Using Cryptographic Hardware to Secure Applications

Bruce Momjian

This presentation explains how to use cryptographic hardware in client applications.

https://momjian.us/presentations

Last updated: September 2023

Outline

1. *Openssh* Configuration

```
# host does not allow password authentication
$ ssh postgres@momjian.us
Permission denied (publickey).

# can also use ssh-keygen -D opensc-pkcs11.so -e
# use the PIV AUTH key (1)
$ pkcs15-tool --read-ssh-key 1 --output ssh.pub
Using reader with a card: Yubico Yubikey 4 OTP+U2F+CCID 00 00
Please enter PIN [PIV Card Holder pin]:

$ cat ssh.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDBrGGJqMxb…

$ sudo sh -c 'cat ssh.pub >> ~postgres/.ssh/authorized_keys'
$ rm ssh.pub

$ ssh -I "$OPENSC" postgres@momjian.us
Enter PIN for 'PIV_II (PIV Card Holder pin)':
Last login: Wed Aug 16 22:52:21 2017 from momjian.us
$ id
uid=109(postgres) gid=117(postgres) groups=117(postgres),111(ssl-cert)
```

Add PKCS#11 Provider for a Host

```
$ cp ~/.ssh/config ~/.ssh/config.orig

# OPENSC set previously
$ echo "
> Host momjian.us
> PKCS11Provider $OPENSC" >> ~/.ssh/config

# -I not needed
$ ssh postgres@momjian.us
Enter PIN for 'PIV_II (PIV Card Holder pin)':
Last login: Fri Aug 18 15:23:09 2017 from momjian.us
$
```

https://ef.gy/hardening-ssh

Use *ssh-agent* To Avoid Repeated PIN Entry

```
# restore config file since we are going to use ssh-agent, not the library directly
$ mv ~/.ssh/config.orig ~/.ssh/config

$ eval $(ssh-agent -s)
Agent pid 9103
$ ssh-add -s "$OPENSC"
Enter passphrase for PKCS#11:
Card added: /usr/lib/x86_64-linux-gnu/opensc-pkcs11.so

$ ssh postgres@momjian.us
Last login: Sat Aug 19 10:05:01 2017 from momjian.us
$

$ ssh -I "$OPENSC" postgres@momjian.us
Enter PIN for 'PIV_II (PIV Card Holder pin)':
```

https://wikitech.wikimedia.org/wiki/Yubikey-SSH
https://utcc.utoronto.ca/~cks/space/blog/sysadmin/Yubikey4ForSSHKeys

ssh-agentd: Script for controlling *ssh-agent*, Part 1

```
$ cat > ssh-agentd <END
[ "$#" -gt 1 -o \( "$#" -eq 1 -a "$1" != "-k" -a "$1" != "-r" -a "$1" != "-s" \) ] &&
        echo "Usage:  $(basename $0) -[krs]" 1>&2
# can't 'exit' since we are being sourced into the shell, we assume no args

# -k stop the running daemon
# -r reload the keys (non-SSH access to the device disconnects ssh-agent)
# -s status
# We don't restart for -r because it would change the required environment
# settings for other sessions.

# Export environment variables for connecting to ssh-agent,
# and optionally start it.
# Should be dot-sourced to set env variables, e.g., ". ssh-agentd",
# -k only stops the running daemon.

# The ssh-agent daemon launched by sshd and gnome-session doesn't
# understand PKCS11 so we have to launch our own and set environment
# variables to point to our own.  This is why we can't use SSH_AUTH_SOCK
# to determine if we have a valid ssh-agent.
```

ssh-agentd: Part 2

```
# Only stop daemon?
if [ "$1" = "-k" ]
then    if [ -s ~/.ssh-agent.pid ]
        then    SSH_AGENT_PID="$(cat  ~/.ssh-agent.pid)" \
                        ssh-agent -k > ~/.ssh-agent.env
                . ~/.ssh-agent.env
                rm ~/.ssh-agent.pid ~/.ssh-agent.env
        fi
else    if [ -s ~/.ssh-agent.pid ] &&
                kill -0 "$(cat ~/.ssh-agent.pid)" >/dev/null 2>&1
        then    # load environment
                if [ "$1" = "-s" ]
                then    echo "Agent pid $(cat ~/.ssh-agent.pid)"
                else    . ~/.ssh-agent.env > /dev/null
                        if [ "$1" = "-r" ]
                        then    ssh-add -e "$OPENSC"
                                ssh-add -s "$OPENSC"
                        fi
                fi
```

ssh-agentd: Part 3

```
        elif [ "$1" != "-s" ]
        then    # execute this if no daemon is running, even with -r
                # start ssh-agent;  save and set environment
                ssh-agent -s > ~/.ssh-agent.env
                . "$HOME"/.ssh-agent.env > /dev/null
                echo "$SSH_AGENT_PID" > ~/.ssh-agent.pid

                # Add PKCS#11 keys
                . /letc/opensc.env
                ssh-add -s "$OPENSC"
        fi
fi
END
```

Consider keychain instead: http://nullprogram.com/blog/2012/06/08/

ssh-agentd: Installation

```
$ chmod +x ssh-agentd
$ chown root:root ssh-agentd
$ sudo cp ssh-agentd /usr/local/bin
```

Various *ssh-agent* scripts: https://stackoverflow.com/questions/18880024/start-ssh-agent-on-login

ssh-agentd: Usage

```
$ . ssh-agentd
$ ssh postgres@momjian.us
Last login: Sat Aug 19 12:32:18 2017 from momjian.us
$

$ pkcs11-tool --module "$OPENSC" --show-info
Cryptoki version 2.20
Manufacturer     OpenSC (www.opensc-project.org)
Library          Smart card PKCS#11 API (ver 0.0)
Using slot 1 with a present token (0x1)
$ ssh postgres@momjian.us
Connection closed by 127.0.0.1
```

ssh-agentd: Usage

```
$ . ssh-agentd -r
Card removed: /usr/lib/x86_64-linux-gnu/opensc-pkcs11.so
Enter passphrase for PKCS#11:
Card added: /usr/lib/x86_64-linux-gnu/opensc-pkcs11.so
$ ssh postgres@momjian.us
Last login: Sat Aug 19 12:33:11 2017 from momjian.us
$

$ . ssh-agentd -s
Agent pid 27825

# must be stopped or it will interfere with gpg's scdaemon
$. ssh-agentd -k
Agent pid 27825 killed
```

2. OpenPGP Configuration

Originally designed for email encryption and signing, OpenPGP supports other applications:

- file encryption
- file signing (e.g., documents, binaries)
- *openssh* and PAM authentication
- *git* commit signing
- Postgres encryption and signing

OpenPGP standard: `https://gnupg.org/ftp/specs/OpenPGP-smart-card-application-3.3.1.pdf`

