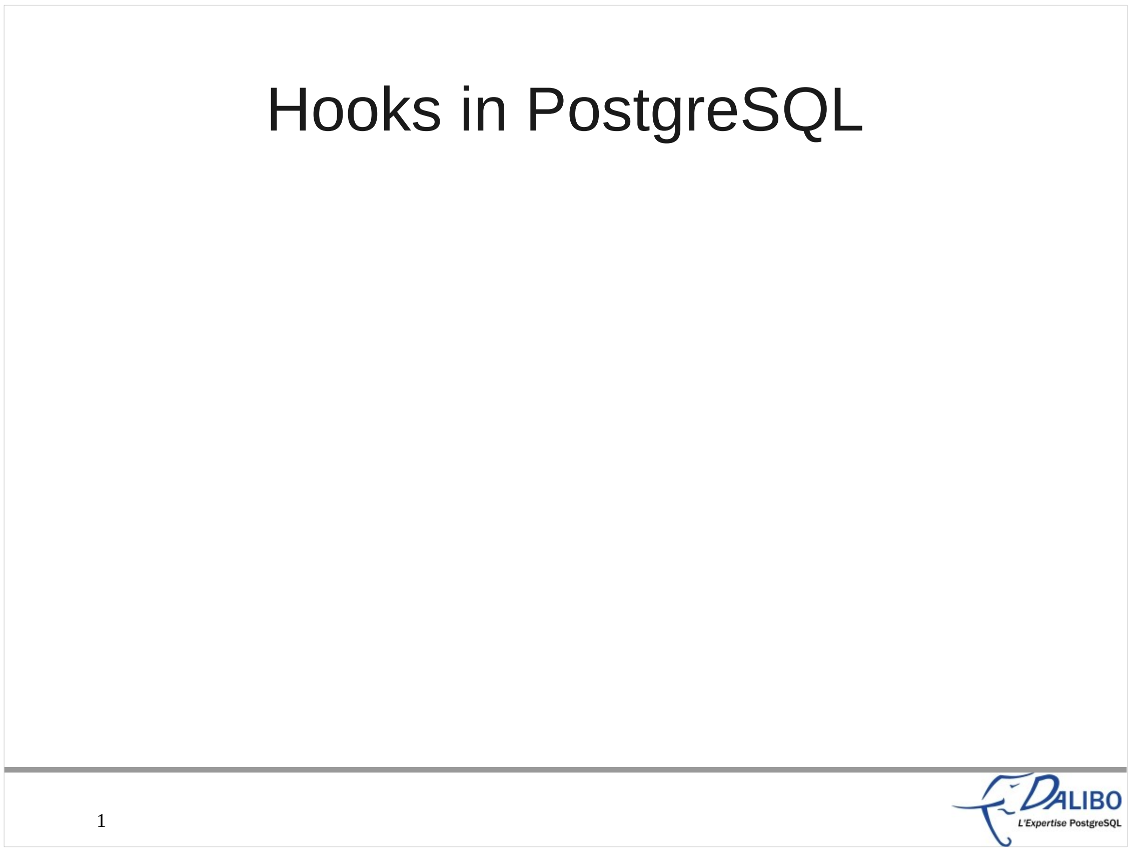


Hooks in PostgreSQL

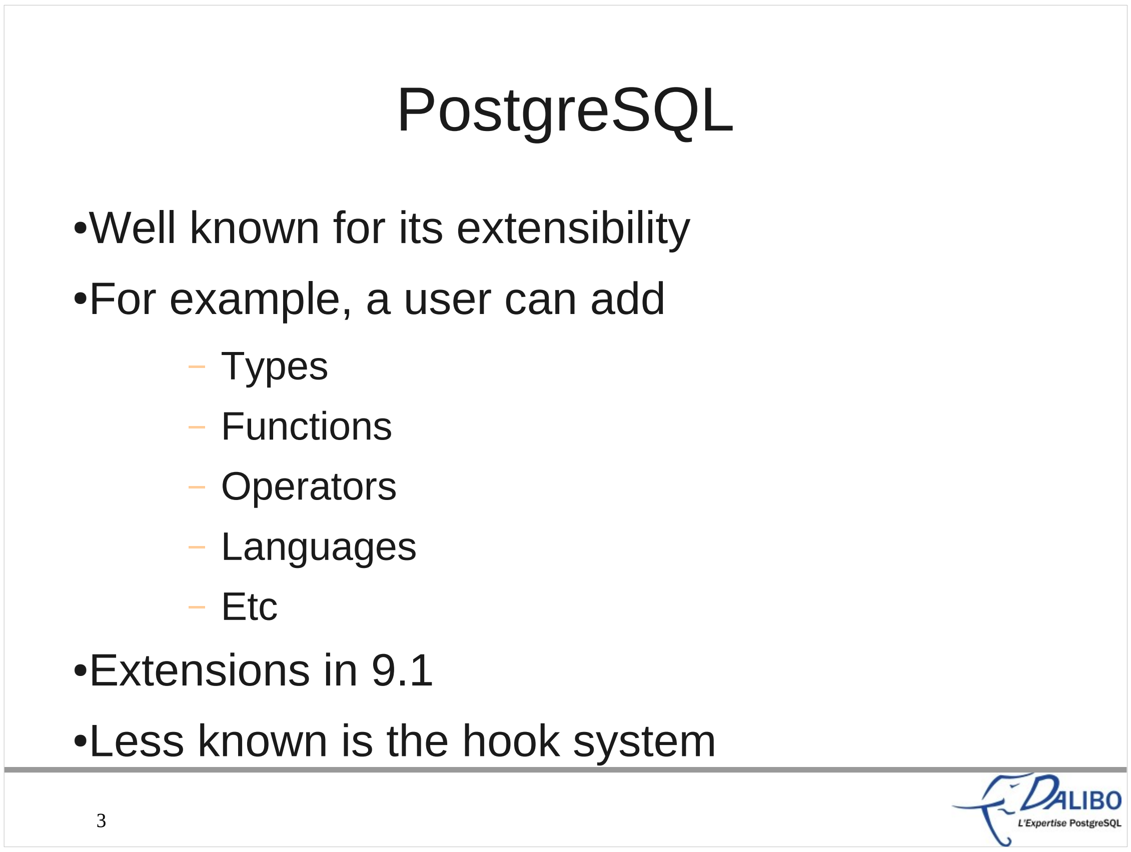
1

This talk will present a quite unknown feature of PostgreSQL: its hook system.

Who's Guillaume Lelarge?

- French translator of the PostgreSQL manual
- Member of pgAdmin's team
- Vice-treasurer of PostgreSQL Europe
- CTO of Dalibo

- Mail: guillaume@lelarge.info
- Twitter: g_lelarge
- Blog: http://blog.guillaume.lelarge.info



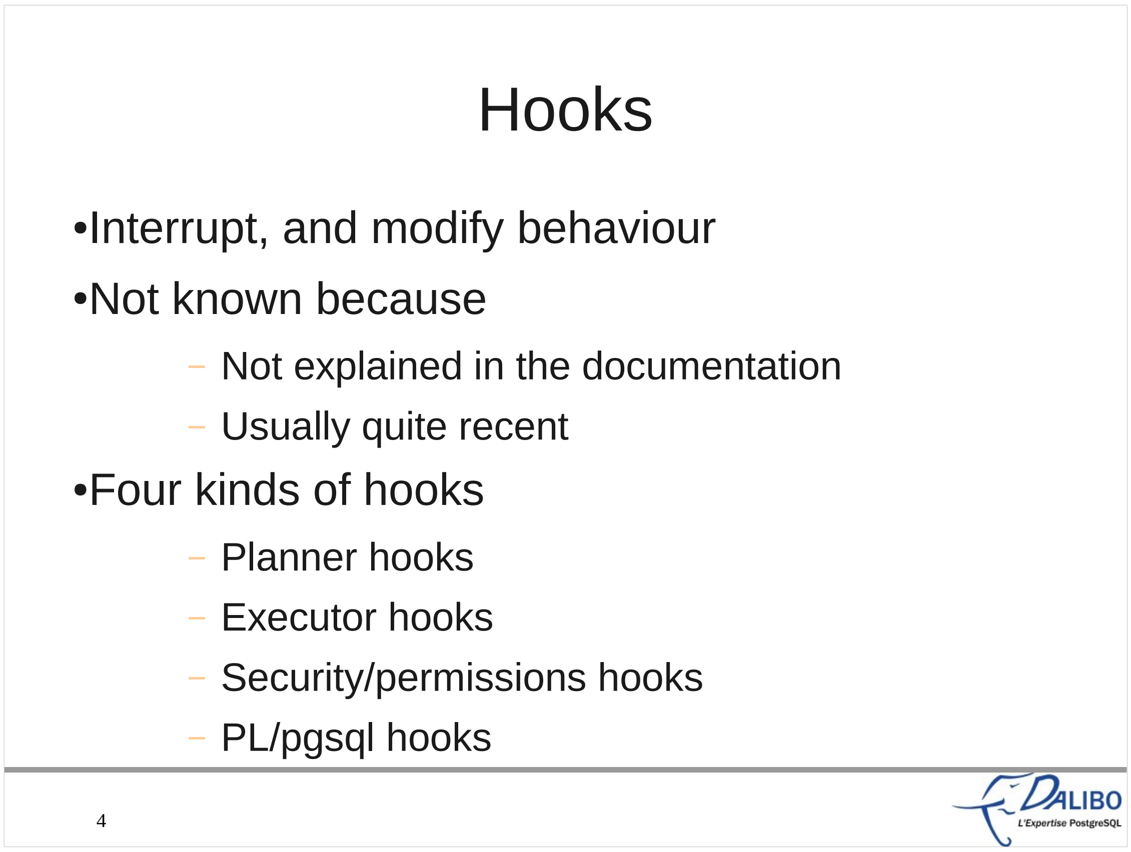
PostgreSQL

- Well known for its extensibility
- For example, a user can add
  - Types
  - Functions
  - Operators
  - Languages
  - Etc
- Extensions in 9.1
- Less known is the hook system

3

PostgreSQL is well known for its extensibility. Many people know that you can add your own user types, add functions that handle them, add operators which use those functions, and lots of other stuff.
Heikki even did an interesting talk at last year's FOSDEM about user types and how to use them.
Many procedural languages are supported.
Actually, the extensibility is so important to the PostgreSQL project that one of the most interesting features of 9.1 is the new EXTENSION object, which helps the handling of external modules, plugins, or whatever you want to call that.

With all this going on with the extensibility, it's quite strange that the hook system is rather unknown, even if the first hooks were available since the 8.3 release.



Hooks

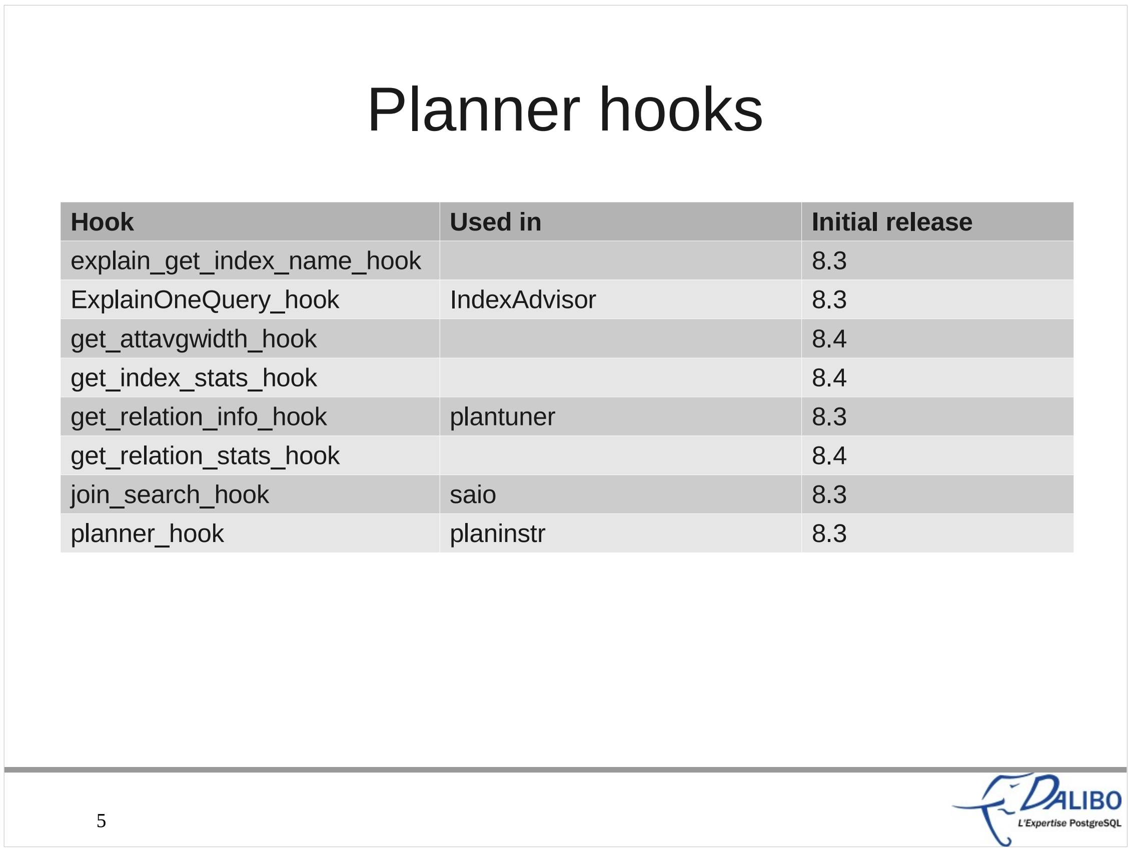
- Interrupt, and modify behaviour
- Not known because
  - Not explained in the documentation
  - Usually quite recent
- Four kinds of hooks
  - Planner hooks
  - Executor hooks
  - Security/permissions hooks
  - PL/pgsql hooks

The aim of hooks is to interrupt and modify the usual behaviour of PostgreSQL. It allows a developer to add new features without having to add it to the core.

It's not well known because it's a rather recent feature. The first hook appeared in 8.3. Actually, 5 hooks appeared in 8.3, 8 more in 8.4, 2 more in 9.0, and 5 more in 9.1. But the biggest issue is probably that it's not discussed in the documentation.

Of course, there are different kinds of hooks, mostly around the planner, the executor, and security/permissions.

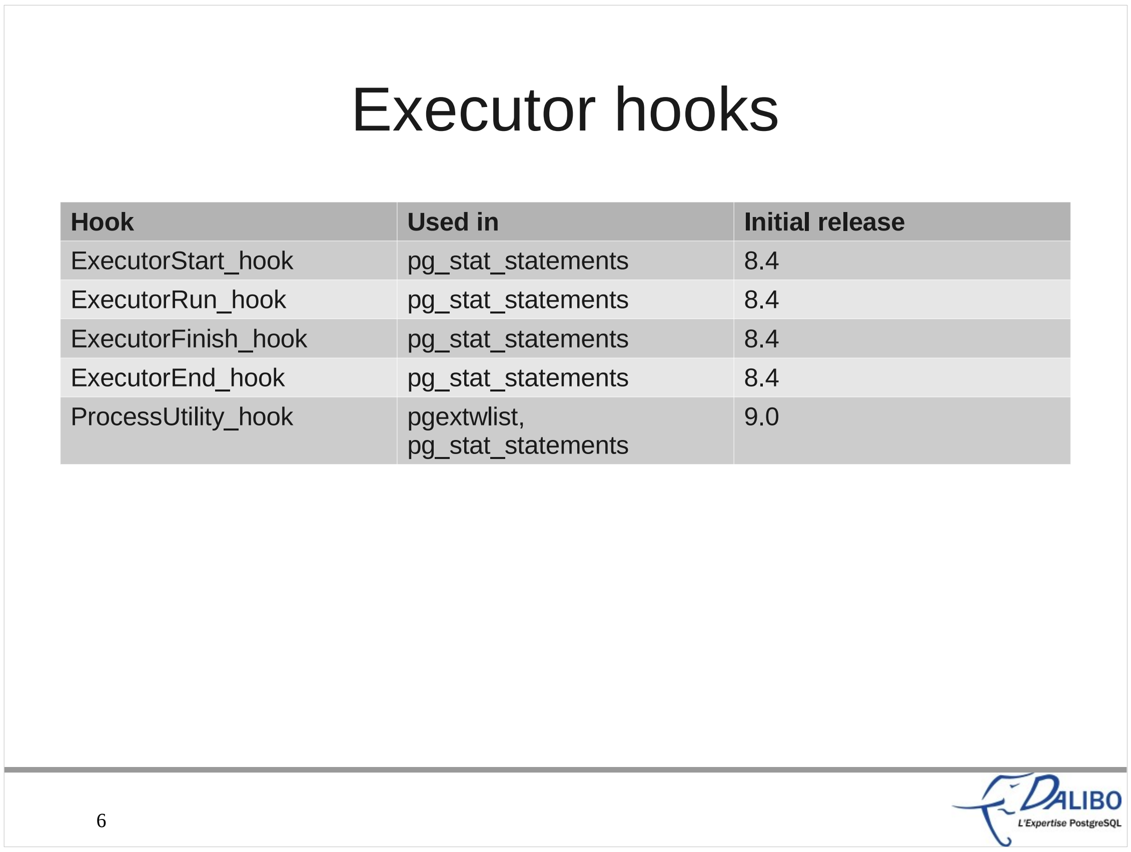
So let's see these hooks...



Planner hooks

Hook	Used in	Initial release
explain_get_index_name_hook		8.3
ExplainOneQuery_hook	IndexAdvisor	8.3
get_attavgwidth_hook		8.4
get_index_stats_hook		8.4
get_relation_info_hook	plantuner	8.3
get_relation_stats_hook		8.4
join_search_hook	saio	8.3
planner_hook	planinstr	8.3

explain_get_index_name_hook, called when explain finds indexes' names, to allow plugins to get control here so that plans involving hypothetical indexes can be explained
ExplainOneQuery_hook see http://archives.postgresql.org/pgsql-patches/2007-05/msg00421.php
get_relation_info_hook, allows modification of expansion of the information PostgreSQL gets from the catalogs for a particular relation, including adding fake indexes (http://www.sai.msu.su/~megera/wiki/plantuner to enable planner hints which allow enable/disable indexes, fix empty table)
join_search_hook, to let plugins override the join search order portion of the planner; this is specifically intended to simplify developing a replacement for GEQO planning, example module saio (http://pgxn.org/dist/saio/), a join order search plugin using simulated annealing which provides an experimental planner module that uses a randomised algorithm to try to find the optimal join order
planner_hook, runs when the planner begins, so plugins can monitor or even modify the planner's behavior (http://pgxn.org/dist/planinstr/) to measure planner running time



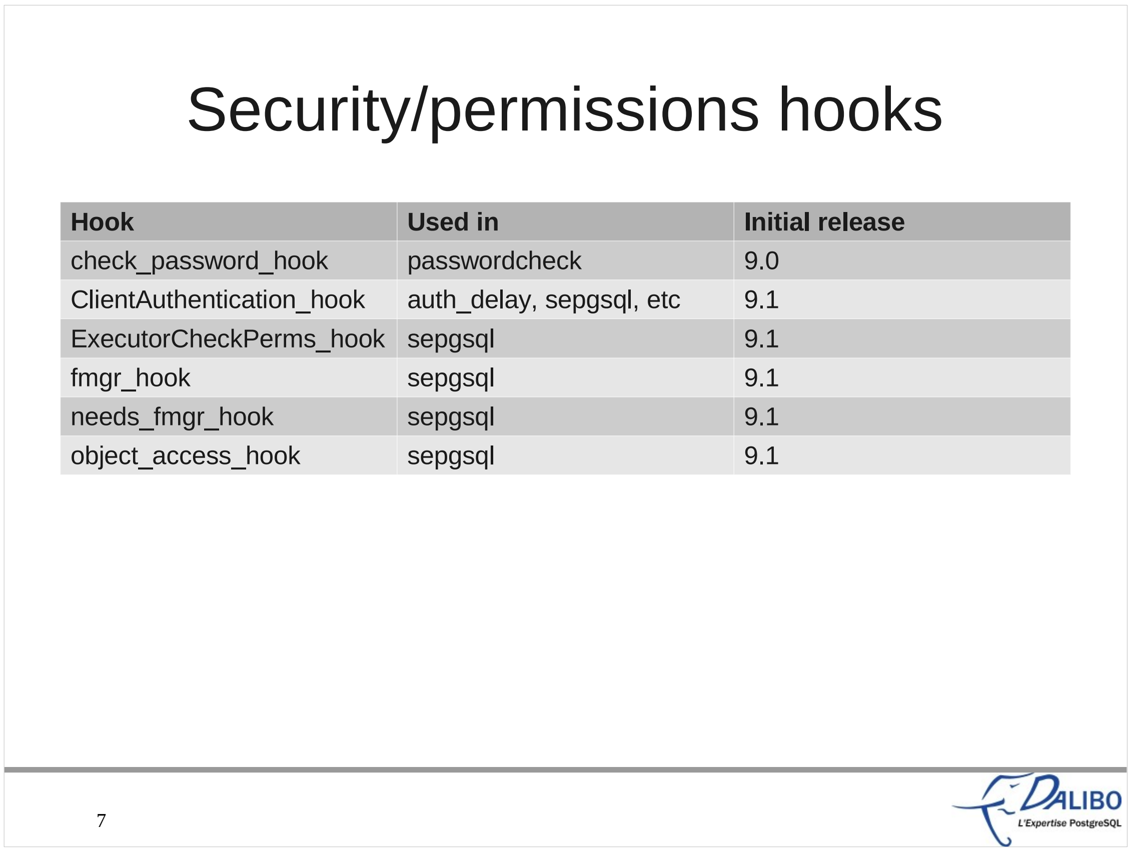
Executor hooks

Hook	Used in	Initial release
ExecutorStart_hook	pg_stat_statements	8.4
ExecutorRun_hook	pg_stat_statements	8.4
ExecutorFinish_hook	pg_stat_statements	8.4
ExecutorEnd_hook	pg_stat_statements	8.4
ProcessUtility_hook	pgextwlist, pg_stat_statements	9.0

All the Executor hooks and the ProcessUtility hook help running functions that will use information from the executor. Mostly used to know which queries are executed, so that you can compute statistics, or log them.

pg_stat_statements uses all of them.

The ProcessUtility hook is used by pgextwlist (see https://github.com/dimitri/pgextwlist for details).



