Sicherheitsattacken auf PostgreSQL

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


CYBERTEC PostgreSQL
Switzerland
SWITZERLAND
CYBERTEC PostgreSQL
Poland
POLAND
CYBERTEC PostgreSQL
Nordic
ESTONIA
CYBERTEC PostgreSQL
International
AUSTRIA
CYBERTEC PostgreSQL
South America
URUGUAY
CYBERTEC PostgreSQL
South Africa
SOUTH AFRICA
CYBERTEC
POSTGRESQL SERVICES & SUPPORT

CYBERTEC
POSTGRESQL SERVICES & SUPPORT

KUNDEN BRANCHEN

- ICT
- Universitäten
- Regierungen
- Automotive
- Industrie
- Handel
- Finanzwesen
- uvm.

Einleitung

Worum geht es in diesem Vortrag

- Schwächen beim Einrichten und Konfigurieren von PostgreSQL
- Schwächen in der Definition von Datenbankobjekten
- **nicht** um „Social Engineering” („sag mir dein Passwort”)
- **nicht** um einzelne Bugs und Vorfälle
- **nicht** ums Betriebssystem

Arten von Sicherheitsattacken

- „Denial of Service”-Attacken
  (den Server dahin bringen, dass er seine Arbeit nicht mehr tun kann)
- Authentisierungsattacken
  (Schwächen der Authentisierung ausnützen, um Zugriff zu bekommen)
- Rechteüberschreitung
  (ein authentisierter Benutzer erschleicht höhere Rechte)

„Denial of Service”-Attacken

Was sind „Denial of Service”-Attacken?

- können mit und ohne Authentisierung vorgehen
  - den Server antwortunfähig machen
  - den Server zum Absturz bringen
  - die Ressourcen am Server erschöpfen

Den Postmaster mit Anfragen überfluten

- hindert legitime Benutzer an der Anmeldung und belegt Resourcen

Wie kann man sich schützen?

- `listen_addresses` auf sichere Netzwerke einschränken
- Firewall, die Spam erkennt und blockiert
- zulässige Client-Maschinen in `pg_hba.conf` einschränken
  - das schützt nicht vor der Attacke, aber verringert den Schaden: eine Verbindung abweisen is billiger als eine versuchte Authentisierung

„Denial of Service” durch authentisierte Benutzer

- alle verfügbaren Verbindungen besetzen (und vielleicht die CPU überlasten)
- den Server „out of memory” gehen lassen (und abstürzen lassen, wenn „Memory Overcommit” nicht deaktiviert ist)
  - zum Beispiel `CREATE INDEX` in vielen gleichzeitigen Sitzungen starten
- die Platte vollschreiben
  - `SELECT * FROM generate_series(1, 10000000000000000000000);`

Schutz vor authentisierten „Denial of Service”-Attacken

- eine Verbindungsobergrenze ist der einzige Schutz dagegen, dass Benutzer alle Verbindungen belegen
- Connection Pool verwenden: hält die Anzahl der Verbindungen klein und verhindert die Überlastung von CPU, Memory und Plattenkapazität
- „Memory Overcommit” deaktivieren, um Abstürze zu vermeiden
- `temp_file_limit` verhindert das Füllen der Platte mit temporären Dateien

Der einzige sichere Schutz ist, nur vertrauenswürdige Benutzer beliebige SQL-Statements ausführen zu lassen.

