PostgreSQL security attacks

Laurenz Albe

www.cybertec-postgresql.com

Senior Consultant

Laurenz Albe

MAIL laurenz.albe@cybertec.at
PHONE +43 670 605 6265
WEB www.cybertec-postgresql.com

CYBERTEC
POSTGRESQL SERVICES & SUPPORT

Introduction

The scope of this talk

- weaknesses in the setup and configuration of PostgreSQL
- weaknesses in the definition of PostgreSQL objects
- **not** about social engineering
- **not** about individual security bugs and incidents
- **not** about the operating system

Kinds of security attacks

- denial of service attacks
  (make the server unresponsive)
- authentication attacks
  (gain access by exploiting weaknesses in authentication methods)
- privilege escalation attacks
  (authenticated users gain higher privileges)

Denial of service attacks

What are denial of service attacks?

- can be with or without authentication
  - render the server unresponsive
  - crash the server
  - exhaust the resources on the server

Spamming the postmaster

- will prevent valid users from logging in and hog resources

How can you protect yourself?

- limit `listen_addresses` to safe networks
- firewall that detects and blocks spam
- limit allowed hosts in `pg_hba.conf`
  - that won't protect you, but reduces the effects: rejecting a connection uses fewer resources than an authentication attempt

Denial of service by authenticated users

- hog all available connections (optionally keep the CPU busy)
- make the server go out of memory (crash it if memory overcommit is on)
  - for example, launch `CREATE INDEX` in many concurrent sessions
- fill the disk
  - `SELECT * FROM generate_series(1, 10000000000000000000000);`

Protection from authenticated denial of service attacks

- apart from a connection limit, there is no way to protect against users hogging connections
- use a connection pool to have a low connection limit — that will prevent overload of CPU, memory and disk bandwidth
- disable memory overcommit to prevent crashes
- set `temp_file_limit` to prevent filling the disk with temporary files

The only certain protection is to prevent untrusted users from executing arbitrary SQL statements.

Authentication attacks

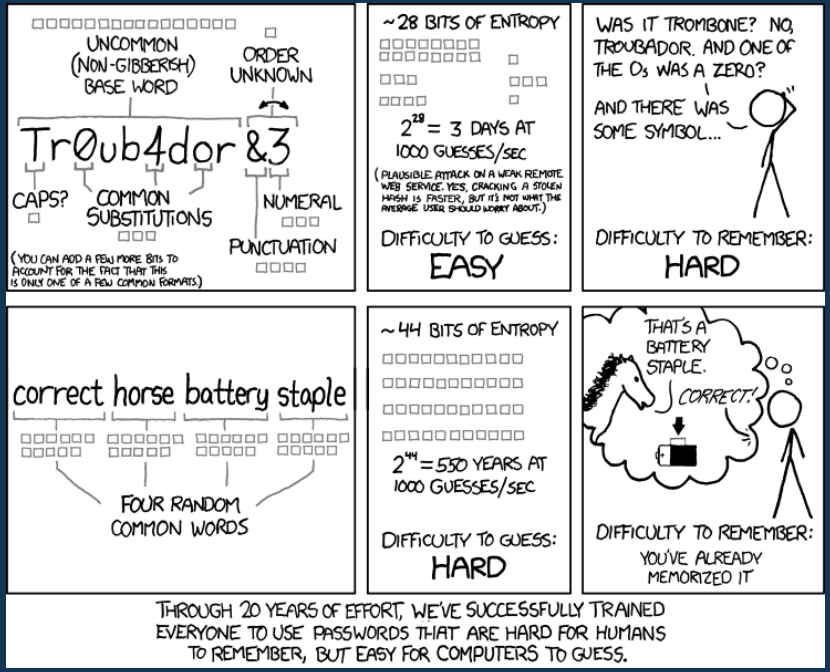
Authentication attacks

- exploit weak or no passwords
- man in the middle attacks
  - eavesdrop on unencrypted connections
  - use `password` authentication to steal passwords
  - crack `md5` authentication

Exploiting weak or no passwords

- gain access to systems where `trust` authentication is possible
- guess the password (could it be `postgres`?)
- brute force attacks with password dictionaries

What is a good password?



Password management

- see https://xkcd.com/936/
- PostgreSQL cannot enforce password rules (the server never sees the clear text password)
- for good security, **don't use passwords in the database**
  - use central identity management for database users (Kerberos, secure LDAP, TLS certificates, . . .)
- use a different user for application, backup, monitoring etc.
  - changing passwords will be less trouble
  - each user gets only the required privileges
  - each user can be configured or locked out differently

Man-in-the-middle attacks

- impersonate the server (e.g., with IP/DNS spoofing)
- pass authentication tokens on to the real server

Protection against **all** man-in-the-middle attacks:

- use TLS encrypted connections
- use a signed server certificate with common name = server name
- use `sslmode=verify-full` on the client

Exploit `password` authentication

- a kind of man-in-the-middle attack
- when a client tries to connect, send a `AuthenticationCleartextPassword` response
- the client will send you the clear text password
- **protect yourself** by rejecting `AuthenticationCleartextPassword` messages
  - with libpq, use `require_auth=!password` (available from v16 on)
- **even better:** use `sslmode=verify-full`

