;
 i
---
```

```
 2
(1 row)
```

Listing 2 demonstrates interdicting access to the PUBLIC schema.

Listing 2. Preventing role user1 from creating any entities

```
postgres=> SET SESSION AUTHORIZATION postgres;
SET
postgres=# REVOKE ALL PRIVILEGES ON SCHEMA PUBLIC FROM user1;
REVOKE
postgres=# SET SESSION AUTHORIZATION user1;
SET

The error message of "ERROR:  permission denied for schema user1" means that this
   defensive measure works:

postgres=> CREATE TABLE X();
ERROR:  permission denied for schema user1
```

Accessing objects under the control of other users

This attack vector, shown in Listing 3 below, assumes that the user has access to the PUBLIC schema; for example, `GRANT USAGE ON SCHEMA PUBLIC TO user1`. It works under the following assumptions:

- All users are by default permitted to connect to any database in the cluster.
- Postures clusters permit users the ability to create and manipulate all entities in the PUBLIC schema.
- An ordinary user account has the right to access system catalogs. Otherwise, the user account can't function properly (intrinsic to PostgreSQL server behavior).

Listing 3. Gleaning information about a table

```
postgres=> SELECT * FROM user1.t2;
ERROR:  permission denied for relation t2
postgres=> insert into user1.t2 values(10);
ERROR:  permission denied for relation t2
postgres=>
postgres=> \d
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 user1  | t1   | table | postgres
 user1  | t2   | table | postgres
(2 rows)

postgres=> \d t?
        Table "user1.t1"
 Column |  Type   | Modifiers
--------+---------+-----------
 i      | integer |

        Table "user1.t2"
 Column |  Type   | Modifiers
--------+---------+-----------
 i      | integer |
```

Although it may not be possible to access the table, the user can still glean information about it.

Listing 4 shows user account user1 obtaining a list of user accounts and their respective properties, too. The ordinary user can't access the passwords themselves.

Listing 4. Obtaining properties of user accounts

```
postgres=> select * from pg_user;
 usename | usesysid | usecreatedb | usesuper | usecatupd | passwd | valuntil | useconfig
----------+----------+-------------+----------+-----------+----------+----------
postgres | 10 | t | t | t | ******** | |
 user1 | 18770 | f | f | f | ******** | |
(2 rows)
```

All users have the default ability of learning the cluster's definitions and schema.

Listing 5 shows a script that gathers information about the cluster's entire definition schema by querying the system catalogs. System catalogs can be modified, or hacked, by the superuser, thus mitigating this threat.

Listing 5. Extracting cluster-wide definitions

```
#!/bin/bash
psql mydatabase << _eof_
set search_path=public,information_schema,pg_catalog,pg_toast;
\t
\o list.txt
SELECT n.nspname||'.'||c.relname as "Table Name"
FROM pg_catalog.pg_class c
 JOIN pg_catalog.pg_roles r ON r.oid = c.relowner
 LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r','')
ORDER BY 1;
\q
_eof_

for i in $( cat list.txt ); do
 psql -c "\d $i"
done
```

Creating and accessing user-defined functions

Functions are either trusted or untrusted. *Trusted* procedural languages execute instructions within the context of the database, such as creating tables, indexes, adding or removing data, etc. *Untrusted* procedural languages, in addition to trusted features, are capable of affecting the real world, too, such as listing the contents of a directory, creating or deleting files, invoking system processes, and even creating socket connections to other hosts.

Listing 6. Adding procedural languages to a database and restoring access to user1

```
postgres=# create language plpgsql;
CREATE LANGUAGE
postgres=# create language plperlu;
CREATE LANGUAGE
postgres=# create language plperl;
CREATE LANGUAGE
postgres=> SET SESSION AUTHORIZATION postgres;
SET
postgres=# GRANT USAGE ON SCHEMA PUBLIC TO user1;
GRANT
```

Listing 7. Trusted vs. untrusted procedural languages

```
postgres=# select lanname as language, lanpltrusted as trusted from pg_language;
 language | trusted
----------+---------
 internal | f
 c        | f
 sql      | t
 plperlu  | f
 plperl   | t
(5 rows)
```

Unlike tables, an ordinary user account does not require special permissions when invoking somebody else's function, even if they've been created by the superuser.

Listing 8. Invoking a superuser's function