Authentisierungsattacken

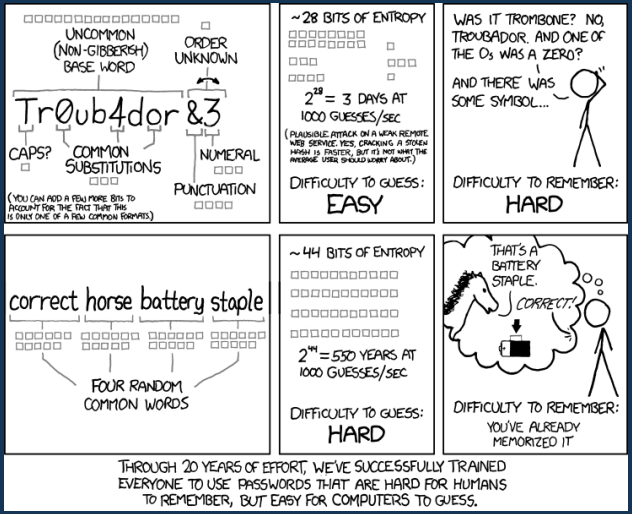
Authentisierungsattacken

- schwache oder nicht verlangte Passwörter ausnützen
- „Man in the Middle”-Attacken
  - unverschlüsselte Verbindungen belauschen
  - mit `password`-Authentisierung Passwörter stehlen
  - `md5`-Authentisierung knacken

Schwache oder nicht verlangte Passwörter ausnützen

- Zugriff auf Systeme erlangen, die sich mit `trust`-Authentisierung anmelden können
- Passwort erraten (könnte es „`postgres`” sein?)
- „Brute Force”-Attacken mit Passwortverzeichnissen

Was ist ein gutes Passwort?



Passwortverwaltung

- PostgreSQL kann keine Passwortkomplexitätsregeln erzwingen (der Server sieht das Passwort nie im Klartext)
- für fortgeschrittene Sicherheitsanforderungen **keine Passwörter in der Datenbank verwenden**
  - zentrale Identitätsverwaltung für Datenbankbenutzer (Kerberos, verschlüsseltes LDAP, TLS-Zertifikate, . . .)
- unterschiedliche Benutzer für Applikation, Sicherung, Monitoring etc.
  - vereinfacht Passwortänderung
  - jeder bekommt nur die Rechte, die er braucht
  - Benutzer können individuell konfiguriert und ausgesperrt werden

„Man in the Middle”-Attacken

- tu so, als wärest du der Server (z.B. mit IP/DNS spoofing)
- reiche alle Nachrichten an den echten Server weiter

Schutz gegen **alle** „Man in the Middle”-Attacken:

- TLS-verschlüsselte Verbindungen verwenden
- signiertes Serverzertifikat verwenden mit „common name” = Servername
- am Client `sslmode=verify-full` verwenden

`password`-Authentisierung ausnützen

- das ist eine Form von „Man in the Middle”-Attacke
- wenn sich der Client verbindet, antworte mit `AuthenticationCleartextPassword`
- der Client sendet dann das Password im Klartext
- **zum Schutz** sollte der Client `AuthenticationCleartextPassword` verweigern
  - mit libpq „`require_auth=!password`” verwenden (verfügbar ab v16)
- **noch besser:** `sslmode=verify-full` verwenden

`md5`-Authentisierung knacken

- durch Belauschen der Datenbankverbindung
- die `AuthenticationMD5Password`-Nachricht vom Server enthält vier Bytes zufällig generiertes „Salt”
- das „Salt” und das gehashte Passwort in der Client-Antwort merken
- wenn man genug Paare hat, „Brute Force”-Attacke starten
- wenn der Server ein bekanntes „Salt” sendet, mit richtigem Hash antworten
- **zum Schutz** verschlüsselte Verbindungen verwenden
- **zum Schutz** `scram-sha-256`-Authentisierung verwenden

Rechteüberschreitung

Den Superuser hineinlegen

- man bringt einen Superuser dazu, eine böse Funktion auszuführen (zum Beispiel ein `INSERT` auf meine Tabelle mit bösem Trigger)

Schutz:

- so wenig wie möglich Superuser verwenden
- nie als Superuser Objekte von nicht vertrauenswürdigen Benutzern verwenden
- nur vertrauenswürdigen Benutzern das `CREATE`-Recht auf Schemas geben
- nur vertrauenswürdigen Benutzern das `TEMP`-Recht auf die Datenbank geben
- in PostgreSQL-Versionen vor v15, **das `CREATE`-Recht auf dem Schema `public` entziehen:**

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


pg_execute_server_program missbrauchen

- jedes Mitglied von pg_execute_server_program kann Superuser werden:

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

- Die Dokumentation warnt:
  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.