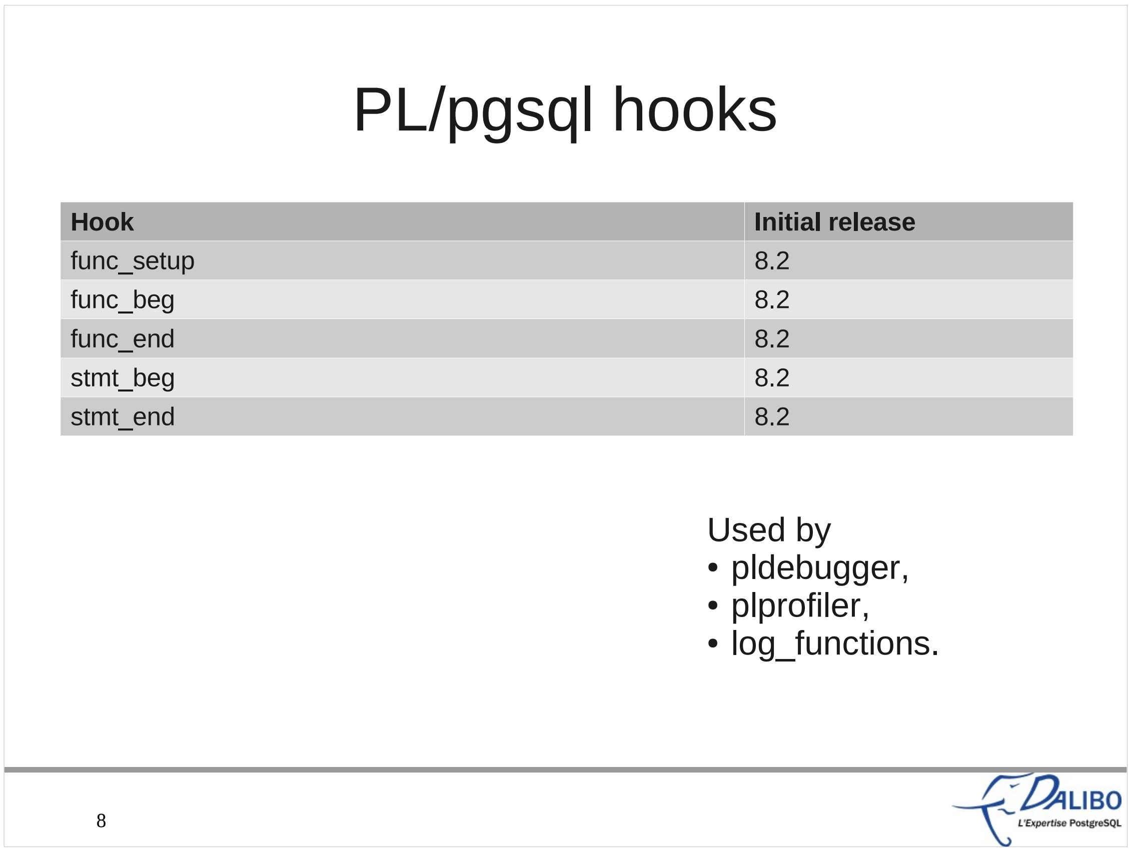
Security/permissions hooks

Hook	Used in	Initial release
check_password_hook	passwordcheck	9.0
ClientAuthentication_hook	auth_delay, sepgsql, etc	9.1
ExecutorCheckPerms_hook	sepgsql	9.1
fmgr_hook	sepgsql	9.1
needs_fmgr_hook	sepgsql	9.1
object_access_hook	sepgsql	9.1

The check_password hook is a way to check passwords according to enterprise ruleswhen a user is created and when he changes his password.

The ClientAuthentication hook makes it possible to add other checks to allow or deny connections.

The other ones are used by sepgsql.

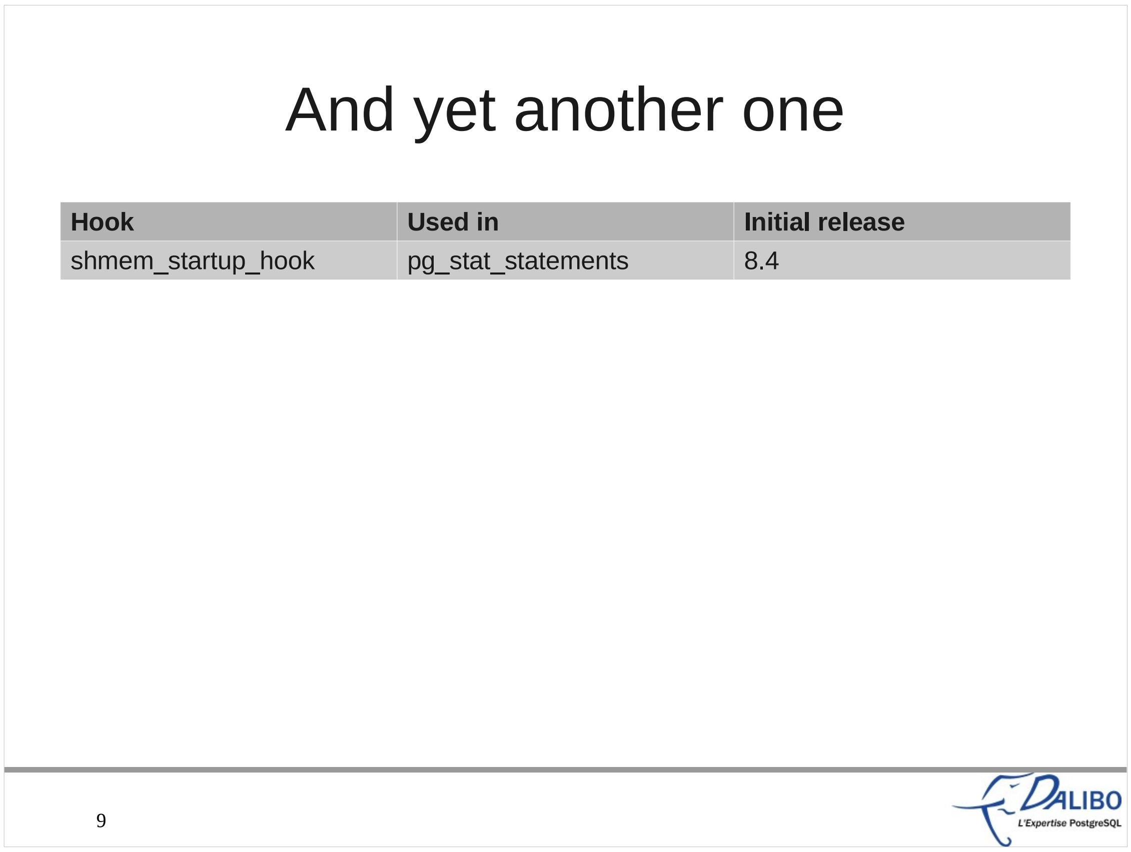


PL/pgsql hooks

Hook	Initial release
func_setup	8.2
func_beg	8.2
func_end	8.2
stmt_beg	8.2
stmt_end	8.2

Used by
- pldebugger,
- plprofiler,
- log_functions.

The PL/pgsql language allows a shared library to hook plugins. AFAIK, its only use is by the debugger, and the profiler written by EnterpriseDB. But there's also a new extension called log_functions which uses them.

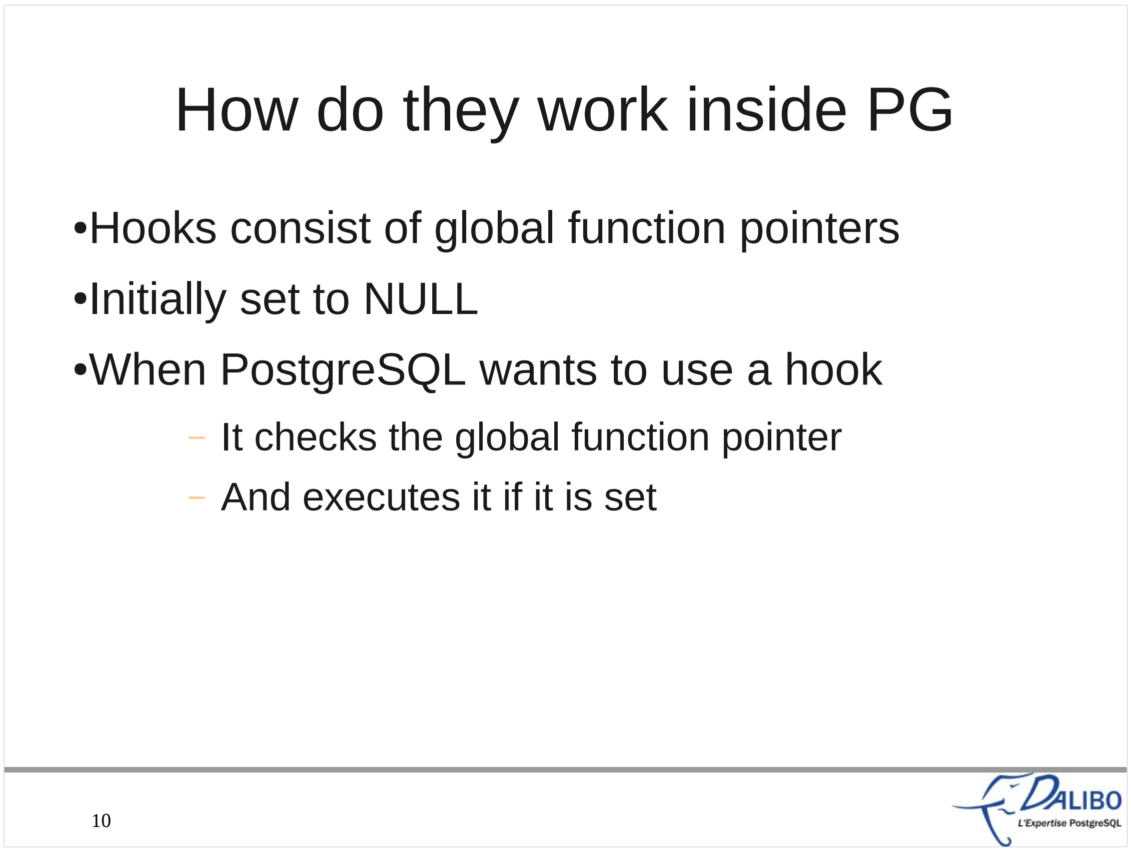


And yet another one

Hook	Used in	Initial release
shmem_startup_hook	pg_stat_statements	8.4

9

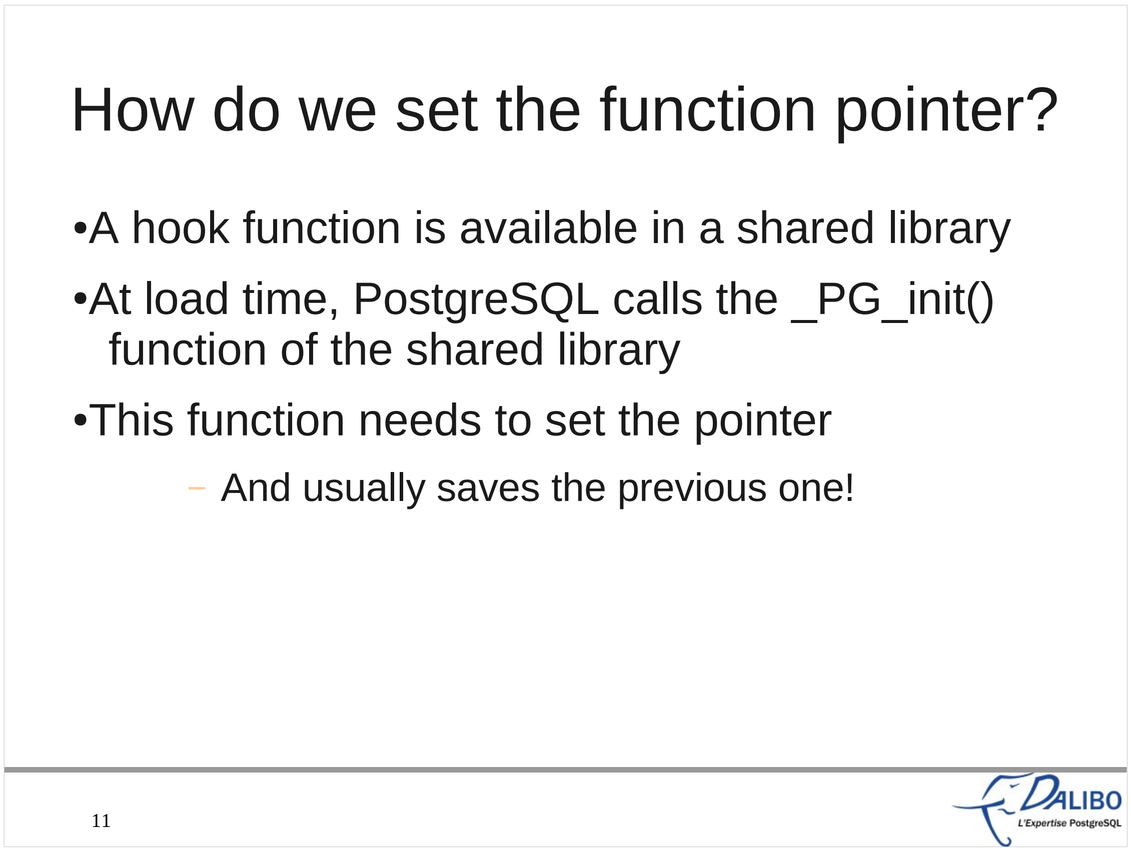
shmem_startup_hook, called when PostgreSQL initializes its shared memory segment



How do they work inside PG

- •Hooks consist of global function pointers
- •Initially set to NULL
- •When PostgreSQL wants to use a hook
  - It checks the global function pointer
  - And executes it if it is set

Each hook consists of a global function pointer. It's initialy set to NULL. When PostgreSQL may have to execute it, it checks if the global function pointer is still set to NULL. If it's set to something else, it executes the function pointer.

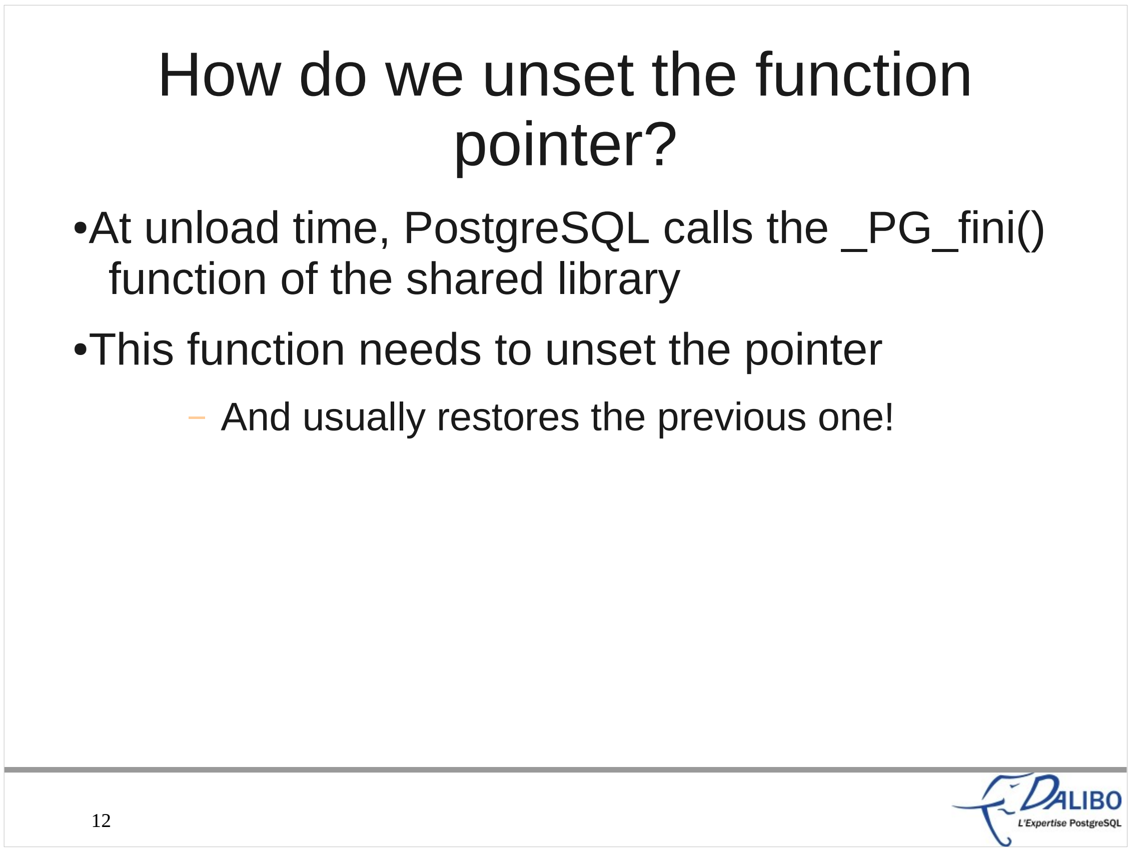


How do we set the function pointer?

- A hook function is available in a shared library
- At load time, PostgreSQL calls the _PG_init() function of the shared library
- This function needs to set the pointer
  - And usually saves the previous one!

A shared library is a .so or .dll file, installed in the lib directory of PostgreSQL.

When PostgreSQL has to load a shared library, it first loads it into memory, and then executes a function called _PG_init. This function is available in many shared libraries, so that they can initialize memory and set up variables. For example, we can use that function to set the global function pointer with our own function. It's usually better to save the previous pointer. We may launch it at the beginning or at the end of our own function. We may reset it at unload time.

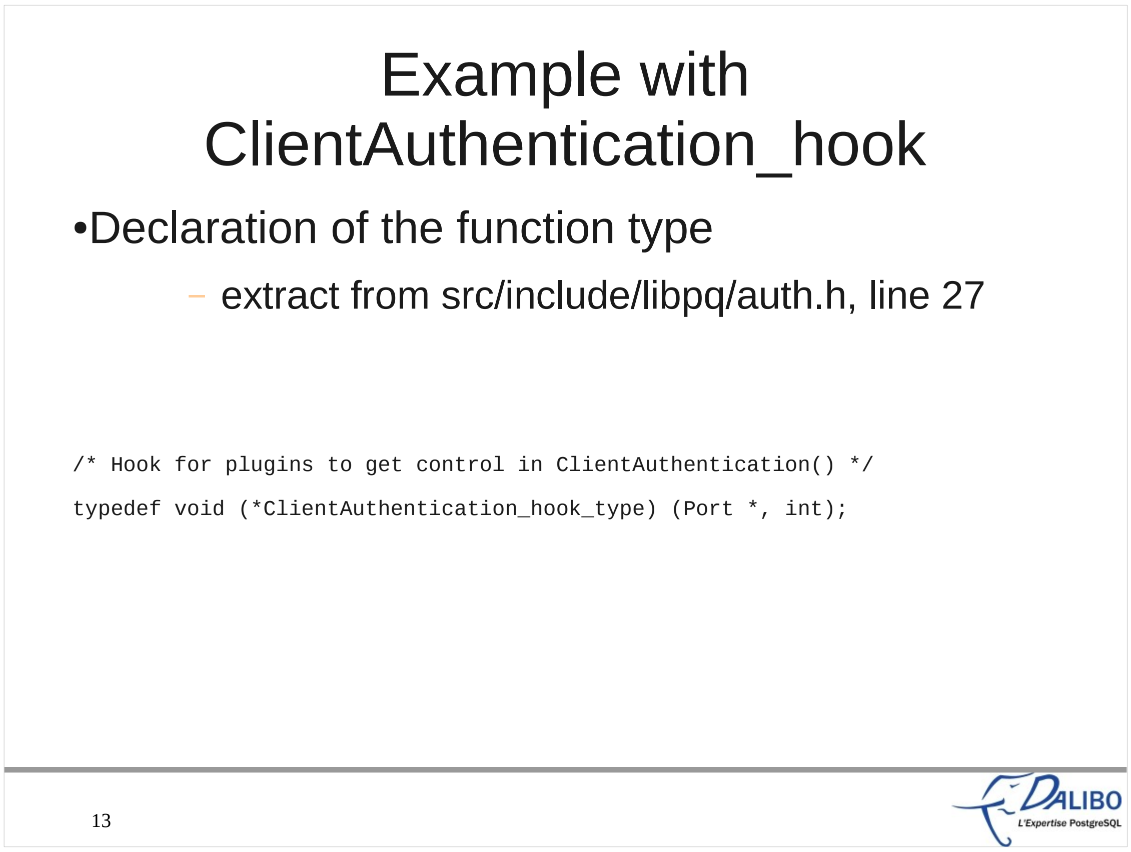


How do we unset the function pointer?

- At unload time, PostgreSQL calls the _PG_fini() function of the shared library
- This function needs to unset the pointer
  - And usually restores the previous one!

We have one function called at load time, we also have one at unload time.

When PostgreSQL needs to unload a shared library, it calls the _PG_fini() function of the shared library. This is the good time to restore the previous value of the function pointer, or at least to set it to NULL.



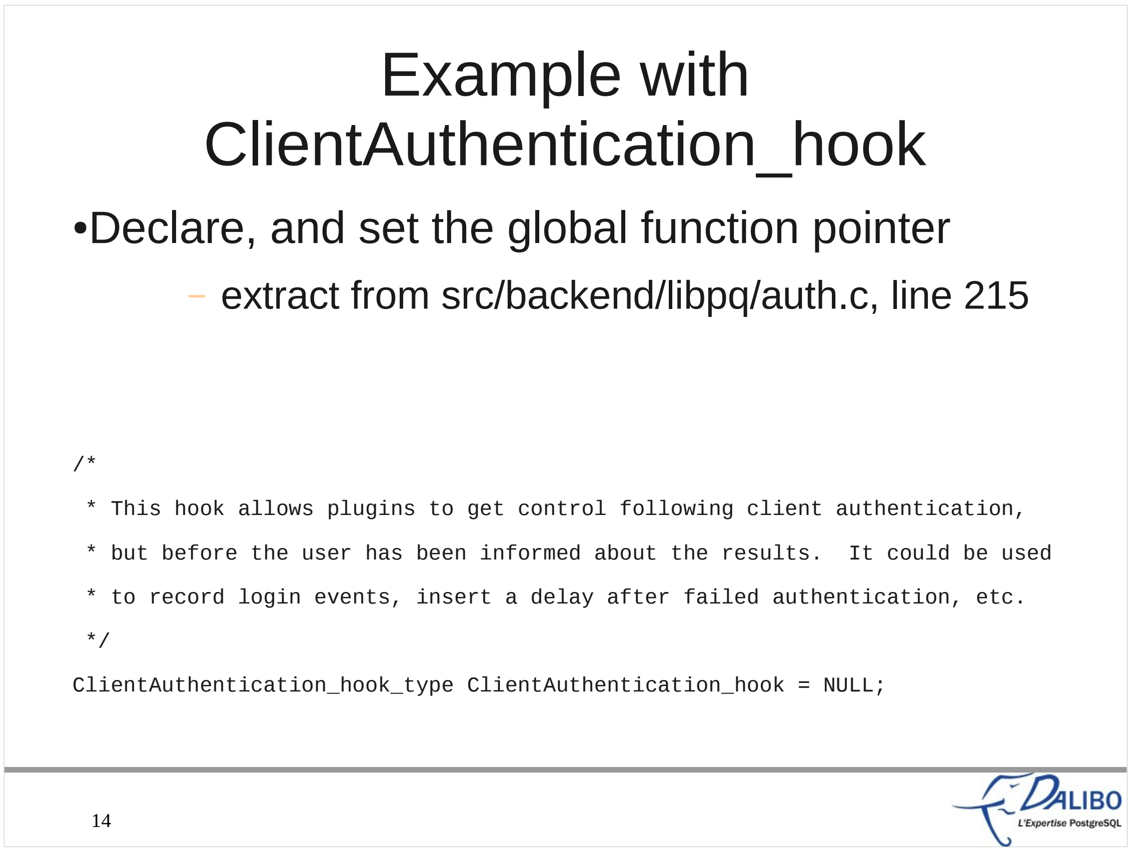
Example with ClientAuthentication_hook

- Declaration of the function type
  - extract from src/include/libpq/auth.h, line 27

```
/* Hook for plugins to get control in ClientAuthentication() */
typedef void (*ClientAuthentication_hook_type) (Port *, int);
```

13

This line declares the ClientAuthentication hook type.