OpenPGP Configuration

- Historically, OpenPGP *(pgp* and *gpg)* contained a single active subkey used for signing, encryption, and authentication
  - It can also contain historical keys and the keys of trusted individuals
- Modern OpenPGP uses subkeys with dedicated roles, e.g., signing, encryption, authentication, like PIV
- Expiration and revocation are also supported
- A primary/master key signs the subkeys and is optionally kept off line
- This more closely matches TLS/SSL certificate authority usage

https://wiki.debian.org/Subkeys

https://support.yubico.com/hc/en-us/articles/360013790259-Using-Your-YubiKey-with-OpenPGP

https://github.com/drduh/YubiKey-Guide/blob/master/README.md

Operating System Software

```
$ sudo apt-get install gnupg2 scdaemon
```

Check the Card's Status

```
$ gpg --card-status
Application ID ...: D2760001240102010006062515440000
Version ..........: 2.1
Manufacturer .....: Yubico
Serial number ....: 06251544
Name of cardholder: [not set]
Language prefs ...: [not set]
Sex ..............: unspecified
URL of public key : [not set]
Login data .......: [not set]
Signature PIN ....: not forced
Key attributes ...: 2048R 2048R 2048R
Max. PIN lengths .: 127 127 127
PIN retry counter : 3 0 3
Signature counter : 0
Signature key ....: [none]
Encryption key....: [none]
Authentication key: [none]
General key info..: [none]
```

Blue fields can be populated manually using *gpg –card-edit*.

https://developers.yubico.com/PGP/Card_edit.html

gpg-agentd: Script for controlling *gpg-agent*, Part 1

```
$ cat > gpg-agentd <END
[ "$#" -gt 1 -o \( "$#" -eq 1 -a "$1" != "-k" -a "$1" != "-r" -a "$1" != "-s" \) ] &&
        echo "Usage:  $(basename $0) -[krs]" 1>&2
# can't 'exit' since we are being sourced into the shell, we assume no args

# -k stop the running daemon
# -r reload the connection
# -s status
# We don't restart for -r because it would change the required environment
# settings for other sessions.

# Export environment variables for connecting to gpg-agent,
# and optionally start it.
# Should be dot-sourced to set env variables, e.g., ". gpg-agentd",
# -k only stops the running daemon
```

http://lorgor.blogspot.com/2017/01/yubikey-gpgssh-great-security-but.html

gpg-agentd: Part 2

```
# Only stop daemon?
if [ "$1" = "-k" ]
then    if [ -s ~/.gpg-agentd.pid ]
        then    kill "$(cat  ~/.gpg-agentd.pid)"
                unset GPG_AGENT_INFO
                rm ~/.gpg-agentd.pid ~/.gpg-agentd.env
                echo 'Agent stopped'
        fi
else    if [ -s ~/.gpg-agentd.pid ] &&
                kill -0 "$(cat ~/.gpg-agentd.pid)" >/dev/null 2>&1
        then    # load environment
                if [ "$1" = "-s" ]
                then    echo "Agent pid $(cat ~/.gpg-agentd.pid)"
                else    . ~/.gpg-agentd.env > /dev/null
                        if [ "$1" = "-r" ]
                        then    gpg-connect-agent reloadagent /bye
                                #gpg-connect-agent "SCD RESET" /bye
                        fi
                fi
```

gpg-agentd: Part 3

```
        elif [ "$1" != "-s" ]
        then    # execute this if no daemon is running, even with -r
                # start gpg-agent;  save and set environment
                gpg-agent -s --enable-ssh-support --daemon > ~/.gpg-agentd.env
                . "$HOME"/.gpg-agentd.env > /dev/null
                echo "$GPG_AGENT_INFO" | awk -F: '{print $2}' > ~/.gpg-agentd.pid
        fi
fi
END
```

gpg-agentd: Installation and Running

```
$ chmod +x gpg-agentd
$ chown root:root gpg-agentd
$ sudo cp gpg-agentd /usr/local/bin

$ . gpg-agentd
```

Reset the OpenPGP Mode

```
$ rm -rf ~/.gnupg

$ openpgp-tool --erase
Using reader with a card: Yubico YubiKey OTP+FIDO+CCID 00 00
Erase card
```

https://developers.yubico.com/ykneo-openpgp/ResetApplet.html

https://github.com/OpenSC/OpenSC/wiki/OpenPGP-card

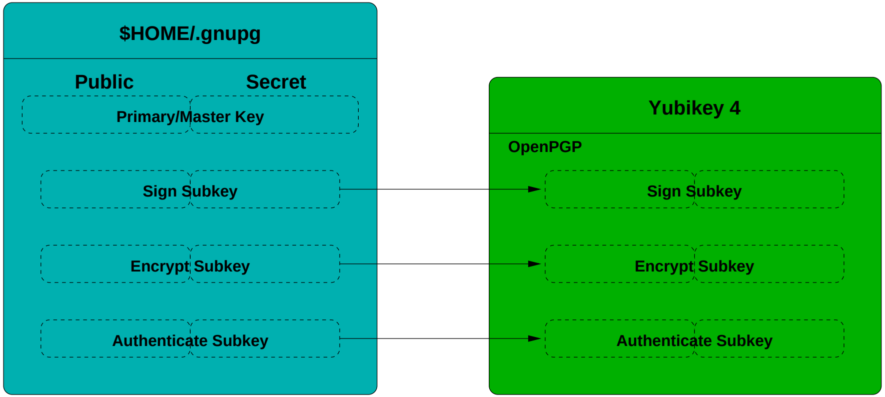
Set configuration of PIN, Reset Code, and Admin Code

```
#!/bin/bash

cd "$HOME" || exit 1
umask 0077

mkdir .yubikey 2> /dev/null
rm -f ~/.yubikey/openpgp.*

PIN="$(dd if=/dev/random bs=1 count=6 2>/dev/null |
       hexdump -v -e '/1 "%u"' | cut -c1-6)"
echo -n "Change PIN (old PIN is '123456') to "
echo "$PIN" | tee ~/.yubikey/openpgp.pin
echo
# clear DISPLAY so we can paste in the new value
DISPLAY="" gpg --change-pin
```

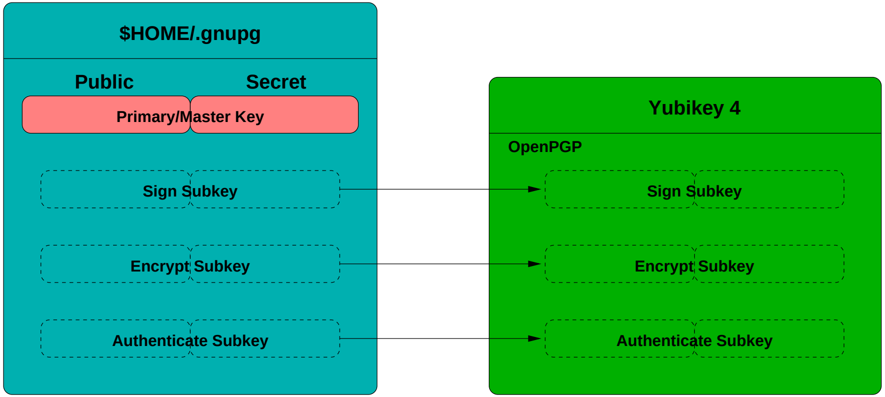
Set configuration of PIN, Reset Code, and Admin Code

```
# same as PIN
RESET="$(dd if=/dev/random bs=1 count=8 2>/dev/null |
         hexdump -v -e '/1 "%u"' | cut -c1-8)"
echo -n "SET Reset Code (PUK) (Admin PIN is '12345678') "
echo "$RESET" | tee ~/.yubikey/openpgp.reset
echo
DISPLAY="" gpg --change-pin

ADMIN="$(dd if=/dev/random bs=1 count=8 2>/dev/null |
         hexdump -v -e '/1 "%u"' | cut -c1-8)"
echo -n "Change Admin PIN (old Admin PIN is '12345678') "
echo "$ADMIN" | tee ~/.yubikey/openpgp.admin
echo
DISPLAY="" gpg --change-pin
```