Break `md5` authentication

- performed by eavesdropping on the database connection
- the server's `AuthenticationMD5Password` response contains a random 4-byte "salt"
- remember the salt and the hashed password in the client response
- once you know enough salts, launch a brute force attack on the server
- wait until the server sends a salt you know and respond with the correct hashed password
- **protect yourself** with encrypted connections
- **protect yourself** by using `scram-sha-256` authentication

Privilege escalation attacks

Trick the superuser

- get a superuser to execute an evil function
  (for example, make them `INSERT` into my table with an evil trigger)

Protection:

- use superusers as little as possible
- as superuser, never use objects belonging to users that are not trustworthy
- only give trustworthy users the `CREATE` privilege on schemas
- only give trustworthy users the `TEMP` privilege on databases
- if you are using a PostgreSQL version older than v15, **revoke the `CREATE` privilege on schema `public`**:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

Abusing pg_execute_server_program

- any members of pg_execute_server_program can become superuser:

```
COPY (SELECT 42) TO PROGRAM
   $$psql -c 'ALTER ROLE laurenz SUPERUSER'$$;
```

- The documentation warns:
  As these roles are able to access any file on the server file system, they [...] could be used to gain superuser-level access, therefore great care should be taken when granting these roles to users.

- **don't give pg_execute_server_program, pg_write_server_files and pg_read_server_files to anybody**

Abusing CREATEROLE

- normal users with CREATEROLE can grant themselves membership in any role:

```
GRANT pg_execute_server_program TO laurenz;
```

- as seen above, this is enough to become superuser
- **don't give users CREATEROLE in v15 and lower**
- from PostgreSQL v16 on, you can grant membership in a role only if
  - you created that role or
  - you were granted ADMIN on that role or
  - you are a superuser

Subverting view security (setup)

- views can be used to show only part of the data:

```
CREATE TABLE data (
   id bigint PRIMARY KEY,
   category varchar(1) NOT NULL,
   data text NOT NULL
);

INSERT INTO data VALUES
   (1, 'p', 'public data'),
   (2, 's', 'secret data');

CREATE VIEW pubdata AS
   SELECT * FROM data WHERE category <> 's';

GRANT SELECT ON pubdata TO PUBLIC;
```


Subverting view security (demo)

- an unprivileged user can subvert security:

```
CREATE FUNCTION echo_secret(bigint, varchar, text)
   RETURNS boolean LANGUAGE plpgsql
   COST 0.001 AS
$$BEGIN
   IF $2 = 's' THEN
      RAISE NOTICE 'Secret data % is: %', $1, $3;
   END IF;
   RETURN FALSE;
END;$$;

SELECT * FROM pubdata
WHERE echo_secret(id, category, data);
NOTICE:  Secret data 2 is: secret data
```


Subverting view security (explanation)

- PostgreSQL checks if the view owner has permission on the table and replaces the view with the view definition
- the optimizer executes the “cheap” condition before the view condition:

```
EXPLAIN (COSTS OFF)
SELECT * FROM pubdata
WHERE echo_secret(id, category, data);

                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on data
   Filter: (echo_secret(id, category, data) AND
           ((category)::text <> 's'::text))
(2 rows)
```

Subverting view security (remedy)

- **all views that serve security purposes must be declared with `security_barrier` set to on:**

  ```
  ALTER VIEW pubdata SET (security_barrier = on);
  ```

- then the optimizer will execute all view conditions before user supplied conditions, unless the latter use `LEAKPROOF` functions
- `LEAKPROOF` means that a function has no side effects (cannot “leak” data)
- `LEAKPROOF` can only be set by a superuser
- comparison operators on standard types are typically `LEAKPROOF`

Abusing `SECURITY DEFINER` (setup)

- functions created with `SECURITY DEFINER` run with the privileges of the owner
- powerful tool to allow unprivileged users privileged operations in a controlled fashion
- like all powerful tools, it is dangerous
- the following function is harmless, isn't it?

```
CREATE FUNCTION harmless(integer) RETURNS integer
   SECURITY DEFINER
   LANGUAGE sql AS
'SELECT $1 + 1';
```

Abusing SECURITY DEFINER (attack)

```
CREATE FUNCTION public.sum(integer, integer) RETURNS integer
   LANGUAGE sql AS
'ALTER ROLE laurenz SUPERUSER; SELECT $1 OPERATOR(pg_catalog.+) $2';

CREATE OPERATOR public.+
   (LEFTARG = integer, RIGHTARG = integer, FUNCTION = public.sum);

SET search_path = public, pg_catalog;

SELECT harmless(41);
 harmless
----------
       42

\du laurenz
      List of roles
 Role name | Attributes
-----------+------------
 laurenz   | Superuser
```

Abusing SECURITY DEFINER (remedy)

- **set search_path to a safe value on all functions:**

```
ALTER FUNCTION harmless SET search_path = pg_catalog;
```

- revoke the EXECUTE privilege on SECURITY DEFINER functions from PUBLIC and grant it only to the users that should have it
- use “new style” SQL functions whenever you can — they don’t depend on search_path:

```
CREATE OR REPLACE FUNCTION harmless(integer) RETURNS integer
   SECURITY DEFINER RETURN $1 + 1;
```

Conclusion

What you should remember

- expose your database as little as possible


- use encrypted connections with `sslmode=verify-full`
- choose a secure authentication method
- give your users as little privileges as possible (revoke `CREATE` on the `public` schema)
- security relevant views must have `security_barrier = on`
- set `search_path` on all functions, most importantly on `SECURITY DEFINER` functions