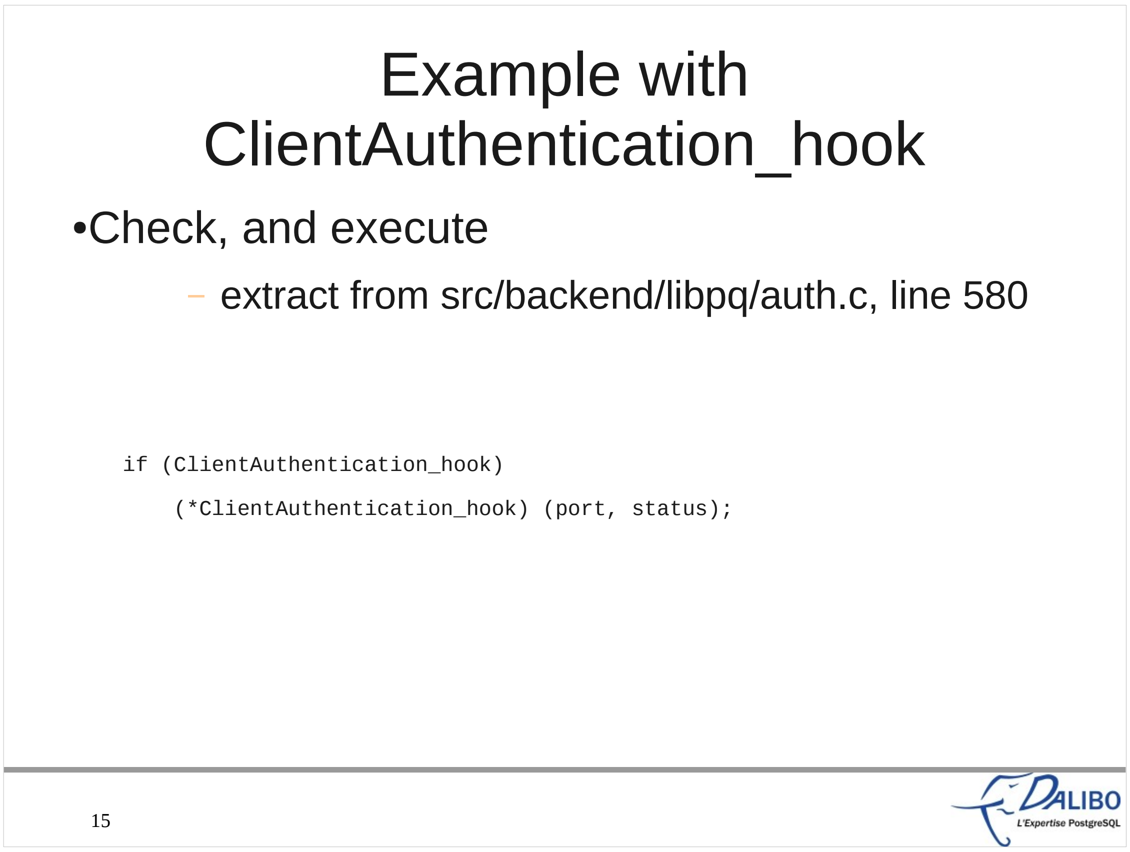


Example with ClientAuthentication_hook

- Declare, and set the global function pointer
  - extract from src/backend/libpq/auth.c, line 215

```
/*
 * This hook allows plugins to get control following client authentication,
 * but before the user has been informed about the results.  It could be used
 * to record login events, insert a delay after failed authentication, etc.
 */
ClientAuthentication_hook_type ClientAuthentication_hook = NULL;
```

This line declares the ClientAuthentication_hook global function pointer and sets its initial value to NULL.



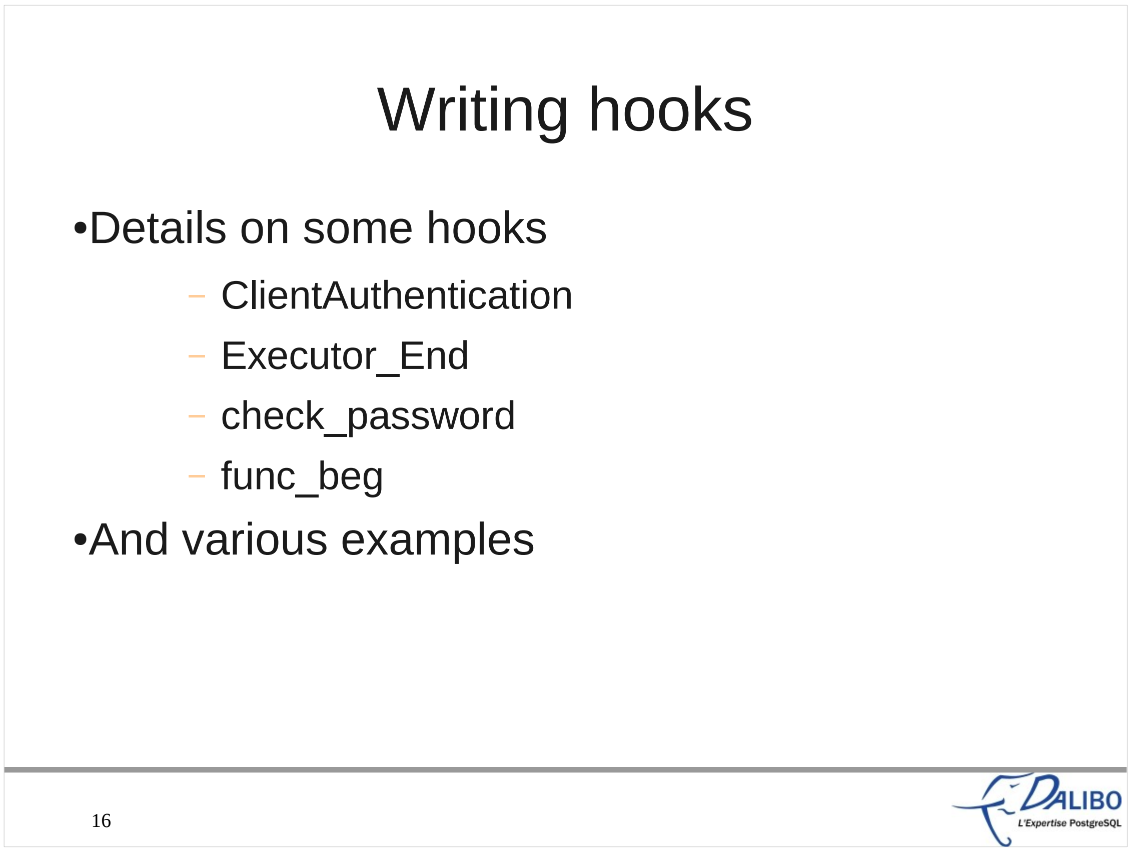
Example with ClientAuthentication_hook

•Check, and execute

– extract from src/backend/libpq/auth.c, line 580

```
if (ClientAuthentication_hook)
    (*ClientAuthentication_hook) (port, status);
```

These two lines check if the ClientAuthentication hook has been set up. If it has, the function is executed.

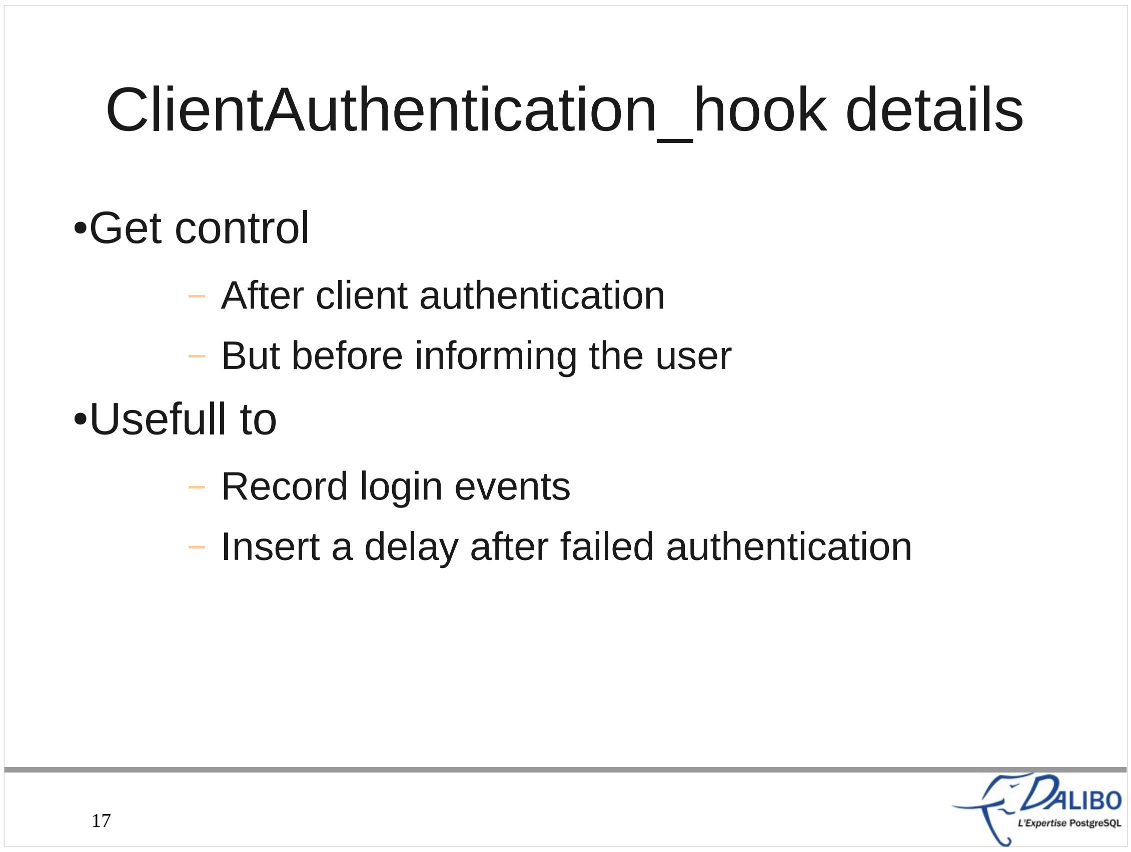


Writing hooks

- Details on some hooks
  - ClientAuthentication
  - Executor_End
  - check_password
  - func_beg
- And various examples

This part will go into much greater details on some of the available hooks: ClientAuthentication, the Executor_End, check_password, and func_beg.

We'll explain how usefull they are, list the already available extensions using them. We'll also see how to write a shared library that uses each of these hooks



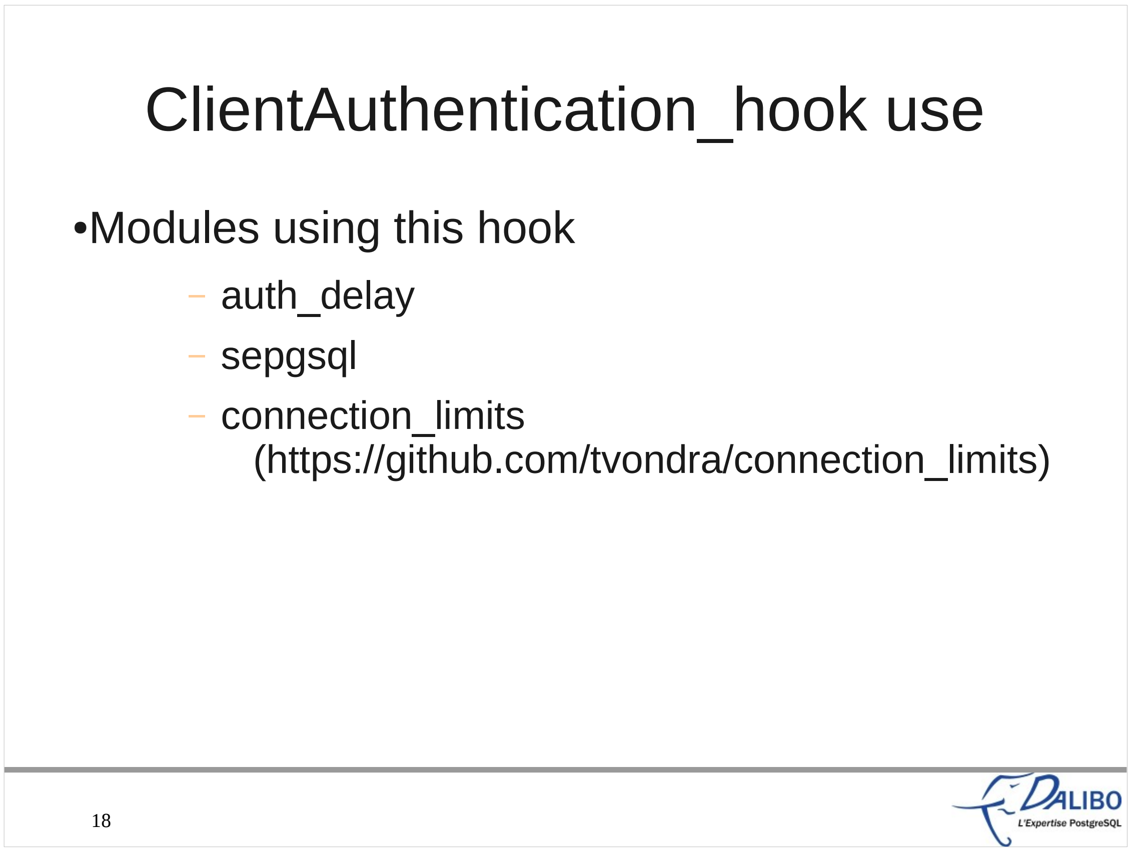
ClientAuthentication_hook details

- Get control
  - After client authentication
  - But before informing the user
- Usefull to
  - Record login events
  - Insert a delay after failed authentication

17

The ClientAuthentication_hook helps a plugin to get control after the client authentication, but before the client is informed of the result of the authentication.

Therefore, the plugin can do other stuff, like record login events (with the result of the authentication), or insert a delay after a failed authentication to avoid DOS attacks.

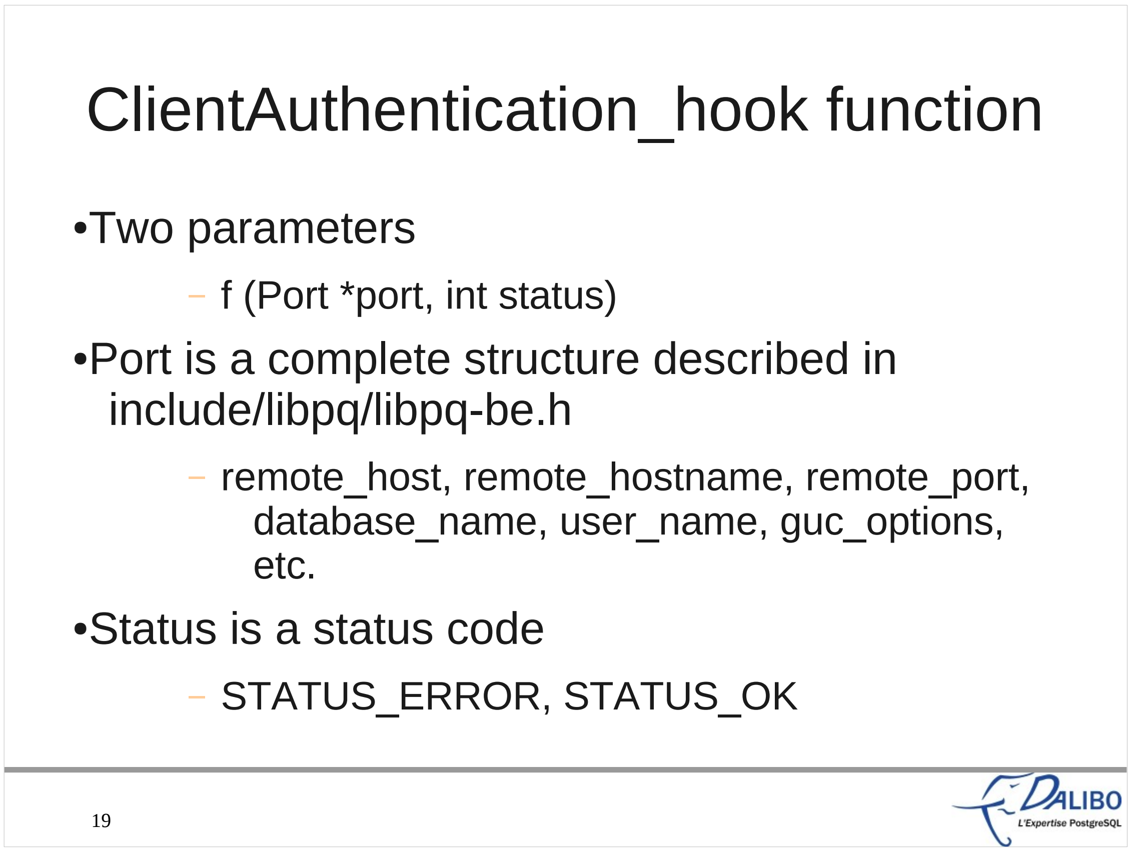


ClientAuthentication_hook use

- Modules using this hook
  - auth_delay
  - sepgsql
  - connection_limits (https://github.com/tvondra/connection_limits)

Three extensions already use this hook:

- auth_delay adds a configurable delay (auth_delay.milliseconds GUC) after a failed attempt to connect
- sepgsql requires specific SELinux context to allow a connection
- connection_limits, written by Tomas Vondra, and available on GitHub, gives more control on the limit of connections than the max_connections GUC (per user, per database, and per IP)



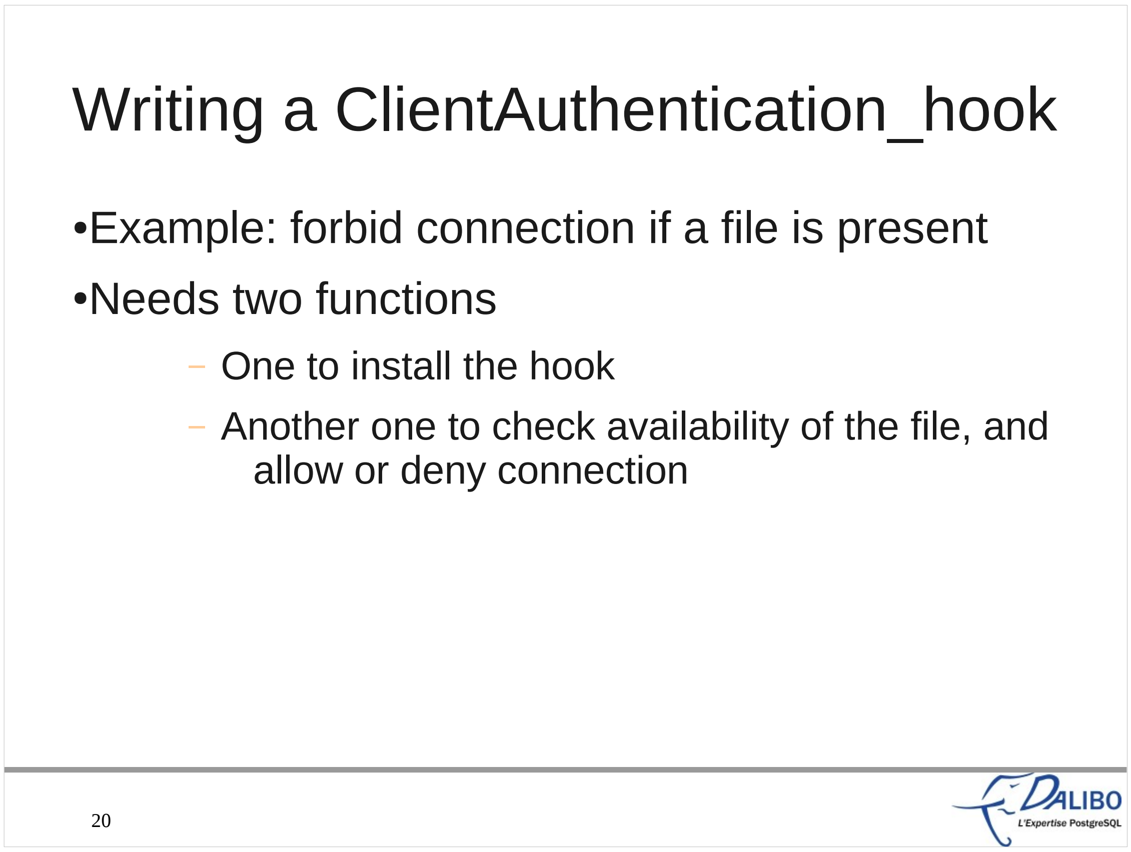
ClientAuthentication_hook function

- Two parameters
  - f (Port *port, int status)
- Port is a complete structure described in include/libpq/libpq-be.h
  - remote_host, remote_hostname, remote_port, database_name, user_name, guc_options, etc.
- Status is a status code
  - STATUS_ERROR, STATUS_OK

The ClientAuthentication_hook function requires two parameters: a Port structure, and a status code.

The first one gives lots of information on the connection to the hook function: user name, database name, GUC options, etc.

The second one is a status code, mostly a boolean value (OK or error).

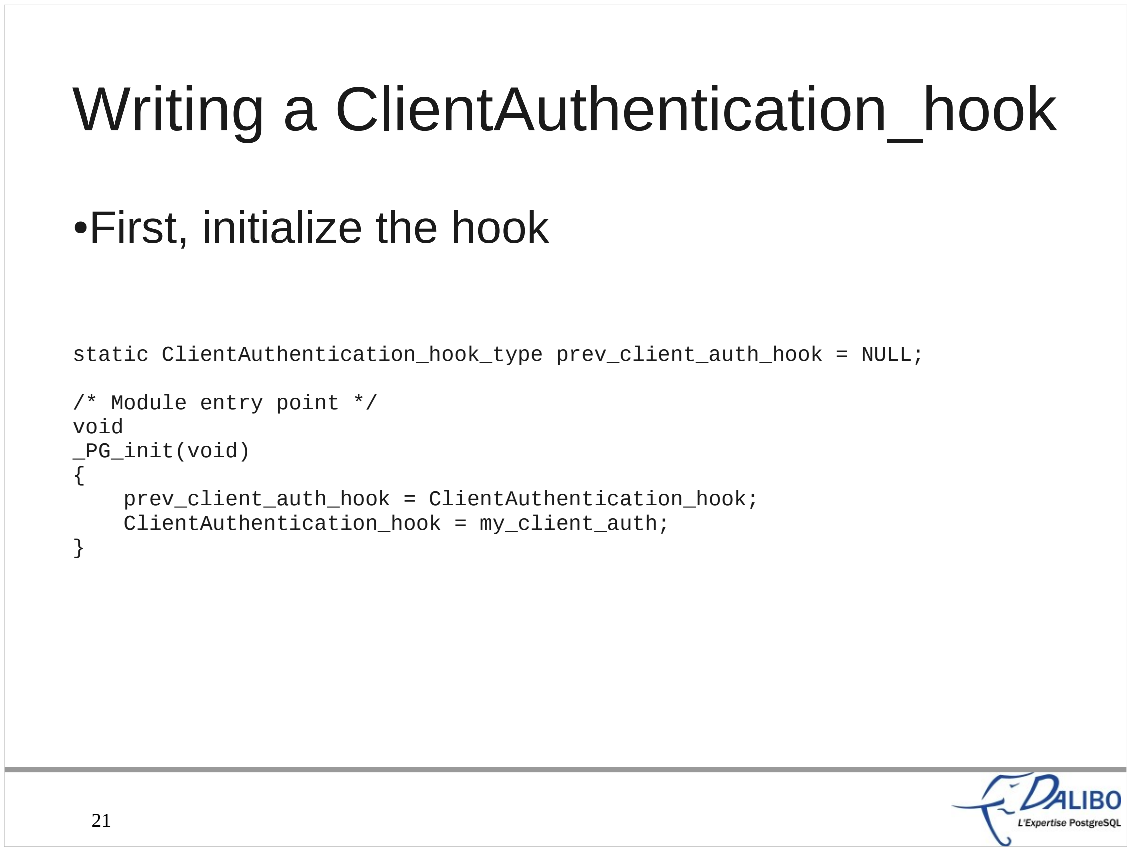


Writing a ClientAuthentication_hook

- Example: forbid connection if a file is present
- Needs two functions
  - One to install the hook
  - Another one to check availability of the file, and allow or deny connection

Here is an example of a new extension using the ClientAuthentication_hook.
Our example will deny connections if a specific file is present.
We need two functions:

- The first one will install the hook (IOW, set the ClientAuthentication_hook global function pointer) ,
- The second one will check the availability of the file, and choose to allow or deny connections.



