2ndQuadrant®
PostgreSQL

Using SSL with PostgreSQL

Andrew Dunstan
andrew.dunstan@2ndquadrant.com

A note on terminology

When we say SSL we really mean TLS, the successor to SSL.

Why use SSL/TLS

- talk securely
- no-one should be able to listen in
- make sure you are talking to the right party

Two basic functions

- Encryption
- Authentication

Managed by X.509 certificates

- These contain
  - a public encryption key
  - identity information
  - a signature
  - other stuff

For every certificate there is a key

- The certificate is useless without the key.

How can we trust a certificate?

- if it's been signed in a way we trust
- if the party presenting the certificate proves they have the key
- if the certificate contains the name we are expecting

Types of Cryptography

- Symmetric
- Asymmetric

Symmetric cryptography

- Encrypt(plaintext, k) => ciphertext
- Decrypt(ciphertext, k) => plaintext

- Note the same key k is used in both operations
  - Need to keep k secure on both sides
  - Communicating the key securely is hard

Asymmetric cryptography

- Encrypt(plaintext, ke) => ciphertext
- Decrypt(ciphertext, kd) => plaintext

- The same key is not used.
  - Side doing encrypting doesn't need kd

Public Key Cryptography

- An asymmetric cryptography system where knowing ke doesn't help you to discover kd.
- Best known system is RSA
  - relies on the difficulty in factoring the product of two very large prime numbers.
- You can publish ke quite safely as long as you keep kd secure.

How this works

- (simplified)
- C: Hi, I'd like to talk securely
- S: Here's my certificate with my public key
- C: Here's something encrypted with the public Key.
- S: Here's your thing back decrypted, proving I have the key.
- C: That worked, so here's a symmetric key encrypted with the public key
- S: Got it, we'll use that for the rest of this conversation

Why switch to a symmetric key?

- Far far cheaper computation
- Doesn't require the client to have a certificate
- Almost all PK systems use this hybrid technique

Things to note

- So far the server isn't authenticated
- The client hasn't used certificate or key of its own.
  - only the server's certificate is ever used for encryption

How do I know you're who you claim to be?

- Authentication
  - Is the name in the certificate what I expect?
  - Is the certificate signed in a way I trust?
  - Has the other side proved they have the key that goes with this certificate? (yes)

Certificate trust

- Certificates can be
  - self-signed
  - signed by a Certificate Authority

Self-signed certificates

- Useful for testing
- Should not be used in production

Certificate Authorities

- Root CAs
- Intermediate CAs
  - delegated by a Root CA
  - or another intermediate CA

Certificate will contain a signature

- signature is verified against the certificate of the Root CA
- if signed by an intermediate CA, the certificate must include a chain of CA certificates back to the Root CA.

Types of names

- In Postgres, the name can be one of three things
- a Host Name (server certificate)
- an IP address (server certificate)
- a User Name (client certificate)

Host name checking

- If the subject has Subject Alternative Names, the host name must match one of those.
- Otherwise, the host name must match the Common Name (CN) field of the certificate's Subject.
- the host name checked is the one that the client connects to, i.e. the `host` field in a connection string

IP Address checking

- must match the CN of the certificate subject field
- currently no support of SANs for IP addresses
- used when the host is specified by address rather than by name

Client name checking

- Done by the server when a client certificate is used
- must match the CN field of the certificate subject
- must match connecting user or a user name map system-username

Connection Modes

- `libpq` and `jdbc` have 6 `sslmode` values
- 4 unverified
- 2 verified

Unverified Connection modes

- `disabled` (do not use SSL)
- `allow` (try non-SSL, then SSL)
- `prefer` (try SSL, then non-SSL)
- `require` (only try SSL)

- None of the above do any authentication. They will accept any server certificate with any name and signature.

Verified Connection Modes

- `verify-ca` - only use SSL and verify the server certificate signature.
- `verify-full` - only use SSL and verify the server certificate signature and host name / IP Address
  - equivalent to what web browsers do when connecting to SSL enabled sites.

Which CA to use?

