Secure Communication with TLS

BRUCE MOMJIAN

TLS/SSL forms the backbone of secure digital communication. This presentation explains how it works for websites and Postgres.

Creative Commons Attribution License

http://momjian.us/presentations

Last updated: January, 2018

Outline

1. Why you should care
2. Secure digital protocol
3. Authentication
4. Browser certificate usage
5. Postgres certificate usage
6. Conclusion

1. Why You Should Care

- Interactions are increasingly digital
- Attackers are increasingly distant
- Attackers are increasingly sophisticated
- New threats prompt new security requirements
- Security software must be regularly updated to be effective

This presentation shows only modern security practices.

What Cryptography Can Accomplish

Authenticity	Verify who is on the other end of the communication channel
Confidentiality	Only the other party can read the original messages
Integrity	No other party can change or add messages

Other features are often desired, e.g. non-repudiation.

This Talk Is Not Enough

Of course, secure digital communication is just one aspect of security. If done properly, attacks will happen somewhere else:

- Protocol attacks
- Trojan horses
- Viruses
- Electromagnetic monitoring (TEMPEST)
- Physical compromise
- Blackmail/intimidation of key holders
- Operating system bugs
- Application program bugs
- Hardware bugs
- User error
- Physical eavesdropping
- Social engineering
- Dumpster diving

(List from *Applied Cryptography,* Bruce Schneier, 1996) For an entertaining video about hardware exploits that bypass encryption, see *Crypto Won't Save You Either,* `https://youtu.be/_ahcUuNO4so`.

2. Secure Digital Protocol

A secure digital protocol needs a way to:

- Negotiate a secret session key in public, e.g. ECDHE
- Encrypt/decrypt a block of data using a key, e.g. AES256
- Encrypt/decrypt a variable-length message, e.g. GCM
- Verify the message was generated by someone who knows the key, e.g. GCM
- Identify that the other party is authentic, e.g. RSA
- Verify someone is not in the middle viewing, modifying, or adding messages, e.g. RSA

This is not a protocol that does one thing — it must do multiple things, and with resourceful and persistent attackers intermixed at all levels of the protocol.

TLS/SSL Protocol

- SSL (Secure Socket Layers) developed by Netscape in 1995
- TLS (Transport Layer Security) approved by the IETF (Internet Engineering Task Force) in 1999
- IETF deprecated SSL 2.0 in 2011 and SSL 3.0 (the last version) in 2015 due to discovered vulnerabilities
- TLS 1.2 is the most recent version (1.3 is in draft)

TLS protocol details and its interaction with certificates, key exchange, and attacks are covered in `https://security.stackexchange.com/questions/20803/how-does-ssl-tls-work`.

What *Not* To Do

- Do not encrypt the *message* using RSA
  - too slow to encrypt
  - messages can expose parts of the secret key
- Do not encrypt the *secret key* using RSA
  - post-session exposure of a persistent RSA private key would expose the message, i.e., does not allow forward secrecy
  - too slow to generate per-session RSA key pairs

Application of Cryptographic Methods

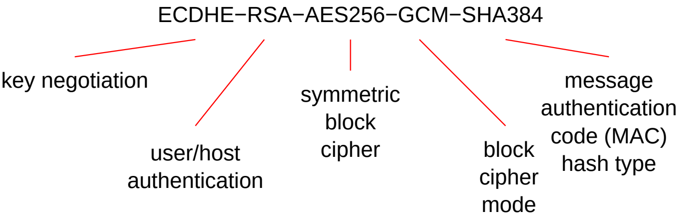
- Use ephemeral Diffie–Hellman (DHE or ECDHE) to negotiate a secret session key
- Use RSA for user authentication (more on this in the next section)
  - Server RSA-signs a *hash* of the current DHE or ECDHE exchange ($e.g.\ g^y \bmod p$) to:
    - proves identity (knowledge of the certificate's RSA private key)
    - prevents a man-in-the-middle from altering the server's DHE or ECDHE parameters
  - Client sends its DHE part ($e.g.\ g^x \bmod p$), optionally signed (authenticated) by a client certificate
- Session tickets allow the reuse of parameters from recent sessions
- TLS 1.3 (draft) will require DHE or ECDHE key negotiation, e.g. passing the key via RSA encryption will no longer be supported

Pieces of the Puzzle

- ECDHE for secret key exchange
- RSA for authenticating users and the secret key
- AES for confidentiality
- GCM and SHA for integrity of encrypted blocks

Modern TLS Connection

```
$ openssl s_client -connect momjian.us:443
…
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES256-GCM-SHA384
```



ECDHE−RSA−AES256−GCM−SHA384

key negotiation

user/host authentication

symmetric block cipher

block cipher mode

message authentication code (MAC) hash type

generated using OpenSSL 1.0.1t 3 May 2016

TLS in the Wild

ECDHE-RSA-AES256-GCM-SHA384	**ECDHE-RSA-AES128-GCM-SHA256**
`nsa.gov`	`google.com`
`whitehouse.gov`	`postgresql.org`
`microsoft.com`	`ibm.com`
`mindtv.org`	`twitter.com`
`mail.ru`	`oracle.com`
`www3.nhk.or.jp`	`verizon.com`

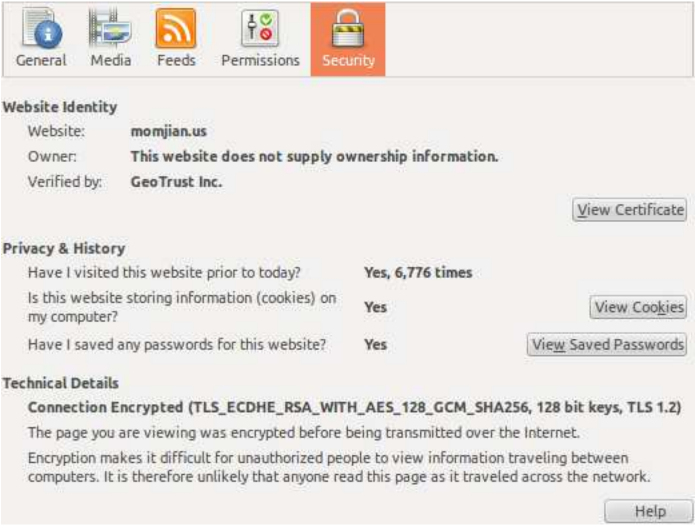
Tested using *openssl s_client -connect* hostname*:443*

as of 2017-05-02

Cryptographic Standards Are Evolving

- Some browsers do not prefer the strongest encryption settings, e.g. Firefox 52 prefers AES128 over AES256
- Chrome 58 prefers elliptic curve DSA
- Check your browser's preferred ciphers: `https://www.ssllabs.com/ssltest/viewMyClient.html`