```
postgres=# SET SESSION AUTHORIZATION postgres;
SET
postgres=# CREATE OR REPLACE FUNCTION public.f1 (
postgres(# OUT x text
postgres(# ) AS
postgres-# $body$
postgres$# select 'hello from f1()'::text;
postgres$# $body$
postgres-# LANGUAGE SQL;
CREATE FUNCTION
postgres=# SET SESSION AUTHORIZATION user1;
SET
postgres=>
postgres=> SELECT * FROM f1();
 x
-----------------
 hello from f1()
(1 row)
```

The function in Listing 9 below has been created by the superuser using `plperlu`. It returns the contents of the directory; user1 can invoke this function. An ordinary user can invoke both trusted and untrusted functions. The best method to mitigate this threat is to deny access to the function by revoking the privilege.

Listing 9. Exploiting functions by an unauthorized user

```
postgres=> SET SESSION AUTHORIZATION postgres;
SET
postgres=# CREATE OR REPLACE FUNCTION public.f2 (
postgres(# OUT x text
postgres(# ) AS
postgres-# $body$
postgres$# # output the root directory contents into standard output
postgres$# # notice the use of the single back ticks
postgres$# $a = `ls -l / 2>/dev/null`;
postgres$# $message = "\nHere is the directory listing\n".$a;
postgres$# return $message;
postgres$# $body$
postgres-# LANGUAGE PLPERLU;
CREATE FUNCTION
postgres=# SET SESSION AUTHORIZATION user1;
SET
postgres=> SELECT * FROM f2();
 x
--------------------------------------------------------------------------------
```

```
 Here is the directory listing
 total 120
 drwxr-xr-x 2 root root 4096 Aug 29 07:03 bin
 drwxr-xr-x 3 root root 4096 Oct 11 05:17 boot
 drwxr-xr-x 3 root root 4096 Nov 26 2006 build
 lrwxrwxrwx 1 root root 11 Aug 22 2006 cdrom -> media/cdrom
 drwxr-xr-x 15 root root 14960 Oct 12 07:35 dev
 drwxr-xr-x 118 root root 8192 Oct 12 07:36 etc
(1 row)
```

Listing 10. Securing against user1 and group PUBLIC

```
postgres=# SET SESSION AUTHORIZATION postgres;
SET
postgres=# REVOKE ALL ON FUNCTION f2() FROM user1, GROUP PUBLIC;
REVOKE
postgres=# SET SESSION AUTHORIZATION user1;
SET
postgres=> SELECT * FROM f2();
ERROR: permission denied for function f2
postgres=>
```

Listing 11 involves intelligence gathering.

Listing 11. Obtaining the function's source code

```
postgres=> SET SESSION AUTHORIZATION user1;
SET
postgres=> select prosrc as "function f3()" from pg_proc where proname='f3';

 function f3()
---------------
# output the root directory contents into standard output
# notice the use of the single back ticks
 $a = `ls -l / 2>/dev/null`;
 $message = "\nHere is the directory listing\n".$a;
 return $message;
(1 row)
```

To hide the source code:

- Write your function as a module in its native language environment (C, Perl, Python, etc.) and store it on the host's hard drive. Then create an abstracted user-defined function in PostgreSQL that invokes the module.
- Consider writing the source code in a table and dynamically create your function as it's required.
- Write your user-defined function in another database in the cluster, which is then called by an authorized user account using the `dblink` module.

Using the security definer

The *security definer* executes the function with the privileges of the user that created it. Thus, a user can access a table that, under normal circumstances, is otherwise unavailable.

For example, as shown in Listing 12, a table with two columns is created in the schema Postgres by the superuser postgres. The ordinary user, user1, will invoke a function using the security definer parameter and obtain a value based on an input value.

Listing 12. Creating a table and function

