# Introducing a PgBouncer authentication layer into our database architecture

February 13, 2023 · Evans Akai Bekoe

Connection Pool Database Access PgBouncer PostgreSQL

We recently decided to factor out PostgreSQL authentication into dedicated PgBouncers for different teams, and no longer allow direct remote access to the database cluster.

Apart from its main use as a connection pooler, we've realized that the isolation afforded by a PgBouncer layer simplifies the management of authentication and connection settings of the different client teams, and frees PostgreSQL of these mundane duties.

## Motivations

In the highly distributed and multi-tenant environment that our databases participate in, even seemingly minor issues easily bloat into an unsustainable mess. We got to the stage where we realized that we needed some formal overarching patterns across all our database servers.

#### Visibility within HBA Rules

Since application servers of teams do not necessarily have contiguous IP addresses, the ip-masks of pg_hba.conf were made to cover very wide subnets. Apart from the potential security issues, this meant that within the HBA rules we lost track of where our clients *are expected to come from*. We could list the different IPs and use comments to separate different team servers, but then the HBA file could become huge and unmanageable since all preceding lines are considered in the HBA logics e.g. a `Reject` on line 1 could impact a different rule on line 500. Without a scalable formalism, a central datateam could not manage all HBAs of all teams, for all our PostgreSQL clusters.

#### Security

In addition to HBA rules allowing connections from very wide subnets, we generally didn't have an ideal security setup w.r.t dedicated users per use-case, apt authentication methods, etc. Going forward, we wanted to enforce the following constraints:

- Avoid SSH access to database servers to client teams, just because of authentication.
- Local connections should use the `peer` or `md5` auth-method.
- Remote connections should come from specific users and IP-addresses, and preferably have SCRAM or `md5` auth-method.
- Remote connections which cannot present a password (e.g. haproxy's pgsql-check option) must have very limited privileges. This means that postgres foreign-data-wrapper superusers (before pg13 passwordless connections are only allowed with a superuser) must first be switched to non-superusers before connecting to other remote database servers.

#### Connection settings

Most connection settings can be made on a PostgreSQL user or database level, but that's not the case at least with `max_connections`. We however want to be able to allot different connection limits to different users and possibly even on different databases. This way one service user is not able to exhaust all available connections, thereby DOS-ing all other service users.

#### Maintenance

Given its clear conceptual separation, it's desirable to pull out authentication and connection settings from PostgreSQL into a separate dedicated layer. Furthermore, in order to delineate ownership and scope, it is desirable to group assets of different teams into different files and instances. Such isolation helps scope deployment, monitoring and troubleshooting. Eventually, we wanted a paradigm where we could present an authentication and connection configuration interface which could be deployed independently, preferably even by the teams themselves, without the chance of breaking another team's instance, and without touching PostgreSQL itself.

Hence we decided that, on each database server, every team should have a dedicated PgBouncer instance to manage their authentication and connections.

## PgBouncer Authentication Layer

Since most teams are already using PgBouncer for connection pooling, it's not such a surprise that we pounced on it as our authentication layer. As to the specific setup, there is not much wiggle room given our motivations.

> **NB:** it is highly recommended to use different users for different use-cases e.g. for foreign-data-wrappers (FDW), for migration, for application, etc; this is so that the `search_path` and other contextual settings within transactional pools remain consistent

> **NB:** with a PgBouncer layer atop PostgreSQL, teams which use an additional application connection pooler could hijack database connections from some client servers, while other client servers are deprived of a PgBouncer pool slot. This materializes into stale PostgreSQL connections which are `idle` and haven't seen a `pg_stat_activity.state_change` for days, and eventually PgBouncer's `client_wait` queue spilling over. The remedy would be to `pg_terminate()` such stale connections, and figure out suitable timeout settings on the application connection pooler.

Below are extracts from the different PgBouncer configuration files.

**INI File**

```
; leave an interface for haproxy health-check with pgsql-check option
; we'll install a user_lookup function in the database=postgres
; and exclude user=haproxy from the auth_file
; hence both pgbouncer and postgresql are included in the haproxy health check
haproxy = host=/run/postgresql/ port=5432 dbname=postgres user=haproxy pool_size=10

; incoming database connections
service1_db = host=/run/postgresql port=5432
...

; outgoing FDW connections to pgbouncers on remote servers
haproxy_remote1 = host=remote1.adjust.com port=8432 dbname=haproxy user=haproxy pool_
remote1_db = host=remote1.adjust.com port=8432 user=$dedicated_non_superuser1
...


logfile = /var/log/pgbouncer/pgbouncer_<%= $instance_name %>.log
pidfile = /run/pgbouncer/pgbouncer_<%= $instance_name %>.pid

; NB: pgbouncer crashes in certain cases when auth_user=pgbouncer
; https://github.com/pgbouncer/pgbouncer/issues/568#issuecomment-840101192
auth_user = pgbouncer_auth
auth_query = SELECT * FROM public.user_lookup($1)
auth_type = hba
auth_file = /etc/pgbouncer/userlist_<%= $instance_name %>.txt
auth_hba_file = /etc/pgbouncer/hba_<%= $instance_name %>.txt

admin_users = pgbouncer, postgres, etc
stats_users = zabbix, etc

; we reserved the port 6432
;listen_port = 7432

...
```

**Auth File**

```
; NB: haproxy user MUST NOT be included here, so that health lookups to PG is made
; other users MUST be listed here, otherwise the user_lookup function has to be creat

"zabbix" ""
"postgres" ""
"pgbouncer" ""
"pgbouncer_auth" ""
"etc" ""

"service1_user" "scram..."
"human1_user" "scram..."
...
```

