Authentication and Configuration

PGCon 2023 2023/05/31

Michael Paquier (he/him)

Senior Database Developer
AWS - RDS

The lecturer

- French, based in Tokyo.
- PostgreSQL contributor since 2009
  - Patches, reviews and bug fixes.
  - Blogging.
- Committer since 2018.

Agenda

Authentication methods

Configuration

PostgreSQL 16

Authentication methods

Authentication methods

- Password
  - Plain text
  - MD5
  - SCRAM-SHA-256
  - RADIUS, ldap, pam, BSD…
- Certificates, peer
- Kerberos, SSPI (Windows)
- https://www.postgresql.org/docs/current/static/auth-methods.html

Trust method

- Zero security.
- Just allow connections in.
- N (N > 2) instances with an open port available there.
- Use cases.
  - Unix domain sockets (local) for debugging.
  - Personal laptop and development.
  - Regression tests, with unix_socket_directories and 0700 umask.

Plain text

- Password sent in clear text

```
Server: Please send your password
Client: “hoge”
Server: OK, good to go
```

- Weak to password sniffing, use SSL!
- Works if pg_authid stores MD5 or SCRAM.
- Server may not know the password.

MD5

- Password hash sent:

```
Server: Here is a salt (4 random bytes),
please compute md5(md5(password || username), salt)
Client: “ad22f1df5331cfa7603c67a2092c6159”
Server: OK, good to go
```

- Again use SSL!
- Cold backups.
- Server may not know the password.

SCRAM-SHA-256

- Challenge-based exchange, added in v10.

```
Client: Here is a random nonce (18 bytes)
r=Random_Nonce
Server: Here is my random nonce, salt and iteration count
r=Random_Nonce,s=Salt,i=Iterations
Client: Proof that I know the password.
p=Client_Proof
Server: Proof that I also know the password.
v=ServerProof
```

SCRAM security

- Replay attacks => longer nonces
- Hash stored in pg_authid cannot be used directly.
  - Dictionary attacks
  - Iteration count can be used as parameter
  - Computation of connection proof is costly (connection startup)
- Still use SSL, channel_binding=require.
- Server **has** to know the password, client **should** check the proof.

SCRAM Channel binding

- MITM prevention, by “binding” FE/BE
- RFC 5929: https://tools.ietf.org/html/rfc5929
- Ensure that the point where a connection is done is still the same.
- Channel types:
  - Only tls-server-endpoint.
  - No tls-exporter (TLSv3), yet.

Client/server and HBA entries

Verifier Type	password	md5	scram-sha-256
MD5	O [1]	O	X
SCRAM	O [1]	O [2]	O

- [1]: Plain text is used, hash generated server-side.
- [2]: SCRAM is used.

Peer

- Unix socket connections (local)
- Relies on getpeereid()
- pg_ident.conf + static service files?
  - Local WAL archiver.
  - Monitoring agent.

LDAP

- Server-side implementation
- Useful for large organizations
- Cleartext password for the client
- Format supported
  - prefix+suffix, or simple bind
  - search+bind
- SSL mandatory: ldaptls=1 and hostssl
- Password policies

GSS/SSPI

- GSS/SSPI
- Uses Kerberos.
  - Active directory available
  - No password prompt.
- User mapping with pg_ident.conf.
- Again use SSL!
- Encryption gssenmode.

Certificates

- No password prompt.
- CN field checked for match with database user.
- User mapping in pg_ident.conf.
- Only over SSL.

Regression tests

- src/test/
  - authentication/: hba, SCRAM, SSPI, peer.
  - kerberos/
  - ldap/
  - ssl/, certificates and channel binding
- PG_TEST_EXTRA
- PROVE_TESTS
- PG_TEST_NOCLEAN
- Mandatory for new features.

Configuration

Code

- Backend, src/backend/libpq
  - auth.c, auth-scram.c for authentication.
  - be-secure*.c for SSL and SSPI.
  - hba.c for administration.
- Frontend (libpq), src/interfaces/libpq:
  - fe-auth.c, fe-auth-scram.c for authentication.
  - fe-secure*.c for SSL and GSS/SSPI.

pg_hba.conf

- Connection policies with rules
  - User (list possible)
  - Database (list possible)
  - Host
  - Authentication Type
  - Extra options (map, etc.)
- First match on user, database and connection type (SSL, etc.)
- listen_addresses in postgresql.conf.
- Role membership with '+' and "all".

