Hooks in PostgreSQL

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
  - Etc
- Less known is the hook system

Hooks

- Interrupt, and modify behaviour
- Different kinds of hooks
- Not known because
  - not explained in the documentation
  - Usually quite recent

Most used hooks

Hook	Initial release
check_password_hook	9.0
ClientAuthentication_hook	9.1
ExecutorStart_hook	8.4
ExecutorRun_hook	8.4
ExecutorFinish_hook	8.4
ExecutorEnd_hook	8.4
ExecutorCheckPerms_hook	9.1
ProcessUtility_hook	9.0

Other hooks

Hook	Used in	**Initial release**
explain_get_index_name_hook		8.3
ExplainOneQuery_hook	IndexAdvisor	8.3
fmgr_hook	sepgsql	9.1
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
needs_fmgr_hook	sepgsql	9.1
object_access_hook	sepgsql	9.1
planner_hook	planinstr	8.3
shmem_startup_hook	pg_stat_statements	8.4

And one plugin

- PlpgSQL_plugin
- Used by EDB's PL/pgsql debugger, and profiler

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

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!

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!

Example with ClientAuthentication_hook

- Declaration
  - extract from src/include/libpq/auth.h, line 27

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

Example with ClientAuthentication_hook

- Set
  - 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;
```

Example with ClientAuthentication_hook

- Check, and execute
  - extract from src/backend/libpq/auth.c, line 580

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

Writing hooks

- Details on some hooks
  - ClientAuthentication
  - Executor
  - check_password
- And various examples

ClientAuthentication_hook details

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

ClientAuthentication_hook use

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

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

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

Writing a ClientAuthentication_hook

- First, initialize the hook

```
static ClientAuthentication_hook_type next_client_auth_hook = NULL;
/* Module entry point */
void
_PG_init(void)
{
    next_client_auth_hook = ClientAuthentication_hook;
    ClientAuthentication_hook = my_client_auth;
}
```

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 (next_client_auth_hook)
        (*next_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!!")));
}
```

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

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/

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 :-)

Writing a 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 = pgluq_ExecutorEnd;
}
```

Writing a 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
pgluq_ExecutorEnd(QueryDesc *queryDesc)
{
  Assert(query != NULL);

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

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

Writing a ExecutorEnd_hook

- Finally, uninstall the hook

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

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 :-/

check_password hook use

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

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

Writing a check_password_hook

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

Writing a check_password_hook

- First, install the hook

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

Writing a check_password_hook

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

```
static void
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")));
  }
}
```

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 – 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
```

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'
```

Using hooks

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

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"
```

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
```

Future hooks?

- Logging hook, by Martin Pihlak
  - https://commitfest.postgresql.org/action/patch_view?id=717
- Planner hook, by Peter Geoghegan
  - parse_analyze() and parse_analyze_varparams()
  - Query normalisation within 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

Hooks in PostgreSQL

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
  - Etc
- Less known is the hook system

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 quite unknown, even if the first hooks were available since the 8.3 release.

Hooks

- Interrupt, and modify behaviour
- Different kinds of hooks
- Not known because
  - not explained in the documentation
  - Usually quite recent

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.

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

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 in 8.4, 2 in 9.0, and 5 in 9.1. But the biggest issue is probably that it's not discussed in the documentation.

Most used hooks

Hook	Initial release
check_password_hook	9.0
ClientAuthentication_hook	9.1
ExecutorStart_hook	8.4
ExecutorRun_hook	8.4
ExecutorFinish_hook	8.4
ExecutorEnd_hook	8.4
ExecutorCheckPerms_hook	9.1
ProcessUtility_hook	9.0

There are many hooks available. These are the most used hooks. We'll discuss them in the rest of these slides.

All the Executor hooks 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.

The check_password hook is a way to check passwords according to enterprise rules.

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

Other hooks

Hook	Used in	Initial release
explain_get_index_name_hook		8.3
ExplainOneQuery_hook	IndexAdvisor	8.3
fmgr_hook	sepgsql	9.1
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
needs_fmgr_hook	sepgsql	9.1
object_access_hook	sepgsql	9.1
planner_hook	planinstr	8.3
shmem_startup_hook	pg_stat_statements	8.4

shmem_startup_hook, called when PostgreSQL initializes its shared memory segment

explain_get_index_name_hook, called when explain finds indexes' names

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

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)

ExplainOneQuery_hook see http://archives.postgresql.org/pgsql-patches/2007-05/msg00421.php

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

explain_get_index_name, to allow plugins to get control here so that plans involving hypothetical indexes can be explained

fmgr_hook, function manager hook (security definer stuff?)

object_access_hook, module sepgsql

And one plugin

- PlpgSQL_plugin
- Used by EDB's PL/pgsql debugger, and profiler

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.

Another name (plugin), but same idea behind.

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!

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 most of shared libraries, so that they can initialize memory and stuff like that. 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
  - extract from src/include/libpq/auth.h, line 27

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


These two lines declare the ClientAuthentication hook.

Example with ClientAuthentication_hook

- Set
  - 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 and sets the ClientAuthentication_hook to its initial value: 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 say that the ClientAuthentication hook will be launched if it has been set previously.

Writing hooks

- Details on some hooks
  - ClientAuthentication
  - Executor
  - check_password
- And various examples

This part will go into much greater details on some of the available hooks: ClientAuthentication, the Executor ones, and check_password

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 use 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

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 adds specific SELinux context to allow a connection
- connection_limits, written by Tomas Vondra, and available on GitHub, allows 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 next_client_auth_hook = NULL;
/* Module entry point */
void
_PG_init(void)
{
    next_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 (next_client_auth_hook)
        (*next_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).

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), and a last one to write the log if the user has the SUPERUSER attribute.

Writing a 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 = pgluq_ExecutorEnd;
}
```


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

Writing a 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
pgluq_ExecutorEnd(QueryDesc *queryDesc)
{
  Assert(query != NULL);

  if (superuser())
    elog(log_level, "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.

Writing a 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

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. 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. Crypt was available until the 8.4 release.

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

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.

Writing a check_password_hook

- First, install the hook

```
void _PG_init(void)
{
  check_password_hook = 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
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.

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
```


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.

Using hooks

- 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.

Future hooks?

- Logging hook, by Martin Pihlak
  - https://commitfest.postgresql.org/action/patch_view?id=717
- Planner hook, by Peter Geoghegan
  - parse_analyze() and parse_analyze_varparams()
  - Query normalisation within pg_stat_statements

For 9.2, there is at least one patch offering a new hook. It is a logging hook. The main idea behind this hook is to send logs to something else than PostgreSQL or syslog.

Another patch, not yet available, is written by Peter Geoghegan to get query normalisation inside 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