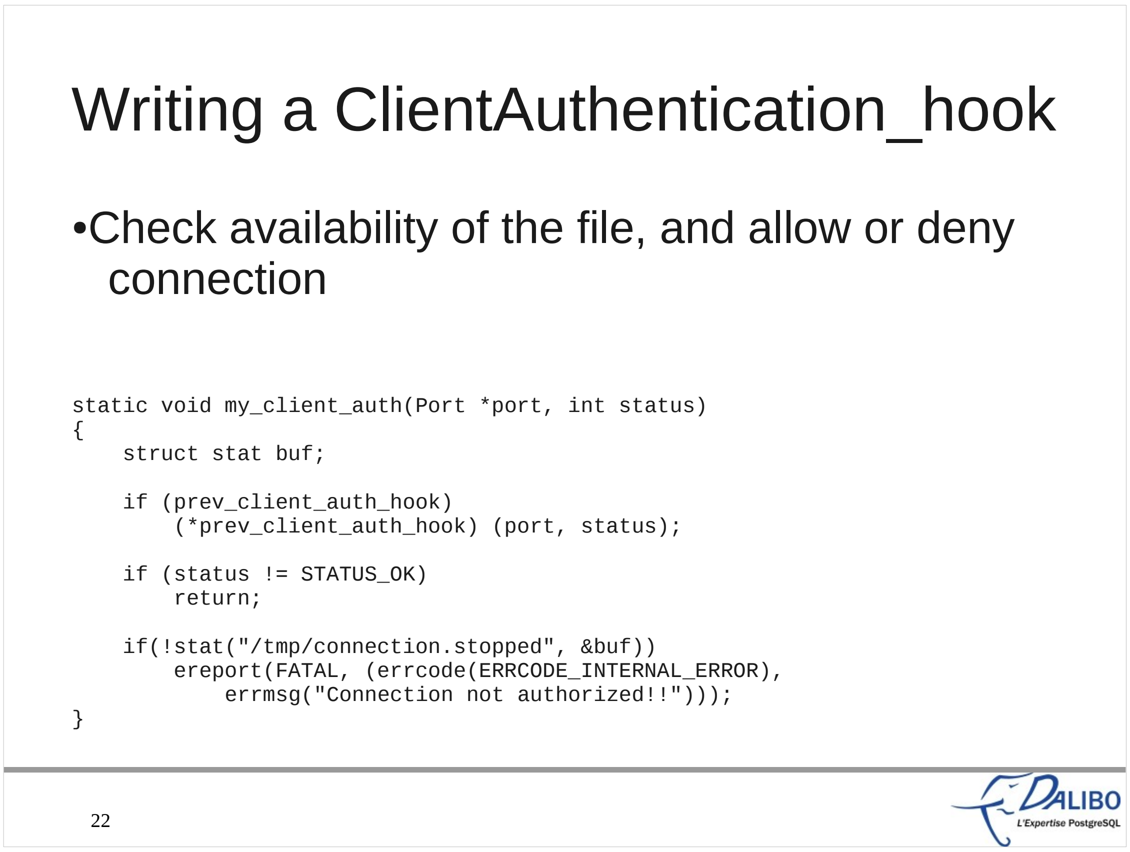
Writing a ClientAuthentication_hook

•First, initialize the hook

```
static ClientAuthentication_hook_type prev_client_auth_hook = NULL;

/* Module entry point */
void
_PG_init(void)
{
    prev_client_auth_hook = ClientAuthentication_hook;
    ClientAuthentication_hook = my_client_auth;
}
```

The initialization of the hook must happen in the _PG_init function. This function is called when PostgreSQL loads the shared library.

The first line saves the previous ClientAuthentication_hook. The second line changes the hook with our own function.



Writing a ClientAuthentication_hook

- Check availability of the file, and allow or deny connection

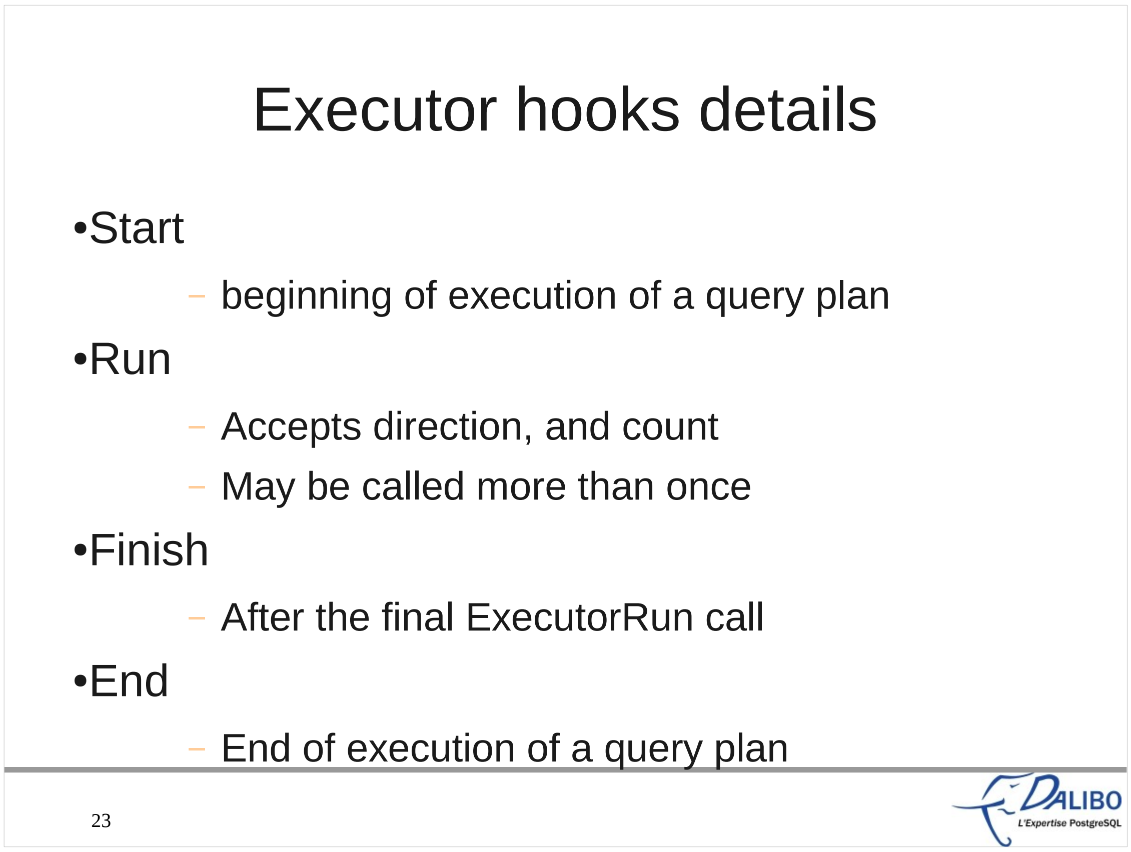
```
static void my_client_auth(Port *port, int status)
{
    struct stat buf;

    if (prev_client_auth_hook)
        (*prev_client_auth_hook) (port, status);

    if (status != STATUS_OK)
        return;

    if(!stat("/tmp/connection.stopped", &buf))
        ereport(FATAL, (errcode(ERRCODE_INTERNAL_ERROR),
            errmsg("Connection not authorized!!")));
}
```

Here is the function that does the actual work.

If a previous hook was set, we first call it.

If the result of its execution is to deny the connection, there is no need to execute our own code. We simply return with a “not OK” status.

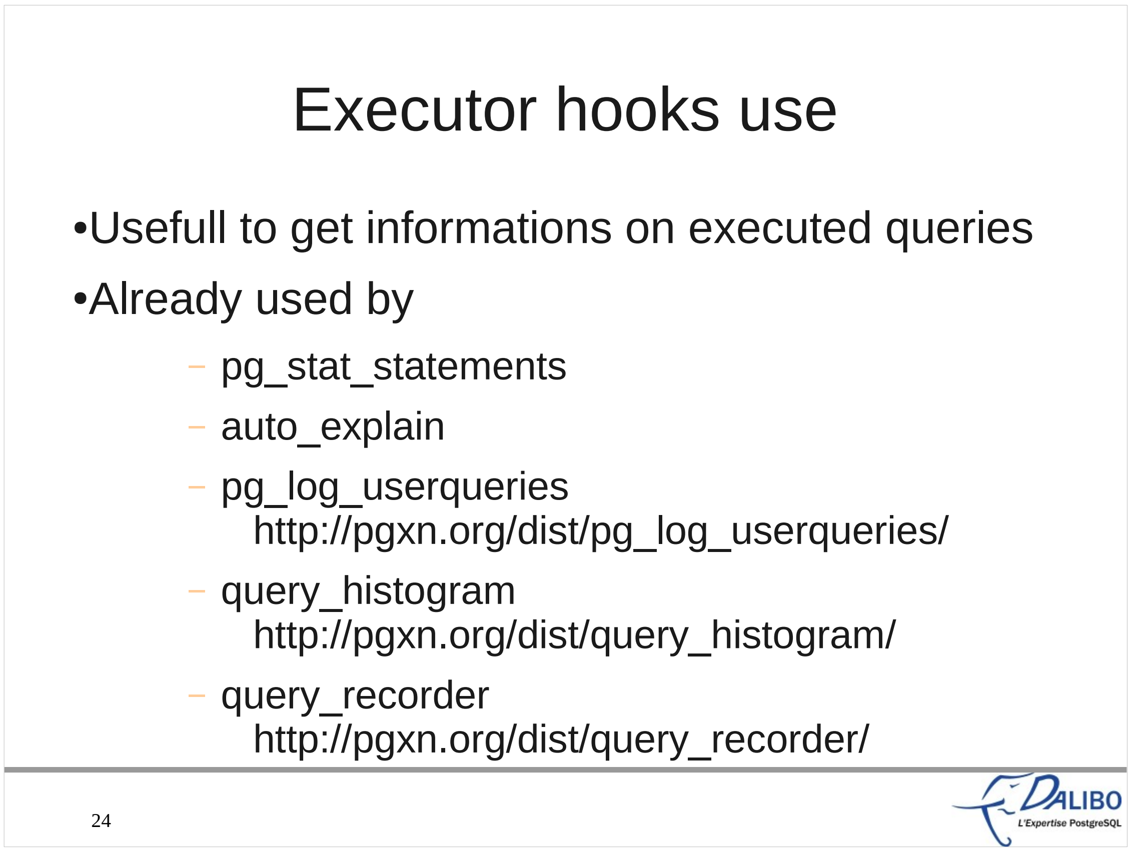
If the previous hook allows the connection, we then need to check for the presence of the file (here, /tmp/connection.stopped). If it cannot find the file, we use ereport() to deny properly the connection.



Executor hooks details

- Start
  - beginning of execution of a query plan
- Run
  - Accepts direction, and count
  - May be called more than once
- Finish
  - After the final ExecutorRun call
- End
  - End of execution of a query plan

There are four hooks for the Executor. The ExecutorStart_hook is executed at the beginning of the execution of a query plan. The ExecutorRun_hook may be called more than once, to process all tuples for a plan. Sometimes, it may stop before processing all tuples. It accepts direction (forward, or backward), and tuples count. The ExecutorFinish_hook is executed after the final ExecutorRun call, and before the ExecutorEnd. This last hook function is called at the end of the execution of the query plan.



Executor hooks use

- Usefull to get informations on executed queries
- Already used by
  - pg_stat_statements
  - auto_explain
  - pg_log_userqueries
    http://pgxn.org/dist/pg_log_userqueries/
  - query_histogram
    http://pgxn.org/dist/query_histogram/
  - query_recorder
    http://pgxn.org/dist/query_recorder/

The executor hooks are the most used hooks in PostgreSQL. There are two contrib modules, and three extensions available that use these hooks.

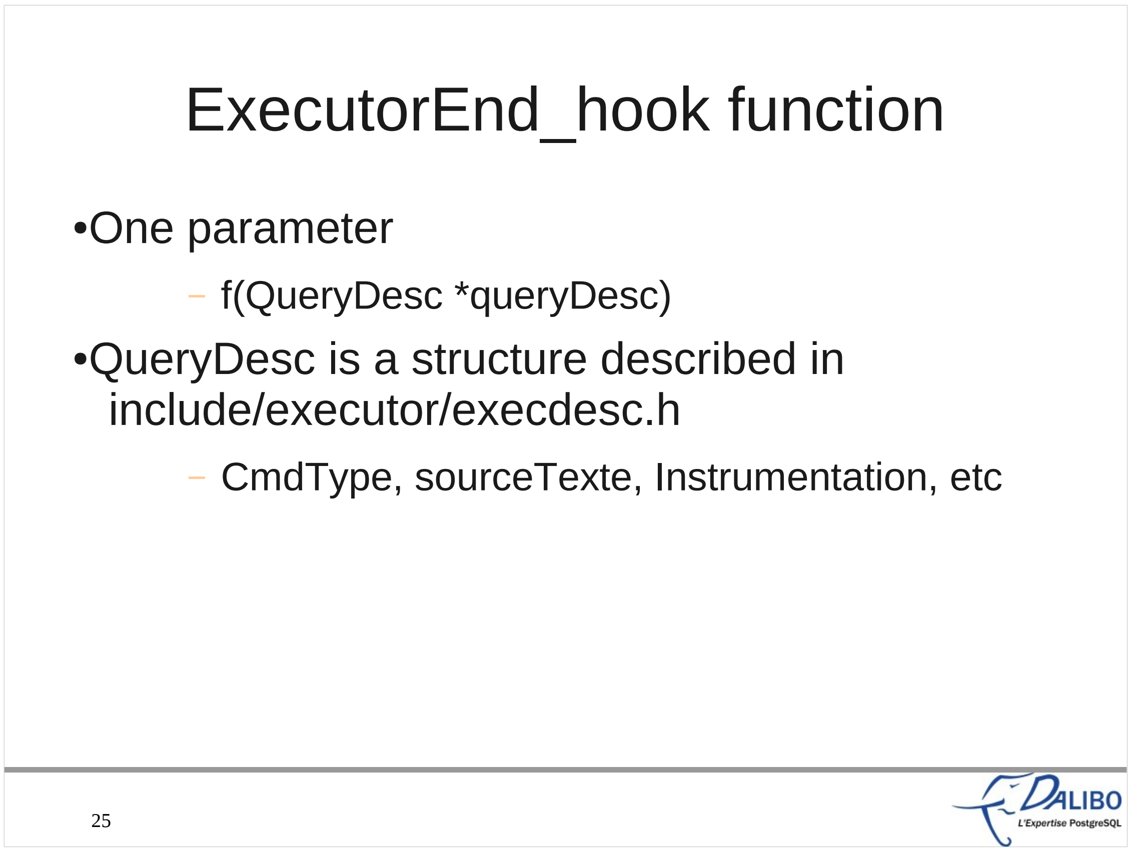
pg_stat_statement is a contrib module that grabs some statistics on the queries executed.

auto_explain uses the hooks to automatically log the explain plan of each query.

pg_log_userqueries is an extension that logs all queries according to some new GUC (per database, user, user attribute).

query_histogram is another extension that builds a duration histogram of the executed queries.

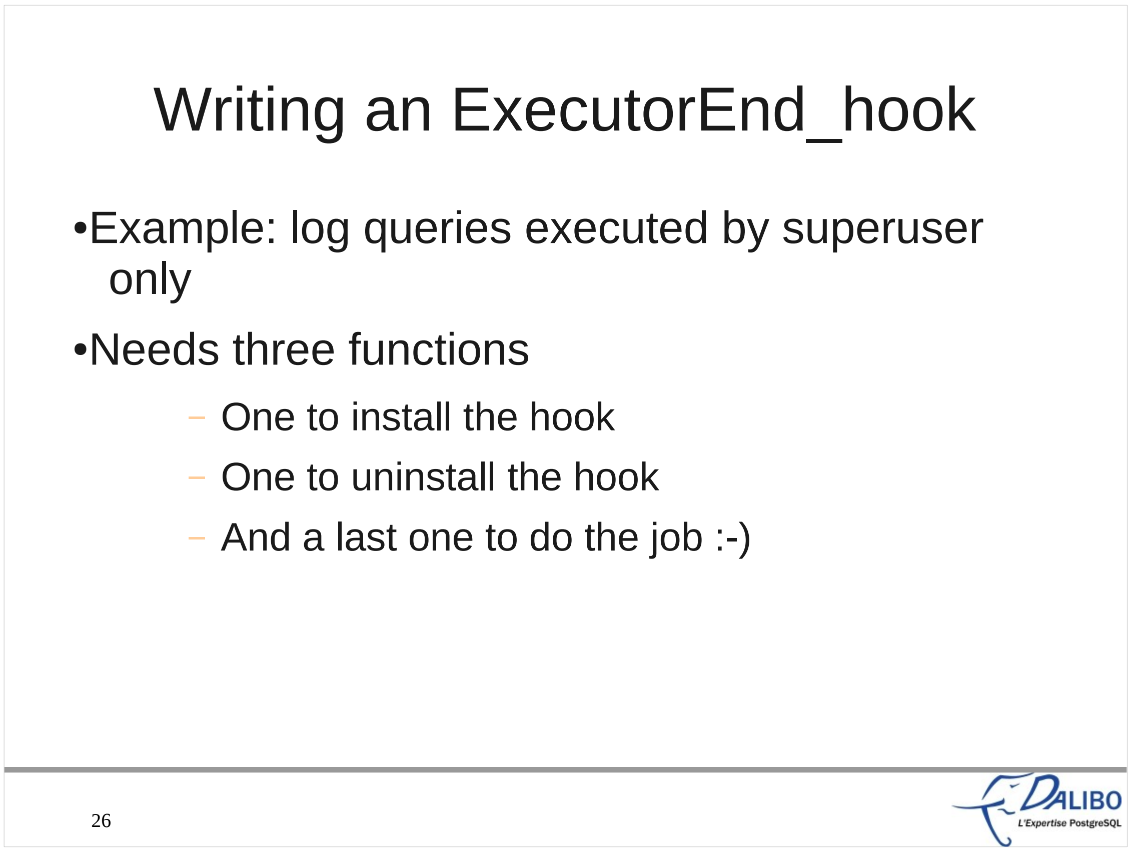
query_recorder is yet another extension to log queries in one or more files, according to the configuration (GUC parameters).



ExecutorEnd_hook function

- One parameter
  - f(QueryDesc *queryDesc)
- QueryDesc is a structure described in include/executor/execdesc.h
  - CmdType, sourceTexte, Instrumentation, etc

This hook takes only one parameter, of type QueryDesc. This parameter gives a lot of information on the executed query (like the texte of the query, the state of the query and the transaction, lots of instrumentation details).

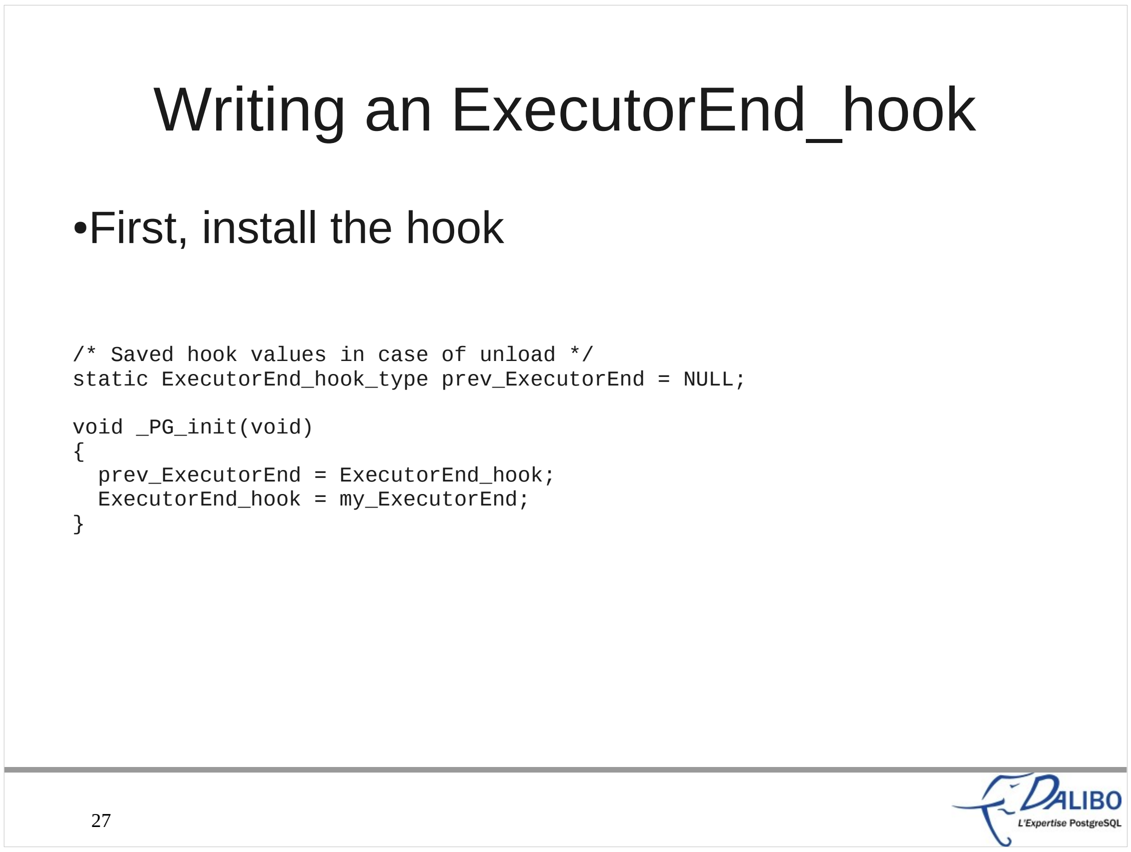


Writing an ExecutorEnd_hook

- Example: log queries executed by superuser only
- Needs three functions
  - One to install the hook
  - One to uninstall the hook
  - And a last one to do the job :-)

For this example, we'll log queries executed only by superusers.

To do that, we need three functions. One to install the hook, one to uninstall it (which is optional for us actually), and a last one to write the log if the user has the SUPERUSER attribute.

