Tell MD5 to SCRAM!

JONATHAN S. KATZ

MAY 30, 2019

crunchy data

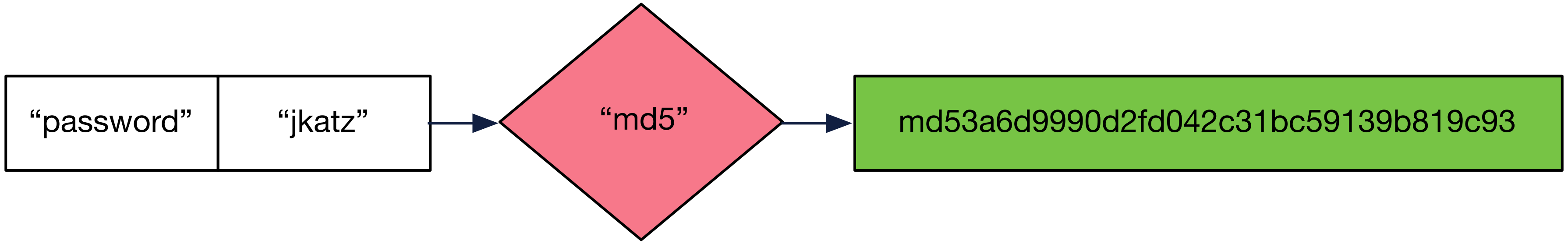
A Brief History of PostgreSQL Password Management

crunchydata

Before PostgreSQL 10: "password"

- Stored the password as plaintext in the database
- Which is fine if you:
  - Only authenticate with the password over encrypted connections
  - Trust your database superusers
  - Trust your system superusers
  - Never use your database password anywhere else. Ever.
- There were reason to use this method, e.g. your PostgreSQL connection driver did not support the MD5 method.
  - This reason is no longer valid.

Before PostgreSQL 10: MD5



“password”

“jkatz”

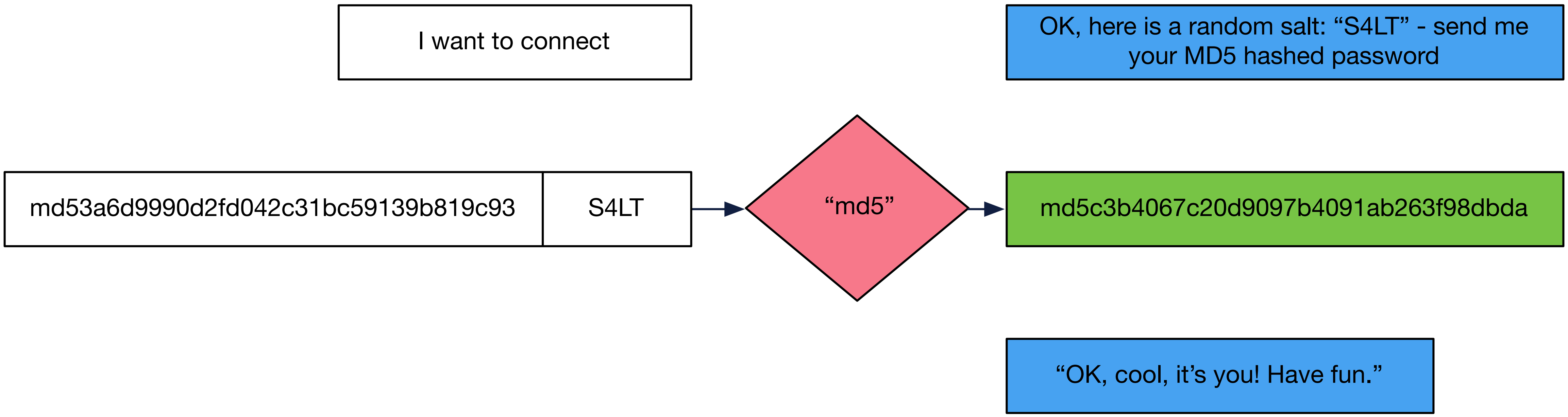
“md5”

md53a6d9990d2fd042c31bc59139b819c93

- Stored the password as a salted MD5 hash, where the salt is the username
- Prepends "md5" so PostgreSQL knows that it is a MD5 stored password

Before PostgreSQL 10: MD5

Client



I want to connect

OK, here is a random salt: “S4LT” - send me your MD5 hashed password

md53a6d9990d2fd042c31bc59139b819c93

S4LT

“md5”

md5c3b4067c20d9097b4091ab263f98dbda

“OK, cool, it’s you! Have fun.”

- When authentication with the MD5 method, PostgreSQL sends over a random salt and asks the client to send a MD5 hash over with the md5 hashed password and the salt

MD5: Of Course It's Safe!

```
$ pg_dumpall

--
-- Roles
--

CREATE ROLE jkatz;
ALTER ROLE jkatz WITH LOGIN PASSWORD 'md53a6d9990d2fd042c31bc59139b819c93';
```

- It is provably very difficult to gain access to one's MD5 hash, even by accident.
- And even more challenging to authenticate with it.