```
postgres=# SET SESSION AUTHORIZATION postgres;
SET
postgres=# CREATE TABLE postgres.t4(x serial,y numeric);
NOTICE: CREATE TABLE will create implicit sequence "t4_x_seq" for serial column "t4.x"
CREATE TABLE
postgres=# INSERT INTO postgres.t4(y) VALUES (random()::numeric(4,3));
INSERT 0 1
postgres=# INSERT INTO postgres.t4(y) VALUES (random()::numeric(4,3));
INSERT 0 1
postgres=# INSERT INTO postgres.t4(y) VALUES (random()::numeric(4,3));
INSERT 0 1
postgres=# INSERT INTO postgres.t4(y) VALUES (random()::numeric(4,3));
INSERT 0 1
postgres=# INSERT INTO postgres.t4(y) VALUES (random()::numeric(4,3));
INSERT 0 1
postgres=# CREATE OR REPLACE FUNCTION public.f4 (
postgres(# IN a int,
postgres(# OUT b numeric
postgres(# ) RETURNS SETOF numeric AS
postgres-# $body$
postgres$# select y from postgres.t4 where x=$1 limit 1;
postgres$# $body$
postgres-# LANGUAGE SQL SECURITY DEFINER;
CREATE FUNCTION
```

Listing 13 shows that user account user1 can now access the desired information.

Listing 13. Unprivileged role accessing a table with a function call

```
postgres=# SET SESSION AUTHORIZATION user1;
SET
postgres=> SELECT b as "my first record" FROM f4(1);
 my first record
-----------------
 0.379
(1 row)

postgres=> SELECT b as "my second record" FROM f4(2);
 my second record
------------------
 0.200
(1 row)
```

Cracking the PostgreSQL password

Effective password administration is key to security in a DBMS. It's the DBA's job to enforce an approved password policy. A password should consist of randomly chosen alphanumeric characters that don't have a discernible pattern. Common practice dictates that passwords have at least six characters and are changed frequently.

PostgreSQL user accounts and passwords

The PostgreSQL user account security policy is centered on the SQL commands that create and administrate the user's account:

- `CREATE ROLE`
- `ALTER ROLE`

- `DROP ROLE`

The following SQL statements belong to the older style of user account administration (although valid, you should use the newer technique of managing users as roles):

- `CREATE GROUP`
- `ALTER GROUP`
- `DROP GROUP`
- `CREATE USER`
- `ALTER USER`
- `DROP USER`

Passwords are stored as unencrypted or encrypted. *Unencrypted* passwords are stored in the clear and can be read by the superuser. *Encrypting* the password involves generating and storing its MD5 hash, which can't be read. One validates the password at login by hashing and comparing it to what's already been stored in the data cluster.

Below are some example invocations that create and administrate the password:

- An account is created without a password:
  `CREATE ROLE user1 WITH LOGIN;`
- An account is created with an unencrypted password:
  `CREATE ROLE roger WITH LOGIN UNENCRYPTED PASSWORD '123'`
- An account is altered and assigned an encrypted password:
  `ALTER ROLE user1 WITH ENCRYPTED PASSWORD '123'`

Executing a SQL query by the superuser against the catalog table `pg_shadow` returns the user's account name and its password. Listing 14 shows the code.

Listing 14. Getting a user's password from the catalog

```
postgres=# select usename as useraccount,passwd as "password" from pg_shadow where
length(passwd)>1 order by usename;

 useraccount | password
-------------+-------------------------------------
 user1 | md5173ca5050c91b538b6bf1f685b262b35
 roger | 123
(2 rows)
```

Listing 15 shows how you can generate the MD5 hash for user1 with the password `123`.

Listing 15. Generating an MD5 password

```
postgres=# select 'md5'||md5('123user1') as "my own generated hash",
                  passwd as "stored hash for user1"
           from pg_shadow where usename='user1';