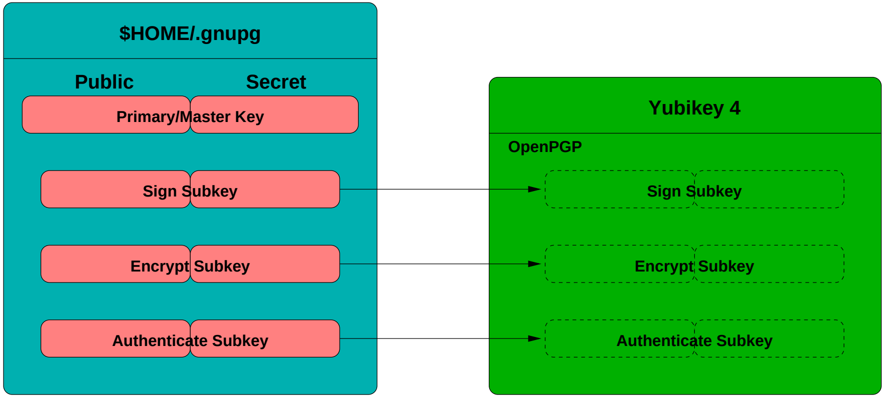
The Key Creation Process



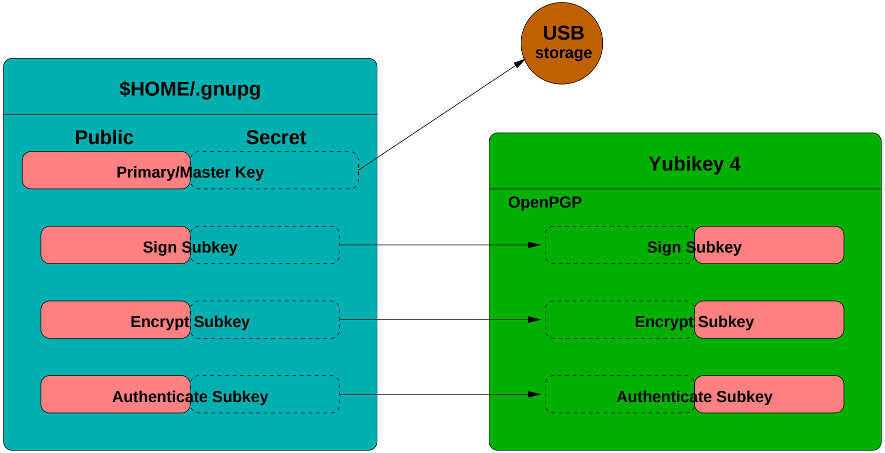
The Key Creation Process



The Key Creation Process



The Key Creation Process



Create the Primary/Master Key

```
$ gpg --gen-key
gpg (GnuPG) 2.2.40; Copyright (C) 2022 g10 Code GmbH
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
…
GnuPG needs to construct a user ID to identify your key.

Real name: Bruce Momjian
Email address: bruce@momjian.us
You selected this USER-ID:
    "Bruce Momjian <bruce@momjian.us>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O gpg --gen-key
You need a Passphrase to protect your secret key.
```

https://blog.liw.fi/posts/2017/05/29/using_a_yubikey_4_for_ensafening_one_s_encryption/

Create a Sign Subkey

```
KEYID="$(gpg --list-keys | awk '{if (out == "Y") {print; exit;}; if ($1 == "pub") out = "Y"}')"

$ gpg --expert --edit-key "$KEYID"
Secret key is available.
sec  rsa3072/F299ED0A3C65CC11
     created: 2023-09-19  expires: 2025-09-18  usage: SC
     trust: ultimate      validity: ultimate
ssb  rsa3072/5A0F6646D28DDE76
     created: 2023-09-19  expires: 2025-09-18  usage: E
[ultimate] (1). Bruce Momjian <bruce@momjian.us>
```

Subkeys: https://security.stackexchange.com/questions/112059/purpose-of-secret-subkeys

Create a Sign Subkey

```
gpg> addkey
Please select what kind of key you want:
   (3) DSA (sign only)
   (4) RSA (sign only)
   (5) Elgamal (encrypt only)
   (6) RSA (encrypt only)
   (7) DSA (set your own capabilities)
   (8) RSA (set your own capabilities)
  (10) ECC (sign only)
  (11) ECC (set your own capabilities)
  (12) ECC (encrypt only)
  (13) Existing key
  (14) Existing key from card
Your selection? 4
```

Create a Sign Subkey

```
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (3072)
Requested keysize is 3072 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y
Really create? (y/N) y
```

gpg Codes

Usage types:

A	authenticate
C	certificate creation
E	encrypt
S	sign

Key types:

pub	public primary/master key
ssb	secret (private) subkey
sec	secret (private) primary/master key
sub	public subkey

`https://unix.stackexchange.com/questions/31996/how-are-the-gpg-usage-flags-defined-in-the-key-details-listing`

Sign Subkey Created

```
sec  rsa3072/F299ED0A3C65CC11
     created: 2023-09-19  expires: 2025-09-18  usage: SC
     trust: ultimate      validity: ultimate
ssb  rsa3072/5A0F6646D28DDE76
     created: 2023-09-19  expires: never       usage: E
ssb  rsa3072/A883914CF02039DE
     created: 2023-09-19  expires: never       usage: S
[ultimate] (1). Bruce Momjian <bruce@momjian.us>

gpg> save
```

Backup the Primary and Subkeys to USB Storage

```
$ df
$ cd /media/laptop11/1F22-32CC

$ gpg --armor --export-secret-keys "$KEYID" > master_with_sub.key

$ gpg --armor --export-secret-subkeys "$KEYID" > sub.key
```

This backs up the contents of the secret keys. Once they are copied to the Yubikey, export only backs up the links to the Yubikey. (Links can be recreated by running *gpg –card-status*.)

Remove the Secret Primary Key

```
$ gpg --delete-secret-keys "$KEYID"

sec  rsa3072/F299ED0A3C65CC11 2023-09-19 Bruce Momjian <bruce@momjian.us>
Delete this key from the keyring? (y/N) y
This is a secret key! - really delete? (y/N) y

$ gpg --import sub.key
gpg: key F299ED0A3C65CC11: "Bruce Momjian <bruce@momjian.us>" not changed
gpg: To migrate 'secring.gpg', with each smartcard, run: gpg --card-status
gpg: key F299ED0A3C65CC11: secret key imported
gpg: Total number processed: 1
gpg:              unchanged: 1
gpg:       secret keys read: 1
gpg:   secret keys imported: 1
```

This can also be accomplished by deleting files: https://wiki.debian.org/Subkeys

Move the Encrypt Secret Subkey to the Card

```
$ gpg --expert --edit-key "$KEYID"
…
gpg> key 1

pub  rsa3072/F299ED0A3C65CC11
     created: 2023-09-19  expires: 2025-09-18  usage: SC
     trust: ultimate      validity: ultimate
ssb* rsa3072/5A0F6646D28DDE76
     created: 2023-09-19  expires: never       usage: E
ssb  rsa3072/A883914CF02039DE
     created: 2023-09-19  expires: never       usage: S
[ultimate] (1). Bruce Momjian <bruce@momjian.us>
```

Move the Encrypt Secret Subkey to the Card

```
gpg> keytocard
Please select where to store the key:
   (2) Encryption key
Your selection? 2

pub  rsa3072/F299ED0A3C65CC11
     created: 2023-09-19  expires: 2025-09-18  usage: SC
     trust: ultimate      validity: ultimate
ssb* rsa3072/5A0F6646D28DDE76
     created: 2023-09-19  expires: never       usage: E
ssb  rsa3072/A883914CF02039DE
     created: 2023-09-19  expires: never       usage: S
[ultimate] (1). Bruce Momjian <bruce@momjian.us>
```

Moving keys to the card must be done after they are backed up.