Firefox Defaults for momjian.us


General
Media
Feeds
Permissions
Security

Website Identity

Website: **momjian.us**

Owner: **This website does not supply ownership information.**

Verified by: **GeoTrust Inc.**

View Certificate

Privacy & History

Have I visited this website prior to today? **Yes, 6,776 times**

Is this website storing information (cookies) on my computer? **Yes**

View Cookies

Have I saved any passwords for this website? **Yes**

View Saved Passwords

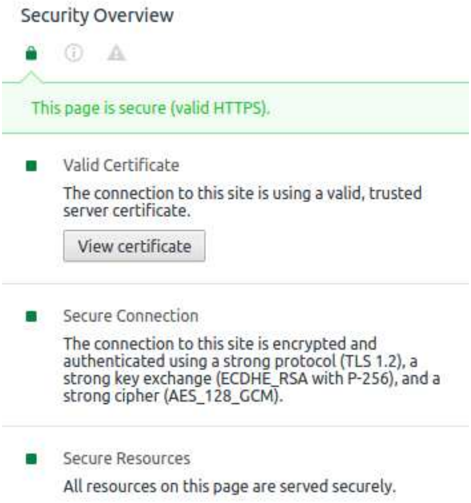
Technical Details

Connection Encrypted (TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256, 128 bit keys, TLS 1.2)

The page you are viewing was encrypted before being transmitted over the Internet.

Encryption makes it difficult for unauthorized people to view information traveling between computers. It is therefore unlikely that anyone read this page as it traveled across the network.

Chrome Defaults for momjian.us



Security Overview

This page is secure (valid HTTPS).

🟩 Valid Certificate

The connection to this site is using a valid, trusted server certificate.

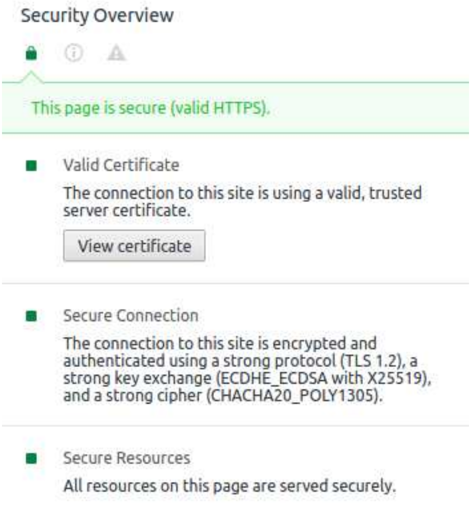
🟩 Secure Connection

The connection to this site is encrypted and authenticated using a strong protocol (TLS 1.2), a strong key exchange (ECDHE_RSA with P-256), and a strong cipher (AES_128_GCM).

🟩 Secure Resources

All resources on this page are served securely.

Chrome Defaults for mail.google.com



Security Overview

This page is secure (valid HTTPS).

🟩 Valid Certificate

The connection to this site is using a valid, trusted server certificate.

🟩 Secure Connection

The connection to this site is encrypted and authenticated using a strong protocol (TLS 1.2), a strong key exchange (ECDHE_ECDSA with X25519), and a strong cipher (CHACHA20_POLY1305).

🟩 Secure Resources

All resources on this page are served securely.

Elliptic Curves Are the Future

The following rows have comparable security:

	Bits			
Symmetric Cipher	**Elliptic Curve**	**RSA & DHE**	**Bit Ratio Cols 2 & 3**	**Computational Ratio**
80	160	1024	6	262
112	224	2048	9	84
128	256	3072	12	1728
192	384	7680	20	8000
256	521	15360	29	25625

`https://www.globalsign.com/en/blog/elliptic-curve-cryptography/`

For columns two and three, doubling the bit count increases computations by 8x (cubic). The chart's asymmetry is caused by non-brute-force attacks against linear finite-field algorithms with complexity less than $O(\sqrt{n})$ (e.g. index calculus) which do not apply to elliptic curves. Simplistically, this is because, in *linear* finite fields, it is easy to find valid integers, while for elliptic curves it is difficult to find valid points *(x,y)*; for ideas see `https://crypto.stackexchange.com/questions/8301/trying-to-better-understand-the-failure-of-the-index-calculus-for-ecdlp`.

Hello Postgres

```
$ psql "sslmode=require host=momjian.us dbname=postgres"
psql (10)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384,\
bits: 256, compression: off)
…
$ psql "dbname=… host=ec2-54-83-59-154.compute-1.amazonaws.com \
user=… password=… port=5432 sslmode=require" # Heroku
psql (10, server 9.4.10)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384,\
bits: 256, compression: off)
```

Postgres TLS Introspection

```
$ psql "sslmode=require host=momjian.us dbname=postgres"
psql (10)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384,\
bits: 256, compression: off)
CREATE EXTENSION sslinfo;

SELECT ssl_is_used(), ssl_version(), ssl_cipher();
 ssl_is_used | ssl_version |         ssl_cipher
-------------+-------------+-----------------------------
 t           | TLSv1.2     | ECDHE-RSA-AES256-GCM-SHA384
```

`sslinfo` also includes functions to query client certificates.

Postgres Fanciness

```
SELECT type, mode
FROM   unnest(string_to_array(ssl_cipher(), '-'),
        '{key exchange, user authentication, symmetric block cipher,
          block cipher mode,  MAC hash type}'::text[])
       AS f(type, mode);
  type  |          mode
--------+------------------------
 ECDHE  | key exchange
 RSA    | user authentication
 AES256 | symmetric block cipher
 GCM    | block cipher mode
 SHA384 | MAC hash type
```

3. Authentication

BIFURCATION
CORE
THE DELTA
IS WHERE
RIDGES

https://www.flickr.com/photos/93243105@N0

3.1 Certificates Creation: Their Purpose

So far, Diffie–Hellman (DHE and ECDHE) allow negotiation of a secret key in public, but who are you negotiating with?

- Is there someone impersonating your intended participant?
- Is there someone between you and your intended participant, passing through all the messages but:
  - viewing them?
  - modifying them?
  - creating fake messages?

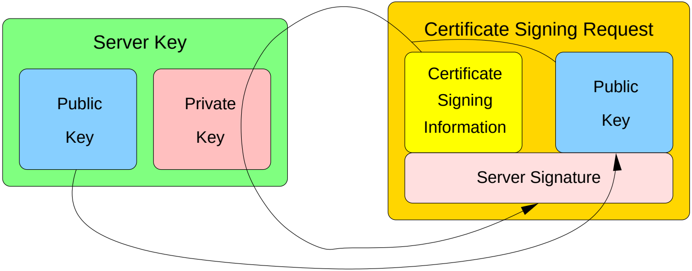
While the two parties could establish a secret privately that could be used later to prove identity, the more common method is to create an X.509 certificate, which is information RSA-signed by someone that both parties trust (a certificate authority).