 my own generated hash | stored hash for user1
-----------------------------------+-----------------------------------
 md5173ca5050c91b538b6bf1f685b262b35 | md5173ca5050c91b538b6bf1f685b262b35
(1 row)
```

Ready for another scare? There are few mechanisms within PostgreSQL that can enforce an iron-clad password policy.

Possible security limitations include:

- The superuser cannot enforce a minimum number of characters to be used for the password.
- Although there is a default parameter in the configuration settings for how the password is to be stored (unencrypted or encrypted as an MD5 hash), the user cannot be forced to use a particular storage method by the superuser.
- There is no mechanism that imposes a life span on the user account.
- The mechanism controlling the effective life span of the user account password becomes irrelevant when the connection method is other than either PASSWORD or MD5 in the cluster's client authentication configuration file, pg_hba.conf.
- User runtime parameters that are altered by the `ALTER ROLE` statement, and which have been set by the superuser or by the defaulted configuration settings in the file postgresql.conf, can be changed by the owner of the user account at will.
- Renaming a user account clears its password if it has been encrypted.
- It's not possible to track who made changes to the user accounts or when these changes occurred.

An aggressive architecture with careful editing of the system catalogs can reward the vigilant DBA.

Because of the fascinating potential for mischief, the security limitations of user accounts and passwords are worthy of another, separate article.

Cracking the password

Enforcing a strongly typed password is a worthy goal, but there's just no way of judging its strength until somebody cracks it. Cracking utilities are based upon two approaches, as follows.

Brute force
: The methodical testing of the hash. It begins with a few letters, increasing in length as the attack continues. This method is recommended for testing short passwords.

Dictionary attacks
: Use a social-engineering approach. A dictionary of words, used by the cracking utility, is the starting point. Thereafter, combinations of those words are generated and tested against the captured hash. This attack takes advantage of the erroneous belief that a long character string consisting of a mnemonic combination of strings and characters is safer than a slightly shorter length of randomly chosen ones.

Depending on the strength of the password and the hardware used, the crack can take anywhere from a few seconds to several months.

DBAs are interested in identifying passwords of fewer than six characters in length.

The command-line utility MDCrack uses brute force to test the password. This Windows binary works just fine on Linux under Wine.

Entering `wine MDCrack-sse.exe --help` returns the configuration switches. A few are shown below:

```
Usage: MDCrack [options...] --test-hash|hash
       MDCrack [options...] --bench[=PASS]
       MDCrack [options...] --resume[=FILENAME]|--delete[=FILENAME]
       MDCrack [options...] --help|--about
```

The simplest command-line invocation is `wine MDCrack-sse.exe --algorithm=MD5 --append=$USERNAME $MD5_HASH`, where `$USERNAME` is the user name and `$MD5_HASH` is the MD5 hash in the pg_shadow catalog table.

MDCrack can run in session mode, as follows, so you can stop a cracking operation and continue later.

Listing 16. MDCrack running in session mode

```
# start session
wine MDCrack-sse.exe --algorithm=MD5 --append=$USERNAME $MD5_HASH\
 --session=mysessionfile.txt

# resume using the last session mode
wine MDCrack-sse.exe --algorithm=MD5 --append=$USERNAME $MD5_HASH\
 --resume=mysessionfile.txt
```

The default character set is abcdefghijklmnopqrstuvwxyz0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ. You could end up with a hung process if the candidate password includes a character that is not part of the defaulted character set. You can change it to any combination of alphanumeric characters you want. For example, you may want to include control characters and punctuation.

Adjusting the character set is done on the command line. The variable `$CHARSET` represents the actual set of characters that will be used:

```
wine MDCrack-sse.exe --algorithm=MD5 --append=$USERNAME $MD5_HASH --charset=$CHARSET
```

The example below is an invocation to crack a Postgres password of 123. Ignoring the first three characters gives you the MD5 hash value md5173ca5050c91b538b6bf1f685b262b35. You can determine the password with the following invocation (tip: grep for the string `Collision found`). This crack takes approximately 0.32 seconds:

```
wine MDCrack-sse.exe --algorithm=MD5 --append=user1 173ca5050c91b538b6bf1f685b262b35\
| grep "Collision found"
```

Listing 17 demonstrates cracking the password in the system catalog pg_shadow.

Listing 17. Cracking the password

```
wine MDCrack-sse.exe --algorithm=MD5 --append=user1 \
`psql -t -c "select substring(passwd,4) from pg_shadow where usename='user1';"` \
| grep "Collision found"
```

Authentication models

Now that you've seen what can go wrong, it's time to explore what you can do to make things right. Authentication is a vast subject, so only the basics are covered here.

Under the "authentication" umbrella, there are several means of controlling access to the Postgres cluster:

- UNIX domain sockets
- Ident server authentication
- LDAP server authentication
- PAM
- Kerberos
- SSL

UNIX domain sockets

A UNIX domain socket is a two-way communication pipe that resembles a file in many respects. The server creates the domain socket that's waiting for clients to open the file via the file system. A typical PostgreSQL domain socket is shown below.

Listing 18. Typical domain socket