- You can use any convenient CA
  - Lets-encrypt
  - Any commercial provider
    - Digicert, Entrust etc.
  - Your corporate internal CA
  - Roll your own

- Whichever you use, you need the root certificate for verification

OpenSSL commands

We use these commands from the `openssl` suite:

- `openssl req` - to generate a Certificate Signing Request (csr) and key
- `openssl req -x509` - to generate a self-signed certificate and key
- `openssl x509` - to sign requests or display certificate info
- `openssl ca` - to sign requests
- `openssl pkcs8` - to convert a key to PKCS#8 format for jdbc use
- `openssl rand` - for generating random passwords

Sample scripts

- Following examples are based on the sample scripts
- https://github.com/adunstan/ssl-scripts

Roll your own CA

- Instructions for Redhat/Centos/Fedora - adjust to taste
-
```
SUBJ='/C=US/ST=North Carolina/L=Apex/O=test/OU=eng'
rm -rf cadir; mkdir cadir; cd cadir
DIR=`pwd`
capw=`openssl rand -base64 30`
cp /etc/pki/tls/openssl.cnf .
sed -i -e "s,^dir.*,dir = $DIR," -e 's/#unique_subject/unique_subject/ \
    openssl.cnf
sed -i -e 's/# copy_extensions/copy_extensions/' openssl.cnf
mkdir certs private newcerts
echo $capw > private/ca.pw # not in production
chmod 700 .; echo 1000 > serial; touch index.txt; echo 01 > crlnumber
openssl req -passout pass:$capw -new -x509 -days 3650 -extensions v3_ca \
   -extfile openssl.cnf -subj "$SUBJ/CN=My Root CA" -keyout private/cakey.pem \
   -out cacert.pem >/dev/null 2>&1
cd ..
```

Intermediate CAs

- A root CA is one that signs its own certificate.
- An intermediate CA is one where the certificate is signed by
  - a root CA , or
  - another intermediate CA.

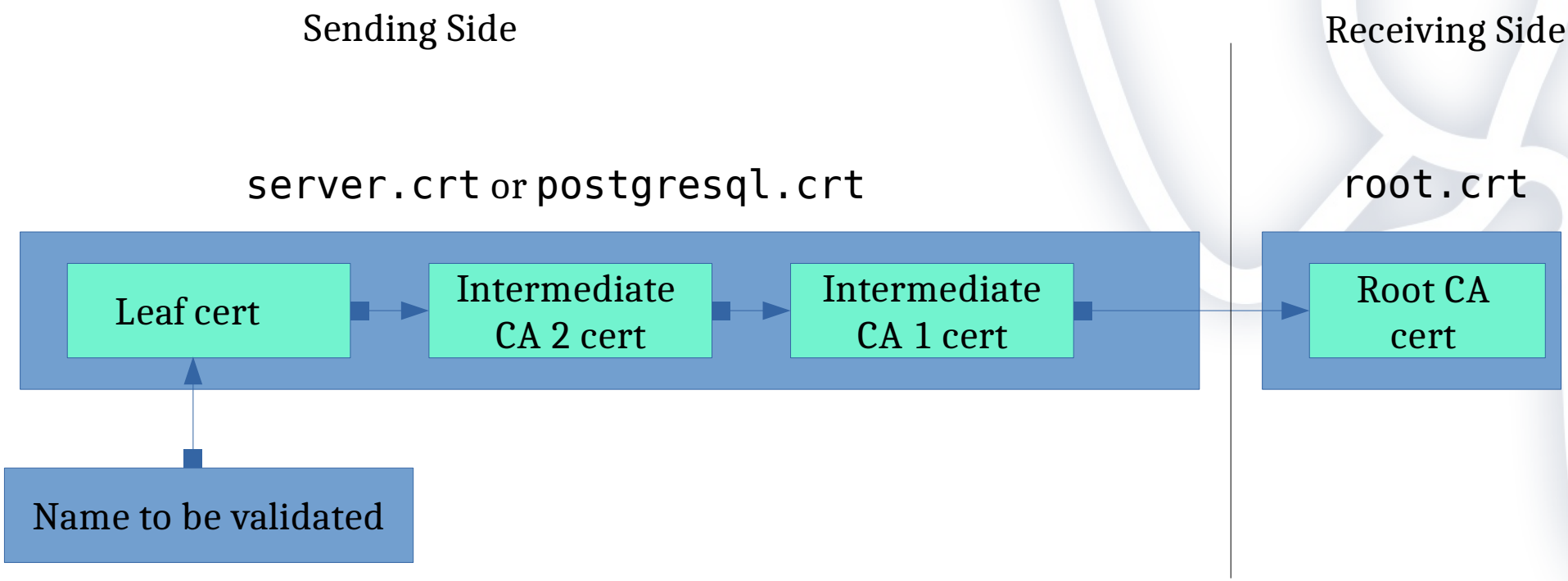
- In effect each signature delegates its authority to the intermediate CA whose certificate it is attached to.