X.509 Certificate Creation

Create an X.509 certificate that is signed by a trusted root certificate authority:

- Create a certificate signing request (CSR) by answering some questions that uniquely identify your website
- A public and private RSA key pair will also be generated
- Email the CSR (which includes the public RSA key) to a publicly-trusted certificate authority
- Receive an X.509 certificate that is RSA-signed by a trusted entity
- Install the trusted certificate in the web server

The RSA Key and Its Signed CSR



Server Key

Public Key

Private Key

Certificate Signing Request

Certificate Signing Information

Public Key

Server Signature

Generating a Certificate Signing Request (CSR)

```
$ openssl req -new -nodes -newkey rsa:2048 \
          -keyout momjian.us.key -out momjian.us.csr
Generating a 2048 bit RSA private key
...............................................................+++
......................................+++
writing new private key to 'momjian.us.key'
…fill in prompts
Country Name (2 letter code) [AU]:
…
$ ls
momjian.us.csr  momjian.us.key
$ openssl req -in momjian.us.csr -noout -text
Certificate Request:
        …
        Subject: C=US, ST=Pennsylvania, L=Newtown Square, \
                 O=Bruce Momjian, OU=website, \
                 CN=momjian.us/emailAddress=bruce@momjian.us
```

The Full CSR

```
$ openssl req -in momjian.us.csr -noout -text
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: C=US, ST=Pennsylvania, L=Newtown Square, O=Bruce Momjian,
                 OU=website, CN=momjian.us/emailAddress=bruce@momjian.us
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:cb:33:cf:07:bf:5a:61:08:47:50:d4:6f:18:d3:
                    …
                    82:a8:11:cb:e2:a1:1a:7d:cf:92:e4:43:59:6a:d3:
                    25:65
                Exponent: 65537 (0x10001)
        Attributes:
            a0:00
    Signature Algorithm: sha256WithRSAEncryption
         12:b1:21:09:b4:b4:3e:fe:2b:9f:ce:96:cd:98:17:80:d0:83:
         …
         0e:09:70:4b
```

The CSR-Generated RSA Key Pair

```
$ openssl rsa -in momjian.us.key -text -noout
Private-Key: (2048 bit)
modulus:
    00:cb:33:cf:07:bf:5a:61:08:47:50:d4:6f:18:d3:
    …
publicExponent: 65537 (0x10001)


privateExponent:
    00:a0:6c:7a:9a:47:3b:f7:37:2d:f6:66:80:7f:3d:
    …
prime1:
    00:ff:e9:0a:97:00:2b:36:ca:dc:35:b5:f5:18:e5:
    …
prime2:
    00:cb:46:09:ea:00:8b:20:36:e6:69:45:e1:e2:c7:
    …
exponent1:
    1c:65:57:6f:79:ed:51:9f:20:e0:34:d8:85:72:bd:
    …
exponent2:
    59:50:c0:f2:6c:a2:b4:d8:ea:8c:bf:03:ed:9d:53:
    …
coefficient:
    32:ee:3b:ef:87:e4:f3:e4:ba:2e:7d:0d:51:ab:97:
```

RSA Key Pair Internals

```
$ rsadump.sh momjian.us.key
key bits = 2047.6
n (pq) = 2565…
e = 65537

d (1/e mod lcm(p-1,q-1)) = 2025…
p = 1797…
q = 1427…

exp1 (dp, d mod (p-1)) = 1994…
exp2 (dq, d mod (q-1)) = 6271…
c (qinv, 1/q mod p) = 3576…
…
```

The final three fields are used to speed computations involving the private key. Script available at `http://momjian.us/main/writings/pgsql/rsadump.sh`, derived from `http://www.vidarholen.net/contents/junk/files/decode_rsa.bash`.

RSA Key Pair Internals

```
$ rsadump.sh -e momjian.us.key
key bits = 2047.6
n (pq) ˜= 2e616
e = 65537

d (1/e mod lcm(p-1,q-1)) ˜= 2e616
p ˜= 1e308
q ˜= 1e308

exp1 (dp, d mod (p-1)) ˜= 1e307
exp2 (dq, d mod (q-1)) ˜= 6e307
c (qinv, 1/q mod p) ˜= 3e307
…
```

RSA and X.509 Files

Key	contains the public and private keys
Pub	contains the public key
Csr	contains the public key and data to be signed
Crt	contains the public key and data and the signature of a certificate authority

TLS/SSL Acronyms

ASN1	Abstract syntax notation used to represent a hierarchical data structure
DER	Binary representation of an ASN1 structure, used by *openssl* and other TLS/SSL tools
PEM	Base64 encoding of DER data, with dashed armor before and after; typical file extensions: *pem, key, pub, csr, crt, crl*

PEM describes the storage format, not the contents, i.e., a PEM file can contain a public/private key pair, public key, certificate request, signed certificate (one or many (a chain)), or certificate revocation list.

Prove the CSR Was Signed by the RSA Private Key

- Study the ASN1 structure of the CSR
- Find the hash method, e.g. SHA256
- Hash the ASN1 section containing the user-supplied parameters
- Extract the signed stored hash in the CSR
- Use the RSA public key to reverse the signature to produce the stored hash
- Compare the computed hash with the stored hash

The ASN1 CSR Structure

```
$ oopenssl req -in momjian.us.csr -outform DER |
> openssl asn1parse -i -inform DER
    …
    0:d=0  hl=4 l= 739 cons: SEQUENCE
    4:d=1  hl=4 l= 459 cons:  SEQUENCE
    8:d=2  hl=2 l=   1 prim:   INTEGER           :00
   11:d=2  hl=3 l= 157 cons:   SEQUENCE
  …
  117:d=3  hl=2 l=  19 cons:    SET
  119:d=4  hl=2 l=  17 cons:     SEQUENCE
  121:d=5  hl=2 l=   3 prim:      OBJECT            :commonName
  126:d=5  hl=2 l=  10 prim:      UTF8STRING        :momjian.us
  …
  171:d=2  hl=4 l= 290 cons:   SEQUENCE
  175:d=3  hl=2 l=  13 cons:    SEQUENCE
  177:d=4  hl=2 l=   9 prim:     OBJECT            :rsaEncryption
  188:d=4  hl=2 l=   0 prim:     NULL
  190:d=3  hl=4 l= 271 prim:    BIT STRING
  465:d=2  hl=2 l=   0 cons:   cont [ 0 ]
  467:d=1  hl=2 l=  13 cons:  SEQUENCE
  469:d=2  hl=2 l=   9 prim:   OBJECT            :sha256WithRSAEncryption
  480:d=2  hl=2 l=   0 prim:   NULL
  482:d=1  hl=4 l= 257 prim:  BIT STRING
```

Hash the User-Supplied-Parameter Section

```
$ # openssl asn1parse can't process text before
$ # the PEM armor so convert it to DER first.
$ openssl req -in momjian.us.csr -outform DER |
> openssl asn1parse -i -inform DER -strparse 4 \
          -out /dev/stdout -noout |
> openssl dgst -sha256 -binary |
> xxd -p -c 999
f405afa2d4b242ecb320071f37ba2e00b249b7fd05f91db7bc35882380e2c25e
```