```
robert@wolf:~$ ls -la /tmp|grep PGSQL
srwxrwxrwx 1 robert robert 0 2007-10-15 12:47 .s.PGSQL.5432
-rw-------  1 robert robert   33 2007-10-15 12:47 .s.PGSQL.5432.lock
```

Notice that the port number is appended to the file's name. Reconfiguring the server to sit on a different TCP/IP port also changes the domain socket's name.

Three parameters in the postgresql.conf configuration file control permissions for a domain socket:

- unix_socket_directory (the file PATH)
- unix_socket_group (the user group)
- unix_socket_permissions (defaults to 0777)

The domain socket's location varies according to the Linux distribution:

- PostgreSQL source code installs and puts the socket in the /tmp directory.
- BSD locates the socket in the /tmp directory.
- RedHat derivatives locate the socket in the /tmp directory.
- Debian derivatives locate the socket in /var/run/postgresql with permissions only for the postgresqlaccount.

There are some weird things about domain sockets. Consider the implications of the following example.

Sudo

`sudo` is a powerful command with many possible configurations that allows users to run programs with the security privileges of another user (normally the superuser, or root). Similar to the Windows command `runas`.

A cluster is created in robert's (superuser) home directory with an authentication of trust. On server startup, though, the domain socket's permissions permits logins except for `robert`. User `robert` logs in with TCP, but is refused on the domain socket. However, `robert` can log in via domain socket after he `sudo`s to `nobody`.

This example shows the versatility of file permissions, mitigating the damage caused by a cracker becoming the superuser.

Listing 19. Permissions

```
robert@wolf:~$ initdb -A trust -U postgres ~/data

robert@wolf:~$ pg_ctl -D ~/data/ -l ~/logfile.txt \
-o "-c unix_socket_permissions=007 -c unix_socket_directory=/tmp" start
server starting

robert@wolf:~$ psql -h localhost -U postgres -c "select 'superuser:this works' as msg"
         msg
----------------------
 superuser:this works
(1 row)

robert@wolf:~$ psql -h /tmp -U postgres -c "select 'superuser:this fails' as msg"
psql: could not connect to server: Permission denied
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?

robert@wolf:~$ sudo su nobody
[sudo] password for robert:

$ psql -h localhost -U postgres -c "select 'nobody:this works' as msg"
        msg
-------------------
 nobody:this works
(1 row)


$ psql -h /tmp -U postgres -c "select 'nobody:this still works' as msg"
           msg
------------------------
 nobody:this still works
(1 row)
```

Ident

The Ident server answers a simple question: What user initiated the connection that goes out of your port X and connects to my port Y? In the context of a PostgreSQL server, it informs the DBMS of the Identity of the user account that is making a login attempt. PostgreSQL then takes that answer and permits or denies permission to login by following a rule set configured by the DBA in the appropriate configuration files.

The PostgreSQL Ident server authentication mechanism works by mapping the PostgreSQL user accounts to the UNIX user accounts using the host's own Ident server.

The following examples assume that all UNIX user accounts have been mapped in PostgreSQL to be able to log in into any database, provided they use the same account name in PostgreSQL. The login fails if the UNIX user name doesn't exist as a user account in the PostgreSQL server, or if an attempt is made to log in using another PostgreSQL user account name.

Suppose you have SSH'd into the host: `ssh -l robert wolf`.

Listing 20. A failed and successful login

```
robert@wolf:~$ psql -U robert robert
Welcome to psql 8.2.4, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

robert@wolf:~$ psql -U postgres robert
psql: FATAL:  Ident authentication failed for user "postgres"

-- This works, su to become the UNIX user account postgres
```

PostgreSQL uses two files to administrate and control all login sessions for users that have been authenticated by the Ident server:

pg_hba.conf
: Controls access via records that are defined on a single line.

pg_Ident.conf
: Comes into play when the Ident service is used as the user account's authenticator. For example, the `METHOD` is identified as *Ident* in the pg_hba.conf file.

Listing 21. Simple configuration examples

```
Example 1: A LOCALHOST connection enforces unix account robert to access database robert
exclusively.  There is no authentication on UNIX domain sockets.

(pg_hba.conf)
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD    OPTION
  host    all         all         127.0.0.1/32        Ident     mymap
  local   all         all                             trust
(pg_Ident.conf)
# MAPNAME     Ident-USERNAME    PG-USERNAME
  mymap       robert            robert


Example 2: A domain socket connection enforces unix account robert to access any database
as pg account robert; unix account postgres can access any database as user robert.

