

# Access control

## Connection and authentication



### Copyright

© Postgres Professional, 2015–2022

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Translated by Alexander Meleshko

### Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

### Feedback

Please send your feedback, comments and suggestions to:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

### Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Configuration files

Simple authentication methods

Password-based authentication

External authentication and name mapping

# Steps during connection



## Identification

- identify the database user name
- the name may differ from the one specified (for external authentication)

## Authentication

- is the user really who they claim they are?
- some sort of confirmation is usually required (e.g., a password)

## Authorization

- is this user allowed to connect to the server?
- partially overlaps with privileges

3

When a client initiates a connection, the server must perform several tasks. Firstly, the server must identify the user, that is, determine their user name. To do this, the server asks the user to provide their user name. The specified name may differ from the name of the database user (for example, if the user is registered under their OS user name).

Next, the server authenticates the user, or verifies that they are who they claim to be. A simple way to achieve that is by requesting a password.

Lastly, the server authorizes the user, that is, determines whether they are allowed to connect to the server (this task is partially overlapped with privileges).

All three tasks are often referred to as “authentication”. PostgreSQL offers significant flexibility when it comes to configuring the authentication process.

So far, we've been connecting to the server without any sort of authentication. In Ubuntu, the default settings allow users to connect to a local server without authorization, if the database user name matches the OS user name. Additionally, in the course VM, PostgreSQL is additionally configured to allow any local connections.

## pg\_hba.conf

the configuration file, has to be reloaded for any changes to apply  
lines are composed of fields, space or tab-separated  
empty lines and any text after a comment sign (#) are ignored

### Fields

|  |   |                       |
|--|---|-----------------------|
| connection type                                    | } | connection parameters |
| database name                                      |   |                       |
| user name  |   |                       |
| host address                                       |   |                       |
| authentication method                              |   |                       |
| optional parameters in the <i>key=value</i> format |   |                       |

Authentication settings are stored in a configuration file. The file functions in a similar way to postgresql.conf, but has a different format. The file is called pg\_hba.conf (from “host-based authentication”). Its location is determined by the *hba\_file* parameter. For any changes made to the configuration to apply, the file must be reloaded (by using `pg_ctl reload` or calling the `pg_reload_conf` function).

The pg\_hba.conf file contains a number of lines, each constituting a separate record. Empty lines and comments (anything after a # sign) are ignored. A line contains a number of fields separated by tabs or spaces.

The number of fields may vary depending on the type of content. See the slide for details.

<https://postgrespro.com/docs/postgresql/13/auth-pg-hba-conf>

# Processing the file

Records are read from top to bottom

The first record that corresponds to the attempted connection (type, database, user, and address) is applied

authentication and verification of the CONNECT privilege are performed

if the result is negative, access is denied

if none of the records correspond to the connection parameters, access is denied

| #     | TYPE   | DATABASE | USER | ADDRESS      | METHOD |
|-------|--|----------|------|--------------|--------|
| #     | "local" is for Unix domain socket connections only |          |      |              |        |
| local |  | all      | all  |              | trust  |
| #     | IPv4 local connections:                            |          |      |              |        |
| host  |  | all      | all  | 127.0.0.1/32 | trust  |
| #     | IPv6 local connections:                            |          |      |              |        |
| host  |  | all      | all  | :::1/128     | trust  |

5

The configuration file is processed from top to bottom. Each record is matched against the parameters of the connection requested by the client (by checking the connection type, database name, user name and IP address). If a corresponding record is found, the authentication method specified in the record is performed. Upon successful authentication, the connection is permitted, otherwise it is denied (no other records are checked after this point).

If no records correspond to the connection parameters, access is also denied.

Thus, the records in the file should go from top to bottom from more specific to more general.

At the bottom of the slide is a fragment of the default file you end up with when building from source (may be different when installing from a package). In this example, there are three records. The first one refers to local non-TCP connections (local) for all databases (all) and users (all). The second one is for remote connections (host) from the address 127.0.0.1 (localhost), and the third is the same, but for IPv6.

So, by default PostgreSQL allows only local connections (including local network connections).

Some of the possible field values are discussed more closely later in this topic.