Move the Sign Secret Subkey to the Card

```
gpg> # reset key selection
gpg> key 0
gpg> key 2

pub  rsa3072/F299ED0A3C65CC11
     created: 2023-09-19  expires: 2025-09-18  usage: SC
     trust: ultimate      validity: ultimate
ssb  rsa3072/5A0F6646D28DDE76
     created: 2023-09-19  expires: never       usage: E
ssb* rsa3072/A883914CF02039DE
     created: 2023-09-19  expires: never       usage: S
[ultimate] (1). Bruce Momjian <bruce@momjian.us>
```

Move the Sign Secret Subkey to the Card

```
gpg> keytocard
Please select where to store the key:
   (1) Signature key
   (3) Authentication key
Your selection? 1

pub  rsa3072/F299ED0A3C65CC11
     created: 2023-09-19  expires: 2025-09-18  usage: SC
     trust: ultimate      validity: ultimate
ssb  rsa3072/5A0F6646D28DDE76
     created: 2023-09-19  expires: never       usage: E
     card-no: 0006 06251544
ssb* rsa3072/A883914CF02039DE
     created: 2023-09-19  expires: never       usage: S
[ultimate] (1). Bruce Momjian <bruce@momjian.us>
```

Create an Authenticate Subkey on the Card

```
gpg> addcardkey
Signature key ....: [none]
Encryption key....: [none]
Authentication key: [none]
```

Authenticate keys are easier to replace than sign or encrypt keys, so having a backup is not as critical.

Create an Authenticate Subkey on the Card

```
Please select the type of key to generate:
   (1) Signature key
   (2) Encryption key
   (3) Authentication key
Your selection? 3

Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y
Really create? (y/N) y
```

Authenticate Subkey on the Card Created

```
sec  rsa3072/7E2DFFC443D68AA9
     created: 2023-09-15  expires: 2025-09-14  usage: SC
     trust: ultimate      validity: ultimate
ssb  rsa3072/A9875F59540C7942
     created: 2023-09-15  expires: 2025-09-14  usage: E
ssb  rsa3072/96D9385D52DAF123
     created: 2023-09-15  expires: never       usage: S
ssb  rsa2048/A43F20F060C0C6DF
     created: 2023-09-15  expires: never       usage: A
     card-no: 0006 06251544
[ultimate] (1). Bruce Momjian <bruce@momjian.us>

gpg> save
```

Check the Card’s Status

```
$ gpg --card-status
Reader ...........: 1050:0407:X:0
Application ID ...: D2760001240102010006062515440000
Application type .: OpenPGP
Version ..........: 2.1
Manufacturer .....: Yubico
Serial number ....: 06251544
Name of cardholder: [not set]
Language prefs ...: [not set]
Salutation .......:
URL of public key : [not set]
Login data .......: [not set]
Signature PIN ....: not forced
Key attributes ...: rsa3072 rsa3072 rsa2048
Max. PIN lengths .: 127 127 127
PIN retry counter : 3 3 3
Signature counter : 0
Signature key ....: 8793 7479 076C AF81 0A45  9CD8 0C77 C262 688B 5033
      created ....: 2023-09-16 00:11:53
Encryption key....: EB13 6D97 593A 883E D5AE  FF46 A16F 6A31 BC90 FB91
      created ....: 2023-09-16 00:08:37
Authentication key: DE5E 5BCF 4947 6DF2 53F1  F314 C13A 6EF9 93B8 CA9B
      created ....: 2023-09-16 00:14:35
```

Check the Card's Status

```
General key info..: sub  rsa3072/0C77C262688B5033 2023-09-16 Bruce Momjian <bruce@momjian.us>
sec#  rsa3072/6BC1DEA2A39C4F58  created: 2023-09-16  expires: 2025-09-15
ssb>  rsa3072/A16F6A31BC90FB91  created: 2023-09-16  expires: 2025-09-15
                                card-no: 0006 06251544
ssb>  rsa3072/0C77C262688B5033  created: 2023-09-16  expires: never
                                card-no: 0006 06251544
ssb>  rsa2048/C13A6EF993B8CA9B  created: 2023-09-16  expires: never
                                card-no: 0006 0625154
```

# indicates the secret key is missing, and > indicates a pointer to the secret key.

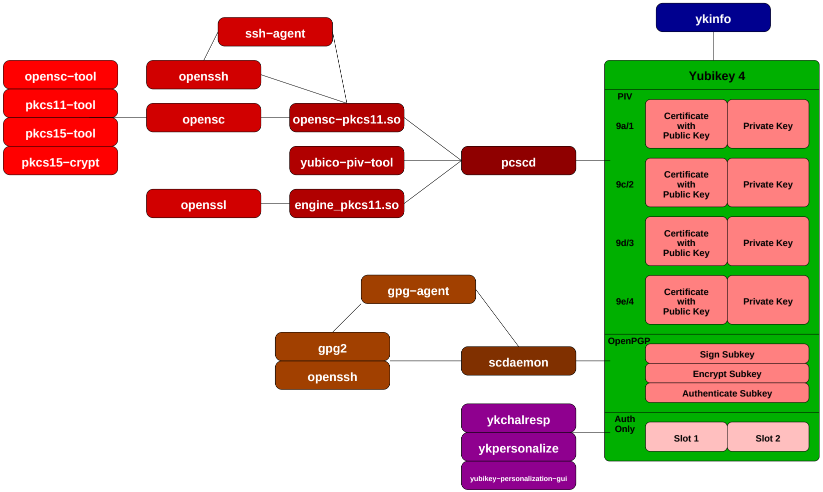
3. OpenPGP Usage: Encrypt and Sign

```
$ echo test | gpg --encrypt --armor --recipient "$KEYID" |
      gpg --decrypt --armor --passphrase-file .yubikey/openpgp.pin
gpg: encrypted with 2048-bit RSA key, ID 3AF0B4AC, created 2017-08-24
      "Bruce Momjian <bruce@momjian.us>"
test

$ echo test | gpg --armor --clearsign --default-key "$KEYID" | gpg
test
gpg: Signature made Thu 24 Aug 2017 07:18:14 PM EDT using RSA key ID 28B2789A
gpg: Good signature from "Bruce Momjian <bruce@momjian.us>" [ultimate]
```

OpenPGP and *openssh*

```
# host does not allow password authentication
$ ssh postgres@momjian.us
Permission denied (publickey).

# this uses gpg-agentd
$ ssh-add -L > ssh.pub

$ cat ssh.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDADEZEXhtp…

$ sudo sh -c 'cat ssh.pub >> ~postgres/.ssh/authorized_keys'
$ rm ssh.pub

$ ssh  postgres@momjian.us
Last login: Sat Aug 19 12:33:29 2017 from momjian.us
$ id
uid=109(postgres) gid=117(postgres) groups=117(postgres),111(ssl-cert)
```

Applications, Software, and Tools Illustrated

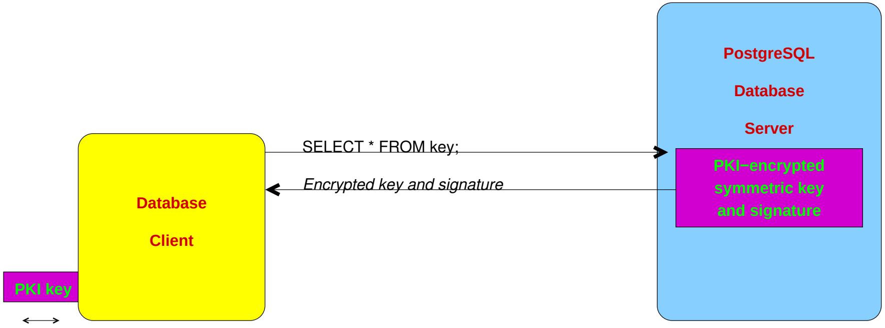


For instructions on using *pcscd* and *scdaemon* tools together, see `https://wiki.archlinux.org/index.php/GnuPG#GnuPG_with_pcscd_.28PCSC_Lite.29`.