(pg_hba.conf)
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD    OPTION
  local   all         all                             Ident     mymap
  host    all         all         127.0.0.1/32        trust

(pg_Ident.conf)
# MAPNAME     Ident-USERNAME    PG-USERNAME
  mymap       robert            robert
  mymap       postgres          robert


Example 3: A domain socket connection enforces that unix account can connect to any
```

```
database with its postgres database namesake using the
keyword "sameuser".  pg_Ident.conf is not necessary here. Local host
connections via TCP-IP are rejected.

(pg_hba.conf)
# TYPE  DATABASE                USER        CIDR-ADDRESS          METHOD    OPTION
  local   template0,template1    all                              Ident     sameuser
  host    all                    all         127.0.0.1/32         reject

Ex4: (all users can connect with their own user names only to the databases postgres
and robert)

(pg_hba.conf)
# TYPE  DATABASE                USER        CIDR-ADDRESS          METHOD    OPTION
  local   template0,template1    all                              Ident     sameuser
```

Remember the following caveats:

- Configuration changes take effect as soon as you've reloaded the files, such as pg_ctl -D mycluster reload.
- Various configuration settings can cause finicky behavior. Look for failed configuration messages in the log.
- Ident was designed and implemented when the machines themselves were considered secure. Any remote server seeking authentication must be considered suspect.
- The Ident server is only used to authenticate localhost connections.

Data encryption

There are many ways you can inadvertently expose yourself to a cracker inside an intranet.

Let's do a sniff. Suppose you execute the following command on your localhost, 192.168.2.64: `tcpdump -i eth0 -X -s 3000 host 192.168.2.100 and port 5432`.

On a remote host, 192.168.2.100, you connect into your local host's PostgreSQL server, which is already listening on port 5432: `psql -h 192.168.2.64 -p 5432 -U postgres postgres`. Now alter the password of your superuser account, postgres: `ALTER USER postgres WITH ENCRYPTED PASSWORD 'my_new_password';`.

Listing 22. Discerning the password in a sniffed data dump

```
16:39:17.323806 IP wolf.56336 > laptop.postgresql: P 598:666(68) ack 470 win 3068
<nop,nop,timestamp 9740679 9589666>
 0x0000: 4500 0078 4703 4000 4006 6d88 c0a8 0264 E..xG.@.@.m....d
 0x0010: c0a8 0240 dc10 1538 6a4f 7ada 6a71 e77c ...@...8jOz.jq.|
 0x0020: 8018 0bfc 1a9d 0000 0101 080a 0094 a187 ................
 0x0030: 0092 53a2 5100 0000 4341 4c54 4552 2055 ..S.Q...CALTER.U
 0x0040: 5345 5220 706f 7374 6772 6573 2057 4954 SER.postgres.WIT
 0x0050: 4820 454e 4352 5950 5445 4420 5041 5353 H.ENCRYPTED.PASS
 0x0060: 574f 5244 2027 6d79 5f6e 6577 5f70 6173 WORD.'my_new_pas
        0x0070:  7377 6f72 6427 3b00                      sword';.
```

SSH tunnels using port forwarding

IP forwarding is a tunneling technology that forwards Internet packets from one host to another. It allows your PostgreSQL clients, such as `psql`, `pgadmin`, and even `openoffice`, to connect to the remote Postgres server with an SSH connection.

Consider the following questions:

- What happens if there is no `psql` client on the remote PostgreSQL server?
- What happens if you need to upload or download data between your workstation and the remote host?
- What do you do when you need to use database clients because they can perform certain tasks that the `psql` client can't do as well, or can't do at all?
- How do you tunnel your network so your team can connect remotely to a database sitting behind a firewall?

This example connects a client (localhost) to a remote host (192.168.2.100). A proxied connection on the workstation's port of 10000 is created. The client, upon connecting to port 10000, is forwarded to the remote host's PostgreSQL server, which is listening on port 5432: `ssh -L 10000:localhost:5432 192.168.2.100`.

Adding the `-g` switch permits other hosts to take advantage of your forwarding connection, which turns this into an instant virtual private network (VPN) for Postgres connections: `ssh -g -L 10000:localhost:5432 192.168.2.100`.

Some tunnel caveats:

- The database client and server are under the impression that they are communicating with their own localhost.
- Remember to configure the file pg_hba.conf to set up the correct authentication for localhost connections using TCP/IP.
- Ports below 1024 are exclusively controlled by root.
- SSH sessions require an existing user account on the PostgreSQL/SSH server.

SSL-encrypted sessions

Encrypted sessions with PostgreSQL require that the server be compiled with the `--with-openssl` switch. Linux distro binaries have this function. Clients such as `psql` and `pgadmin` also have the requisite capabilities.

You can verify the server using the `pg_config` command-line utility, as follows.