Reverse the Signed Hash

```
$ openssl req -in momjian.us.csr -outform DER |
> openssl asn1parse -i -inform DER -strparse 482 \
          -out /dev/stdout -noout |
> openssl pkeyutl -verifyrecover -inkey momjian.us.key |
> openssl asn1parse -i -inform DER
    0:d=0  hl=2 l=  49 cons: SEQUENCE
    2:d=1  hl=2 l=  13 cons:  SEQUENCE
    4:d=2  hl=2 l=   9 prim:   OBJECT            :sha256
   15:d=2  hl=2 l=   0 prim:   NULL
   17:d=1  hl=2 l=  32 prim:  OCTET STRING      [HEX DUMP]: \
F405AFA2D4B242ECB320071F37BA2E00B249B7FD05F91DB7BC35882380E2C25E
```

Extract the Reverse Signed Hash and Compare

```
$ openssl req -in momjian.us.csr -outform DER |
> openssl asn1parse -i -inform DER -strparse 482 \
          -out /dev/stdout -noout |
> openssl pkeyutl -verifyrecover -inkey momjian.us.key |
> openssl asn1parse -i -inform DER -strparse 17 \
          -out /dev/stdout -noout |
> xxd -p -c 999
f405afa2d4b242ecb320071f37ba2e00b249b7fd05f91db7bc35882380e2c25e

$ # The hash we computed earlier on the user-supplied params
$ openssl req -in momjian.us.csr -outform DER |
> openssl asn1parse -i -inform DER -strparse 4 \
          -out /dev/stdout -noout |
> openssl dgst -sha256 -binary |
> xxd -p -c 999
f405afa2d4b242ecb320071f37ba2e00b249b7fd05f91db7bc35882380e2c25e

$ openssl req -in momjian.us.csr -verify -key momjian.us.key -noout
verify OK
```

Website Certificate (Signed CSR)

```
$ openssl x509 -in momjian.us.pem -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            23:30:40:e9:4c:3e:b1:63:4e:15:2b:1a:e2:00:a1:01
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=GeoTrust Inc., CN=RapidSSL SHA256 CA
        Validity
            Not Before: Mar  3 00:00:00 2016 GMT
            Not After : May  2 23:59:59 2018 GMT
        Subject: CN=momjian.us
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
        …
```

3.2 Certificate Verification: Manually Verify Each Certificate Link

```
$ openssl x509 -in momjian.us.pem -subject -issuer -noout
subject= /CN=momjian.us
issuer= /C=US/O=GeoTrust Inc./CN=RapidSSL SHA256 CA

$ openssl x509 -in RapidSSL_SHA256_CA.pem -subject -issuer -noout
subject= /C=US/O=GeoTrust Inc./CN=RapidSSL SHA256 CA
issuer= /C=US/O=GeoTrust Inc./CN=GeoTrust Global CA

$ openssl x509 -in /etc/ssl/certs/GeoTrust_Global_CA.pem \
          -subject -issuer -noout
subject= /C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
issuer= /C=US/O=GeoTrust Inc./CN=GeoTrust Global CA
```

Verify the Certificate Chain to the Root CA

```
$ # momjian.us.pem needs two certificates to reach the root CA
$ # so we concatenate them together.  -CApath /dev/null prevents
$ # the local certificate store from being used.
$ cat /etc/ssl/certs/GeoTrust_Global_CA.pem RapidSSL_SHA256_CA.pem |
> openssl verify -CApath /dev/null -CAfile /dev/stdin momjian.us.pem
momjian.us.pem: OK

$ openssl verify -CApath /dev/null -CAfile \
          /etc/ssl/certs/GeoTrust_Global_CA.pem RapidSSL_SHA256_CA.pem
RapidSSL_SHA256_CA.pem: OK

$ openssl verify -CApath /dev/null -CAfile \
          /etc/ssl/certs/GeoTrust_Global_CA.pem \
          /etc/ssl/certs/GeoTrust_Global_CA.pem
/etc/ssl/certs/GeoTrust_Global_CA.pem: OK
```

Experimenting With Verification

```
$ # Certificate order doesn't matter in this case.
$ cat RapidSSL_SHA256_CA.pem /etc/ssl/certs/GeoTrust_Global_CA.pem |
> openssl verify -CApath /dev/null -CAfile /dev/stdin momjian.us.pem
momjian.us.pem: OK

$ # momjian.us can't verify itself
$ openssl verify -CApath /dev/null -CAfile momjian.us.pem momjian.us.pem
momjian.us.pem: CN = momjian.us
error 20 at 0 depth lookup:unable to get local issuer certificate

$ # error and success with a root certificate
$ openssl verify -CApath /dev/null /etc/ssl/certs/GeoTrust_Global_CA.pem
/etc/ssl/certs/GeoTrust_Global_CA.pem: C = US, O = GeoTrust Inc., \
CN = GeoTrust Global CA
error 18 at 0 depth lookup:self signed certificate
OK
```

Rules of Certificate Verification, Part 1

These rules apply to public and private CA verification:

- Receive the first remote certificate; assume the remote server has its private key
- Receive any additional remote certificates
  - these should be intermediate certificates that create a chain toward a root certificate
  - traverse up the chain as far as possible
    - find a certificate whose subject name matches the current certificate issuer's subject name, e.g. if the certificate issuer is "RapidSSL SHA256 CA", find a certificate with that subject name
    - verify that the issuer's public key can decrypt the current certificate's signature and matches the hash of the certificate body
    - continue until no more matches or a root certificate is found

Rules of Certificate Verification, Part 2

- Using the top-most certificate found, look in the local certificate store for similar matches
    - if a root certificate was already received, check that is also exists in the local certificate store
    - if not, using the previous instructions, continue until a local-certificate-store root certificate is found, or fail

For more details, see the “Verify Operation” section of the `verify` manual page.

Installing Certificates in Apache 2.4

- Place the signed server/leaf certificate in a new file
- Append any intermediate certificates that should be sent to the client to help the traverse to a locally-stored root certificate, in reverse-signing order (for compatibility)
- You can place the new file in any directory accessible by Apache, or in `/etc/ssl/certs` (or where specified by the Apache `SSLCACertificatePath` directive)
- Record the file path in `/etc/apache2/sites-enabled/000-default-ssl.conf` using the `SSLCertificateFile` directive

Introspecting a Certificate Bundle

It is possible to show all the certificates in a *certificate bundle:*

```
$ # There is no value in including the root certificate
$ # because it must exist on the remote side for success.
$ cat RapidSSL_SHA256_CA.pem momjian.us.pem \
      > /etc/ssl/certs/momjian.us.bundle.pem

$ openssl crl2pkcs7 -nocrl -certfile momjian.us.bundle.pem |
> openssl pkcs7 -print_certs -text -noout | grep 'Subject:'
        Subject: CN=momjian.us
        Subject: C=US, O=GeoTrust Inc., CN=RapidSSL SHA256 CA
```

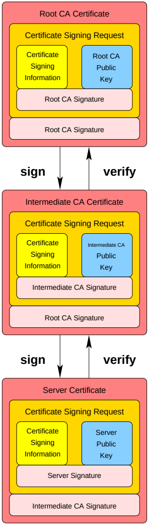
3.3 Certificate Authority Creation

To create a local root certificate authority and a chain underneath it:

- Create a certificate signed by its own key pair (the root CA)
- Create an intermediate CA whose certificate is signed by the root CA's private key
- Create a server certificate signed by the intermediate CA

Details about certificate creation and management can be found at `https://jamielinux.com/docs/openssl-certificate-authority/introduction.html`.