Generating an intermediate CA

- ```
  openssl req -new -nodes -text -out intermediate.csr \
     -keyout intermediate.key -subj "$SUBJ/CN=My Intermediate CA 1"
  chmod og-rwx intermediate.key
  openssl x509 -req -in intermediate.csr -text -days 1825 -extfile openssl.cnf \
    -extensions v3_ca -CA cacert.pem -CAkey private/cakey.pem -CAcreateserial \
    -out intermediate.crt
  rm intermediate.csr
  ```
  ```

Validating a certificate signed by a non-root CA

- To validate a leaf certificate you need the whole chain of certificates back to the root CA certificate.



Create a simple server certificate

- Using your Root CA from above
- ```
  openssl req -new -days 365 -config cadir/openssl.cnf  -nodes -out server.req \
     -keyout server.key  -subj "$SUBJ/CN=foo.bar.com" > /dev/null 2>&1
  openssl ca -config cadir/openssl.cnf -in server.req -out server.crt \
    -cert cadir/cacert.pem -keyfile cadir/private/cakey.pem -batch
  chmod 600 server.key
  rm -f server.req
  ```
- Will ask for key for CA
  ```

Deploying a server certificate

- On server:
  - `mv server.key server.crt $PGDATA`
  - In postgresql.conf:
    - `ssl = on`
- Then
  - `pg_ctl restart`

Verifying a server certificate

- On client:
  - `mv cacert.pem ~/.postgresql/root.crt`
- or on Windows:
  - `move cacert.pem %APPDATA%\postgresql\root.crt`
- Connect with `sslmode=verify-ca` or `sslmode=verify-full`

pg_hba.conf settings for SSL

- no SSL on `local` connections
- `host` lines match both SSL and non-SSL connections
- `hostssl` lines only match SSL connections
- `hostnossl` lines only match non-SSL connections

Authentication methods for SSL

- any usual authentication method can be used, e.g. `scram-sha-256`
- `cert` method uses an SSL client certificate
  - only works with SSL
- other methods can use option `clientcert=1`
  - requires a trusted client certificate to be presented
    - Only works with SSL connections
  - a kind of Multi Factor Authentication
    - certificate/key is something you have
    - password is something you know

Multi-name certificates

- standard x.509 extension
  - rfc5280
- harder to create certificates
- allows you to deploy the same certificate on multiple hosts
- only applies to host names
  - not IP addresses
  - not user names
- supported by both libpq and jdbc

Multi-name certificate example

- hosts will be curly larry and mo
- ```
  cat > /tmp/san.cnf < -EOF
      [ req ]
      default_bits       = 2048
      distinguished_name = req_distinguished_name
      req_extensions     = req_ext
      [ req_distinguished_name ]
      countryName                 = Country Name (2 letter code)
      stateOrProvinceName         = State or Province Name (full name)
      localityName               = Locality Name (eg, city)
      organizationName           = Organization Name (eg, company)
      commonName                 = Common Name (e.g. server FQDN or YOUR name)
      [ req_ext ]
      subjectAltName = @alt_names
      [alt_names]
      DNS.1 = curly
      DNS.2 = larry
      DNS.3 = mo
      EOF
  openssl req -new -days 365 -config /tmp/san.cnf -nodes -out server.req \
      -keyout server.key -subj "$SUBJ/CN=many names" > /dev/null 2>&1
  ```
- Sign as before, CA must have copy_extensions enabled
  ```

CN is deprecated for host names

- Although in wide use, Subject CN fields have been deprecated for HTTPS servers since 2011 (see https://tools.ietf.org/html/rfc6125 Appendix B section 3.1.)
- At some stage in the future PostgreSQL might well follow suit.
- It's probably best to get into the habit of using SANs for host names,even though it's more cumbersome to generate.

- Some people recommend putting the most common host name in a CN field as well.
  - `libpq` ignores the CN if a SAN is present.

Generating client certificates (libpq)

- Using the CA from above:
- ```
  openssl req -new -days 365 -text  -nodes  -out client.req \
    -keyout client.key -subj "$SUBJ/CN=myuser" > /dev/null 2>&1
  openssl ca -config cadir/openssl.cnf -in client.req -out client.crt \
    -cert cadir/cacert.pem -keyfile cadir/private/cakey.pem -batch
  chmod 600 client.key
  rm -f client.req
  ```
  ```

Generating Client certificates (jdbc)