## pg\_hba.conf contents

Location of the configuration file:

```
=> SHOW hba_file;

          hba_file
-----
/etc/postgresql/13/main/pg_hba.conf
(1 row)
```

View the file (without comments and empty strings):

```
student$ sudo egrep '^[^#]' /etc/postgresql/13/main/pg_hba.conf

local  all             postgres                                trust
local  all             all                                           trust
host   all             all             127.0.0.1/32                md5
host   all             all             ::1/128                    md5
local  replication     all                                           trust
host   replication     all             127.0.0.1/32                md5
host   replication     all             ::1/128                    md5
```

When configuring the virtual machine, we have made some changes to the default pg\_hba.conf. Namely, the method peer was changed to trust, so that alice, bob and charlie could connect locally under the OS user student.

Connection type

Database name

Host address

Role name

# Connection type



## local

local connection via a Unix domain socket

## host

TCP/IP connection

(usually the *listen\_addresses* parameter has to be changed)

## hostssl

encrypted SSL connection over TCP/IP

(the server must be compiled with SSL support, and the *ssl* parameter must be set)

## hostnossl

unencrypted TCP/IP connection

8

The connection type field contains one of the values listed below.

“local” allows a local connection via a Unix domain socket (without using a network connection).

“host” allows any TCP/IP connection. Since by default PostgreSQL listens to connections only from the local address (localhost), you will most likely need to set a different address using the *listen\_address* server parameter.

“hostssl” allows only an encrypted SSL connection over TCP/IP. Such connections require that the server is compiled with SSL support. In addition, you need to set the *ssl* = on parameter.

“hostnossl” allows only unencrypted TCP/IP connections.



**all**

connecting to any database

**sameuser**

a database which name matches the user role name

**samerole**

a database which name matches the user role name or a group name that the user is a member of

**replication**

a special permission for the replication protocol

**database**

a specific database name (may be in quotes)

***name[, name...]***

several names from the list

In the database field, you can specify one of the values listed below, or several such values separated by commas.

The word “all” corresponds to any database.

The word “sameuser” corresponds to a database that matches the user name.

The word “samerole” corresponds to a database that matches the name of any role that the user is a member of (including the user's own, since the user is also a role).

Any specific database name can be listed here, too.

A list of database names can be stored in an external file and linked to using the @ sign. The external file can store names separated by commas, spaces, tabs or line breaks. Nested file links (@) and comments (#) are allowed.

**all**

any IP address

***IP address/mask\_length***

specified IP address range (i.e. 172.20.143.0/24)

or an alternative form with two fields (172.20.143.0 255.255.255.0)

**samehost**

server IP address

**samenet**

any IP address from any subnet to which the server is connected

***domain\_name***

the IP address matching the specified name (i.e. domain.com)

any part of name can be specified, starting with a dot (.com)

The address field may contain one of the following values.

“all” corresponds to any client IP address.

IP address with a subnet mask length (CIDR) defines the range of valid IP addresses. Alternatively, the IP address can be specified in one field and the subnet mask in the next. IP addresses in the IPv6 notation are also supported.

“samehost” corresponds to the IP address of the server (this is an alternative of 127.0.0.1 for systems where such an address is not allowed).

“samenet” corresponds to any IP address from any subnet to which the server is connected.

Lastly, the address can be specified as a domain name (or a part of it, starting with a dot). PostgreSQL will determine whether the client’s IP address belongs to the domain. To do this, the domain name is first looked up using the IP address (reverse lookup), and then PostgreSQL checks if the source IP address really corresponds to such a domain (forward lookup). This matches the network owner with the domain name owner, thus blocking out compromised addresses:

[https://en.wikipedia.org/wiki/Forward-confirmed\\_reverse\\_DNS](https://en.wikipedia.org/wiki/Forward-confirmed_reverse_DNS)

*all*

any role

*role*

a role with a specific name (possibly in quotes)

*+role*

a role that is a member of the specified role

*name[, name...]*

multiple names in the formats given above

In the user name field, you can specify one of the values listed below, or several such values separated by commas.

“all” corresponds to any client IP address.