Root CA, Intermediate CA, and Server Certificate


Root CA Certificate
Certificate Signing Request
Certificate Signing Information
Root CA Public Key
Root CA Signature
Root CA Signature
sign
verify
Intermediate CA Certificate
Certificate Signing Request
Certificate Signing Information
Intermediate CA Public Key
Intermediate CA Signature
Root CA Signature
sign
verify
Server Certificate
Certificate Signing Request
Certificate Signing Information
Server Public Key
Server Signature
Intermediate CA Signature

Create the Root CA

```
$ openssl req -new -nodes -subj "/CN=CA-root" -text \
          -keyout CA-root.key > CA-root.csr

$ # -extfile needed to enable v3_ca
$ openssl x509 -req -in CA-root.csr -text -days 3650 \
          -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
          -signkey CA-root.key > CA-root.crt
```

Certificate Marked as a CA

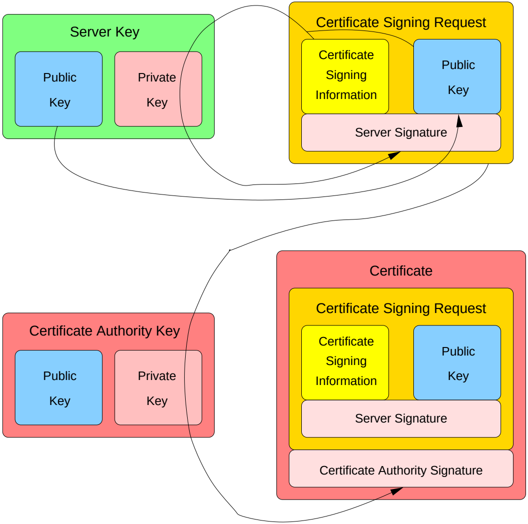
```
$  openssl x509 -in CA-root.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            9e:40:f6:e3:7c:e8:94:ec
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=CA-root
        …
            X509v3 Basic Constraints:
                CA:TRUE
```

Create the Intermediate CA

```
$ openssl req -new -nodes -subj "/CN=CA-intermediate" -text \
          -keyout CA-intermediate.key > CA-intermediate.csr

$ openssl x509 -req -in CA-intermediate.csr -text -days 1825 \
        -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
        -CA CA-root.crt -CAkey CA-root.key -CAcreateserial  \
         > CA-intermediate.crt
```

CA Signing the CSR


Server Key
Public Key
Private Key
Certificate Signing Request
Certificate Signing Information
Public Key
Server Signature
Certificate Authority Key
Public Key
Private Key
Certificate
Certificate Signing Request
Certificate Signing Information
Public Key
Server Signature
Certificate Authority Signature

Proving the Certificate Signature

```
$ # Get offsets of user-supplied parameter section
$ openssl x509 -in CA-intermediate.crt -outform DER |
> openssl asn1parse -i -inform DER
    0:d=0  hl=4 l= 767 cons: SEQUENCE
    4:d=1  hl=4 l= 487 cons:  SEQUENCE
   …
   39:d=2  hl=2 l=  18 cons:   SEQUENCE
   41:d=3  hl=2 l=  16 cons:    SET
   43:d=4  hl=2 l=  14 cons:     SEQUENCE
   45:d=5  hl=2 l=   3 prim:      OBJECT            :commonName
   50:d=5  hl=2 l=   7 prim:      UTF8STRING        :CA-root
   …
   91:d=2  hl=2 l=  26 cons:   SEQUENCE
   93:d=3  hl=2 l=  24 cons:    SET
   95:d=4  hl=2 l=  22 cons:     SEQUENCE
   97:d=5  hl=2 l=   3 prim:      OBJECT            :commonName
  102:d=5  hl=2 l=  15 prim:      UTF8STRING        :CA-intermediate
  …
  495:d=1  hl=2 l=  13 cons:  SEQUENCE
  497:d=2  hl=2 l=   9 prim:   OBJECT            :sha256WithRSAEncryption
  508:d=2  hl=2 l=   0 prim:   NULL
  510:d=1  hl=4 l= 257 prim:  BIT STRING
```

Hash the User-Supplied-Parameter Section

```
$ openssl x509 -in CA-intermediate.crt -outform DER |
> openssl asn1parse -i -inform DER -strparse 4 \
          -out /dev/stdout -noout |
> openssl dgst -sha256 -binary |
> xxd -p -c 999
9b7a02bdaff1412c5843b812255cbd023b1c6f6ae1ac3b93cd3d5fd001e0bc21
```

Reverse the Signed Hash

```
$ openssl x509 -in CA-intermediate.crt -outform DER |
> openssl asn1parse -i -inform DER -strparse 510 \
          -out /dev/stdout -noout |
> openssl pkeyutl -verifyrecover -inkey CA-root.key |
> openssl asn1parse -i -inform DER
    0:d=0  hl=2 l=  49 cons: SEQUENCE
    2:d=1  hl=2 l=  13 cons:  SEQUENCE
    4:d=2  hl=2 l=   9 prim:   OBJECT            :sha256
   15:d=2  hl=2 l=   0 prim:   NULL
   17:d=1  hl=2 l=  32 prim:  OCTET STRING      [HEX DUMP]: \
9B7A02BDAFF1412C5843B812255CBD023B1C6F6AE1AC3B93CD3D5FD001E0BC21
```

Extract the Reverse Signed Hash and Compare

```
$ openssl x509 -in CA-intermediate.crt -outform DER |
> openssl asn1parse -i -inform DER -strparse 510 \
          -out /dev/stdout -noout |
> openssl pkeyutl -verifyrecover -inkey CA-root.key |
> openssl asn1parse -i -inform DER -strparse 17 \
          -out /dev/stdout -noout |
> xxd -p -c 999
9b7a02bdaff1412c5843b812255cbd023b1c6f6ae1ac3b93cd3d5fd001e0bc21
```

```
$ # The hash we computed earlier on the user-supplied params
$ openssl x509 -in CA-intermediate.crt -outform DER |
> openssl asn1parse -i -inform DER -strparse 4 \
          -out /dev/stdout -noout |
> openssl dgst -sha256 -binary |
> xxd -p -c 999
9b7a02bdaff1412c5843b812255cbd023b1c6f6ae1ac3b93cd3d5fd001e0bc21
```

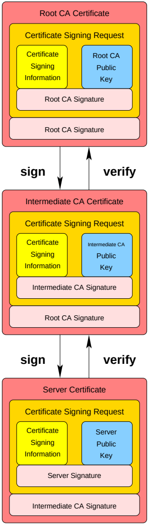
```
$ openssl verify -CAfile CA-root.crt CA-intermediate.crt
CA-intermediate.crt: OK
```