MD5 Needs to SCRAM

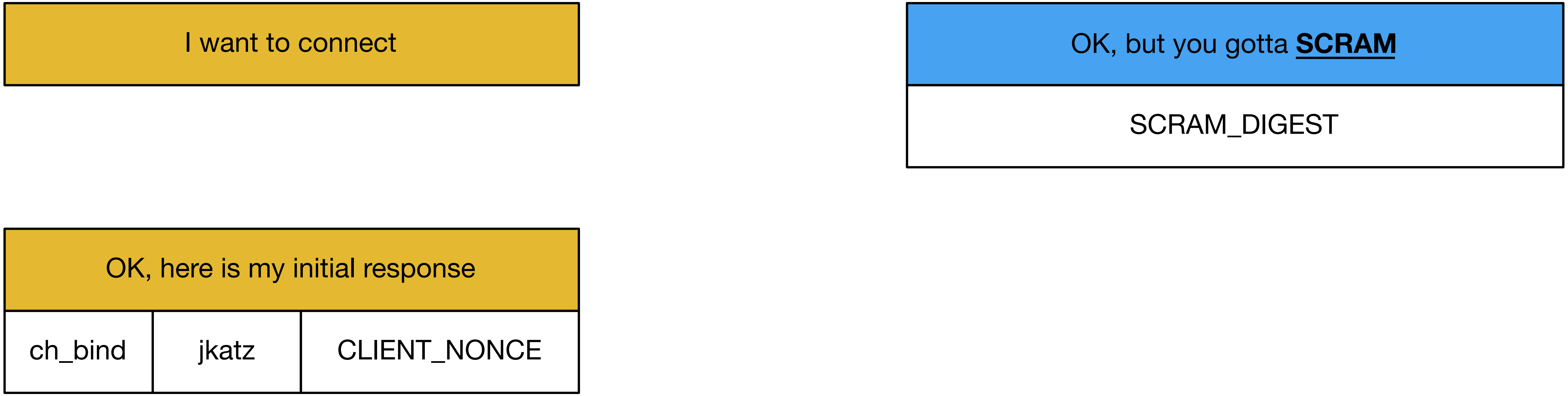
SCRAM? That Seems Rude...

- "**S**alted **C**hallenge **R**esponse **A**uthentication **M**ethod"
- It's a standard! RFC5802
- Defines a method for a client and server to authenticate **without ever sharing the password**
- Also allows client + server to validate each others i

```
<DIGEST>$<ITERATIONS>:<SALT>$<STORED_KEY>:<SERVER_KEY>
```

Authentication the SCRAM Way

Client



I want to connect

OK, but you gotta **SCRAM**

SCRAM_DIGEST

OK, here is my initial response

ch_bind	jkatz	CLIENT_NONCE

Authentication the SCRAM Way

Client

Alright, so it looks like you append to my nonce. Cool. I'm going to generate a **PROOF** for you to validate that I know the PASSWORD.

I will take the plaintext PASSWORD that I *think* is correct, initializing with ***SALT***, and then apply HMAC using **SCRAM_DIGEST** for ***ITERATIONS*** which gives me a SALTED_PASSWORD

To finish the proof, I will derive the ***STORED_KEY***, which is the **SCRAM_DIGEST** of the HMAC of SALTED_PASSWORD with "Client Key".

I build a CLIENT_SIGNATURE which is the HMAC using STORED_KEY and information about this session

ch_bind	CLIENT_NONCE + SERVER_NONCE	CLIENT_KEY XOR CLIENT_SIGNATURE

Oh yeah? Well, I'm going to send you some stuff to see if we can both come to the same conclusion about the password

CLIENT_NONCE + SERVER_NONCE	SALT	ITERATIONS

Authentication the SCRAM Way

Client

Oh you think you're so clever?

I can compute CLIENT_SIGNATURE because I have the ***STORED_KEY*** and the session information.

I'll XOR that with the proof and get the CLIENT_KEY.

If your CLIENT_KEY is valid, its **SCRAM_DIGEST** and it will be the same as *STORED_KEY*.

So you can trust me, I'll send you a SERVER_SIGNATURE which is the HMAC with ***SERVER_KEY*** and the session information

SERVER_SIGNATURE

OK, I can create SERVER_SIGNATURE as I can derive the SERVER_KEY using a HMAC with SALTED_PASSWORD with "Server Key" and then see if I can match SERVER_SIGNATURE.

If it does, I trust that you authenticated me, and we can move forward.

Upgrading to SCRAM

- In postgresql.conf set `password_encryption` to `scram-sha-256`
- Keep `md5` as your authentication method in pg_hba.conf until all your users have re-hashed their passwords
  - ...have your users re-hash their passwords. Best way is `\password`
- Once all of your users have re-hashed their password, switch your authentication method to `scram-sha-256`

But wait there's more!

- Channel binding, introduced in PostgreSQL 11, allows SCRAM to use elements of TLS to
  - Ensure the SSL handshake is still the same when verifying identities
- Prevents man-in-the-middle attacks!

Wow, did I do that in five minutes?

Jonathan S. Katz
@jkatz05

crunchy data