4. PIV vs OpenPGP

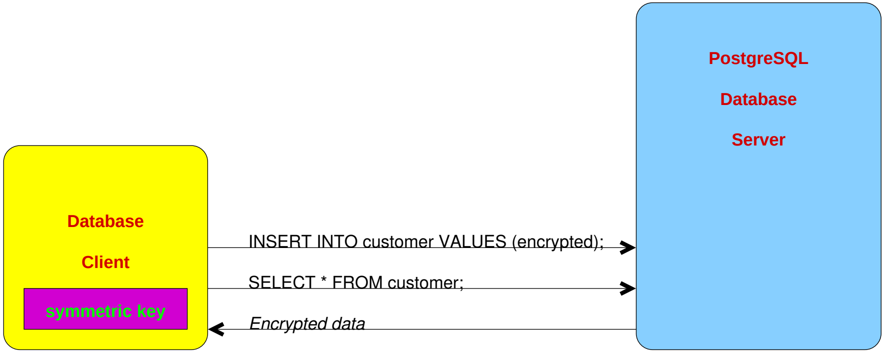
- Storage differences
  - PIV stores all per-user information on removable media
  - OpenPGP
    - requires storage of per-user OpenPGP public key information in the file system
    - optionally stores private/secret information on removable media
    - unsuited for multiple users using the same card reader or USB slot
- Application support
  - OpenSSH supports both
  - OpenSSL supports PIV
  - *pgp, gpg, git commit* support OpenPGP
  - many email programs support OpenPGP through S/MIME

5. Postgres Usage: Client Side


PostgreSQL
Database
Server
PKI−encrypted symmetric key and signature
SELECT * FROM key;
Encrypted key and signature
Database
Client
PKI key

The signature detects key modifications. The key and data can still be deleted with proper permissions.

Client Uses Decrypted Received Key



The unencrypted symmetric key never appears on the server.

Create Key Table

```
CREATE TABLE user_key (
        username        NAME PRIMARY KEY,
        enc_sym_key     BYTEA,
        signed_hash     BYTEA
);
```

Compute Key and Signature

```
\set sym_key "openssl rand -hex 32 | tr -d '\n'"
\echo :sym_key
e00c82d36d31411987054e8c004c09a0323e4166726de963a35de66394f6edd6

-- use 0:2 because of signature requirement
\set enc_sym_key "echo :'sym_key' | openssl rsautl -engine pkcs11 -keyform
  engine -encrypt -inkey 2 -passin file:"$HOME"/.yubikey/piv.pin |
  xxd -plain | tr -d '\n'"

-- create a signed hash of enc_sym_key to detect unauthorized changes
-- This could be done with openssl using -encrypt then -sign, but openssl
-- version 1.0.1t doesn't support input data longer than 245 bytes for PIV
-- rsautl and doesn't support any PIV pkeyutl operations.  Therefore, we
-- manually generate the hash, sign it, and store it in a separate column.
\set signed_hash "echo :'enc_sym_key' | openssl dgst -sha256 -binary |
  openssl rsautl -engine pkcs11 -keyform engine -sign -inkey 2 -passin
  file:~/.yubikey/piv.pin | xxd -plain | tr -d '\n'"
```

Populate Key Table

```
INSERT INTO user_key VALUES (CURRENT_USER,
                             decode(:'enc_sym_key', 'hex'),
                             decode(:'signed_hash', 'hex'));

SELECT * FROM user_key WHERE username = CURRENT_USER;
 username |          enc_sym_key              |          signed_hash
----------+-----------------------------------+------------------------------
 user1    | \x994c87a3ca52d8aa43fcf55…        | \xc2a52d5465d853cdb76e6b3bd…
```

Retrieve Encrypted Key and Verify Signature

```
SELECT enc_sym_key, signed_hash FROM user_key WHERE username = CURRENT_USER
\gset

-- check signature, these two hex values should match:
\echo "echo :'signed_hash' | cut -c3- | xxd -plain -revert | openssl rsautl -engine
  pkcs11 -keyform engine -verify -inkey 2 -passin
  file:"$HOME"/.yubikey/piv.pin | xxd -plain | tr -d '\n'"
107dd77e826db987bd1dcb0487de65ba47f1b937fc3355bb84c1e7b24a932481
\echo "echo :'enc_sym_key' | cut -c3- | openssl dgst -sha256 -binary |
  xxd -plain | tr -d '\n'"
107dd77e826db987bd1dcb0487de65ba47f1b937fc3355bb84c1e7b24a932481
```

Get Symmetric Key

```
\set sym_key "echo :'enc_sym_key' | cut -c3- | xxd -plain -revert | openssl rsautl
  -engine pkcs11 -keyform engine -decrypt -inkey 2 -passin
  file:~/.yubikey/piv.pin"

-- symmetric key
\echo :sym_key
e00c82d36d31411987054e8c004c09a0323e4166726de963a35de66394f6edd6
```

With Cryptographic Hardware Removed

```
\set sym_key "echo :'enc_sym_key' | cut -c3- | xxd -plain -revert | openssl rsautl
  -engine pkcs11 -keyform engine -decrypt -inkey 2 -passin
  file:"$HOME"/.yubikey/piv.pin"
engine "pkcs11" set.
Invalid slot number: 1
PKCS11_get_private_key returned NULL
cannot load Private Key from engine
140427660957328:error:26096080:engine routines:ENGINE_load_private_key:failed
loading private key:eng_pkey.c:124:
unable to load Private Key
```

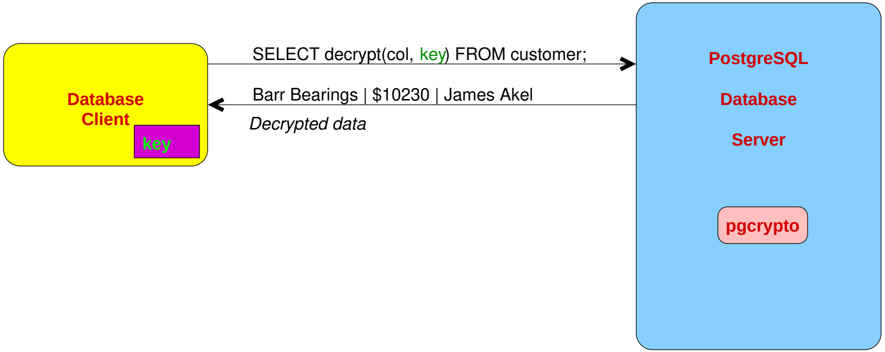
Create Survey Table and Populate

```
CREATE TABLE survey1 (id SERIAL, username NAME, enc_result BYTEA);

\set enc `echo 'secret_message' | openssl enc -aes-256-cbc -pass pass::sym_key |
  xxd -plain | tr -d '\n'`
INSERT INTO survey1 VALUES (DEFAULT, CURRENT_USER, decode(:'enc', 'hex'));

SELECT * FROM survey1 WHERE username = CURRENT_USER;
 id | username |                    enc_result
----+----------+--------------------------------------------------------------
  1 | user1    | \x53616c7465645f5fa4cbc7d81c989cfa9611e9e4be7bddbf8b4c…
```