Create a Server/Leaf Certificate

```
$ openssl req -new -nodes -subj "/CN=$(hostname)" \
          -text -keyout server.key > server.csr

$ # This is an leaf so don't use the "v3_ca" extension.
$ openssl x509 -req -in server.csr -text -days 913 \
        -CA CA-intermediate.crt -CAkey CA-intermediate.key \
        -CAcreateserial > server.crt
```

3.4 Certificate Hierarchies: Our Simplistic Configuration



Root CA Certificate

Certificate Signing Request

Certificate Signing Information

Root CA Public Key

Root CA Signature

Root CA Signature

sign **verify**

Intermediate CA Certificate

Certificate Signing Request

Certificate Signing Information

Intermediate CA Public Key

Intermediate CA Signature

Root CA Signature

sign **verify**

Server Certificate

Certificate Signing Request

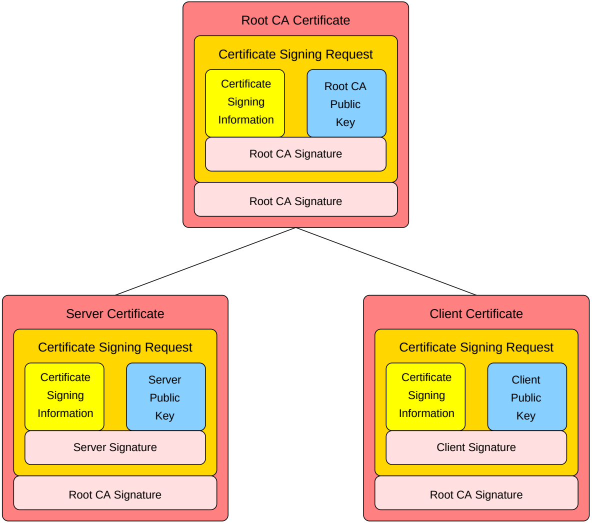
Certificate Signing Information

Server Public Key

Server Signature

Intermediate CA Signature

Root Signs Server and Client Certificates



Root CA Certificate

Certificate Signing Request

Certificate Signing Information

Root CA Public Key

Root CA Signature

Root CA Signature

Server Certificate

Certificate Signing Request

Certificate Signing Information

Server Public Key

Server Signature

Root CA Signature

Client Certificate

Certificate Signing Request

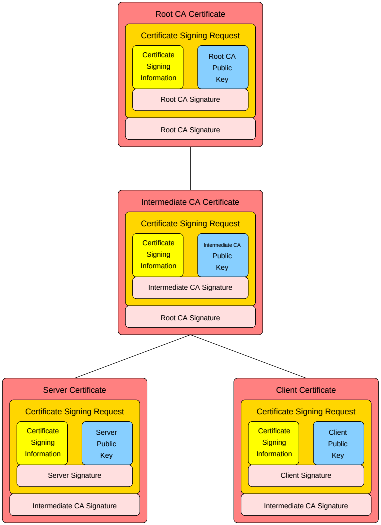
Certificate Signing Information

Client Public Key

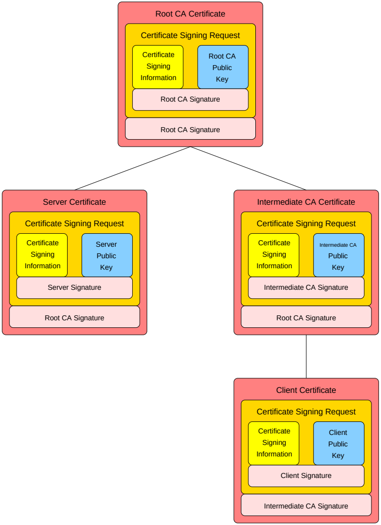
Client Signature

Root CA Signature

Intermediate Signs Server and Client Certificates


Root CA Certificate
Certificate Signing Request
Certificate Signing Information
Root CA Public Key
Root CA Signature
Root CA Signature
Intermediate CA Certificate
Certificate Signing Request
Certificate Signing Information
Intermediate CA Public Key
Intermediate CA Signature
Root CA Signature
Server Certificate
Certificate Signing Request
Certificate Signing Information
Server Public Key
Server Signature
Intermediate CA Signature
Client Certificate
Certificate Signing Request
Certificate Signing Information
Client Public Key
Client Signature
Intermediate CA Signature

Intermediate Signs Client



Root CA Certificate

Certificate Signing Request

Certificate Signing Information

Root CA Public Key

Root CA Signature

Root CA Signature

Server Certificate

Certificate Signing Request

Certificate Signing Information

Server Public Key

Server Signature

Root CA Signature

Intermediate CA Certificate

Certificate Signing Request

Certificate Signing Information

Intermediate CA Public Key

Intermediate CA Signature

Root CA Signature

Client Certificate

Certificate Signing Request

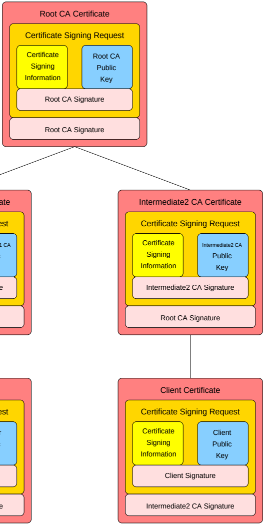
Certificate Signing Information

Client Public Key

Client Signature

Intermediate CA Signature

Separate Server/Client Intermediates


Root CA Certificate
Certificate Signing Request
Certificate Signing Information
Root CA Public Key
Root CA Signature
Root CA Signature
Intermediate1 CA Certificate
Certificate Signing Request
Certificate Signing Information
Intermediate1 CA Public Key
Intermediate1 CA Signature
Root CA Signature
Intermediate2 CA Certificate
Certificate Signing Request
Certificate Signing Information
Intermediate2 CA Public Key
Intermediate2 CA Signature
Root CA Signature
Server Certificate
Certificate Signing Request
Certificate Signing Information
Server Public Key
Server Signature
Intermediate1 CA Signature
Client Certificate
Certificate Signing Request
Certificate Signing Information
Client Public Key
Client Signature
Intermediate2 CA Signature

3.5 Certificate Tips with Your Own Root CA

- Create a root certificate (root CA) signed by its own password-protected key pair
- Create an intermediate CA whose certificate is signed by the root CA's private key
- Transfer the root CA's private key to offline storage
- Use the intermediate certificate to sign leaf certificates
- Append the intermediate certificate when sending the leaf certificate
- Use the root certificate when validating remote certificates
- Because remotes verify using only a root certificate, intermediate and leaf certificates can be replaced incrementally
- For more details see `https://github.com/ssllabs/research/wiki/SSL-and-TLS-Deployment-Best-Practices`

3.6 Certificate Revocation List (CRL)

A certificate revocation list (CRL) records non-expired certificates that should no longer be trusted due to:

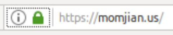
- Unauthorized exposure of a certificate's private key (certificates are public)
- Device containing the certificate's private key was or is not under trusted control
- Departure of staff that had access to the certificate's private key

