PostgreSQL Pass-The-Hash protocol design weakness

Date: Monday, 2nd March 2015
Version tested: PostgreSQL v9.5, older ones are affected as well
Credits:

- Jens Steube: jens.steube __AT__ gmail.com
- Philipp Schmidt: philipp.smd __AT__ gmail.com

The PostgreSQL Challenge-Response Authentication using the AUTH_REQ_MD5 method or simply configuring "md5" as the Host Based Authentication (HBA) in pg_hba.conf is the default setting on many linux distributions as well as recommended in the default configuration on github:

METHOD can be "trust", "reject", "md5", "password", "gss", "sspi", "ident", "peer", "pam", "ldap", "radius" or "cert".
Note that "password" sends passwords in clear text; "md5" is preferred since it sends encrypted passwords.

It has a severe protocol design weakness. The weakness we have found here, and that we can demonstrate with a proof of concept (POC) code, is also known as a pass-the-hash (PTH) vulnerability [1].

The basic idea and problem of PTH is that an attacker can authenticate to a service (in this case to the PostgreSQL database server) without knowing the actual password. The only information he needs to know is a valid combination of username and the hashed password which is stored in the database. This hash could be directly dumped from the disk, from a database backup or could originate from a very different database in case the credentials (username and password) were reused.

PTH is most prominently known from mimikatz [2] where it is used to authenticate to windows services (for instance the ntlm and older samba protocols). In general, the information security industry classifies a PTH vulnerability as a very critical and severe problem since the attacker does not need to invest many resources (money, hardware, time etc) to guess the plaintext of the hash to come up with the actual password [3]. Instead, the service login can be simply accomplished by getting and sending the hash as-is, without any need to have knowledge of the users credentials.

The fact that the PostgreSQL protocol is able to use an encrypted communication using SSL does not affect the ability to exploit the PTH weakness.

Technical description of the weakness in PostgreSQL

PostgreSQL server store user passwords in the following salted and hashed format:

MD5 ($pass . $username)

where $username denotes the PostgreSQL account (role) and $pass is the actual plaintext password. From now on we will use $salt and $username indistinguishable; the concatenation of $pass and $username will be denoted by P (except if the contrary is explicitly stated).

The current PostgreSQL Challenge-Response Authentication protocol using AUTH_REQ_MD5 works like this:

1. Server generates R = 4 random bytes
2. Server sends R to the client
3. Client computes F_c = H(H(P) . R)
4. Client sends F_c to the server
5. Server has H(P) stored in the database
6. Server computes F_s = H(H(P) . R) and compares F_s with F_c

Notes:

- The hashing algorithm H is always MD5 for the AUTH_REQ_MD5 method
- F_c and F_s should be identical but were distinguished here with a suffix to emphasize that F_c is always computed on client side, while F_s is newly and independently computed on server side

By analyzing step number 3, one can easily identify the main weakness of this protocol, i.e. the H(P) value is hashed again (by concatenating the challenge) but during the whole communication (step 1 to 6) an attacker does not need to proof that he has the actual P value (password). An attacker just needs to have H(P) which does not directly proof that he knows the password (for the reasons mentioned before). Furthermore, H(P) is directly stored in the server database, this does weaken the protocol even more, since the client/attacker could have obtained H(P) from there.

Therefore, we would say that this problem could be seen as a protocol design flaw and to proof this, we will attach a patch/diff to this document such that it is easier to understand the actual "shortcut" the attacker could use to circumvent the secure authentication (which should always ensure that the actual user credentials were indeed involved on client side).

Demonstration

The following step-by-step guide will help you in testing the PTH vulnerability. For convenience, we did apply the changes (a patch) directly to the libs in PostgreSQL. Alternative an attacker could simply build a standalone utility that behaves like "psql" or "pg_dumpall" etc.

The attached diff file is not a patch to solve this problem, instead it is a demonstration of the weakness. It will only modify the code of the client.

1. First download the latest PostgreSQL git code (we did use git commit with sha1 checksums up to ee4ddcb38a0abfdb8f7302bbc332a1cb92888ed1):
   git clone git://git.postgresql.org/git/postgresql.git
2. Enter the postgresql directory:
   cd postgresql
3. Apply the patch:
   git apply postgresql_diff_clean.txt
4. Configure the PostgreSQL project:
   ./configure
5. Run make:
   make && make install
6. Depending on your distro (and installation path), you may need to set $PATH:
   export PATH=$PATH:/usr/local/pgsql/bin/
7. Initialize the database (here we use a temp directory ~/postgres_data/):
   pg_ctl -D ~/postgres_data/ init
8. Make sure that md5 is the default authentication method (AUTH_REQ_MD5), if not edit the file ~/postgres_data/pg_hba.conf and double-check that "md5" is used as METHOD:
   vim ~/postgres_data/pg_hba.conf
9. Start the PostgreSQL database server:
   postgres -D ~/postgres_data
10. Set the password in psql (here we did use the unpatched psql binary but it doesn't really matter):
    psql template1
    CREATE ROLE postgres WITH PASSWORD 'hashcat';
    ALTER ROLE postgres WITH LOGIN;
11. Login with the modified psql utility (this won't require the user to input the actual password, we only need the "hash" in the format "md5" + the 32 hexadecimal MD5 hash):
    psql -U postgres

Notes:

- While we demonstrate here specifically with the psql utility which can exploit this PTH weakness, we actually mean all the other PostgreSQL tools that use this front-end libs too (and as noted above it is not limited to this, attackers could develop standalone tools)
- The modified psql utility will ask for a hash instead of a password on the prompt:
  Hash:
  or in case the -U switch is used:
  Hash for user [username]:
- We have developed two different patch files:
  - the first one *postgresql_diff_clean.txt* is meant to be complete and includes some additional and unrequired front end tweaks
  - The second diff file *postgresql_diff_minimal.txt* was instead designed to include just the changes that are necessarily required for this demonstration to work
- An attacker needs to input the hash as demonstrated below, i.e. the fixed string "md5" followed by 32 hexadecimal characters (the actual MD5 hash)

To conclude this proof of concept demonstration, here is an example how an attacker would extract the hash from the database. He is using a database on host “et” on which he gained access already:

```
postgres@et:~$ pg_dumpall -r
...
CREATE ROLE postgres;
ALTER ROLE postgres WITH SUPERUSER INHERIT CREATEROLE CREATEDB LOGIN REPLICATION
PASSWORD 'md5a6343a68d964ca596d9752250d54bb8a';
```

Now that the attacker knows the hash, here's an example how he can use the information to connect to a different remote database on host “sf” using our patched client. The client will ask for the hash instead of the password.

```
postgres@et:~$ psql -h sf
Hash : md5a6343a68d964ca596d9752250d54bb8a
psql (9.5devel, server 9.3.6)
SSL connection (protocol: TLSv1.2, cipher: DHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.
postgres=#
```

Just as a side note the password used for this demo is "hashcat" and user is "postgres":

```
postgres@et:~$ echo -n hashcatpostgres | md5sum
a6343a68d964ca596d9752250d54bb8a -
```

Transitioning to a more secure hash (Preparation)

Before we are going to show how to solve the PTH issue itself let's discuss how to transition existing hash to a more secure one. This can be seen as a preparation as it will come in handy within the PTH solution.

- Hash the current MD5 value with a hash algorithm H of your choice and replace it with the current MD5 value. That means, just for example, store bcrypt(MD5(P)) instead of just MD5(P). This enables the server to automatically convert all existing hashes as it is not required to ask the user for his password. The PostgreSQL database server can test the current hash in the database on startup, find out if the hash is still stored in the "old" format (for instance if the signature starts with "md5") and then automatically convert all hashes
- Alternatively think about dropping MD5 altogether. While it is not that badly broken as many people say, the reputation of MD5 is damaged and there are enough free and much more secure alternatives.
    - The advantage of this is you can use H(P) whenever we stated H(H(P))
    - The disadvantage of this is that you have to ask the user for a new password
- In the protocol description below we assume that the server database only stores H(H(P)) and does neither know H(P), nor is it easy to derive H(P) from H(H(P)). The only reasonable way an attacker could come up with H(P) is to guess the password P itself and this should be made very hard by using a secure hashing algorithm
- Consider about adding a special field for holding a unique salt. Currently you are using the username. This has the following effect:
    - If a user is getting renamed, the hash will not match any longer. Therefore the user is forced to supply a new password (or the same one again). However you can not do these changes without user interaction
    - In a bigger network with several PostgreSQL database installations it is likely that one and the same person has access to several databases using the same username and password. Therefore, by exploiting the PTH issue, it's enough to have the dump of one of the databases to access all databases because the salt is not unique for each database.
    - To make transition easier you can copy the username into the new salt attribute

Proposed solution for the PTH weakness

1. Server generates R = 32 random bytes
2. Server sends R to the client
3. Client computes C_c = HMAC(H(H(P)), R)
4. Client computes T_c = H(P)
5. Client computes F = T_c ^ C_c
6. Client sends F to the server
7. Server has H(H(P)) stored in the database
8. Server computes C_s = HMAC(H(H(P)), R)
9. Server computes T_s = F ^ C_s
10. Server computes H(T_s) and compares it with H(H(P))

Notes:

- Step 1 should use a cryptographically secure pseudo-random number generator (CSPRNG)
- C_c and C_s should be identical but are distinguished here with a suffix to make it clear that C_c was computed on client side while C_s will be newly and independently computed on server side (the same holds for T_c and T_s)
- The weakest point in the described idea above is that if an attacker has access to the database and already has the hash, he can calculate T_s as well. Since T_s is actually just H(P) that would be easier or faster to compute than H(H(P)).
    - Why does an attacker want to reduce H(H(P)) to H(P) and why does he even bother about getting and guessing H(P)?
    - Depending on the hashing algorithm H, H(P) could be much, much faster to crack than H(H(P))
    - The reason the attacker wants to get the clear text password is similar to the reasons described above: the passwords could enable him to access other services and the password may contain sensitive information itself
    - Therefore we highly recommend to use a different hashing algorithm, a much stronger one, instead of MD5, at least for the outer hash
- Use the binary result of MD5 instead of the ascii hex representation. This will result in a better attacker versus defender (A:D) ratio
- You can optionally replace the XOR in step #5 and #9 with a real cryptographically secure cipher. One possible secure way could be to encrypt T_c with C_c as the key using AES in step #5 and decrypt F with C_s as key on the server in step #9

Conclusion

Within this document we have demonstrated what this pass-the-hash (PTH) weakness in PostgreSQL is all about, the technical details that describe the idea behind PTH in general and also specifically applied to the PostgreSQL challenge-response authentication, a proposed solution and finally also how an attacker can misuse all this in real life.
To conclude, we want to emphasize again that a PTH weakness should not be underestimated/neglected and we have all the faith in the PostgreSQL development team/community that this problem will be addressed responsible and as soon as possible.

Links

[1] Wikipedia: Pass the hash
[2] Benjamin Delpy: Mimikatz
[3] SANS Institute: Pass-the-hash attacks: Tools and Mitigation