About files included with @

```
$ cat $PGDATA/pg_hba.conf
#TYPE  DATABASE  USER        METHOD
local   all              @user.list  trust
```

```
$ cat $PGDATA/users.list
user1
user2
user3
```

```
$ cat $PGDATA/users.list
user1,user2
user3 user4
user5
```

pg_ident.conf

- User name mapping
  - Map name
  - OS user (single entry)
  - Database user (single entry)
- For GSSAPI, peer.
- Regexp support for system user, dbuser with optional \1.
- Maps with HBA entries => map=my_map.

Ident entries with same map name

- Equivalent to lists in pg_hba.conf.
- All ident entries with matching map name are checked.

```
$ cat $PGDATA/pg_hba.conf
#TYPE  DATABASE  USER      METHOD
local  all       all       peer map=my_map
```

```
$ cat $PGDATA/pg_ident.conf
# MAPNAME  SYSTEM-USERNAME    PG-USERNAME
my_map     system_user1       pg_user1
my_map     system_user2       pg_user2
```

pg_service.conf

- Centralize connection parameters for clients.
- PGSERVICEFILE.
- Use with pg_ident.conf!
- Service connecting to Postgres
- Connection parameter “service=monitor” or PGSERVICE.

```
[monitor]
host=$DB_HOST_OR_SOCKET_DIR
port=$DB_PORT
user=$DB_USER
```

About sslmode

Modes	Protection		Server-side SSL	
	Eavesdropping	MITM	Disabled	Required
disable	X	X	O	X
allow	X	X	O	O
prefer (default)	X	X	O	O
require	O	X	X	O
verify-ca	O	O	X	O
verify-full	O	O	X	O

PostgreSQL 16

Include, include_dir, include_if_exists

- For pg_ident.conf and pg_hba.conf.
- include_dir, all files suffixed with ".conf".
- include vs include_if_exists:
  - hard vs soft failure.
  - Depends on the setup.
- Rules similar to postgresql.conf.

Inclusion depth

```
$ cat $PGDATA/pg_hba.conf
#TYPE  DATABASE        USER        METHOD
local  @pg_hba.conf    all         peer map=my_map
```

```
LOG:  could not open file "/data/pg_hba.conf": maximum nesting depth
exceeded
CONTEXT: line 2 of configuration file "/data/pg_hba.conf"
        […]
        line 2 of configuration file "/data/pg_hba.conf"
FATAL:  could not load /data/pg_hba.conf
```

More in pg_hba.conf

- Database and roles: regular expressions.
- Begins with ‘/’.
- Can be in lists as single elements.
- Roles beginning with ‘/’ backward-incompatible.
- Failure after HBA entry match => authentication failure.

```
$ cat $PGDATA/pg_hba.conf
#TYPE  DATABASE              USER                   METHOD
local  db1,"/^db\d{2,4}$"    "/^pguser\d{2,4}$"     peer
```

More in pg_ident.conf

- Database user, same rules as HBA entries for roles:
  - Regular expression, with ‘/’
  - Membership, with ‘+’
  - “all”, as alias.

```
$ cat $PGDATA/pg_ident.conf
# MAPNAME  SYSTEM-USERNAME        PG-USERNAME
my_map       system_user1               all
my_map       system_user2               +pguser2
my_map       system_user3               "/^pguser\d{2,4}$"
```

System views

- pg_hba_file_rules
  - Rule number
  - File name
- pg_ident_file_mappings
  - Map order
  - File name

require_auth / libpq

- Filter authentication methods.
- Default does nothing, same as usual.
- Comma-separated lists and negated patterns.
- Combines with others, like channel_binding or sslmode.
- Like:
  - require_auth=none and !none
  - require_auth=password,md5,scram-sha-256,sspi,gss
  - require_auth=!password,!md5

More connection parameters

- sslcertmode=require|allow|disable
- sslrootcert=system
  - Checks the system's CA.
  - Switches sslmode to verify-full by default. Others are **failures**.
- scram_iterations, new GUC.
  - \password
  - Low number => weaker, fast authentication.
  - High number => Harder to crack, slow authentication.
  - Protocol and code flexible with that.

Thank you!

Michael Paquier

paquier@amazon.com