Role name corresponds to the user (or role, which is the same) with the specified name. If the role name is preceded by a + sign, then the name corresponds to any user who is a member of the specified role.

A list of database names can be stored in an external file and linked to using the @ sign. The external file can store names separated by commas, spaces, tabs or line breaks. Nested file links (@) and comments (#) are allowed.

Doesn't check anything

# Simple authentication

trust

allow without authentication

reject

refuse without authentication

Various methods can be specified in the authentication method field. To begin with, let's look at the two simplest ones.

The "trust" method unconditionally trusts the user and does not perform verification. In real life, it should never be used for anything but local connections.

The "reject" method unconditionally denies access. It can be used to cut off any connections of a certain type or from certain addresses (for example, to prohibit unencrypted connections).

## Question

What does the configuration below mean?

| #         | TYPE     | DATABASE | USER    | ADDRESS | METHOD |
|-----------|----------|----------|---------|---------|--------|
| hostnossl | all      |          | all     | all     | reject |
| host      | sameuser |          | all     | samenet | trust  |
| host      | pub      |          | +reader | all     | trust  |

14

1. Unencrypted connections are prohibited.
  2. Users are allowed to access databases that match their user names from server's subnet.
  3. Users who are members of the reader role are allowed access to the pub database.
- Note that the first record cannot be moved down, or the configuration result will change.

## Editing pg\_hba.conf

Back up pg\_hba.conf, so that we could restore it when we are done experimenting.

```
student$ sudo cp -n /etc/postgresql/13/main/pg_hba.conf ~/pg_hba.conf.orig
```

Another way to display the contents of pg\_hba.conf is through the pg\_hba\_file\_rules view:

```
=> SELECT line_number, type, database, user_name, address, auth_method
FROM pg_hba_file_rules;
```

| line_number | type  | database      | user_name  | address   | auth_method |
|-------------|-------|---------------|------------|-----------|-------------|
| 89          | local | {all}         | {postgres} |           | trust       |
| 94          | local | {all}         | {all}      |           | trust       |
| 96          | host  | {all}         | {all}      | 127.0.0.1 | md5         |
| 98          | host  | {all}         | {all}      | ::1       | md5         |
| 101         | local | {replication} | {all}      |           | trust       |
| 102         | host  | {replication} | {all}      | 127.0.0.1 | md5         |
| 103         | host  | {replication} | {all}      | ::1       | md5         |

(7 rows)

The view reads the file itself, not displays previously scanned values. You can use it to check if the changes you make actually apply.

For example, add the following string to pg\_hba.conf:

```
student$ echo 'local all all trust' | sudo tee -a /etc/postgresql/13/main/pg_hba.conf
```

```
local all all trust
```

```
=> SELECT line_number, error
FROM pg_hba_file_rules
WHERE error IS NOT NULL;
```

| line_number | error                                 |
|-------------|---------------------------------------|
| 104         | invalid authentication method "trust" |

(1 row)

Without the help of the view, we would have learned of the error only from the server log and after the configuration file is rescanned.

The server requests a password from the client



**password**

transmitted unencrypted

**md5**

an MD5 hash is transmitted

**scram-sha-256**

the SCRAM protocol is used

During password authentication, the PostgreSQL server requests a password from the user and checks it matches the password stored either in the database itself or in an external service.

<https://www.postgrespro.com/docs/postgresql/13/protocol-flow#id-1.10.5.7.3>

For passwords stored in the database, three methods are supported.

The **md5** method compares the MD5 hash of the password with the MD5 hash stored in the database. Upon request, the server sends the so-called "salt" to the client, the client calculates the MD5 hash of the password, adds the salt, calculates the MD5 hash again and sends it to the server, where it is compared with the stored hash. Thanks to the salt, the same password can result in different hash values. However, the MD5 algorithm is currently considered insufficiently cryptographically secure.

The most secure method **scram-sha-256** uses the SCRAM protocol for authentication and employs the SHA-256 algorithm. The method implements the SASL framework that separates the authentication mechanism from the application protocol.

<https://postgrespro.com/docs/postgresql/13/sasl-authentication>

The **password** method transmits the password in plain text. It should not be used if the client-server connection is not encrypted.