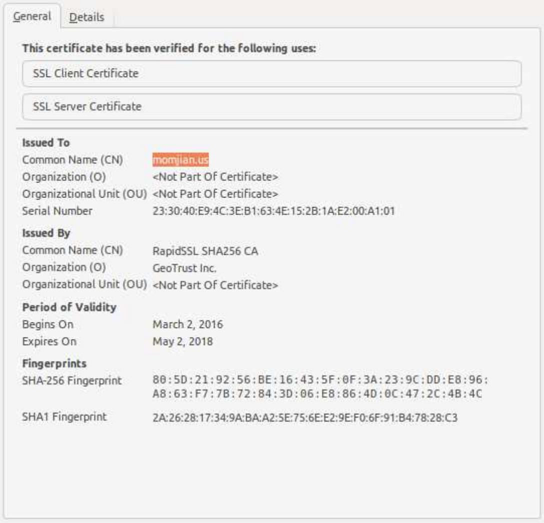
While not required for initial security, it is best to configure CRL files and their distribution method during certificate installation (even if the CRLs are empty) so they are ready when needed.

4. Browser Certificate Usage

https://www.flickr.com/photos/12227067@N02/

The *momjian.us* TLS Certificate





This certificate has been verified for the following uses:

SSL Client Certificate

SSL Server Certificate

Issued To

Common Name (CN)	momjian.us
Organization (O)	<Not Part Of Certificate>
Organizational Unit (OU)	<Not Part Of Certificate>
Serial Number	23:30:40:E9:4C:3E:B1:63:4E:15:2B:1A:E2:00:A1:01

Issued By

Common Name (CN)	RapidSSL SHA256 CA
Organization (O)	GeoTrust Inc.
Organizational Unit (OU)	<Not Part Of Certificate>

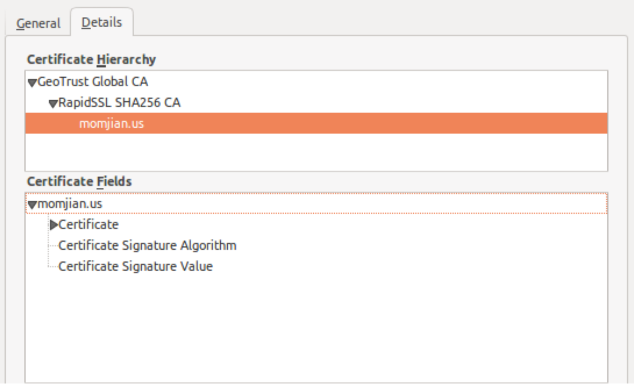
Period of Validity

Begins On	March 2, 2016
Expires On	May 2, 2018

Fingerprints

SHA-256 Fingerprint	80:5D:21:92:56:BE:16:43:5F:0F:3A:23:9C:DD:E8:96: A8:63:F7:7B:72:84:3D:06:E8:86:4D:0C:47:2C:4B:4C
SHA1 Fingerprint	2A:26:28:17:34:9A:BA:A2:5E:75:6E:E2:9E:F0:6F:91:B4:78:28:C3

Certificate Chain to a Trusted Certificate Authority



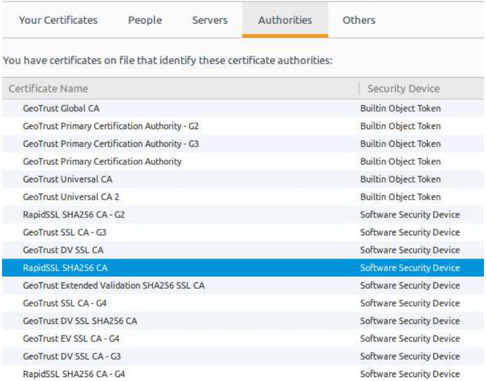
Certificate Hierarchy

- GeoTrust Global CA
  - RapidSSL SHA256 CA
    - momjian.us

Certificate Fields

- momjian.us
  - Certificate
  - Certificate Signature Algorithm
  - Certificate Signature Value

Trusted Certificates



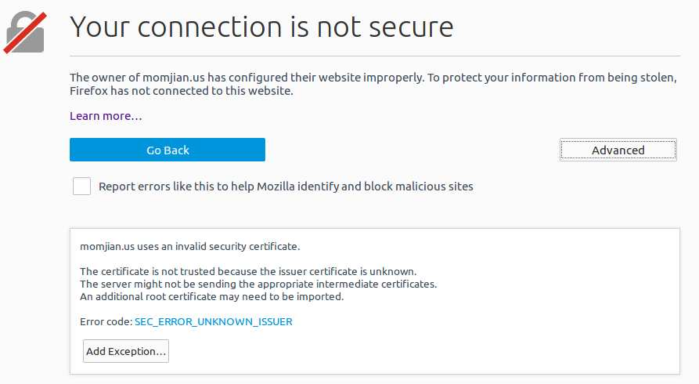
You have certificates on file that identify these certificate authorities:

Certificate Name	Security Device
GeoTrust Global CA	Builtin Object Token
GeoTrust Primary Certification Authority - G2	Builtin Object Token
GeoTrust Primary Certification Authority - G3	Builtin Object Token
GeoTrust Primary Certification Authority	Builtin Object Token
GeoTrust Universal CA	Builtin Object Token
GeoTrust Universal CA 2	Builtin Object Token
RapidSSL SHA256 CA - G2	Software Security Device
GeoTrust SSL CA - G3	Software Security Device
GeoTrust DV SSL CA	Software Security Device
RapidSSL SHA256 CA	Software Security Device
GeoTrust Extended Validation SHA256 SSL CA	Software Security Device
GeoTrust SSL CA - G4	Software Security Device
GeoTrust DV SSL SHA256 CA	Software Security Device
GeoTrust EV SSL CA - G4	Software Security Device
GeoTrust DV SSL CA - G3	Software Security Device
RapidSSL SHA256 CA - G4	Software Security Device

Firefox trusted certificate authorities are at `https://ccadb-public.secure.force.com/mozilla/CACertificatesInFirefoxReport`.

Chrome does not include RapidSSL certificates so, for compatibility, the *momjian.us* web server must send the RapidSSL intermediate certificate to clients.

Removal of *Geotrust Global CA* Prevents Certificate Validation



Your connection is not secure

The owner of momjian.us has configured their website improperly. To protect your information from being stolen, Firefox has not connected to this website.

Learn more…

☐ Report errors like this to help Mozilla identify and block malicious sites

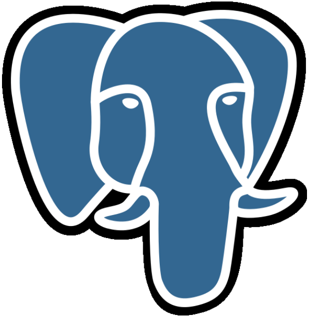
momjian.us uses an invalid security certificate.

The certificate is not trusted because the issuer certificate is unknown.
The server might not be sending the appropriate intermediate certificates.
An additional root certificate may need to be imported.