- **Niemandem pg_execute_server_program, pg_write_server_files und pg_read_server_files geben, dem man nicht auch Superuser geben würde.**

CREATEROLE missbrauchen

- Wer CREATEROLE hat, kann sich zum Mitgleid jeder Gruppe machen:

```
GRANT pg_execute_server_program TO laurenz;
```

- wie eben gesehen, reicht das, um Superuser zu werden
- **unter PostgreSQL v16 niemandem CREATEROLE geben**
- von v16 an kann man jemandem nur dann zum Mitglied einer Rolle machen, wenn
  - man die Rolle selber erzeugt hat oder
  - man ADMIN auf diese Rolle bekommen hat oder
  - man ein Superuser ist

Sicherheit bei Views unterlaufen (Teil 1)

- Views können verwendet werden, um nur einen Teil der Daten zu zeigen:

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

INSERT INTO daten VALUES
   (1, 'p', 'öffentliche daten'),
   (2, 's', 'voll geheim');

CREATE VIEW fuer_alle AS
   SELECT * FROM daten WHERE categorie <> 's';

GRANT SELECT ON fuer_alle TO PUBLIC;
```


Sicherheit bei Views unterlaufen (Teil 2)

- ein unprivilegierter Benutzer kann die Einschränkung unterlaufen:

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

SELECT * FROM fuer_alle
WHERE echo_secret(id, categorie, daten);
NOTICE:  Geheime Daten 2 sind: voll geheim
```


Sicherheit bei Views unterlaufen (Erklärung)

- PostgreSQL überprüft die Rechte auf die zugrundeliegende Tabelle mit dem Viewbesitzer und ersetzt die View mit ihrer Definition
- Der Optimizer wertet die „billigere” Bedingung vor der View-Bedingung aus:

```
EXPLAIN (COSTS OFF)
SELECT * FROM fuer_alle
WHERE echo_secret(id, categorie, daten);

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


Sicherheit bei Views unterlaufen (verhindern)

- **alle Views, die Sicherheitszwecken dienen, müssen mit `security_barrier = on` definiert werden:**

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

- dann plant der Optimizer die Viewbedingungen *vor* allen Bedingingen aus der Abfrage, die nicht `LEAKPROOF` sind
- `LEAKPROOF` bedeutet, dass eine Funktion keine Seiteneffekte hat (kein „Datenleck”)
- nur ein Superuser kann eine Funktion `LEAKPROOF` setzen
- Vergleichsoperatoren auf Standardtypen sind normalerweise `LEAKPROOF`

`SECURITY DEFINER` missbrauchen (Teil 1)

- `SECURITY DEFINER`-Funktionen werden mit den Rechten des Besitzers ausgeführt
- mächtiges Werkzeug, um Benutzern auf kontrollierte Weise Tätigkeiten zu erlauben, die hohe Rechte benötigen
- gefährlich wie alle mächtigen Werkzeuge
- die folgende Funktion ist harmlos, oder?

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


SECURITY DEFINER missbrauchen (Teil 2)

```
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 harmlos(41);
 harmlos
---------
      42

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

SECURITY DEFINER missbrauchen (verhindern)

- **auf allen Funktionen search_path auf einen sicheren Wert setzen:**

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

- PUBLIC das EXECUTE-Recht auf alle SECURITY DEFINER-Funktionen entziehen und nur den Benutzern erteilen, die es brauchen
- wo immer möglich SQL-Funktionen „neuen Stils” verwenden (sind vom aktuellen search_path unabhängig):

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


Zusammenfassung

Was man sich merken sollte

- Datenbanken so wenig wie möglich exponieren (kleine Angriffsfläche)
- verschlüsselte Verbindungen mit `sslmode=verify-full` verwenden
- eine sichere Authentisierungsmethode wählen
- den Benutzern so wenige Rechte wie möglich geben (`CREATE`-Recht auf Schema `public` entziehen)
- sicherheitsrelevante Views müssen `security_barrier = on` haben
- auf allen Funktionen, besonders auf jenen mit `SECURITY DEFINER`, einen `search_path` setzen

Fragen