```
pg_config --configure
```

To prepare the PostgreSQL server for encrypted sessions:

1. Create a self-signed server key (server.key) and certificate (server.crt) using the OpenSSL command-line tool `openssl`.
    1. Create the server key: `openssl genrsa -des3 -out server.key 1024`.
    2. Remove the passphrase `openssl rsa -in server.key -out server.key`.
    3. Create a self-signed certificate for the server: `openssl req -new -key server.key -x509 -out server.crt`.
2. Install the two files, server.key and server.crt, into the data cluster's directory.
3. Edit the postgresql.conf file and set the named pair: `ssl = on`.

4. Restart the server.

Listing 23. Successful SSL encrypted session connection

```
robert@wolf:~$ psql -h 192.168.2.100 -U robert
Welcome to psql 8.2.4, the PostgreSQL interactive terminal.

Type: \copyright for distribution terms
 \h for help with SQL commands
 \? for help with psql commands
 \g or terminate with semicolon to execute query
 \q to quit

SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)

robert=#
```

The server always tests the connection first for encrypted session requests. However, you can control the server's behavior by editing the authentication file pg_hba.conf. On the client side, you can control the client's (`psql`) default behavior for using an encrypted session or not by defining the environment variable *PGSSLMODE*.

Their are six modes (two new modes are specific for V8.4).

Mode	Description
disable	Will attempt only unencrypted SSL connections.
allow	First tries an unencrypted connection and, if unsuccessful, an SSL connection attempt is made.
prefer	The opposite of allow; the first connection attempt is SSL and the second is unencrypted.
require	The client attempts only an encrypted SSL connection.
verify-ca	An SSL connection, and valid client certificate signed by a trusted CA.
Verify-full	An SSL connection, valid client certificate signed by a trusted CA, and the server host-name matches the certificate's host name.

For example: `export PGSSLMODE=prefer`.

SSL certificates

SSL authentication is when the client and server exchange certificates that have been signed by a third party who has unquestioned credentials. This third party is known as a *certificate authority* (CA). The connection is refused by either the server or client when it doesn't receive a legitimate certificate from the other.

Though there's a lot of detail, setting up authentication on PostgreSQL using SSL certificates is straightforward:

1. Edit postgresql.conf, `ssl=on`. Server-side authentication requires that the following files be in its data cluster:

- server.key

- server.crt (which must be signed by a CA)
- root.crt (verifies client authentication)
- root.crl (certificate revocation list, optional)

The file root.crt contains a list of approved CA certificates. There should be an entire collection of certificates available for your particular distribution, which you can add.

The file root.crl is similar to root.crt in that it contains a list of certificates signed by the CA. However, these certificates are of clients that have been revoked of the right to connect. An empty root.crl won't interfere with the authentication process.

Client-side authentication requires that the following files be in the client's home directory, ~/.postgresql:

- postgresql.key
- postgresql.crt
- root.crt (verify server authentication)
- root.crl (certificate revocation list, optional)

As with the server's root.crt, the client's file, root.crt, contains a list of server certificates that have been signed by a reputable third-party CA. The last file, root.crl, is optional and is used to revoke server certificates.

Obtaining certificates requires that both client and server have submitted certificate requests, client.csr and server.csr, to the CA. Certificates can only be created after they have generated their private keys, as follows.