Error code: SEC_ERROR_UNKNOWN_ISSUER

News report of a root certificate being distrusted: https://www.bleepingcomputer.com/news/security/google-outlines-ssl-apocalypse-for-symantec-certificates/

5. Postgres Certificate Usage



Getting the Postgres Server Certificate

```
$ # https://github.com/thusoy/postgres-mitm (by Tarjei Husøy,uses Python)
$ wget https://github.com/thusoy/postgres-mitm/archive/master.zip
$ unzip master.zip
$ cd postgres-mitm-master/
$ # This returns the server certificate, not the entire chain.
$ postgres_get_server_cert.py $(hostname) | openssl  x509 -text -noout
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            90:b1:e0:3f:72:fb:65:8c
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=CA-intermediate
        Validity
            Not Before: May  3 01:47:21 2017 GMT
            Not After : May  1 01:47:21 2027 GMT
        Subject: CN=momjian.us
        …
```

`openssl s_client` cannot be used because Postgres requires custom protocol messages before switching to SSL mode.

Enabling Certificate Authentication

To verify server authenticity, the Postgres client must enable checking of the server certificate by using connection options `sslmode=verify-ca` or `sslmode=verify-full`. The later verifies the server certificate's subject name matches the server's host name. For example, this authenticates the server certificate (but not the subject name):

```
psql "sslmode=verify-ca host=localhost dbname=postgres"
```

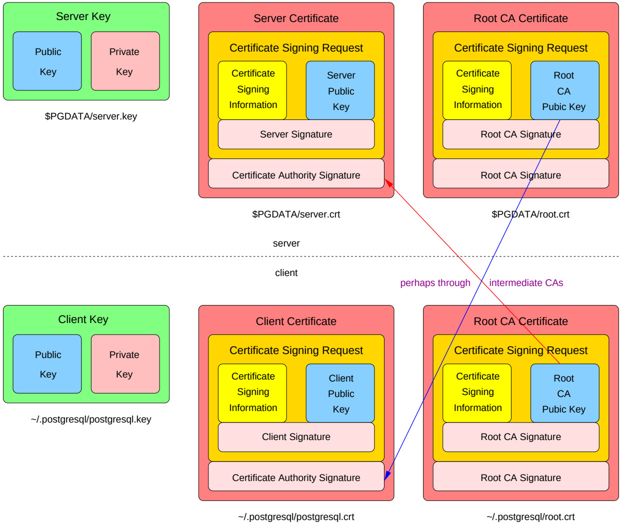
I specified the host name (`localhost`) because SSL is not supported on Unix domain sockets, which is the default connection type on Unix-like platforms.

Private Root CAs

Postgres usually uses private CAs because servers and clients are typically part of the same organization. (Using a public CA provides little value and adds another party that must be trusted.) Therefore, to enable authentication, the Postgres client must store a trusted root certificate, e.g:

```
$ cp root.crt ~postgres/.postgresql/root.crt
```

Certificate Validation


Server Key
Public Key
Private Key
$PGDATA/server.key
Server Certificate
Certificate Signing Request
Certificate Signing Information
Server Public Key
Server Signature
Certificate Authority Signature
$PGDATA/server.crt
Root CA Certificate
Certificate Signing Request
Certificate Signing Information
Root CA Pubic Key
Root CA Signature
Root CA Signature
$PGDATA/root.crt
server
client
perhaps through
intermediate CAs
Client Key
Public Key
Private Key
~/.postgresql/postgresql.key
Client Certificate
Certificate Signing Request
Certificate Signing Information
Client Public Key
Client Signature
Certificate Authority Signature
~/.postgresql/postgresql.crt
Root CA Certificate
Certificate Signing Request
Certificate Signing Information
Root CA Pubic Key
Root CA Signature
Root CA Signature
~/.postgresql/root.crt

Server Cert. with Intermediate and Leaf

Using the previously-created certificates and using:

```
$ cat server.crt CA-intermediate.crt > $PGDATA/server.crt
```

These client `root.crt` contents allow connections:

- `CA-root.crt` (the server provides the intermediate and leaf)
- `CA-intermediate.crt CA-root.crt` (the client intermediate is ignored)
- `CA-root.crt CA-intermediate.crt` (the order of certificates in `root.crt` doesn't matter)
- `CA-root.crt CA-intermediate.crt server.crt` (extra client certificates are ignored)

and these fail because no root certificate is stored on the client:

- `CA-intermediate.crt`
- `server.crt`
- `server.crt CA-intermediate.crt`
- `CA-intermediate.crt server.crt`

Server Certificate With Only Leaf

If we store only the server certificate on the server:

```
$ cat server.crt > $PGDATA/server.crt
```

These client `root.crt` contents allow connections:

- `CA-intermediate.crt CA-root.crt` (the client must supply the intermediate)
- `CA-root.crt CA-intermediate.crt` (order doesn't matter here)
- `CA-root.crt CA-intermediate.crt server.crt` (the client's `server.crt` s is ignored)

and these fail because the root or intermediate certificates are missing:

- `CA-root.crt`
- `CA-intermediate.crt`
- `server.crt`
- `server.crt CA-intermediate.crt`
- `CA-intermediate.crt server.crt`

The First Certificate in `server.crt` Is Special

If we store the intermediate certificate first on the server:

```
$ cat CA-intermediate.crt server.crt > $PGDATA/server.crt
```

the server will not start:

```
FATAL:  could not load private key file "server.key":
        key values mismatch
```

The first certificate in `server.crt` must match the keys in `$PGDATA/server.key`. Additional stored certificates should be appended in reverse-signing order (for compatibility).

Certificate Revocation List (CRL) Testing

```
$ cp CA-root.crt ˜postgres/.postgresql/root.crl
$ sql "sslmode=verify-ca host=$(hostname) dbname=postgres"
psql: SSL error: certificate verify failed

$ cp CA-intermediate.crt ˜postgres/.postgresql/root.crl
$ sql "sslmode=verify-ca host=$(hostname) dbname=postgres"
psql: SSL error: certificate verify failed

$ cp server.crt ˜postgres/.postgresql/root.crl
$ sql "sslmode=verify-ca host=$(hostname) dbname=postgres"
psql: SSL error: certificate verify failed
```

Multiple revoked certificates can be appended to the file.

Client Certificates

Placing certificates on Postgres clients has advantages:

- Servers can validate client certificates by:
  - installing certificates on clients that are signed by a certificate authority the server trusts
  - installing trusted root certificates on the server by specifying `ssl_ca_file` in `$PGDATA/postgresql.conf`
  - adding `clientcert=1` to `hostssl` lines in `pg_hba.conf`
- Clients can even authenticate database user names to servers by setting the certificate subject name

6. Conclusion: What Cryptography Can Accomplish

Authenticity	Verify who is on the other end of the communication channel (X.509, RSA)
Confidentiality	Only the other party can read the original messages (ECDHE, AES)
Integrity	No other party can change or add messages (GCM, SHA)

Conclusion

http://momjian.us/presentations

https://www.flickr.com/photos/thevlue/