## Set a user password

```
[ CREATE | ALTER ] ROLE ...  
  PASSWORD 'password'  
[ VALID UNTIL date_time ];
```

a user with an empty password will be denied access during password authentication

## Passwords are stored in the system catalog

`pg_authid`

the encryption method is determined by the *password\_encryption* parameter

the authentication method must match the encryption method

(md5 automatically switches to scram-sha-256)

So far, we've been creating roles without specifying any passwords. If the password authentication method is set, such users will be denied access.

Passwords are stored in the database in the `pg_authid` table.

To set a password, you must specify it either immediately when creating a role with the `CREATE ROLE` command, or later with the `ALTER ROLE` command. Passwords are stored in encrypted form. The encryption algorithm (MD5 or SCRAM-SHA-256) is determined by the *password\_encryption* parameter.

You can optionally specify a password expiration time.

If a stored password is encrypted with the SCRAM-SHA-256 algorithm and the authentication method is set to use MD5, the more reliable SCRAM-SHA-256 method will be used during communication instead.

# Entering a password



## Manually

### Set the PGPASSWORD variable

inconvenient when connecting to different databases  
not recommended for security reasons

### The passwords file

~/.pgpass at the client host  
lines in the format `host:port:database:username:password`  
may use the \* sign (any value)  
records are checked from top to bottom, the first match is used  
the file must have permission 600 (rw-----)

19

The password can be entered manually every time, or the input can be automated. There are two ways to do it.

First, the password can be set in the PGPASSWORD environment variable (on the client). However, this is inconvenient if you frequently connect to multiple databases. It also poses some security risks.

Otherwise, you can store passwords in the ~/.pgpass file (its location is defined by the PGPASSFILE environment variable). Access to the file must be restricted to the owner alone, or PostgreSQL will ignore it.

## Password-based authentication

Let's set up password-based authentication for local connections for the user student. First, we can check how passwords are encrypted when saved.

```
=> SHOW password_encryption;
```

```
password_encryption
-----
md5
(1 row)
```

We want to use a secure encryption algorithm.

```
=> SET password_encryption='scram-sha-256';
```

SET

Now, set up a password for student. The password may contain any Unicode characters.

```
=> ALTER ROLE student PASSWORD 'p@ssword';
```

ALTER ROLE

What's left is to set up the authentication rules:

```
student$ sudo tee /etc/postgresql/13/main/pg_hba.conf << EOF
local all postgres trust
local all student  scram-sha-256
EOF
```

```
local all postgres trust
local all student  scram-sha-256
```

If the encryption method MD5 is specified instead, the system would still use SCRAM-SHA-256 for student, but other users would have been able to store passwords encrypted with the weaker MD5.

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf
-----
t
(1 row)
```

Try to guess the password:

```
student$ psql 'user=student password=1234'
```

psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed: FATAL: password authentication failed for user "student"

Now, enter the correct one:

```
student$ psql 'user=student password=p@ssword' -c '\conninfo'
```

You are connected to database "student" as user "student" via socket in "/var/run/postgresql" at port "5432".

Restore the original pg\_hba.conf.