```
openssl req -new -newkey rsa:1024 -nodes -keyout client.key -out client.csr
openssl req -new -newkey rsa:1024 -nodes -keyout server.key -out server.csr
```

There is more than one way to execute the `openssl` utility to get what you need. For example, you can put a life span on them, or you can have them generated with self-signed certificates and eliminate the need of a CA.

2. You now have three options for generating your client and server certificates. You can:

- Get client.csr and server.csr signed by a reputable CA.
- Become a CA by using the `openssl` perl utility CA.pl.
- Create self-signed certificates and add them into the server's and client's root.crt files, respectively.

Below is an abbreviated set of commands to use with CA.pl.

- `CA.pl -newca` (create the new CA)
- `CA.pl -newreq` (create a certificate request with a private key)
- `CA.pl -signreq` (sign the certificate request by the CA you created)

For those open source purists, there is always http://www.cacert.org for 'free' certificates.

Listing 24. An example certificate

```
-----BEGIN CERTIFICATE-----
MIIC9TCCAl6gAwIBAgIJAMuhpY+o4QR+MA0GCSqGSIb3DQEBBQUAMFsxCzAJBgNV
BAYTAkFVMRMwEQYDVQQIEwpTb21lLVN0YXRlMSEwHwYDVQQKExhJbnRlcm5ldCBX
aWRnaXRzIFB0eSBMdGQxFDASBgNVBAMTC0NvbW1vbiBOYW1lMB4XDTA3MDIxMjEy
MjExNVoXDTA3MDMxNDEyMjExNVowWzELMAkGA1UEBhMCQVUxEzARBgNVBAgTClNv
bWUtU3RhdGUxITAfBgNVBAoTGEludGVybmV0IFdpZGdpdHMgUHR5IEx0ZDEUMBIG
A1UEAxMLQ29tbW9uIE5hbWUwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAKA4
nX/eBKsPJI1DmtH2wdJE9uZf+IRMUWYrAEDL4F6NEuo2+BsIoOBKS/rrV77Itet9
kduJCQ6k/z2ouAVb4muXpJALDjJpYBXt9wqZf+2p1n9dqDw1rCWBjXIdhOcA3DDv
u0Ig1FUfm8GS97evxM5IJBECRnK/5JZroXCRSHcpAgMBAAGjgcAwgb0wHQYDVR0O
BBYEFElEWNUCV+61itXp86czrDe35vjrMIGNBgNVHSMEgYUwgYKAFElEWNUCV+61
itXp86czrDe35vjroV+kXTBbMQswCQYDVQQGEwJBVTETMBEGA1UECBMKU29tZS1T
dGF0ZTEhMB8GA1UEChMYSW50ZXJuZXQgV2lkZ2l0cyBQdHkgTHRkMRQwEgYDVQQD
EwtDb21tb24gTmFtZYIJAMuhpY+o4QR+MAwGA1UdEwQFMAMBAf8wDQYJKoZIhvcN
AQEFBQADgYEAaFzbUmXcWVzqaVeEpZkNwF/eVh110qIUUxXGdeKZGNXIyK67GCUY
SG/IFkZ/hrGLeqElLrdmU0mHd2Enq2IuvhxnsOVTTickjKospJvlHPYSumkXx0Xp
zey9PhjLh1chpxNGTATKb8ET8YZvBRrDHl/EMPIjLd62iSR/ugFe8go=
-----END CERTIFICATE-----
```

3. Assuming you've generated self-signed certificates, copy them into the correct location and edit root.crt. The client certificate is saved in the server's root.crt and the server's certificate is saved in the client's root.crt.

4. Monitor the log messages upon server restart to confirm everything is configured correctly.

The server's default behavior still uses encryption. This can be disabled by editing the name pair `ssl_ciphers='NULL'` in postgresql.conf and restarting the server. Consider your decision carefully; setting ssl_ciphers to NULL effectively disables encryption.

Conclusion

In this article, you learned some basics about protecting your PostgresSQL database server. There is a lot more material on this topic, but one can only cover so much in a single article. There is not enough being written about PostgreSQL these days. Perhaps, with a little help from you, we could see more about PostgreSQL security.

Downloadable resources

Description	Name	Size
Sample code	os-postgresecurity-listings_src_code.zip	10KB

Related topics

- Check out the tutorial "Total Security In A PostgreSQL Database," which is based upon a series of articles by the author.
- Read the PostgreSQL V8.3.8 Documentation.
- Check out http://c3rb3r.openwall.net/mdcrack/.
- At Wikipedia, learn more about UNIX domain sockets; Ident protocol, an Internet protocol that helps Identify the user of a particular TCP connection; and Certificate authority.
- Get all the news, FAQ, documents, source, and more for the OpenSSL Project.
- Get various patches for tcpdump and libpcap programs.
- Learn more about port forwarding, or tunneling, to forward otherwise insecure TCP traffic through SSH Secure Shell.
- Follow developerWorks on Twitter.