PostgreSQL High-Security

Mailserver Konferenz Berlin, 2014

Hans-Jürgen Schönig

PostgreSQL im Einsatz:

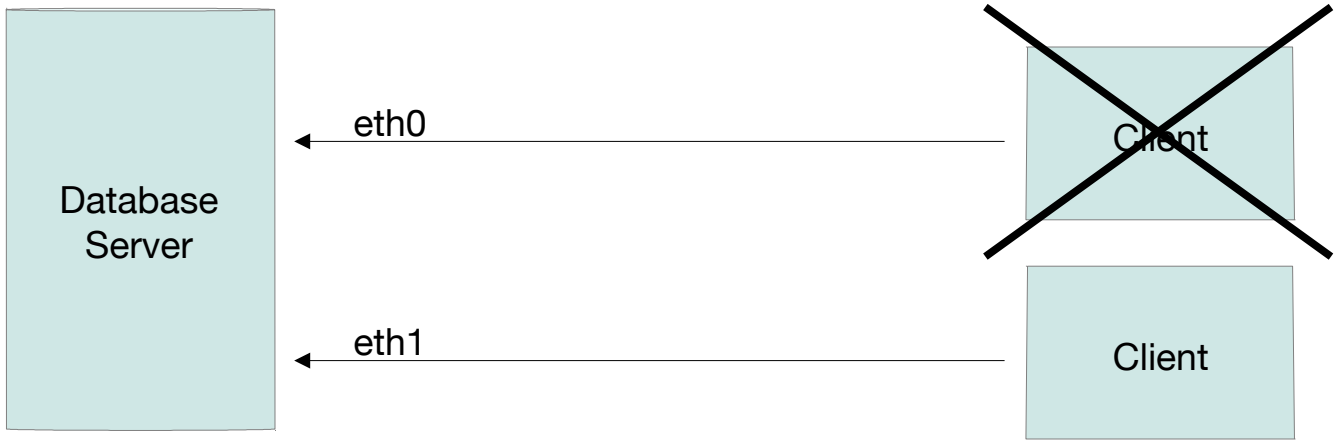
- Storage Backend
  - Heinlein Mail Archiv
  - Archiveopteryx
  - DBMail
  - etc.
- Authentication Backend
  - PAM Module
  - etc

PostgreSQL absichern:

- PostgreSQL verfügt über mehrere Security Layer
  - IP “ja / nein” (= listen_addresses)
  - Host Based Authentication (= pg_hba.conf)
  - Berechtigungen auf Objekte (Schemata, Tabellen, Spalten, etc.)
  - etc.
- SELinux und Verschlüsselung
  - PostgreSQL kann mit SELinux integriert werden
  - PostgreSQL ermöglicht Verschlüsselung via pgcrypto (Daten, etc.)

TCP ja / nein:

- Welche Netzwerkinterfaces werden betrachtet?



- IP “ja / nein” (= listen_addresses)
- Nicht jedes Netzwerkinterface muss berücksichtigt werden.
- Ideal wenn kein IP benötigt wird (nur UNIX Sockets)

pg_hba.conf: Netzwerksicherheit

- Herkunftsorientierte Authentifizierung
  - je nach Herkunft einer Anfrage unterschiedliche Authentifizierungsmethode.
- Mögliche Authentifizierungsmethoden
  - trust, reject, md5, password, gss, sspi, krb5, ident, peer, pam, ldap, radius, cert
  - mittels PAM sind auch andere Methoden möglich

Benutzer, Rollen, Gruppen ...

- User = Gruppe = Rolle

  - früher hatte PostgreSQL “traditionell” ein “User” -> “Gruppe” -> “Welt” Modell

  - mittlerweile basiert das System aus in sich verschachtelten Rollen

  - wesentlich flexiblere Struktur

Gedanken zum Thema User ...

- **Annahme: “Tätigkeit ist stabiler als Personal”**
  - Trennung zwischen “echten” Usern und Rechtelieferanten.
  - Rechte an abstrakte Rollen zuzuweisen ist langfristig einfacher und leichter zu administrieren.
  - Personen zu “tauschen” wird einfacher.

Rollen / User anlegen:

```
test=# \h CREATE ROLE
Command:     CREATE ROLE
Description: define a new database role
Syntax:
CREATE ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

Rechte auf Databaseebene:

- Jede Database kann getrennt gesichert werden:

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]
        | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
        [ WITH GRANT OPTION ]
```

Rechte auf Schemaebene:

- Schemata sind ein Namespace innerhalb einer Database:

```
GRANT { { CREATE | USAGE } [, ...]
          | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
        [ WITH GRANT OPTION ]
```

Rechte auf Tabellenebene:

- Absicherung von Tabellen:

```
GRANT { { SELECT | INSERT | UPDATE | DELETE |
          TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
         [ WITH GRANT OPTION ]
```

Rechte auf Spaltenebene:

- Absicherung von einzelnen Spalten:

```
GRANT { { SELECT | INSERT | UPDATE | REFERENCES }
        ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    TO { [ GROUP ] role_name | PUBLIC } [, ...]
    [ WITH GRANT OPTION ]
```

Verschlüsselung:

- PostgreSQL hat eine Extension namens “pgcrypto”
  - Alle relevanten Verschlüsselungsverfahren werden unterstützt:

    => SHA, md5, DES, Blowfish
  - Verschlüsselung kritischer Infos

SELinux Integration (1):

- **PostgreSQL kann mit SELinux integriert werden.**
  - PostgreSQL unterstützt ein entsprechendes Kommando: SECURITY LABEL
  - Integration mit SELinux ermöglicht ein wesentlich höheres Level an Security

SELinux Integration (2):

```
test=# \h SECURITY LABEL
Command:     SECURITY LABEL
Description: define or change a security label applied to an object
Syntax:
SECURITY LABEL [ FOR provider ] ON
{
  TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE agg_name (agg_type [, ...] ) |
  DATABASE object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN TABLE object_name
  FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  ROLE object_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TABLESPACE object_name |
  TYPE object_name |
  VIEW object_name
} IS 'label'
```

SELinux Integration (3):

- Ein Beispiel:

```
SECURITY LABEL FOR selinux ON TABLE mytable
    IS 'system_u:object_r:sepgsql_table_t:s0';
```

Security barriers (1):

- Der Optimizer legt die Reihenfolge von Filtern fest
  - Das macht aus Performance Gründen sehr viel Sinn
  - Das kann zu einem Security Problem führen.

Security barriers (2):

Aufgabe:

```
CREATE TABLE t_test (id int4);
INSERT INTO t_test SELECT *
        FROM generate_series(1, 20);

CREATE VIEW v AS
    SELECT  *
    FROM    t_test
    WHERE   id % 2 = 0
        AND fast_func(id) = 0;
```

Query:

```
SELECT * FROM v WHERE slow_func(id) = 0;
```

=> Slow function is executed first

Security barriers (3):

- Das Problem:

  - Ein User darf nur eine View aber nicht die darunter liegenden Daten sehen (Security Konzept)

  - Wenn diese Stored Procedures Debug Messages ausgeben, sieht die “slow_func” Daten, die sie nicht sehen soll.

  => Der User kommt “illegal” an Daten

Security barriers (4):

- Die Lösung:

```
CREATE VIEW v WITH (security_barrier) AS
 SELECT  *
 FROM    t_test
 WHERE   id % 2 = 0
        AND fast_func(id) = 0;
```

- Der Optimizer darf die Reihenfolge nicht mehr blind vertauschen, um zu optimieren.

Any questions?

Hans Jürgen Schönig
hs@cybertec.at