- Using the CA from above:
- ```
  openssl req -new -days 365 -text  -nodes  -out client.req \
     -keyout client.key -subj "$SUBJ/CN=myuser" > /dev/null 2>&1
  openssl pkcs8 -topk8 -inform PEM -in client.key -outform DER  \
     -passout pass: -out client.pk8
  openssl ca -config cadir/openssl.cnf -in client.req -out client.crt \
     -cert cadir/cacert.pem -keyfile cadir/private/cakey.pem -batch
  ```
- ```
  chmod 600 client.pk8
  ```
- ```
  rm -f client.req client.key
  ```
  ```

Deploying a client certificate (libpq)

- On client:
  - `mv client.crt ~/.postgresql/postgresql.crt`
  - `mv client.key ~/.postgresql/postgresql.key`
- or on Windows:
  - `move client.crt %APPDATA%\postgresql\postgresql.crt`
  - `move client.key %APPDATA%\postgresql\postgresql.key`

Deploying a client certificate (jdbc)

- On client:
  - `mv client.crt ~/.postgresql/postgresql.crt`
  - `mv client.pk8 ~/.postgresql/postgresql.pk8`
- or on Windows:
  - `move client.crt %APPDATA%\postgresql\postgresql.crt`
  - `move client.pk8 %APPDATA%\postgresql\postgresql.pk8`

Verifying client certificates

- On server:
  - `cp cacert.pem $PGDATA/root.crt`
- In postgresql.conf (default is blank):
  - `sslroot = 'root.crt'`
- Then:
  - `pg_ctl restart`

Password protecting keys

- With slightly different parameters, the commands above can generate keys that are encrypted with a password, which must be supplied when the key is used.
- server has `ssl_passphrase_command` setting that can supply it
- `jdbc` has `sslpassword` setting
- `libpq` doesn't currently have a setting
  - openssl libraries will prompt user
- patches to be released soon to support `sslpassword` in `libpq`
- See the sample scripts repo for examples of generating keys with passwords.

CRL files

- Certificate Revocation List file
- Sometimes we need to stop trusting certain certificates
  - Certificate compromise
    - Someone unauthorized got hold of the key
    - Key holder no longer trusted
  - CA compromise
    - More serious
    - Need to distrust all certificates signed by that CA
  - Various others
    - See rfc5280
  - CRLs are issued by CAs

PostgreSQL and CRL files

- Server:
  `ssl_crl_file = 'mylist.crl'`
- Default is blank, i.e. no file

- Client:
  `sslcrl="mylist.crl"`
- Default is `~/.postgresql/root.crl` (or on Windows `%APPDATA%\postgresql\root.crl`)
- Ignored if file does not exist

Pgbouncer and SSL

Settings:

- `client_tls_*` settings
  - for talking to clients where `pgbouncer` is acting as a server
  - requires a server certificate
- `server_tls_*` settings
  - for talking to the server where `pgbouncer` is acting as a client
  - requires a client certificate, if used
- Only provision for one certificate on each side.

pgbouncer settings on client side

- `client_tls_mode`
  - same setting names as for `libpq/jdbc`
  - same meanings more or less, except:
    - `allow` is the same as `prefer`
    - `verify-ca` is the same as `verify-full`
- `client_tls_cert_file`
- `client_tls_key_file`
- `client_tls_ca_file`
- `client_tls_ciphers`
  - default not necessarily the same as the server
- some others less important

pgbouncer settings on server side

- `server_tls_mode`
  - same setting names and meanings as for `libpq/jdbc`
- `server_tls_cert_file`
- `server_tls_key_file`
- `server_tls_ca_file`
- `server_tls_ciphers`
  - default not necessarily the same as the server

Dealing with multiple pgbouncer users

- You can only have one server_tls_certificate
- But you want to connect as many users
- Solution: use a map in `pg_ident.conf`
- Users.txt:

```
"curly" ""
"larry" ""
"mo" ""
```

-
```
# map name  sysusername  dbusername
bouncer     pgbouncer    larry
bouncer     pgbouncer    curly
bouncer     pgbouncer    mo
```

pgbouncer.ini

```
[databases]
* = host=dbhost port=5432
[pgbouncer]
logfile = ./pgbouncer.log
pidfile = ./pgbouncer.pid
listen_port = 6932
listen_addr = *
client_tls_sslmode = verify-full
client_tls_key_file = pgb_clnt.key
client_tls_cert_file = pgb_clnt.crt   # CN=bouncerhost.foo.com
client_tls_ca_file = root.crt
client_tls_ciphers = HIGH:MEDIUM:+3DES:!aNULL
client_tls_protocols = secure
server_tls_sslmode = verify-full
server_tls_ca_file = root.crt
server_tls_key_file = pgb_srvr.key
server_tls_cert_file = pgb_srvr.crt  # CN=pgbouncer
server_tls_protocols = tlsv1.2
server_tls_ciphers = HIGH:MEDIUM:+3DES:!aNULL
auth_type = cert
auth_file = users.txt
admin_users = postgres
```

2ndQuadrant PostgreSQL

Questions?

Andrew Dunstan
andrew.dunstan@2ndQuadrant.com