Retrieve Data and Decrypt

```
SELECT enc_sym_key, signed_hash FROM user_key WHERE username = CURRENT_USER
\gset

-- required signature verification skipped
\set sym_key `echo :'enc_sym_key' | cut -c3- | xxd -plain -revert | openssl
  rsautl -engine pkcs11 -keyform engine -decrypt -inkey 2 -passin
  file:"$HOME"/.yubikey/piv.pin`

SELECT * FROM survey1 WHERE username = CURRENT_USER
\gset
\set result `echo :'enc_result' | cut -c3- | xxd -plain -revert |
  openssl enc -d -aes-256-cbc -pass pass::sym_key`

\echo :result
secret_message
```

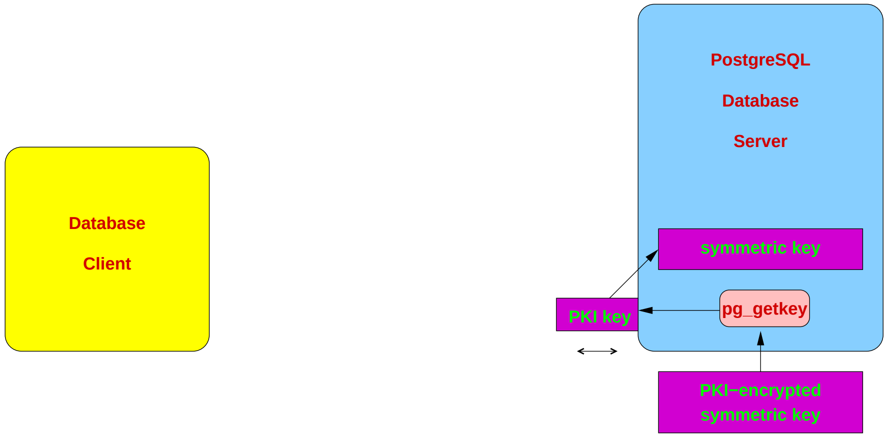
Client-Side Key and Server-Side Processing



Database Client
key
SELECT decrypt(col, key) FROM customer;
Barr Bearings | $10230 | James Akel
Decrypted data
PostgreSQL
Database
Server
pgcrypto

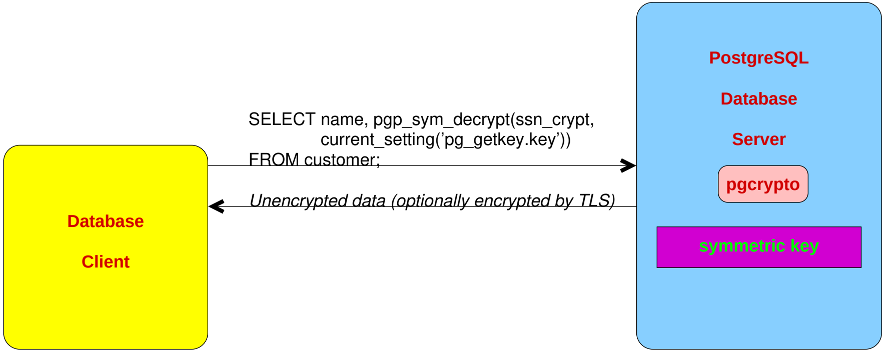
This has the key on the client, the server, the server logs, and over the network.

Server-Side Key: Use *pg_getkey* to Decrypt Stored Key



pg_getkey uses the PKI key to decrypted the on-disk encrypted symmetric key and store it in a server-side variable (GUC).

pgcrypto Uses the Decrypted Key



```
SELECT name, pgp_sym_decrypt(ssn_crypt,
            current_setting('pg_getkey.key'))
FROM customer;
```

pgcrypto can use the key stored in the server-side variable. The unencrypted key is never stored unencrypted in the file system.

Install *pg_getkey*

```
$ wget https://momjian.us/download/pg_getkey-1.0.tgz
$ tar zxf pg_getkey-1.0.tgz
$ make top_builddir=/usr/local/src/pgsql clean install
# modify $DESTDIR/bin/pg_getkey_generate
$ pg_getkey_generate
engine "pkcs11" set.
Wrote public-key encrypted symmetric key to /u/pgsql/data/pg_getkey.key
Additional steps:
 * Customize the bin/pg_getkey script
 * Make sure the Postgres binary directory is in server's PATH
 * Add this to $PGDATA/postgresql.conf and restart:
       shared_preload_libraries = 'pg_getkey.so'
 * If the key cannot be loaded, the server will not start and an error
   message will be written to the Postgres server log
$ echo "shared_preload_libraries = 'pg_getkey.so'" >> $PGDATA/postgresql.conf
```

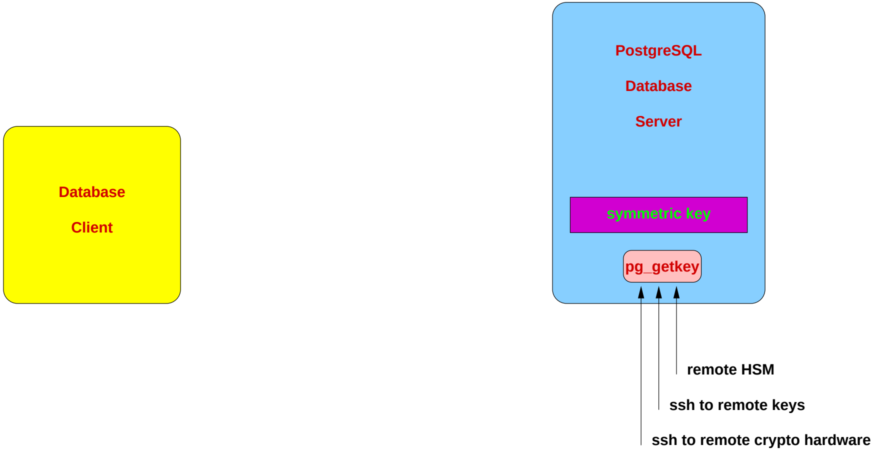
Check *pg_getkey*

```
$ pg_ctl restart
$ psql postgres
SHOW pg_getkey.key;
                          pg_getkey.key
------------------------------------------------------------------
 01c85817bdd7b7de6c5d8047dda80895999da6ac975f04f7596203e399776940

SELECT setting
FROM pg_settings
WHERE name = 'pg_getkey.key';
                             setting
------------------------------------------------------------------
 01c85817bdd7b7de6c5d8047dda80895999da6ac975f04f7596203e399776940

SELECT current_setting('pg_getkey.key');
                         current_setting
------------------------------------------------------------------
 01c85817bdd7b7de6c5d8047dda80895999da6ac975f04f7596203e399776940
```