```
student$ sudo cp ~/pg_hba.conf.orig /etc/postgresql/13/main/pg_hba.conf
```

```
student$ sudo pg_ctlcluster 13 main reload
```

`ldap [parameters]`

passwords are stored on a LDAP server

`radius [parameters]`

passwords are stored on a RADIUS server

`pam [parameters]`

passwords are stored in the PAM plugin

Passwords can be stored by external services outside the database.

The methods “ldap”, “radius” and “pam” use an LDAP server, a RADIUS server, or the Pluggable Authentication Module, respectively. These methods require additional specific parameters. They are not considered in detail in this topic.

Performed outside the database

`peer [map=...]`

query the username from the OS kernel (for local connections)

`cert [map=...]`

authentication using the client's SSL certificate

`gss [map=... and other parameters]`

Kerberos authentication over the GSSAPI protocol

`sspi [map=... and other parameters]`

Kerberos/NTLM authentication for Windows

These methods carry out both identification and authentication outside the database. Upon successful authentication, PostgreSQL receives two names:

1. The name specified during connection (internal DBMS name).
2. The name identified by the external system (external name).

Therefore, all of these methods allow for at least one additional parameter "map". It defines the mapping rules for internal and external names (more on that on the next slide).

The "peer" method requests the user name from the OS kernel. Since the OS has already authenticated this user (most likely by requesting a password), it can be trusted.

The "cert" method uses client certificate-based authentication and is intended for SSL connections only.

The "gss" method uses Kerberos authentication over the GSSAPI protocol (RFC1964 <https://tools.ietf.org/html/rfc1964>). Automatic authentication (single sign-on) is supported.

The "sspi" method uses Kerberos or NTLM authentication on Windows systems. Automatic authentication is supported.

## pg\_ident.conf

another configuration file

lines consist of fields, space or tab-separated

empty lines and any text after a comment sign (#) are ignored

## Fields

the name of the mapping

(specified in the map parameter in pg\_hba.conf)

external name

(if starts with a slash, then considered a regular expression)

internal DB user name

The name matching rules are defined in a separate file `pg_ident.conf`. Its location is determined by the `ident_file` parameter. `pg_ident.conf` structure is similar to that of `pg_hba.conf`. Records consist of three fields: mapping name, external user name, internal user name.

Mapping names are necessary to distinguish between different mapping rules within the same `pg_ident.conf` file (which, in turn, may be required by different records in `pg_hba.conf`).

The external name must match the name returned by the external authentication system or the one listed in the certificate. If this field starts with a slash, then its value is considered a regular expression. This can be used to handle situations where the external and internal names differ only by prefixes or suffixes.

The internal name must match the name of the database user.

Each record mapping an internal user name to an external user name means that the specified external user is allowed to connect to the DBMS as the specified internal user (after a successful authentication, of course).

<https://postgrespro.com/docs/postgresql/13/auth-username-maps>



# Question

What does the configuration below do?

## pg\_hba.conf

| # | TYPE    | DATABASE | USER | ADDRESS  | METHOD |        |
|---|---------|----------|------|----------|--------|--------|
|   | hostssl | sameuser | all  | all      | cert   | map=m1 |
|   | local   | all      | all  |          | peer   | map=m2 |
|   | host    | all      | all  | samehost | md5    |        |

## pg\_ident.conf

| # | MAPNAME | SYSTEM-USERNAME      | PG-USERNAME |
|---|---------|----------------------|-------------|
|   | m1      | /^(.*)@domain\.com\$ | \1          |
|   | m2      | student              | alice       |
|   | m2      | student              | bob         |

SSL connections are authenticated using a client certificate. It is assumed that the name (common name) in the certificate is stored as "user@domain.com", and user is considered the role name. For local connections, PostgreSQL requests the user name from the operating system. The m2 mapping says that the OS user "student" can connect to the DB under the roles "alice" and "bob". The network connection to the local server is authenticated by a password (encrypted by MD5).

Authentication settings are defined in configuration files

Authentication can be done by password (with the password stored within the DBMS or outside of it) or using external authentication services

1. Modify the configuration files (after backing up the originals) in such a way that:  
the superusers `student` and `postgres` are always allowed local connections,  
all users are allowed network connections to all databases with password-based authentication using MD5 encryption.
2. Create a role `alice` with an MD5 encrypted password and a role `bob` with a SCRAM-SHA-256 encrypted password.
3. Verify that the created roles can connect to the database.
4. As a superuser, look at the passwords of `alice` and `bob` in the system catalog.
5. Restore the original configuration files.

## 1. Authentication configuration

Save the original configuration file:

```
student$ sudo cp -n /etc/postgresql/13/main/pg_hba.conf ~/pg_hba.conf.orig
```

Now create a new pg\_hba.conf file from scratch:

```
student$ sudo tee /etc/postgresql/13/main/pg_hba.conf << EOF
local all postgres trust
local all student trust
host all all all md5
EOF
```

```
local all postgres trust
local all student trust
host all all all md5
```

```
student$ sudo pg_ctlcluster 13 main reload
```

## 2. Creating roles

```
=> SHOW password_encryption;
```

```
password_encryption
-----
md5
(1 row)
```

```
=> CREATE ROLE alice LOGIN PASSWORD 'alice';
```

```
CREATE ROLE
```

```
=> SET password_encryption='scram-sha-256';
```

```
SET
```

```
=> CREATE ROLE bob LOGIN PASSWORD 'bob';
```

```
CREATE ROLE
```

## 3. Verifying connection settings

The connection settings require the user to enter a password. We will provide it in the connection string.

Note that for this task, it's better to enter it explicitly to verify that the system asks for it.