**HBA File**

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD

local   haproxy         all                                     peer
local   pgbouncer       all                                     peer

# add rules for FDW superusers to connect to a local pgbouncer for user switching

local   all             postgres                                peer

# allow connections from each client-server
# allow connection for auth_user for user_lookup of remote haproxy
# it might be necessary to add rules to Trust remote FDW connections

host    haproxy         haproxy         $app_server_ip1         trust   # hostname1.a
host    haproxy         pgbouncer_auth  $app_server_ip1         trust   # hostname1.a
host    all             all             $app_server_ip1         md5     # hostname1.a

host    haproxy         haproxy         $app_server_ip2         trust   # hostname2.a
host    haproxy         pgbouncer_auth  $app_server_ip2         trust   # hostname2.a
host    all             all             $app_server_ip2         md5     # hostname2.a

...
```

## PostgreSQL Authentication Layer

`pg_hba.conf` and `pg_ident.conf` have to be adapted in order to allow PgBouncer user to proxy `peer` connections for `auth_user` and `haproxy` .

#### pg_ident.conf

```
ospg            pgbouncer           haproxy
ospg            pgbouncer           pgbouncer_auth
```

#### pg_hba.conf

```
# TYPE    DATABASE        USER                ADDRESS             METHOD

# replication and other special connections

host      replication     postgres            198.168.100.1/32    trust

# proxy peer-auth; pg_ident mappings are required

local     postgres        haproxy                                 peer    map=ospg
local     postgres        pgbouncer_auth                          peer    map=ospg

# pgbouncer instance groups
# the human and service users and hashes could be synced into PostgreSQL from each pg
# each such user should be granted the respective Role of the pgbouncer instance
# e.g. GRANT pgbouncer_instance_name1 TO service1_user;

local     all             +pgb_instance_name1                     md5
local     all             +pgb_instance_name2                     md5
...
local     all             +pgb_instance_nameN                     md5

# default peer-auth

local     all             all                                     peer
```

#### The lookup function

The lookup function is required only in the `postgres` database; this is only used for haproxy health checks.

```
-- the auth_user has to be Granted select on the user_lookup function
-- NB: pgbouncer crashes in certain cases when the function returns Record and the us
-- https://github.com/pgbouncer/pgbouncer/issues/568#issuecomment-1198500665

CREATE OR REPLACE FUNCTION public.user_lookup(i_username text, OUT uname text, OUT pl
  RETURNS SETOF record
  LANGUAGE sql
  SECURITY DEFINER
  SET search_path TO 'public', 'pg_temp'
 AS $function$
         SELECT usename::text, passwd
           FROM pg_catalog.pg_shadow
          WHERE usename = i_username;
     $function$

REVOKE ALL ON FUNCTION public.user_lookup(text) FROM public;
```

## Putting it all together

We can capture all the above configs in the following yaml interface:

```
pgbouncer:
  instances:
    adjust_teamA:
      databases:
        haproxy:
          host: /run/postgresql/
          port: 5432
          dbname: postgres
          user: haproxy
          pool_size: 10
        service1_db:
          host: /run/postgresql
          port: 5432
        haproxy_remote1:
          host: remote1.adjust.com
          port: 8432
          dbname: haproxy
          user: haproxy
          pool_size: 10
          remote: true
        remote1_db:
          host: remote1.adjust.com
          port: 8432
          user: $dedicated_non_superuser1
          remote: true
      config:
        scalar:
          listen_port: 7432
          pool_size: 20
          ...
        vector:
          stats_users:
            - zabbix
          admin_users:
            - pgbouncer
            - postgres
          ...
      hba:
        client_groups:        # ip-addresses are interpreted within the configuratio
          - address: [service1_clients]
            database: all
            user: all
            method: scram
            type: host
          ...
        Client_hosts:         # ip-addresses are listed verbatim
          - address: [198.168.100.1]
            database: all
            user: all
            method: scram
            type: host
      userlist:
        - user: zabbix
          pgb_only: true      # user only connects to pgbouncer database
        - user: pgbouncer
          pgb_only: true
        - user: postgres
          pgb_only: true
        - user: service1_user
          hash: scram...
          acl:                # postgres authorization configuration could go here
        - user: human1_user
          hash: scram...
    applovin_teamA:
      ...
```

## Summary

With the capable authentication facilities of PgBouncer, we managed to strip down PostgreSQL HBA to a bare minimum and shoved most of the details into multiple independent PgBouncer instances. At a mere glance, the scope of the connections the datateam allows is apparent and we can direct our auditors to the different owners of the PgBouncer instances for more context on which team connections are allowed and for which reasons.

The user-authentication process no longer requires oversight or manual input from the datateam. Instead, autonomous teams within their own confines, can deploy their own users and password-hashes and tune other connection settings (they have better contextual info about their services), then our local cronjob checks deployed PgBouncer `auth_files` and ensures PostgreSQL logins are in sync with these. This setup works even for teams with isolated configuration-management repositories; the only requirement is to successfully deploy a PgBouncer instance to the database server.

Teams can also deploy haproxies on their application servers to check the health of their PgBouncer connection endpoint; the health of PostgreSQL is checked implicitly.

The current setup doesn't address authorization on PostgreSQL objects; perhaps we can talk about this sometime in the future.

Scorpevans-Postgresql