Use *pg_getkey*

```
CREATE EXTENSION pgcrypto;

SELECT pgp_sym_encrypt('secret_message', current_setting('pg_getkey.key'));
                  pgp_sym_encrypt
--------------------------------------------
 \xc30d04070302436a9eb71085cdad6fd23f01c79…

SELECT pgp_sym_decrypt(
                pgp_sym_encrypt('secret_message', current_setting('pg_getkey.key')),
                current_setting('pg_getkey.key'));
 pgp_sym_decrypt
-----------------
 secret_message
```

Other *pg_getkey* Key Access Options



Per-User Keys

Using PL/PERLU, you can:

- Store per-user keys in the each database, encrypted with cryptographic hardware
  - PL/PERLU can run operating system commands and store the output
  - Only the super user can create PL/PERLU functions
- Have users call a PL/PERLU function to retrieve their symmetric key, like *pg_getkey*
  - on first call for each user, add a PKI-encrypted symmetric key row to a database table
  - on first call per session, decrypt the stored key using cryptographic hardware
  - cache the result for future calls using a PL/PERLU global variable
- *shared_preload_libraries* speeds up the first use of PL/PERLU in a session
- This is a combination of the client-side and *pg_getkey* approaches

Setup for Per-User Keys

```
CREATE EXTENSION plperlu;

DROP TABLE IF EXISTS user_key;

CREATE TABLE user_key (
        username        NAME PRIMARY KEY,
        enc_sym_key     BYTEA
);

GRANT SELECT, INSERT ON TABLE user_key TO PUBLIC;
```

Per-User Keys Function

```
CREATE OR REPLACE FUNCTION pg_get_user_key() RETURNS TEXT AS $$
    # No precomputed key?
    if (!defined($_SHARED{key}))
    {
        my $rv = spi_exec_query("
            SELECT encode(enc_sym_key, 'hex')
            FROM user_key
            WHERE username = CURRENT_USER", 1);
```

Per-User Keys Function

```
# Add user key row?
if ($rv->{processed} == 0)
{
    my $enc_key = `openssl rand -hex 32 | \\
        openssl rsautl -engine pkcs11 -keyform engine \\
        -encrypt -inkey 3 -passin file:"\$PIN_FILE" | \\
        xxd -plain`;
    $rv = spi_exec_query("
        INSERT INTO user_key VALUES (
            CURRENT_USER, decode('$enc_key', 'hex'))");
    elog(ERROR, "Could not insert key row")
        if ($rv->{processed} != 1);
    $rv = spi_exec_query("
        SELECT encode(enc_sym_key, 'hex')
        FROM user_key
        WHERE username = CURRENT_USER", 1);
}
```

Per-User Keys Function

```
        elog(ERROR, "Could not find key row")
            if ($rv->{processed} == 0);
        my $enc_sym_key = $rv->{rows}[0]->{encode};
        # decrypt the key, use 0:3 because only encryption is required
        my $key = `echo ''$enc_sym_key'' | xxd -plain -revert | \\
             openssl rsautl -engine pkcs11 -keyform engine -decrypt \\
             -inkey 3 -passin file:"\$PIN_FILE"`;
        chomp($key);
        $_SHARED{key} = $key;
    }
    return $_SHARED{key};
$$ LANGUAGE plperlu;
```

Per-User Keys Function Usage

```
$ psql -U postgres test
SELECT * FROM user_key;
 username | enc_sym_key
----------+------------

SELECT pg_get_user_key();
                         pg_get_user_key
------------------------------------------------------------------
 8ae54420dd959ad997580781763732902be5ddef4d587fbd4202d1e258218bcc

SELECT * FROM user_key;
 username |                           enc_sym_key
----------+--------------------------------------------------------
 postgres | \x0401bb58f0d16daea8b29ca2f5bbfd56bf29336ab01d6b3b8388…

SELECT pgp_sym_decrypt(
                pgp_sym_encrypt('secret_message', pg_get_user_key()),
                pg_get_user_key());
 pgp_sym_decrypt
-----------------
 secret_message
```

Add a Second User

```
$ psql -U bob test
SELECT * FROM user_key;
 username |                       enc_sym_key
----------+---------------------------------------------------------------
 postgres | \x0401bb58f0d16daea8b29ca2f5bbfd56bf29336ab01d6b3b8388…

SELECT pg_get_user_key();
                         pg_get_user_key
-------------------------------------------------------------------
 d43e5f52b0776ac34caf2f6b17a4884eab6cf683954ba2d8208e2ef8f348a0da

SELECT * FROM user_key;
 username |                      enc_sym_key
----------+---------------------------------------------------------------
 postgres | \x0401bb58f0d16daea8b29ca2f5bbfd56bf29336ab01d6b3b8388…
 bob      | \x5764364dcc8d3e914682b8642646adc9a559f7e156d636c83616…
```

Show the First User Is Unchanged

```
$ psql -U postgres test
SELECT pg_get_user_key();
                         pg_get_user_key
------------------------------------------------------------------
 8ae54420dd959ad997580781763732902be5ddef4d587fbd4202d1e258218bcc
```

A restricted administrative function could be written that decrypts multiple user keys in the same session.

Transparent Encryption

- Allow encryption to be transparent to application developers, though not to admignistrators
- The unencrypted key is never stored on disk or on backup media, or sent over the network
- Standby servers would need identically-configured cryptographic hardware
- Key backups should be stored securely in a way that standard software can use, e.g., *openssl*
- The examples use *pg_getkey*, but per-user keys could also be used with a more complex permission setup

Create Table and View

```
CREATE EXTENSION IF NOT EXISTS pgcrypto;

DROP TABLE IF EXISTS survey1 CASCADE;

CREATE TABLE survey1 (id SERIAL, username NAME, enc_result BYTEA);

CREATE OR REPLACE VIEW survey1_view AS
SELECT id, username, pgp_sym_decrypt(enc_result,
                             current_setting('pg_getkey.key')) AS enc_result
FROM survey1;
```

Create Trigger

```
CREATE OR REPLACE FUNCTION survey1_view_ins_upd() RETURNS trigger AS $$
BEGIN
    IF (TG_OP = 'INSERT')
    THEN    INSERT INTO survey1 VALUES (DEFAULT, NEW.username,
            pgp_sym_encrypt(NEW.enc_result,
            current_setting('pg_getkey.key')));
    ELSIF (TG_OP = 'UPDATE')
    THEN    UPDATE survey1 SET id = NEW.id, username = NEW.username,
            enc_result = pgp_sym_encrypt(NEW.enc_result,
                                         current_setting('pg_getkey.key'))
        WHERE id = OLD.id;
        IF NOT FOUND
        THEN RETURN NULL;
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER survey1_view_trigger
INSTEAD OF INSERT OR UPDATE ON survey1_view
FOR EACH ROW EXECUTE PROCEDURE survey1_view_ins_upd();
```

Transparent INSERT

```
INSERT INTO survey1_view VALUES (DEFAULT, CURRENT_USER, 'test');

SELECT * FROM survey1_view;
 id | username | enc_result
----+----------+------------
  1 | postgres | test

SELECT * FROM survey1;
 id | username |                       enc_result
----+----------+-------------------------------------------------------------
  1 | postgres | \xc30d040703028c885c43606e062b7bd235011482f275a8d3de4…
```