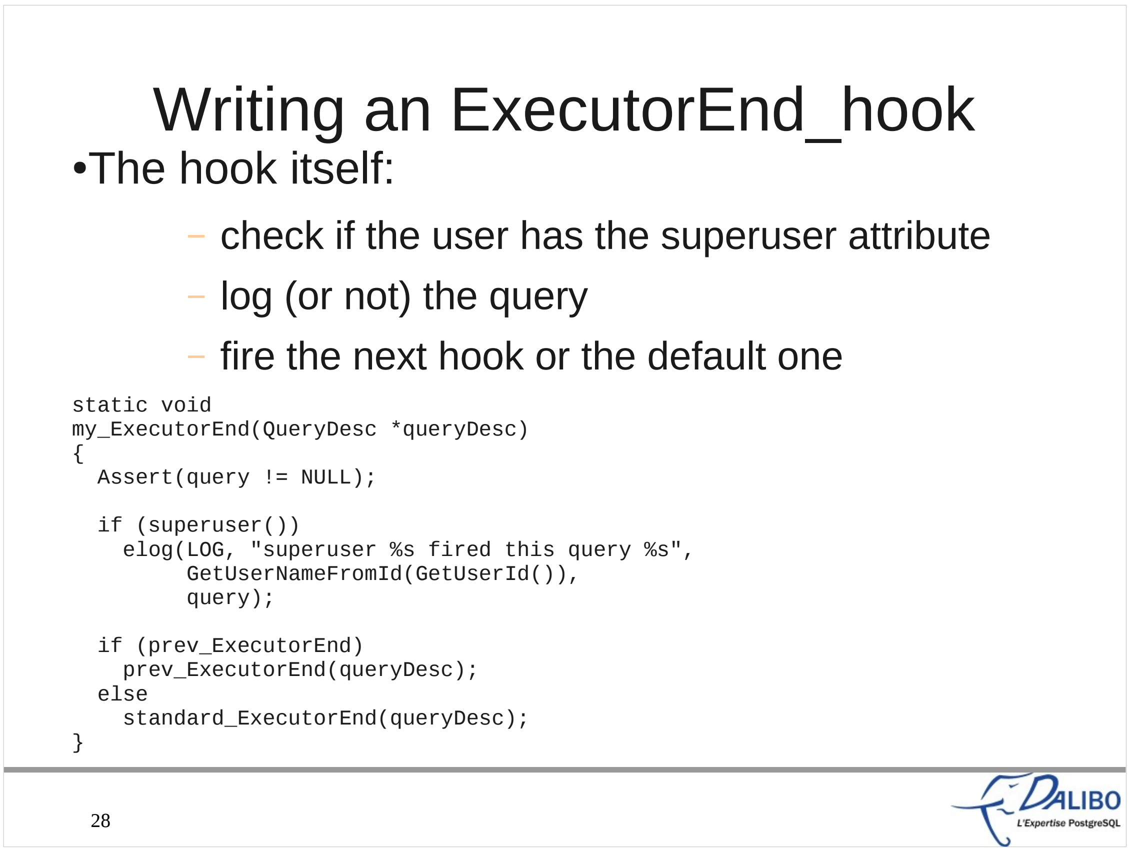


Writing an ExecutorEnd_hook

- First, install the hook

```
/* Saved hook values in case of unload */
static ExecutorEnd_hook_type prev_ExecutorEnd = NULL;

void _PG_init(void)
{
  prev_ExecutorEnd = ExecutorEnd_hook;
  ExecutorEnd_hook = my_ExecutorEnd;
}
```

This function saves the previous hook on ExecutorEnd_hook, and installs our own function as the new hook.


Writing an ExecutorEnd_hook

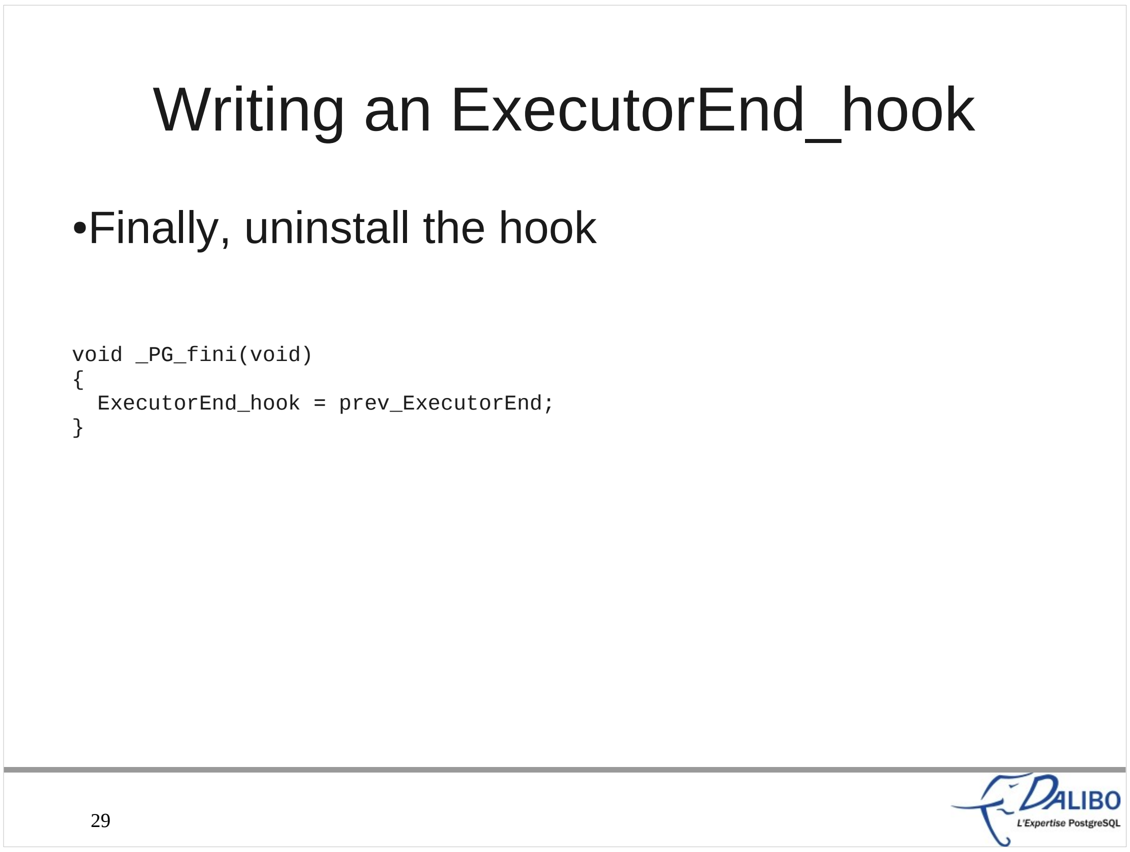
- The hook itself:
  - check if the user has the superuser attribute
  - log (or not) the query
  - fire the next hook or the default one

```
static void
my_ExecutorEnd(QueryDesc *queryDesc)
{
  Assert(query != NULL);

  if (superuser())
    elog(LOG, "superuser %s fired this query %s",
         GetUserNameFromId(GetUserId()),
         query);

  if (prev_ExecutorEnd)
    prev_ExecutorEnd(queryDesc);
  else
    standard_ExecutorEnd(queryDesc);
}
```

This function first checks if the user is a superuser. If he is, it calls elog() to log the query and the username.

Then, it executes the previous ExecutorEnd_hook if there was one.

superuser(), GetUserNameFromId, and GetUserId are functions provided by PostgreSQL.

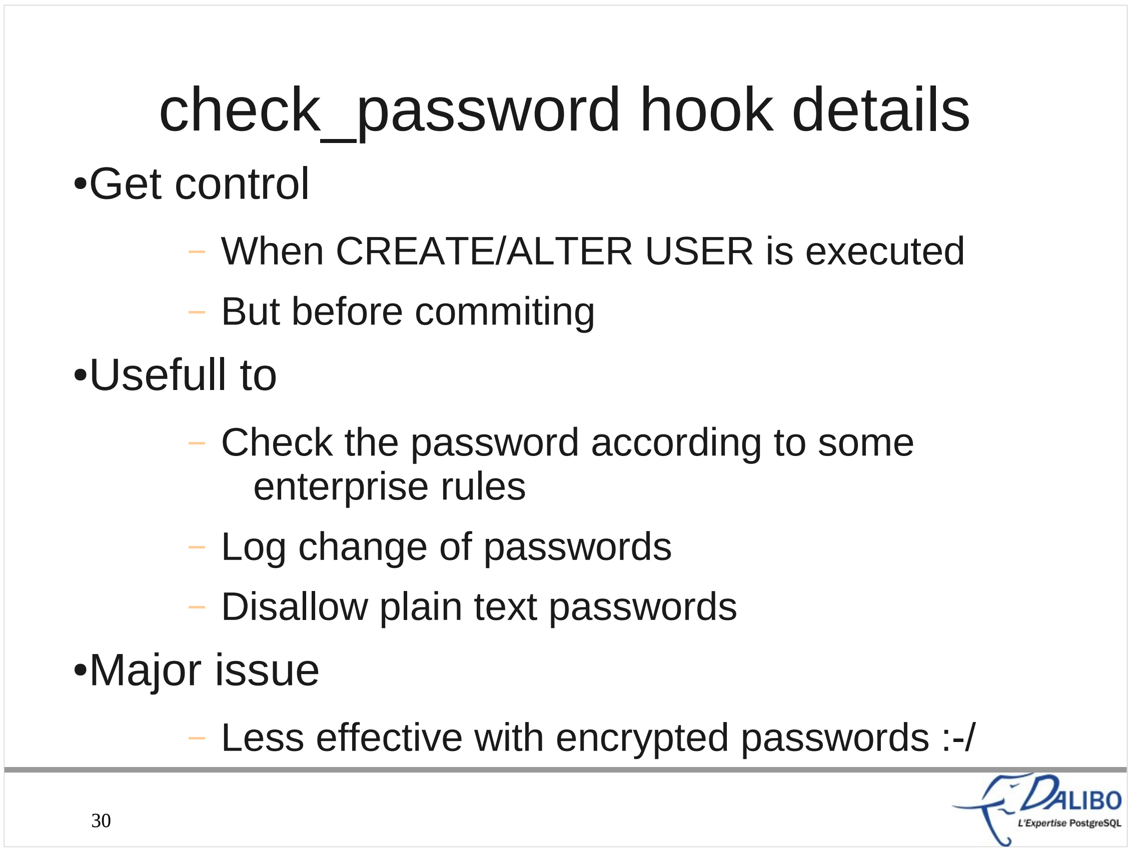


Writing an ExecutorEnd_hook

- Finally, uninstall the hook

```
void _PG_fini(void)
{
  ExecutorEnd_hook = prev_ExecutorEnd;
}
```

And this last function sets the hook with the previous ExecutorEnd_hook.



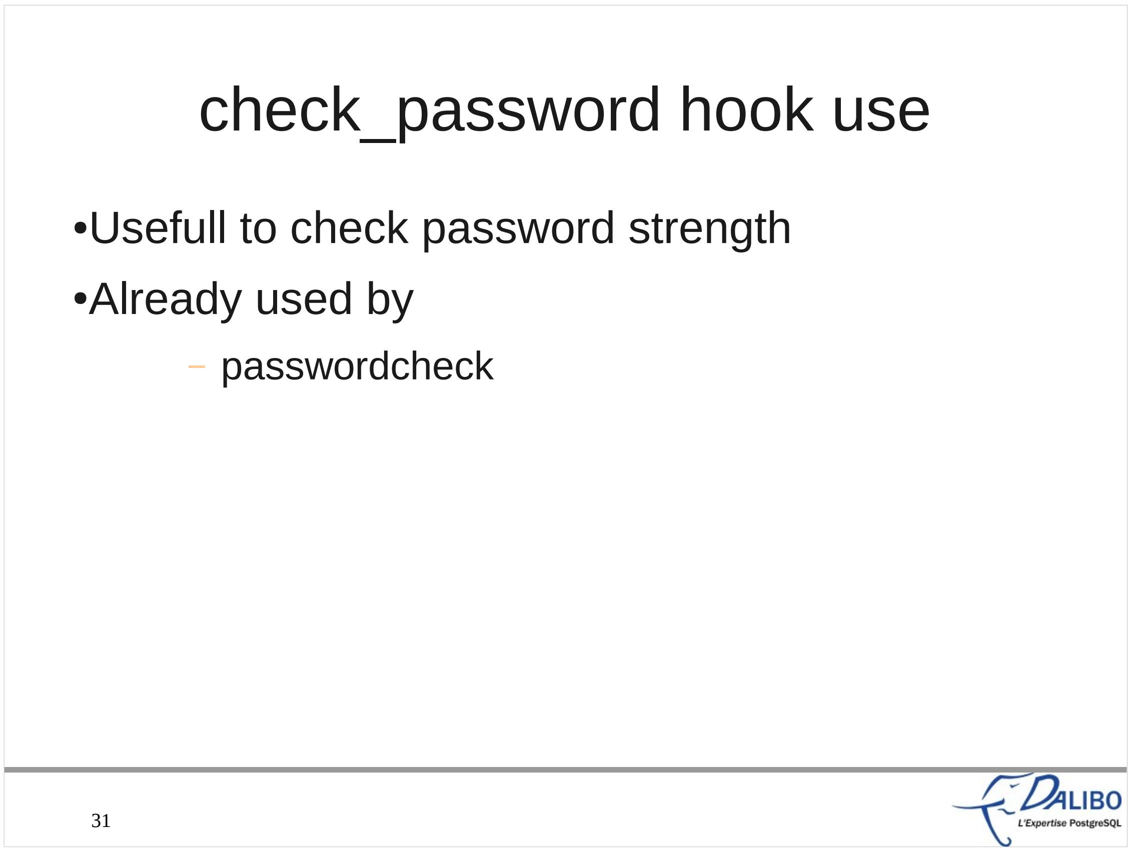
check_password hook details

- Get control
  - When CREATE/ALTER USER is executed
  - But before commiting
- Usefull to
  - Check the password according to some enterprise rules
  - Log change of passwords
  - Disallow plain text passwords
- Major issue
  - Less effective with encrypted passwords :-/

The check_password hook enables an extension to get control when a user executes a CREATE USER or ALTER USER query. It gets control before the statement is commited.

It's pretty usefull to check the password according to some enterprise rules. It can be used to log changes of passwords, and to deny using plain text passwords in CREATE/ALTER USER statements.

It also has a major drawback: it's quite less effective with encrypted passwords. It's much more difficult and time consuming to check the password against a plain text dictionary because you need to compute the MD5 checksum for each word, and compare the result to the encrypted password.



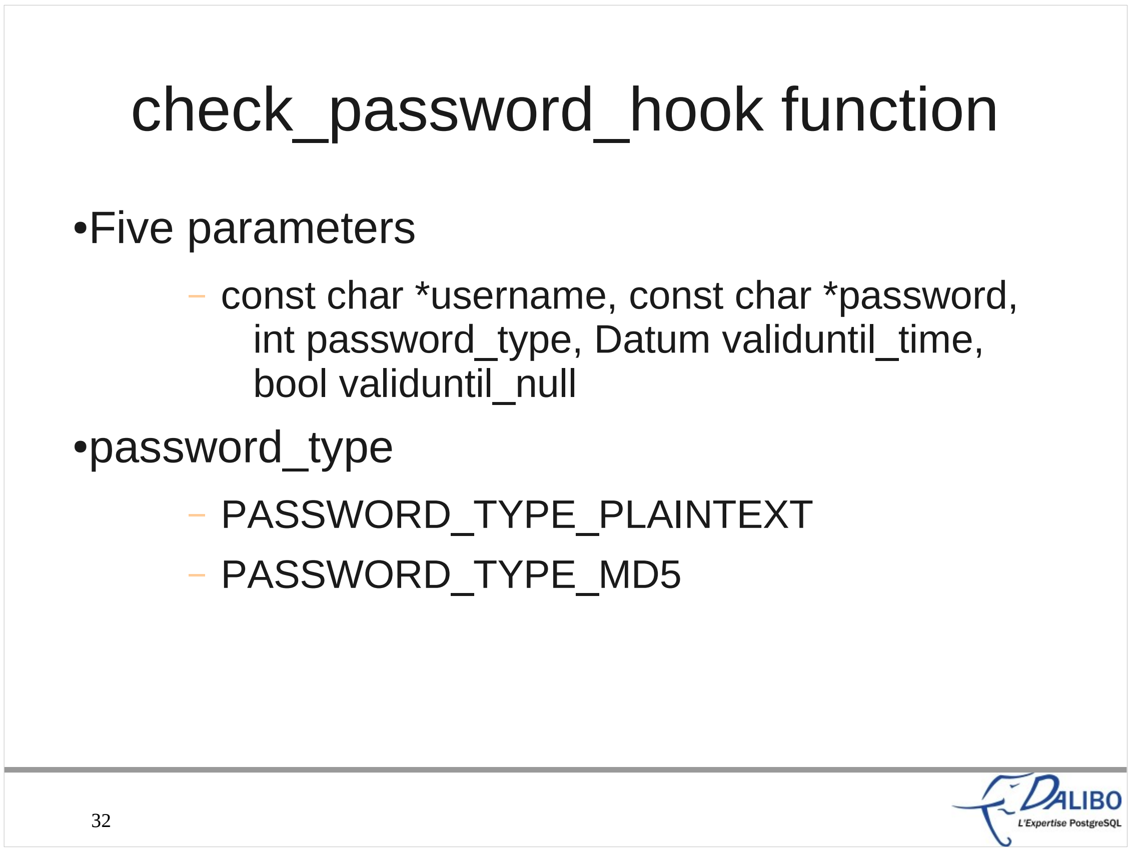
check_password hook use

- Usefull to check password strength
- Already used by
  - passwordcheck

31

The main use of this hook is to check password strength.

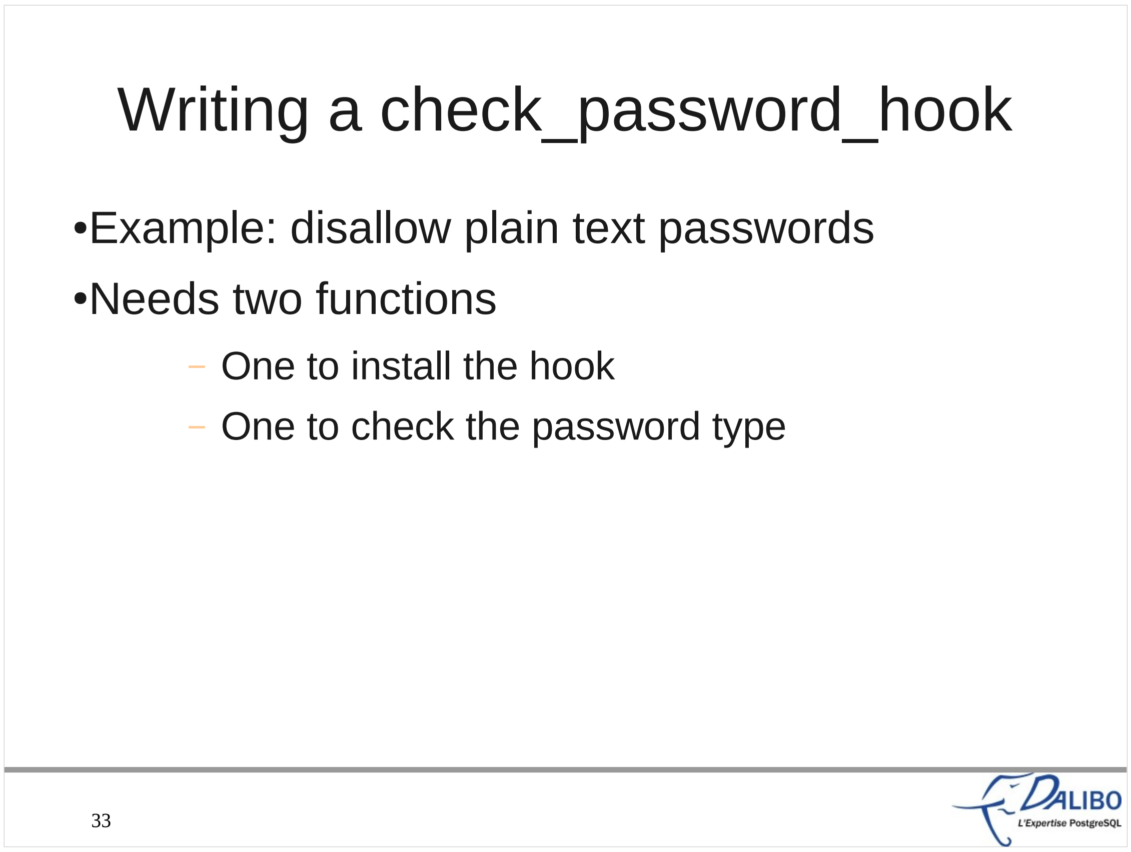
Hence, the only extension known now is passwordcheck, which is a contrib module available in the PostgreSQL distribution. It makes a few checks to be sure the password is not too weak. If you want to use it, make sure you read the source to make the changes you want, so that it really stick to your entreprise rules. Using Cracklib is quite easy to, just a few lines to uncomment.



check_password_hook function

- Five parameters
  - const char *username, const char *password, int password_type, Datum validuntil_time, bool validuntil_null
- password_type
  - PASSWORD_TYPE_PLAINTEXT
  - PASSWORD_TYPE_MD5

This hook function takes much more parameters than the previous one.
Username and password are self explanatory.
password_type allows the hook function to know if it is an encrypted password or a plain text one. An encrypted password is always encrypted with MD5.
The validuntil_* parameters give informations on the validity timestamp limit on the password.

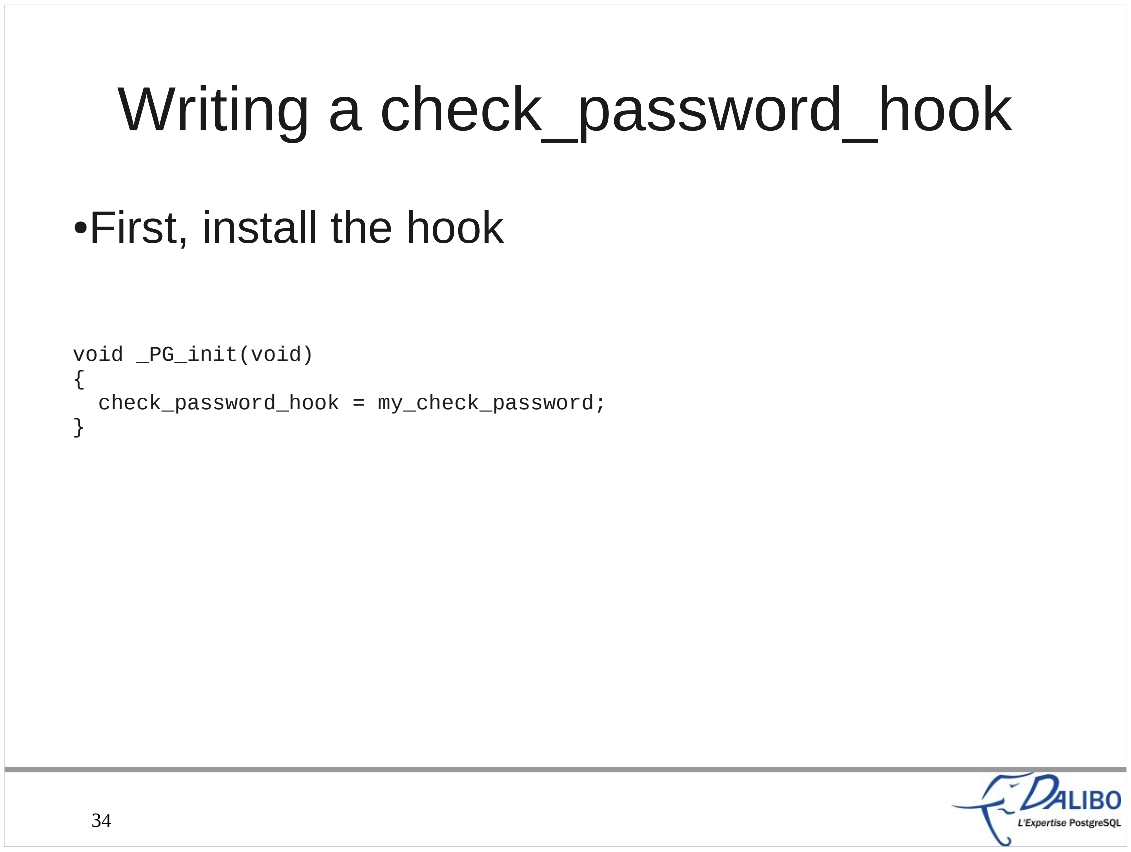


Writing a check_password_hook

- Example: disallow plain text passwords
- Needs two functions
  - One to install the hook
  - One to check the password type

33

For this third example, we'll disallow the use of plain text passwords. We need two functions: one to install the hook, one to check the password type.