```
=> \c "dbname=student user=alice host=localhost password=alice"
```

You are now connected to database "student" as user "alice" on host "localhost" (address "127.0.0.1") at port "5432".

```
=> \c "dbname=student user=bob host=localhost password=bob"
```

You are now connected to database "student" as user "bob".

## 4. Viewing passwords

```
=> \q
```

```
student$ psql
```

```
=> SELECT rolname, rolpassword FROM pg_authid WHERE rolname IN ('alice','bob') \gx
```

```
-[ RECORD 1 ]-----
rolname      | alice
rolpassword  | md5579e43b423b454623383471aeb85cd87
-[ RECORD 2 ]-----
rolname      | bob
rolpassword  | SCRAM-SHA-256$4096:RxfUKG0EtG1gUpu86ZRKiQ==SVQE5lfRIzwftd+kS1pQsJpEV23iGcIMlaTubhT/eM=:sLlRyDwn3yL/nECwzLVKJ1+xSct+WcnLzx17Jf7DI=
```

Passwords are stored as hash function values that cannot be decrypted. The server always compares the encrypted values the encrypted value stored in pg\_authid and the entered password's encrypted hash.

## 5. Restoring the default configuration

```
student$ sudo cp ~/pg_hba.conf.orig /etc/postgresql/13/main/pg_hba.conf
```

```
student$ sudo pg_ctlcluster 13 main reload
```

Certain users, a list of whom is subject to change from time to time, must be allowed local access without authorization. The problem is that changing the list of trusted users requires changing the `pg_hba.conf` file every time.

1. Set up authentication that does not have this problem.
2. Verify that the new configuration works as intended.
3. Restore the original configuration.

1. Use a group role.

## 1. Authentication configuration

Save the original configuration file:

```
student$ sudo cp -n /etc/postgresql/13/main/pg_hba.conf ~/pg_hba.conf.orig
```

We will control what users to authenticate by adding them into the locals group.

Overwrite the existing pg\_hba.conf file:

```
student$ sudo tee /etc/postgresql/13/main/pg_hba.conf << EOF
local all student trust
local all +locals trust
EOF
```

```
local all student trust
local all +locals trust
```

```
student$ sudo pg_ctlcluster 13 main reload
```

Create a group role:

```
=> CREATE ROLE locals;
```

```
CREATE ROLE
```

## 2. Verification

Alice belongs to the locals group:

```
=> CREATE ROLE alice LOGIN;
```

```
CREATE ROLE
```

```
=> GRANT locals TO alice;
```

```
GRANT ROLE
```

Bob does not:

```
=> CREATE ROLE bob LOGIN;
```

```
CREATE ROLE
```

```
student$ psql "dbname=student user=alice" -c "\conninfo"
```

You are connected to database "student" as user "alice" via socket in "/var/run/postgresql" at port "5432".

```
student$ psql "dbname=student user=bob" -c "\conninfo"
```

psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed: FATAL: no pg\_hba.conf entry for host "[local]", user "bob", database "student", SSL off

Grant Bob membership:

```
=> GRANT locals TO bob;
```

```
GRANT ROLE
```

```
student$ psql "dbname=student user=bob" -c "\conninfo"
```

You are connected to database "student" as user "bob" via socket in "/var/run/postgresql" at port "5432".

## 2. Restoring the default configuration

```
student$ sudo cp ~/pg_hba.conf.orig /etc/postgresql/13/main/pg_hba.conf
```

```
student$ sudo pg_ctlcluster 13 main reload
```