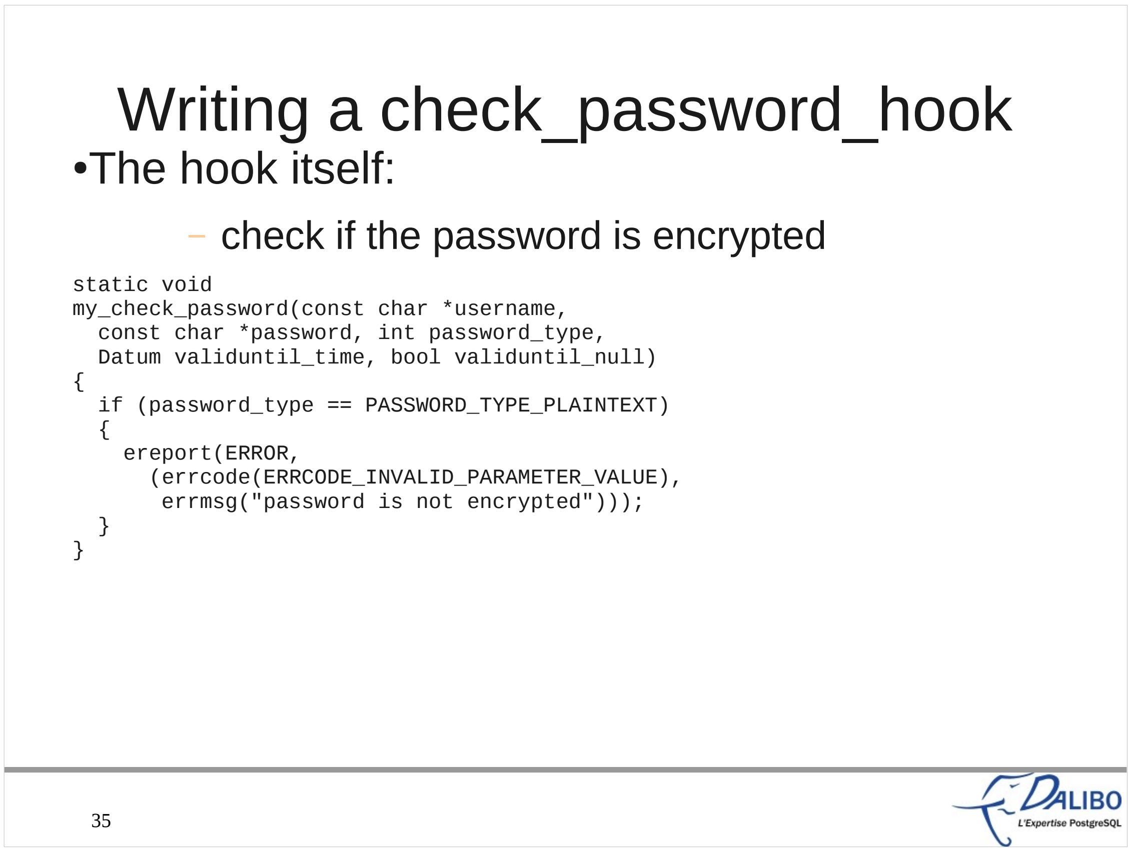


Writing a check_password_hook

- First, install the hook

```
void _PG_init(void)
{
  check_password_hook = my_check_password;
}
```

Installing the hook is really easy. We just need to initialize the global function pointer to our function. We could save the previous value, but don't show this here as we already showed that before.



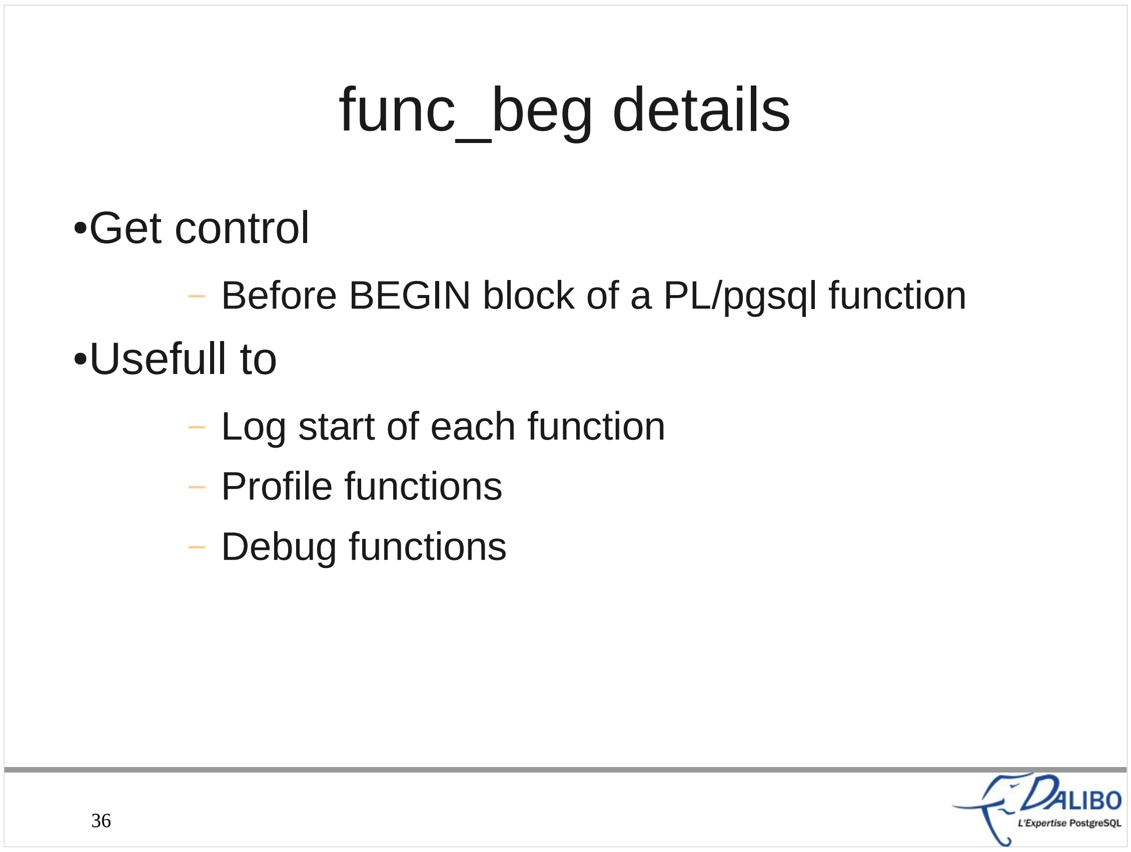
Writing a check_password_hook

- The hook itself:
  - check if the password is encrypted

```
static void
my_check_password(const char *username,
  const char *password, int password_type,
  Datum validuntil_time, bool validuntil_null)
{
  if (password_type == PASSWORD_TYPE_PLAINTEXT)
  {
    ereport(ERROR,
      (errcode(ERRCODE_INVALID_PARAMETER_VALUE),
       errmsg("password is not encrypted")));
  }
}
```

The hook itself is here. It only checks the password type, and calls the ereport() function if it is a plaintext password.

Calling ereport with an ERROR log level will cancel the query.

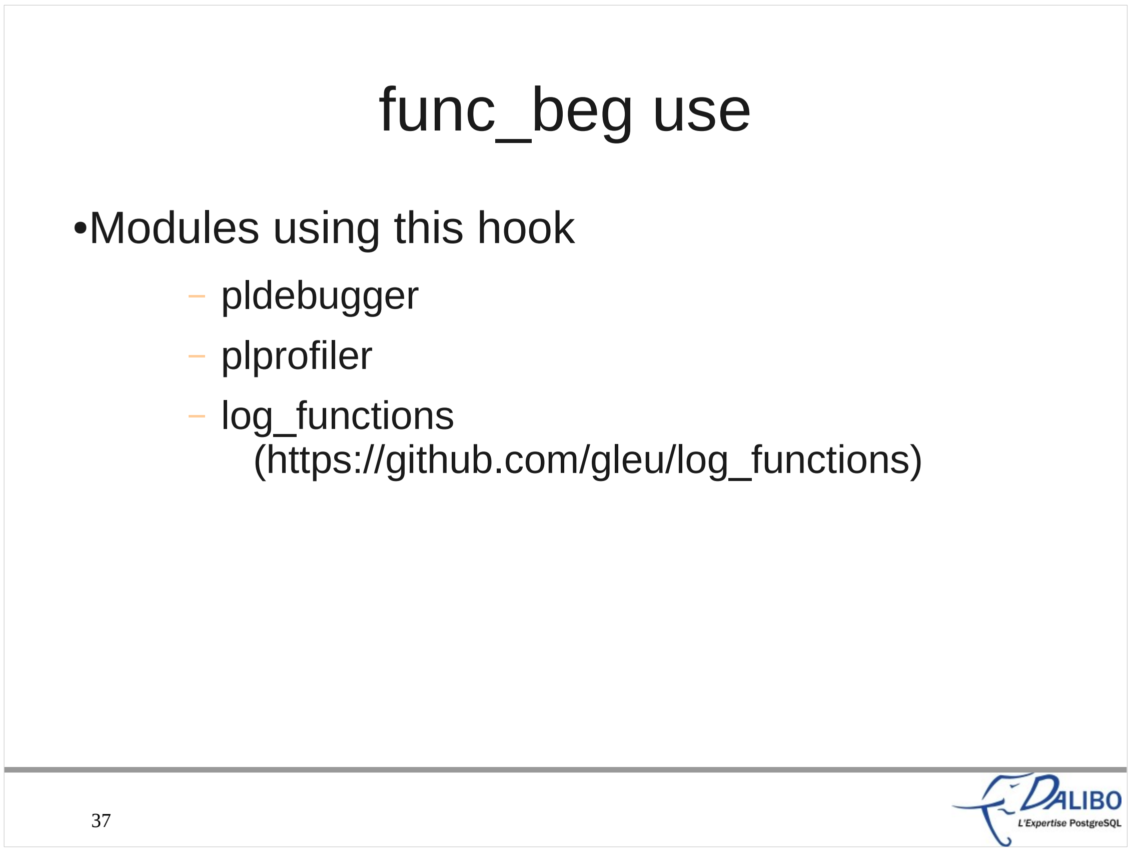


func_beg details

- Get control
  - Before BEGIN block of a PL/pgsql function
- Usefull to
  - Log start of each function
  - Profile functions
  - Debug functions

The func_beg hook helps a plugin to get control at the BEGIN statement of a PL/pgsql stored function.

Therefore, the plugin can log the use of each function, and can help profiling and debugging the function.



func_beg use

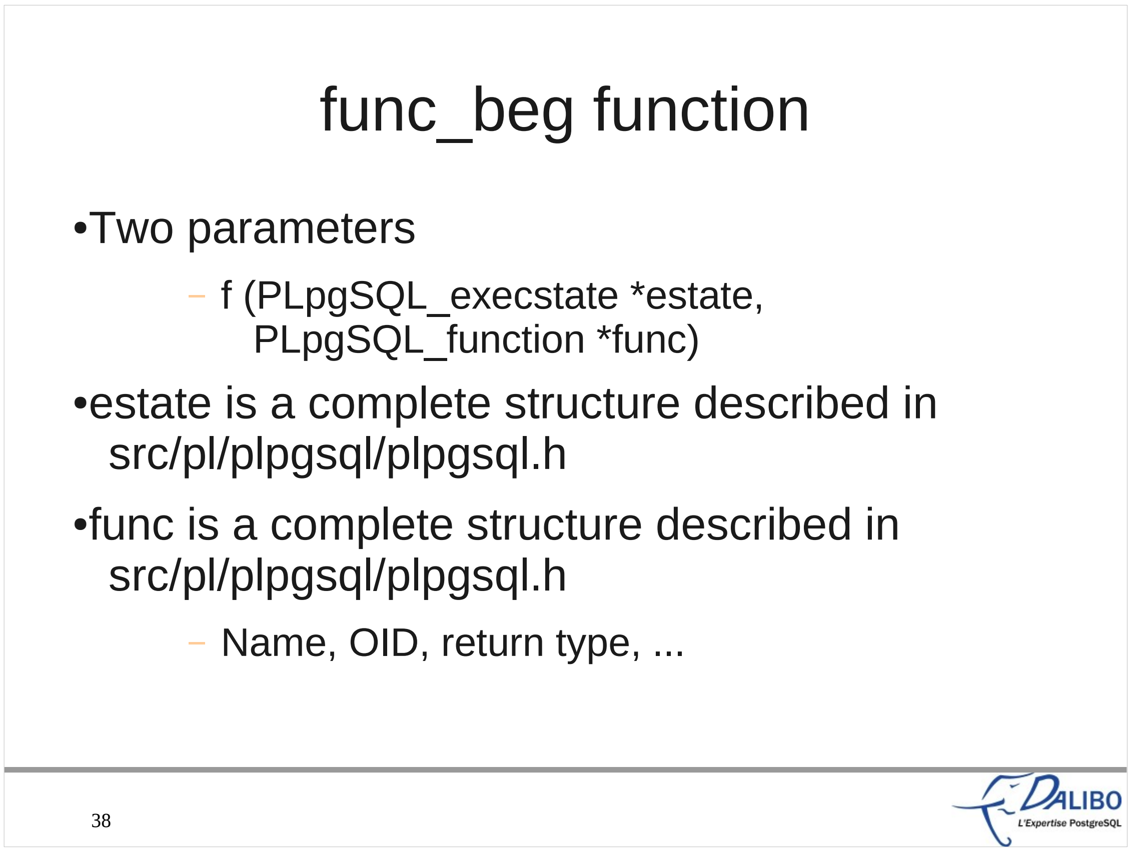
•Modules using this hook

- pldebugger
- plprofiler
- log_functions (https://github.com/gleu/log_functions)

37

Three extensions already use this hook:

- pldebugger provides a debugger for PL/pgsql functions
- plprofiler provides a profiler for PL/pgsql functions
- log_functions, available on GitHub, allows to log the use of each functions, its duration, and the duration of each statement.

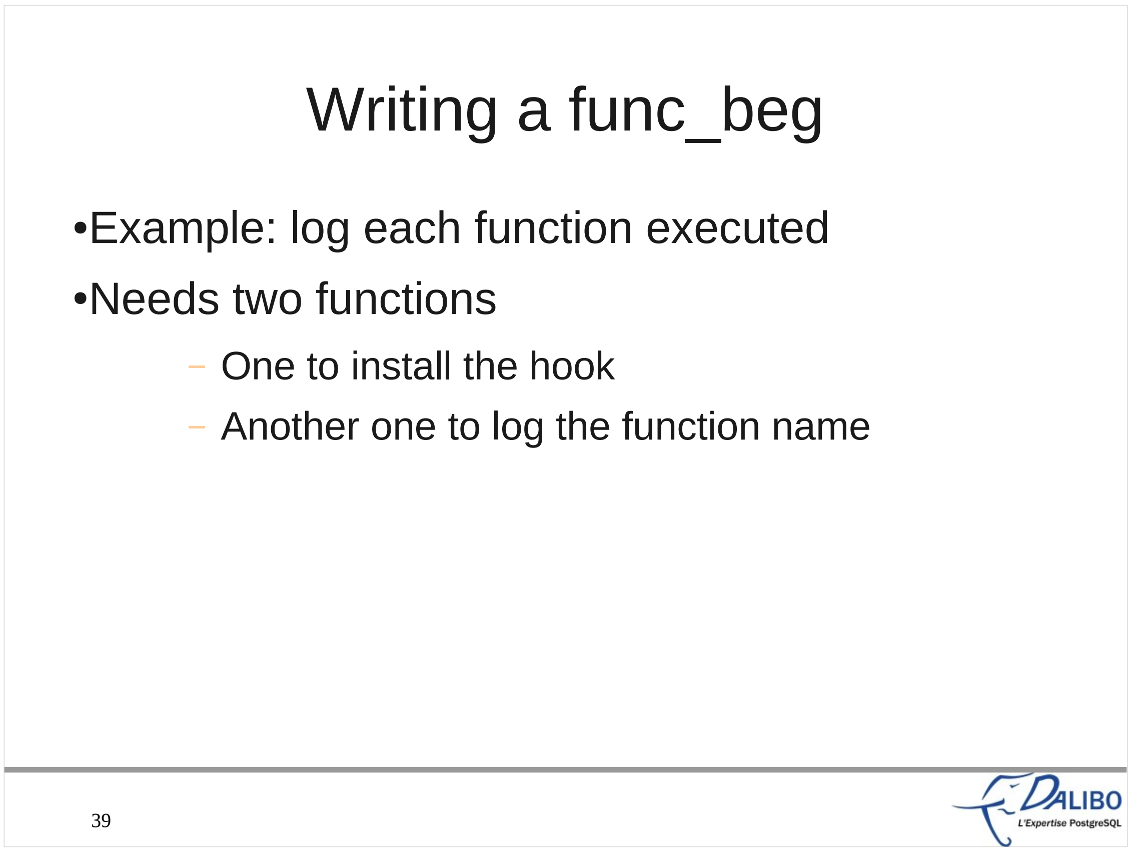


func_beg function

- Two parameters
  - f (PLpgSQL_execstate *estate, PLpgSQL_function *func)
- estate is a complete structure described in src/pl/plpgsql/plpgsql.h
- func is a complete structure described in src/pl/plpgsql/plpgsql.h
  - Name, OID, return type, ...

The func_beg hook function requires two parameters: a PLpgSQL_execstate structure, and a PLpgSQL_function structure.

The first one gives lots of informations on the execution state of the function and the second one gives informations on the function being executed (name, oid, etc.).



Writing a func_beg

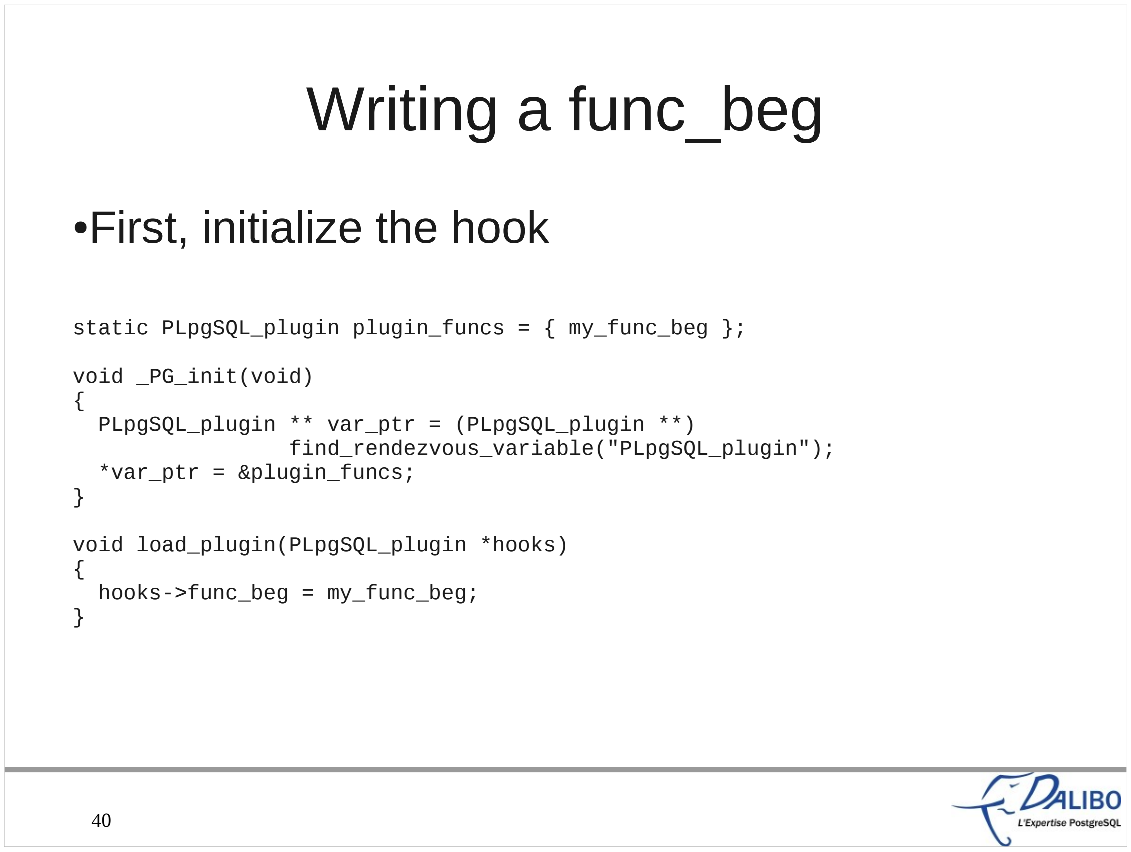
- Example: log each function executed
- Needs two functions
  - One to install the hook
  - Another one to log the function name

Here is an example of a new extension using the func_beg hook.

Our example will log each function executed.

We need two functions:

- The first one will install the hook ;
- The second one will log the function name.



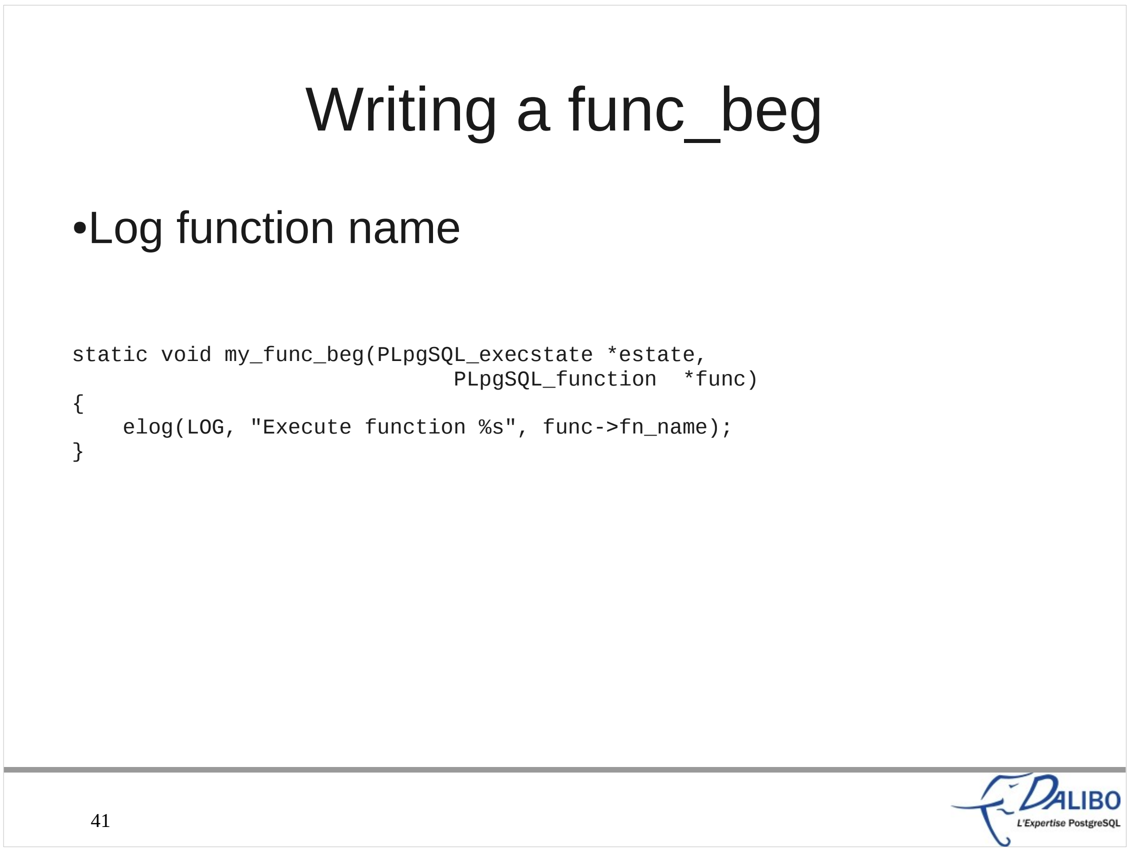
Writing a func_beg

- First, initialize the hook

```
static PLpgSQL_plugin plugin_funcs = { my_func_beg };

void _PG_init(void)
{
  PLpgSQL_plugin ** var_ptr = (PLpgSQL_plugin **)
                 find_rendezvous_variable("PLpgSQL_plugin");
  *var_ptr = &plugin_funcs;
}

void load_plugin(PLpgSQL_plugin *hooks)
{
  hooks->func_beg = my_func_beg;
}
```

The initialization of the hook must happen in the _PG_init function. This function is called when PostgreSQL loads the shared library.

It's a little bit more complicated than the rest of our examples.

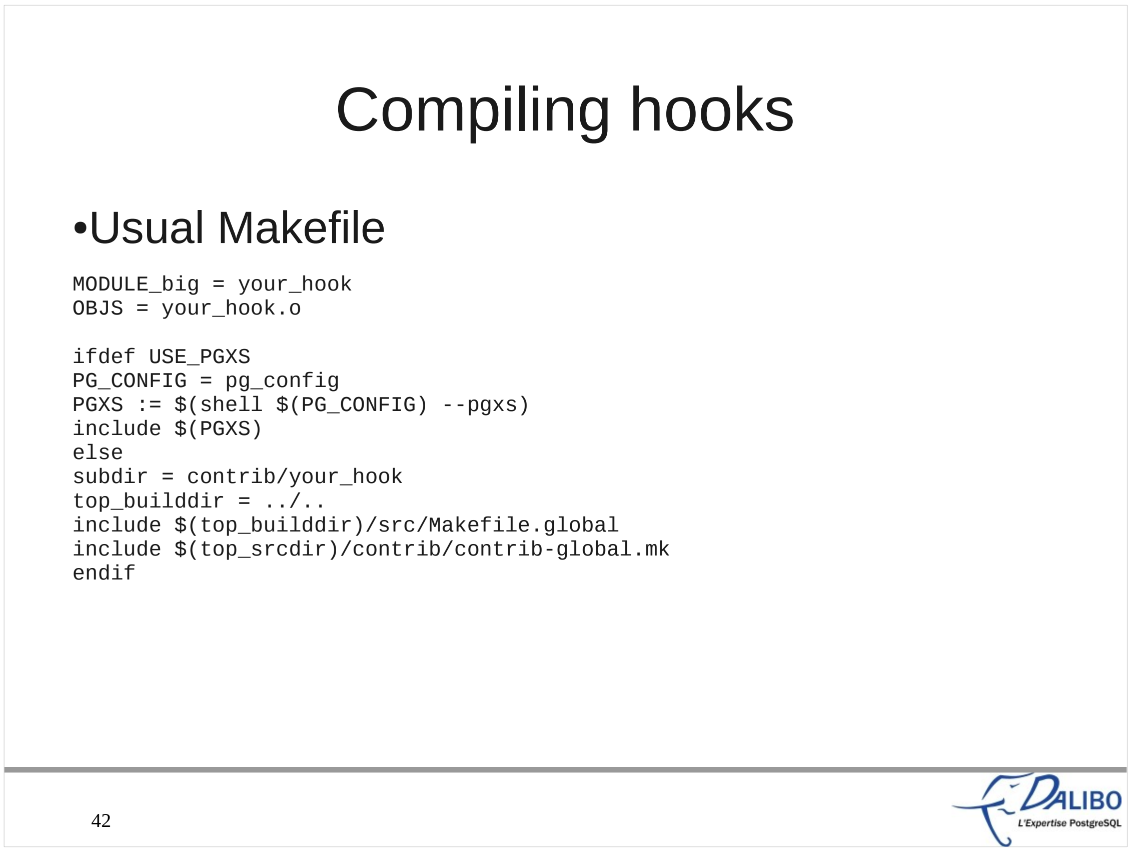


Writing a func_beg

- Log function name

```
static void my_func_beg(PLpgSQL_execstate *estate,
                              PLpgSQL_function  *func)
{
    elog(LOG, "Execute function %s", func->fn_name);
}
```

Here is the function that does the actual work.
It simply calls elog to log the execution of the function, and adds the name of the function (found in the func structure).



Compiling hooks

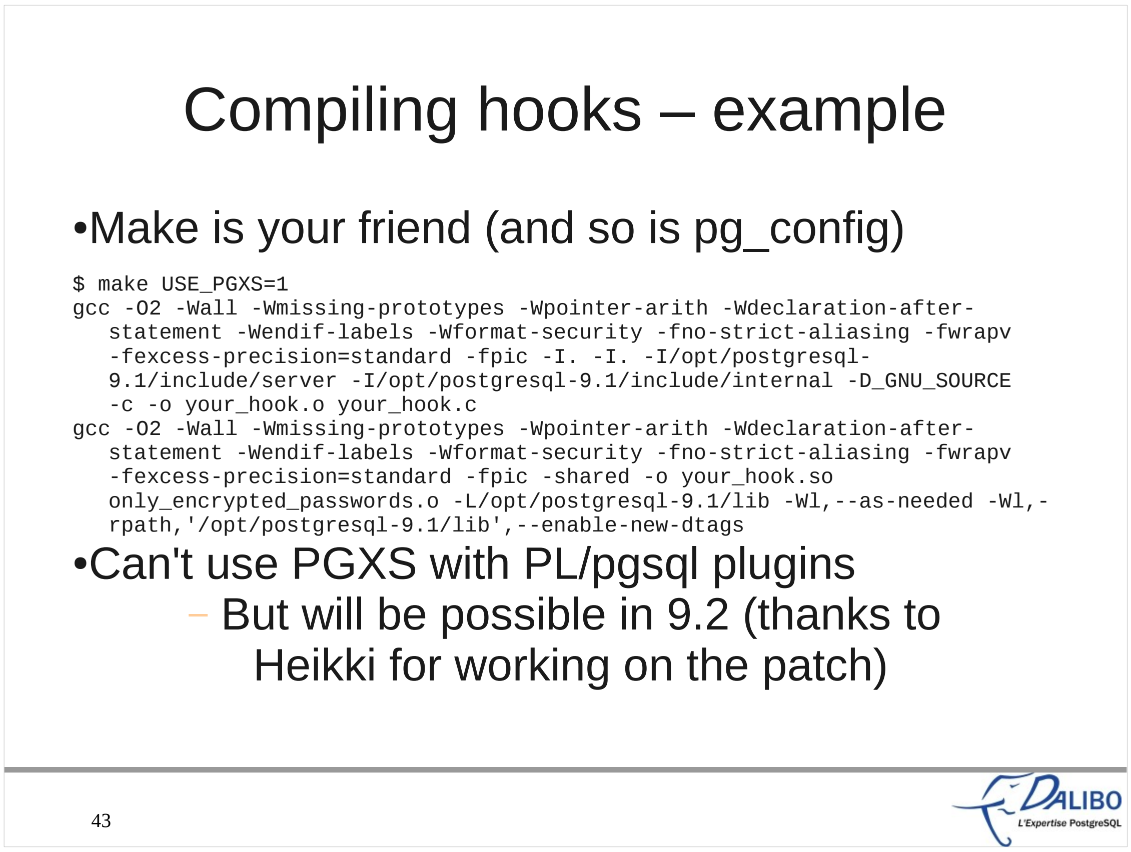
- Usual Makefile

```
MODULE_big = your_hook
OBJS = your_hook.o

ifdef USE_PGXS
PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
else
subdir = contrib/your_hook
top_builddir = ../..
include $(top_builddir)/src/Makefile.global
include $(top_srcdir)/contrib/contrib-global.mk
endif
```

Compiling hooks is really easy. You need this usual Makefile for shared library.

You can compile the code outside of the PostgreSQL source tree if you use PGXS. It relies on pg_config, which may only be available if you install the -devel package of PostgreSQL.

If you already has the source tree, you can simply put the directory of the source in the contrib directory of PostgreSQL. You don't need pg_config if you did that.



Compiling hooks – example

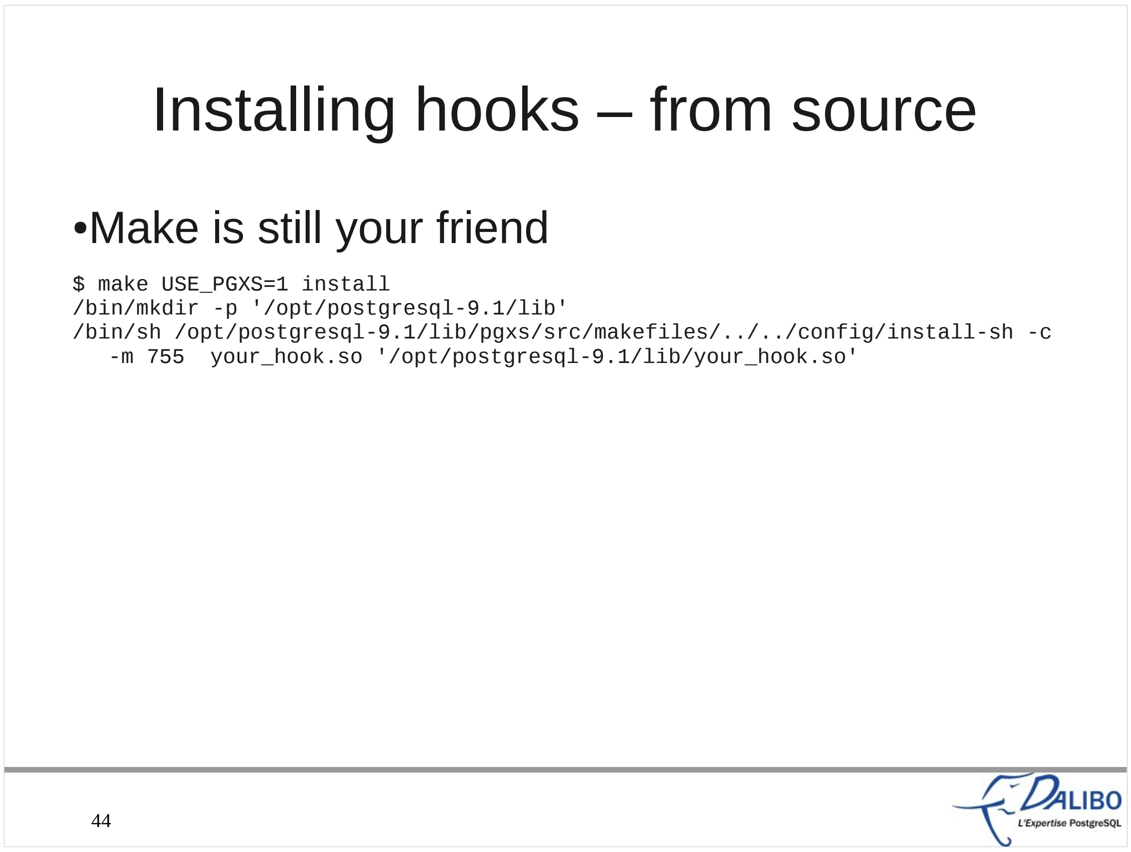
- Make is your friend (and so is pg_config)

```
$ make USE_PGXS=1
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-
   statement -Wendif-labels -Wformat-security -fno-strict-aliasing -fwrapv
   -fexcess-precision=standard -fpic -I. -I. -I/opt/postgresql-
   9.1/include/server -I/opt/postgresql-9.1/include/internal -D_GNU_SOURCE
   -c -o your_hook.o your_hook.c
gcc -O2 -Wall -Wmissing-prototypes -Wpointer-arith -Wdeclaration-after-
   statement -Wendif-labels -Wformat-security -fno-strict-aliasing -fwrapv
   -fexcess-precision=standard -fpic -shared -o your_hook.so
   only_encrypted_passwords.o -L/opt/postgresql-9.1/lib -Wl,--as-needed -Wl,-
   rpath,'/opt/postgresql-9.1/lib',--enable-new-dtags
```

- Can't use PGXS with PL/pgsql plugins
  - But will be possible in 9.2 (thanks to Heikki for working on the patch)

To compile outside of the PostgreSQL source tree, add USE_PGXS=1 to the make command. Remember you need to have the pg_config tool in your PATH.

You don't need to set this environment variable if you had put the source code inside the contrib directory of the PostgreSQL source tree.



Installing hooks – from source

- Make is still your friend

```
$ make USE_PGXS=1 install
/bin/mkdir -p '/opt/postgresql-9.1/lib'
/bin/sh /opt/postgresql-9.1/lib/pgxs/src/makefiles/../../config/install-sh -c
   -m 755  your_hook.so '/opt/postgresql-9.1/lib/your_hook.so'
```

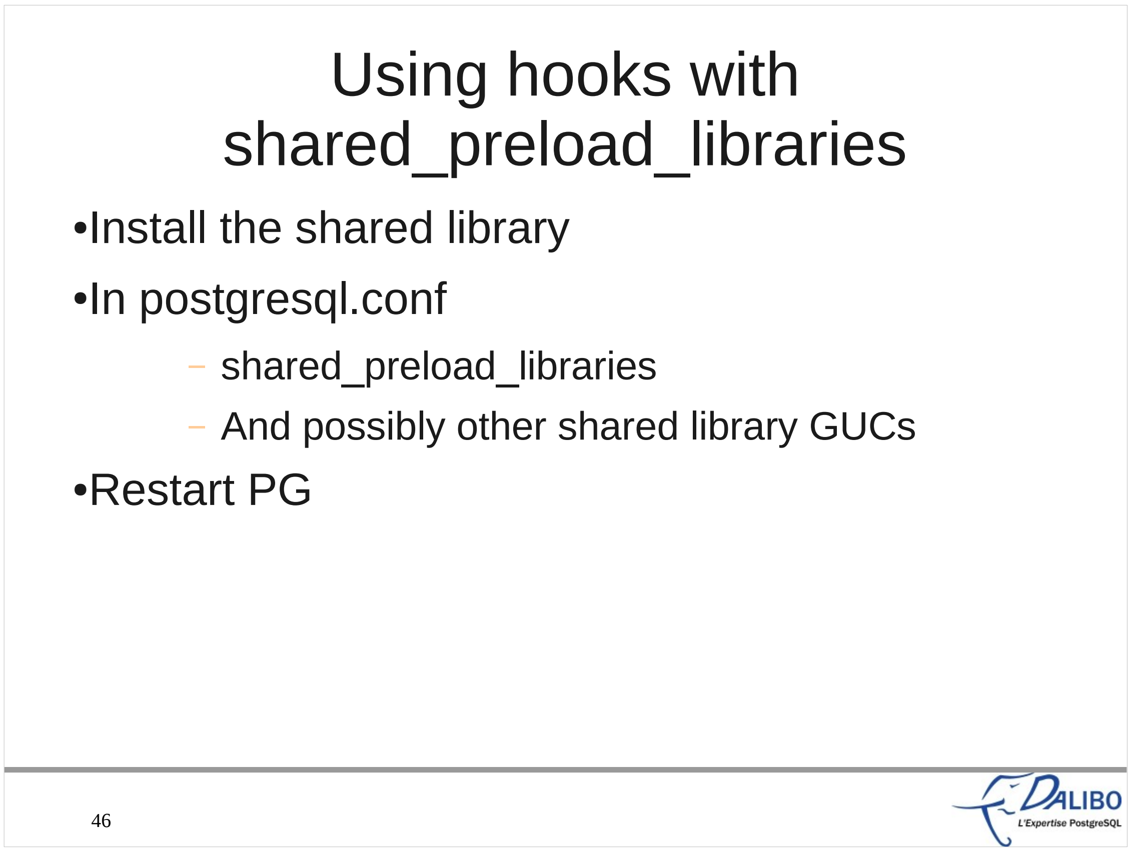
You'll still use make to install the shared library.

PGXS

- It's better to rely only on PGXS (if possible)
- Makefile looks like this:

```
MODULE_big = your_hook
OBJS = your_hook.o

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

- So much simpler...



Using hooks with shared_preload_libraries

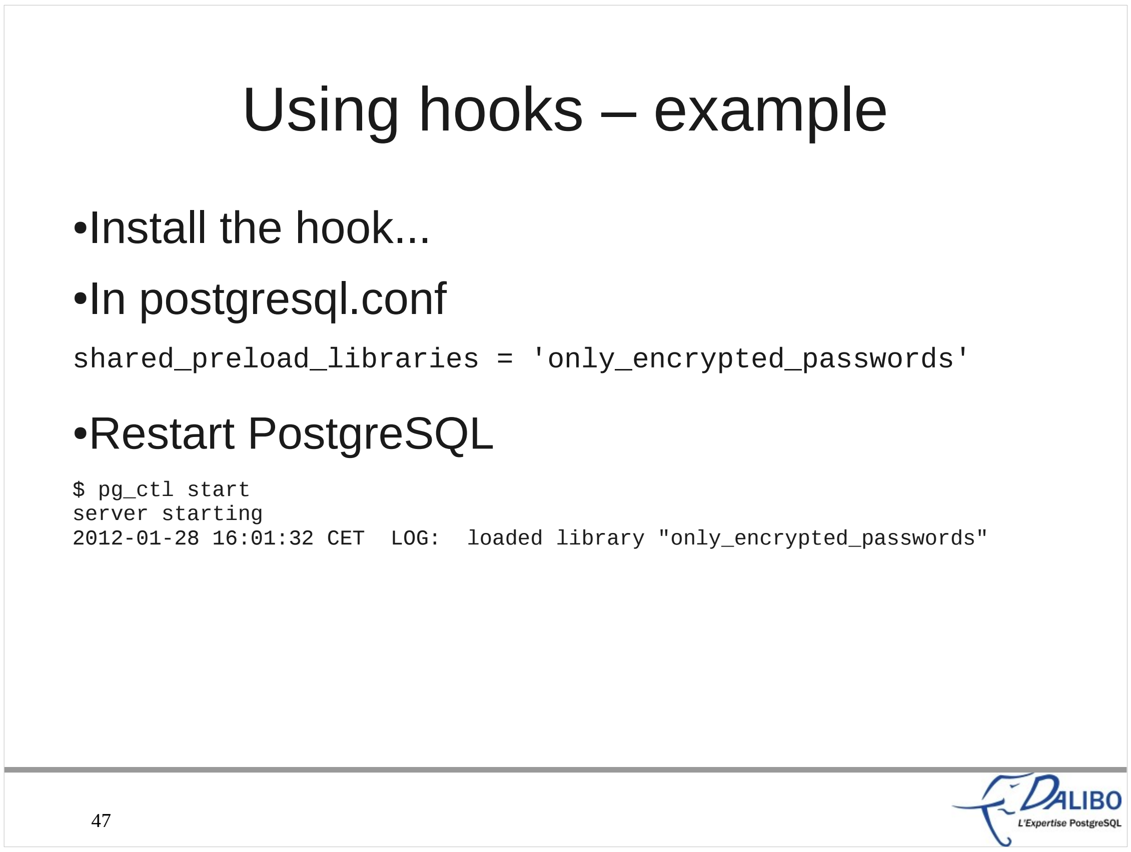
- Install the shared library
- In postgresql.conf
  - shared_preload_libraries
  - And possibly other shared library GUCs
- Restart PG

To use a hook, you first need to install the shared library.

After that, you need to change the configuration in the postgresql.conf file. There is at least one GUC to change (shared_preload_libraries). It consists on a list of library names, separated by commas. For example shared_preload_libraries = 'pg_stat_statements,pg_log_userqueries'

Don't forget to uncomment the line if it's commented.

Then, the only remaining work is to restart PostgreSQL.



Using hooks – example

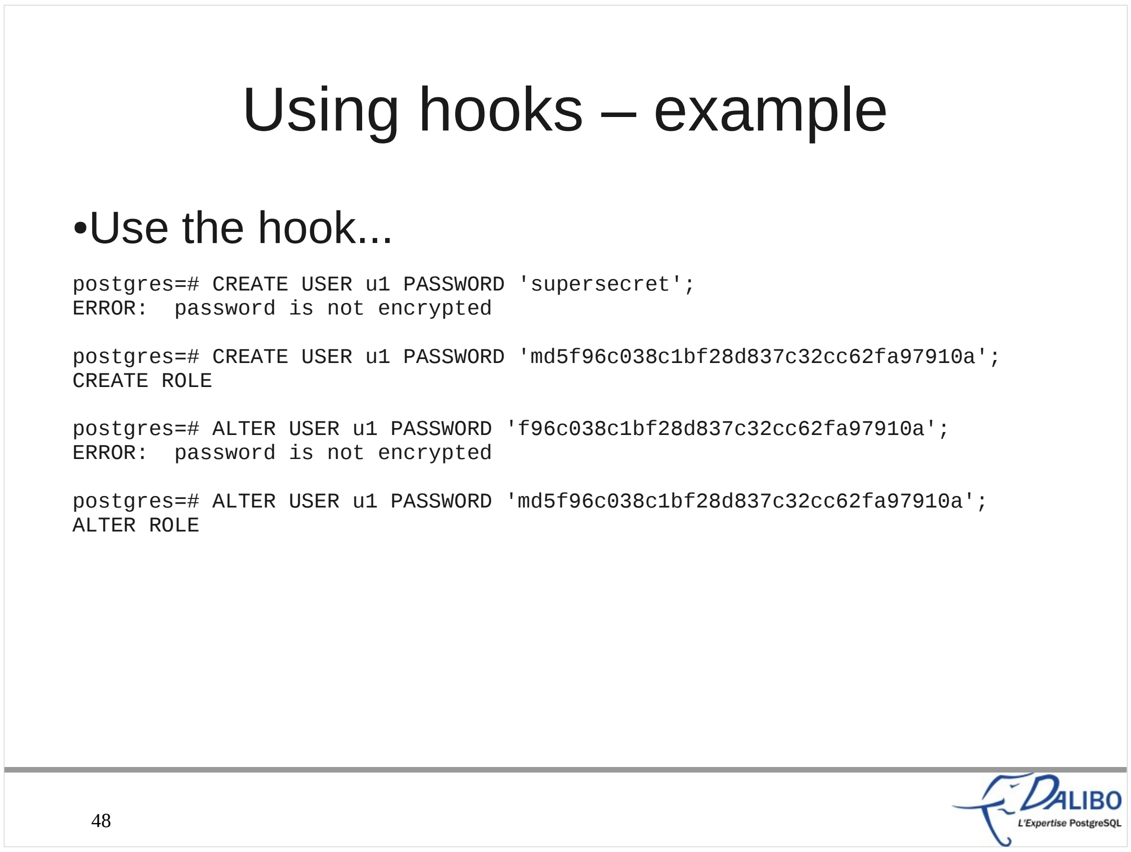
- Install the hook...
- In postgresql.conf

```
shared_preload_libraries = 'only_encrypted_passwords'
```

- Restart PostgreSQL

```
$ pg_ctl start
server starting
2012-01-28 16:01:32 CET  LOG:  loaded library "only_encrypted_passwords"
```

Here is example showing how to install the only_encrypted_password shared library, that used the checkpassword_hook.


Using hooks – example

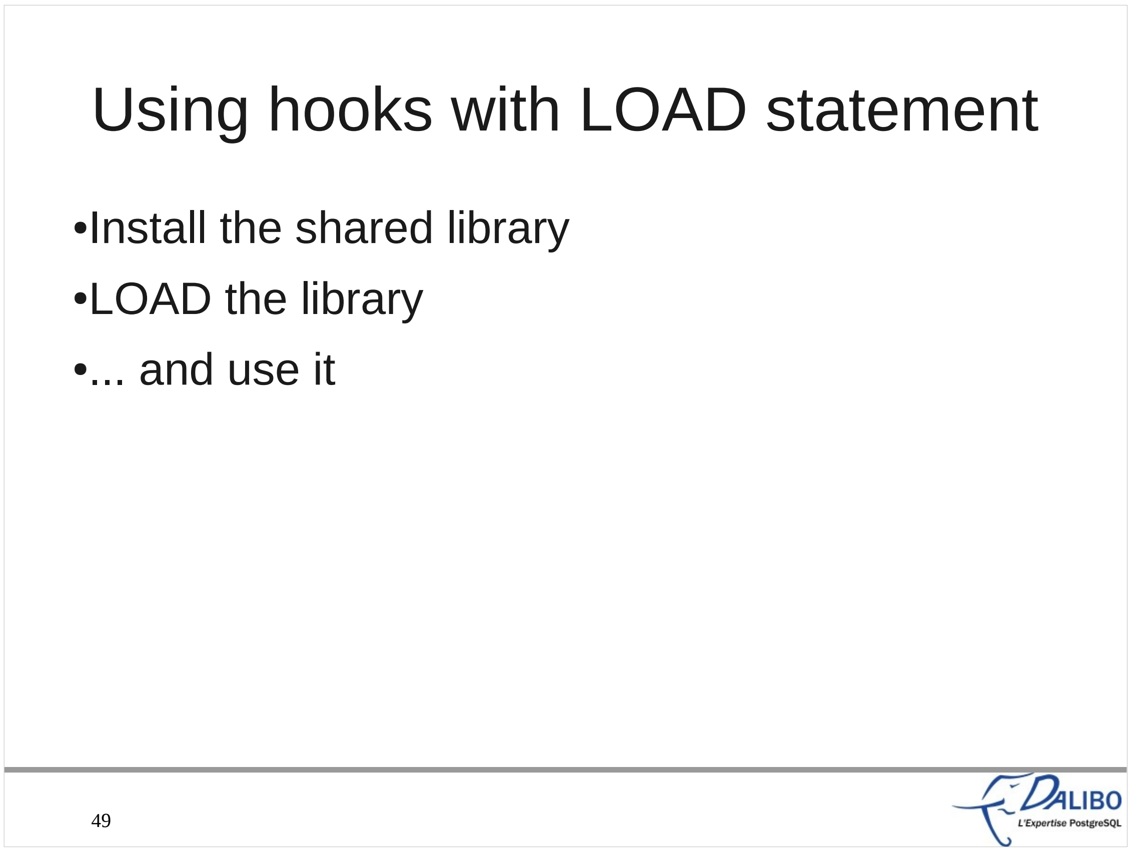
•Use the hook...

```
postgres=# CREATE USER u1 PASSWORD 'supersecret';
ERROR:  password is not encrypted

postgres=# CREATE USER u1 PASSWORD 'md5f96c038c1bf28d837c32cc62fa97910a';
CREATE ROLE

postgres=# ALTER USER u1 PASSWORD 'f96c038c1bf28d837c32cc62fa97910a';
ERROR:  password is not encrypted

postgres=# ALTER USER u1 PASSWORD 'md5f96c038c1bf28d837c32cc62fa97910a';
ALTER ROLE
```

And here is an example that shows its use.

Without an encrypted password, the CREATE or ALTER USER statement will fail, and our error message will appear.

With an encrypted password, everything work fine, as usual.

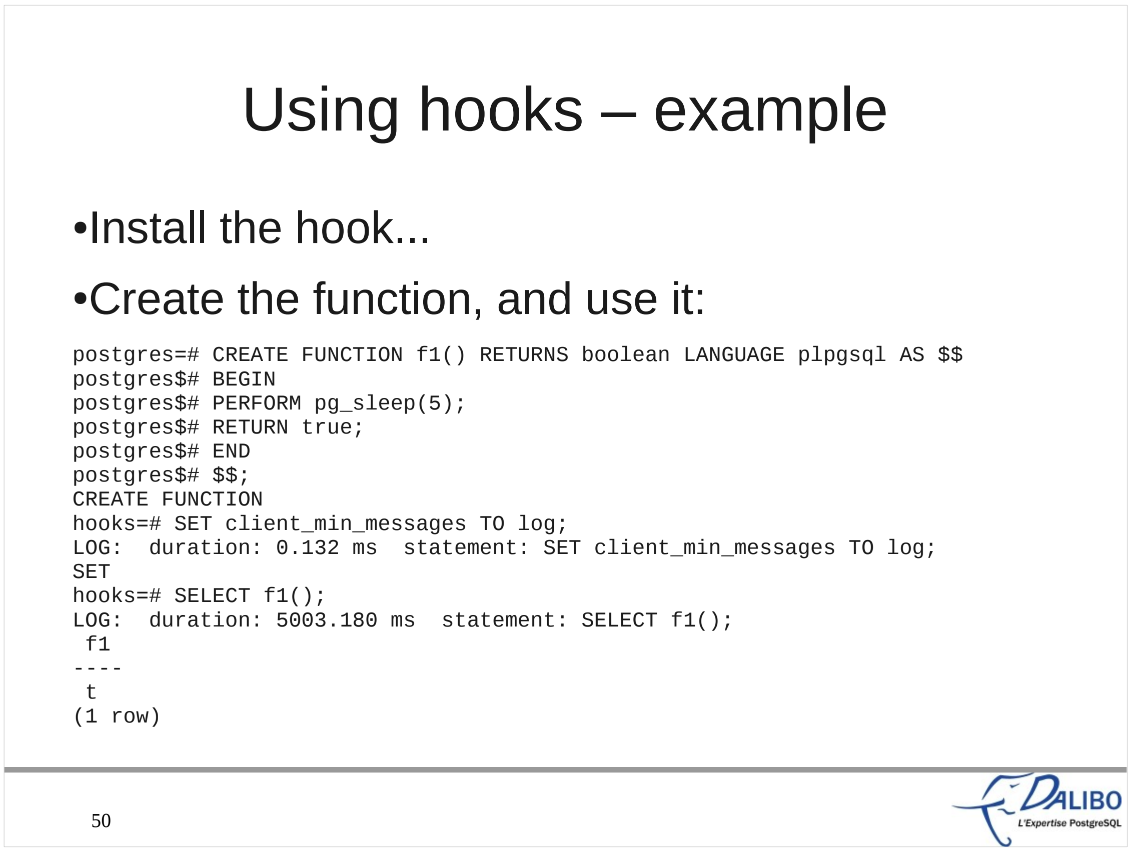


Using hooks with LOAD statement

- Install the shared library
- LOAD the library
- ... and use it

You're not required to preload a library. You can load it when you feel interested by its feature. It won't work for all libraries because some require shared memory which is only given at startup time.

To load a shared library, use the SQL statement LOAD.

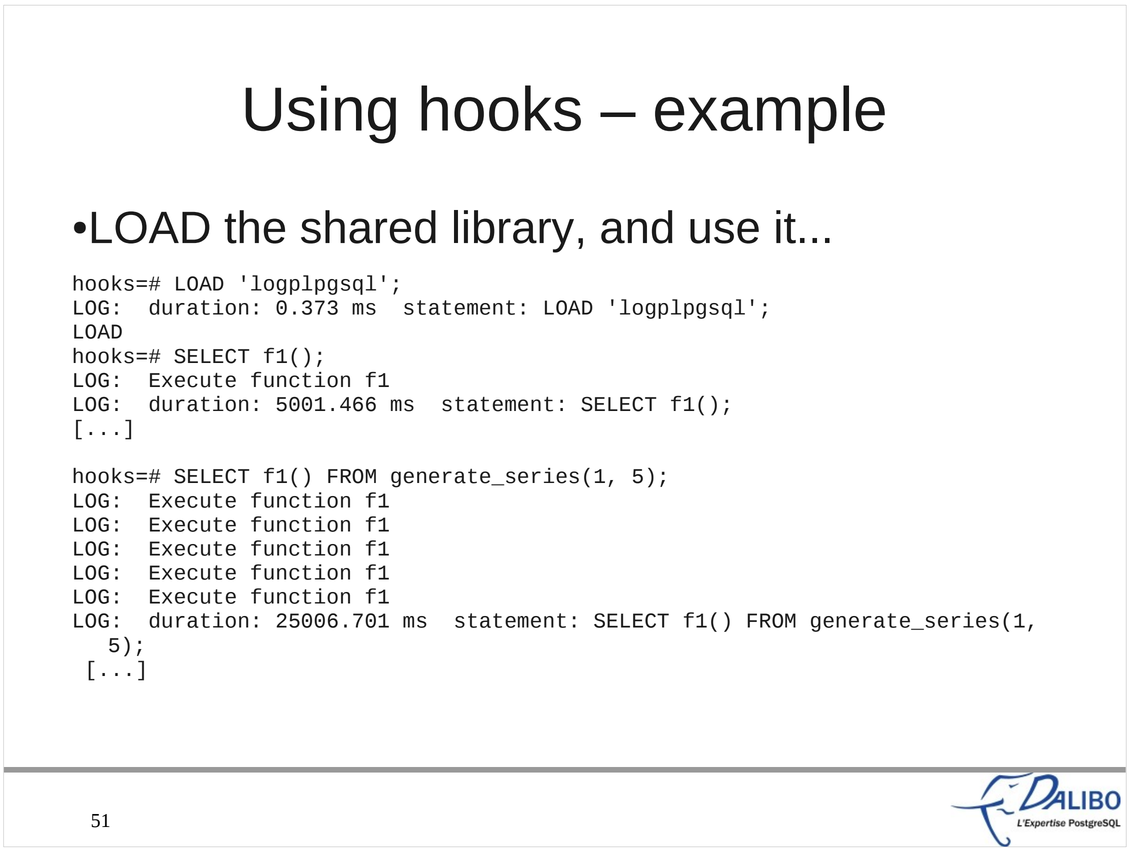


Using hooks – example

- Install the hook...
- Create the function, and use it:

```
postgres=# CREATE FUNCTION f1() RETURNS boolean LANGUAGE plpgsql AS $$
postgres$# BEGIN
postgres$# PERFORM pg_sleep(5);
postgres$# RETURN true;
postgres$# END
postgres$# $$;
CREATE FUNCTION
hooks=# SET client_min_messages TO log;
LOG:  duration: 0.132 ms  statement: SET client_min_messages TO log;
SET
hooks=# SELECT f1();
LOG:  duration: 5003.180 ms  statement: SELECT f1();
 f1
----
 t
(1 row)
```

In this example, we create a PL/pgsql function. Then we set client_min_messages to the LOG level. And finally we execute our function. Nothing special here.

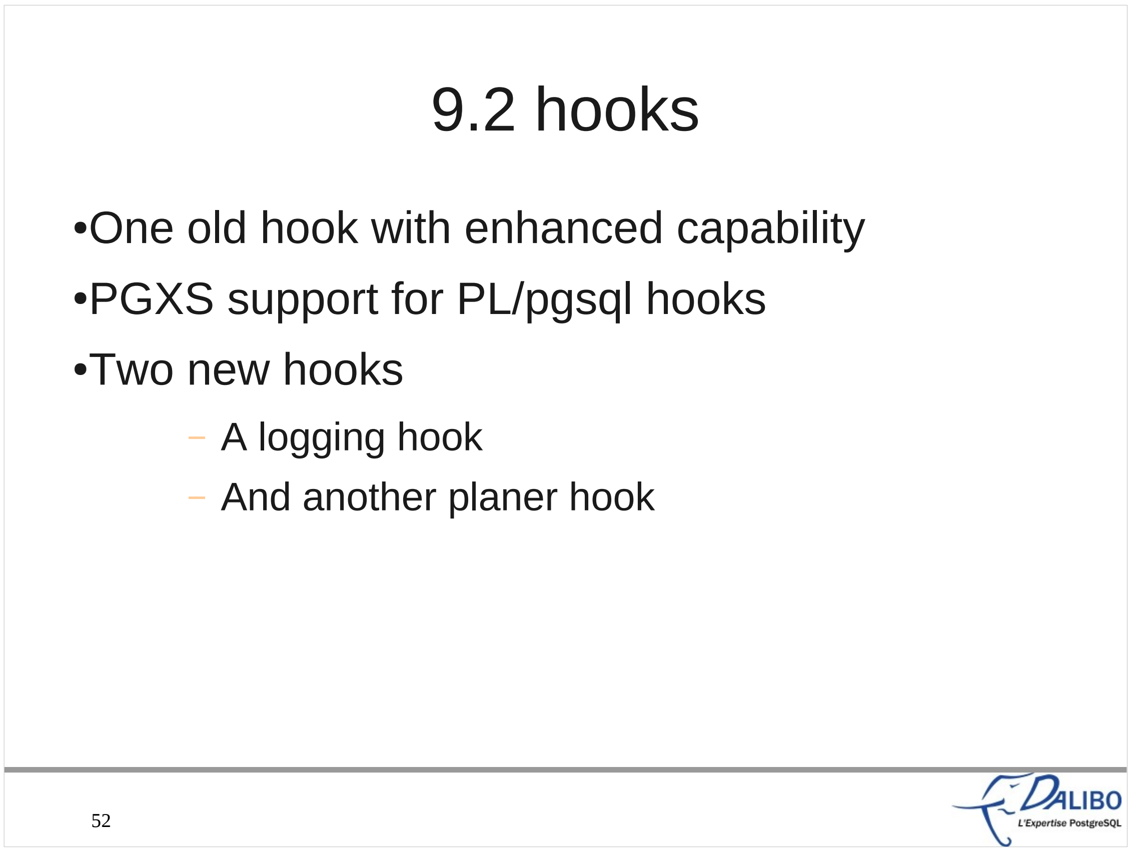


Using hooks – example

- LOAD the shared library, and use it...

```
hooks=# LOAD 'logplpgsql';
LOG:  duration: 0.373 ms  statement: LOAD 'logplpgsql';
LOAD
hooks=# SELECT f1();
LOG:  Execute function f1
LOG:  duration: 5001.466 ms  statement: SELECT f1();
[...]

hooks=# SELECT f1() FROM generate_series(1, 5);
LOG:  Execute function f1
LOG:  Execute function f1
LOG:  Execute function f1
LOG:  Execute function f1
LOG:  Execute function f1
LOG:  duration: 25006.701 ms  statement: SELECT f1() FROM generate_series(1,
   5);
 [...]
```

Now, we load the shared library logplpgsql with the LOAD statement.
We execute the PL/pgsql function, and we see new LOG messages. The shared library did its work.

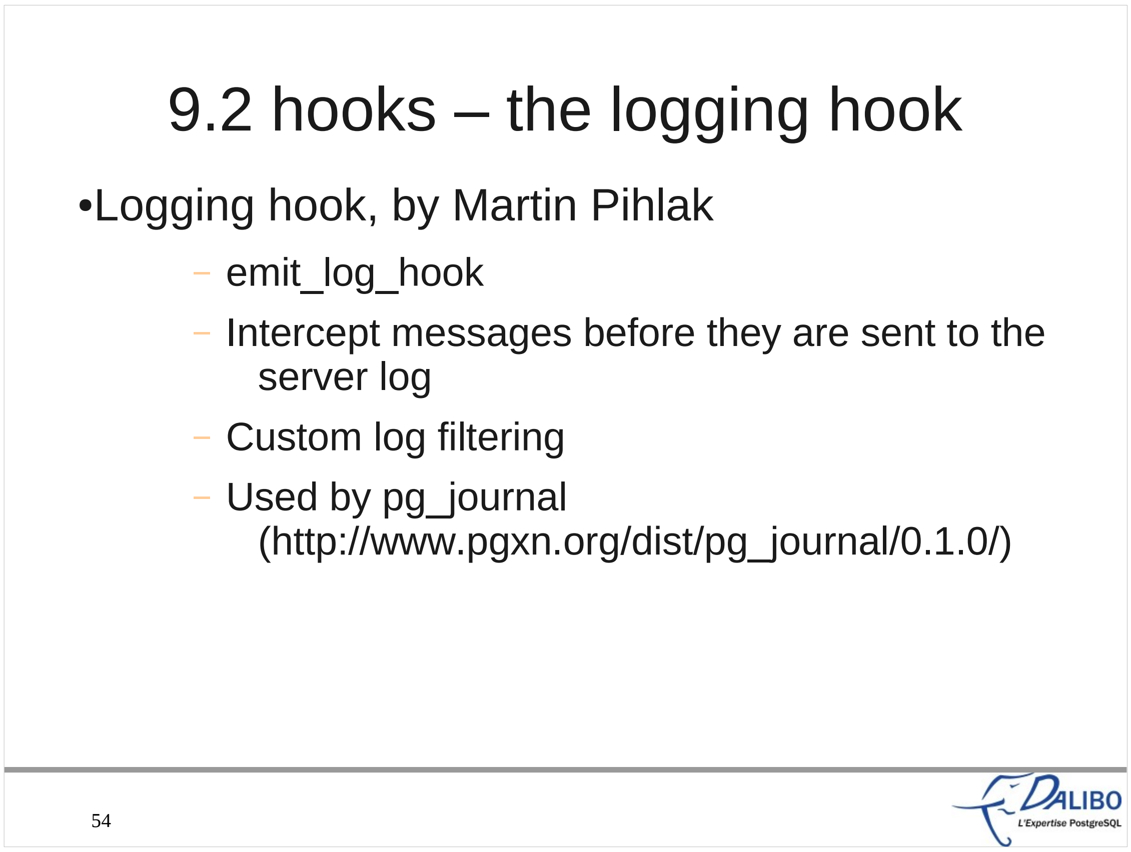


9.2 hooks

- One old hook with enhanced capability
- PGXS support for PL/pgsql hooks
- Two new hooks
  - A logging hook
  - And another planer hook

9.2 – Enhanced object_access_hook

- DROP statement support for object_access_hook
- Used by sepgsql

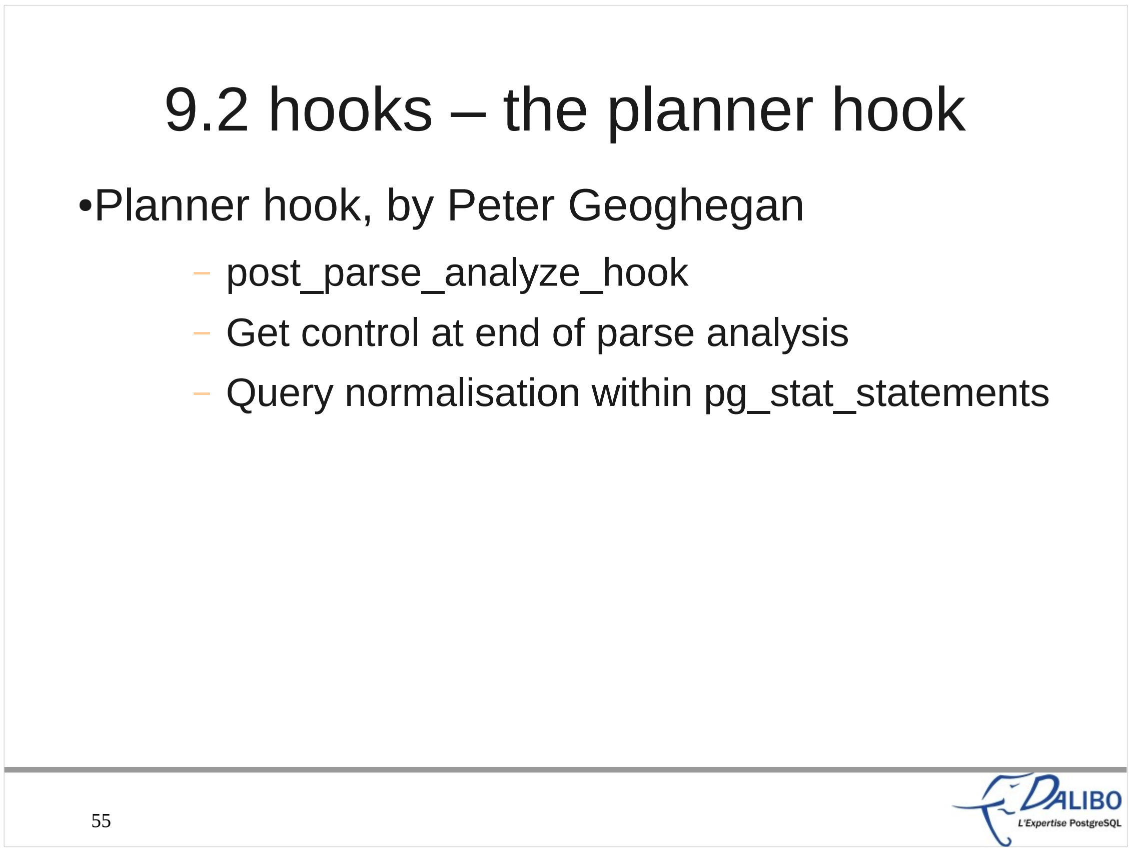


9.2 hooks – the logging hook

- Logging hook, by Martin Pihlak
  - emit_log_hook
  - Intercept messages before they are sent to the server log
  - Custom log filtering
  - Used by pg_journal (http://www.pgxn.org/dist/pg_journal/0.1.0/)

54

The logging hook was written by Martin Pihlak. The main idea behind this hook is to send logs to something else than PostgreSQL logging collector or syslog. It allows filtering, and writing to different log files.



9.2 hooks – the planner hook

- Planner hook, by Peter Geoghegan
  - post_parse_analyze_hook
  - Get control at end of parse analysis
  - Query normalisation within pg_stat_statements

The post_parse_analyze_hook was written by Peter Geoghegan while he was working on a feature for the pg_stat_statements contrib module.

This hook helps to get control at the end of the parse analysis. It allows query normalisation with pg_stat_statements.

Conclusion

- Hooks are an interesting system to extend the capabilities of PostgreSQL
- Be cautious to avoid adding many of them
- We need more of them :-)

- Examples and slides available on:
  - https://github.com/gleu/Hooks-in-PostgreSQL