Transparent UPDATE

```
UPDATE survey1_view SET username = 'user1', enc_result = 'test2'
WHERE enc_result = 'test';

SELECT * FROM survey1_view;
 id | username | enc_result
----+----------+------------
  1 | user1    | test2

SELECT * FROM survey1;
 id | username |                         enc_result
----+----------+---------------------------------------------------------
  1 | user1    | \xc30d0407030242f7c649c9eb708363d236011227744c6b650d
```

Transparent DELETE

```
DELETE FROM survey1_view
WHERE enc_result = 'test2';

SELECT * FROM survey1_view;
 id | username | enc_result
----+----------+-----------

SELECT * FROM survey1;
 id | username | enc_result
----+----------+-----------
```

SELECT Translation

Notice the WHERE clause references the unencrypted value. Internally the SELECT is processed as:

```
EXPLAIN VERBOSE SELECT *
FROM survey1_view
WHERE enc_result = 'test2';
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on public.survey1  (cost=0.00..21.04 rows=3 width=100)
   Output: survey1.id, survey1.username,
           pgp_sym_decrypt(survey1.enc_result,
                           current_setting('pg_getkey.key'::text))
   Filter: (pgp_sym_decrypt(survey1.enc_result,
                            current_setting('pg_getkey.key'::text)) = 'test2'::text)
```

UPDATE Translation

```
EXPLAIN VERBOSE UPDATE survey1_view
SET username = 'user1', enc_result = 'test3'
WHERE enc_result = 'test2';
                              QUERY PLAN
---------------------------------------------------------------------------
 Update on public.survey1_view  (cost=0.00..21.04 rows=3 width=138)
   ->  Seq Scan on public.survey1  (cost=0.00..21.04 rows=3 width=138)
         Output: survey1.id, 'user1'::name, 'test3'::text,
                 ROW(survey1.id, survey1.username,
                     pgp_sym_decrypt(survey1.enc_result,
                     current_setting('pg_getkey.key'::text))),
                 survey1.ctid
         Filter: (pgp_sym_decrypt(survey1.enc_result,
                                  current_setting('pg_getkey.key'::text)) = 'test2'::text)
```

Performance Considerations

- Accessing cryptographic hardware only at server start has minimal performance impact
    - accessing it once per session and caching the key can have a performance impact
    - accessing it for every key access might have an unacceptable performance impact
- Concurrent cryptographic hardware access is controlled by operating system tools
- Yubikey RSA 2048-bit decryption is 40x slower than pure *openssl* software-based decryption
- *openssl* RSA 2048-bit software-based decryption is 2x slower than AES256 decryption (with CPU acceleration)

Indexing Encrypted Data

- Indexing decrypted values cannot be done safely since it would cause unencrypted data to be written to disk.
- Indexing encrypted data requires that all index entries and lookups use the same initialization vector.
- Unfortunately, this causes duplicate values to have identical index entries, causing **possible information leakage.**
- *pgp_sym_encrypt()* uses a different random salt and initialization vector for each encryption.
- Therefore, indexed data must use *encrypt(),* which uses a fixed initialization vector and no salt.
- Encrypted data length can also leak information, though this is not specific to indexes.

Example of Indexing Encrypted Data

```
CREATE TABLE emp (emp_id SERIAL, name TEXT, country BYTEA);

-- bytea fields can be long, so use a hash index
CREATE INDEX i_emp ON emp USING hash (country);

-- i.e., echo -n 'Pakistan' | openssl enc -aes-128-cbc -K pg_getkey.key -iv ''
INSERT INTO emp
VALUES (DEFAULT, 'Andy', encrypt('Pakistan',
        current_setting('pg_getkey.key')::bytea, 'aes'));

SET enable_seqscan = false;

ANALYZE emp;

EXPLAIN SELECT emp_id, name
FROM emp
WHERE country = encrypt('Pakistan',
      current_setting('pg_getkey.key')::bytea, 'aes');
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using i_emp on emp  (cost=0.00..8.02 rows=1 width=9)
   Index Cond: (country = '\xa92e404e54bfc5900c785d4e484bfdd8'::bytea)
```

Data Key Expiration

- Configure the system to store the current and previous data encryption keys
- Store the key version number with the data
- On INSERT, use the current key and store the key version number
- For other operations, try the current and previous keys
- Run a background process to update all the rows that used the previous key to use the current key
    - once complete, remove the previous key

6. Database Encryption Scope

There are two popular uses of encryption in databases:

1. Encryption of data
2. Encryption of keys

Typically, data is encrypted with a symmetric key (1), and the symmetric key is encrypted with a public key and stored (2). The private/decryption/master key can be stored server-side, client-side, or on a network-attached device (e.g., HSM), and can be stored in cryptographic hardware. Data signing is also possible.

Data Encryption Key Storage Options

	Data Encryption Key Storage		
Data Encryption Key Scope	Server File System	Client-Side	Database
Cluster-wide[1]	✓		
User[2]		✓	✓[3]
Database			✓
User/database			✓[3]
Table			✓
Row			✓
Column[4]			✓
Field			✓

1 All database users have the same security access. It blocks access to users with read-only access to the file system and replication. Storage theft is also protected.
2 Data encrypted with per-user keys must be easily identifiable as belonging to that user, e.g., user name column.
3 Stores the encryption is in the the database, but ideally the master key is client-side.
4 It is unclear where to store the encryption key.

Number of Data Encryption Keys

Theoretically you can use a different encryption key for every field. However, practically, anyone with the master key can decrypt the encryption key, so it is really the master key that controls access.
The optimal number of encryption keys is the number of users plus the number of user groups needed to control access. The groups can be:

- predefined, e.g., confidential, top secret
- user-defined, e.g., staff, development
- ad hoc, e.g., allow access to user1, user5, and user12

Predefined Encryption Categories

```
CREATE TABLE user_key (
        username        NAME PRIMARY KEY,
        enc_sym_key     BYTEA,
        confidential    BYTEA,
        secret          BYTEA,
        top_secret      BYTEA
);
```

The same predefined category key, e.g., confidential, is encrypted with each user's public key. A NULL value indicates the user does not have access to that security level. Signing of the encrypted keys would prevent malicious tampering, e.g., changing the key to a known value. Of course, data must be labeled with its security level.

User-Defined Encryption Categories

```
CREATE TABLE group_key (
        groupname           NAME PRIMARY KEY,
        username            NAME[],
        enc_sym_key         BYTEA[]
);
```

The same encryption key is public-key encrypted by each user's public key that is in the group. Applications must identify the group name that encrypted the data and look up their matching username in this table. Data must be labeled with its encryption category.

Ad Hoc Encryption Options

```
CREATE TABLE user_data (
        id                SERIAL PRIMARY KEY,
        enc_sym_data      BYTEA,
        rolename          NAME[],
        enc_sym_key       BYTEA[]
);
```

rolename is a list of user and group names. *enc_sym_key* is the symmetric encryption key encrypted with every role's public key. This does not require data labeling since role names are stored with the data.

7. Private Key Storage Options

While encryption using the public key requires no privileged access, there are several options to store the private key:

- Store in the file system
  - unencrypted
  - encrypted, and require a password to decrypt it
  - encrypted, and require a PIV device and a PIN (and optional touch) to decrypt it

Private Key Storage Options

- Store in dedicated cryptographic hardware:
  - removable PIV card (e.g., CAC) and require a PIN and card reader
  - PIV/USB combined device (e.g., Yubikey 4) and require a PIN and optional touch
  - USB-connected hardware security module (HSM) that can do auditing, complex access control, and store many more keys
  - network-connected HSM, e.g., KMIP
  - external keys allow external (and more secure) logging of key access

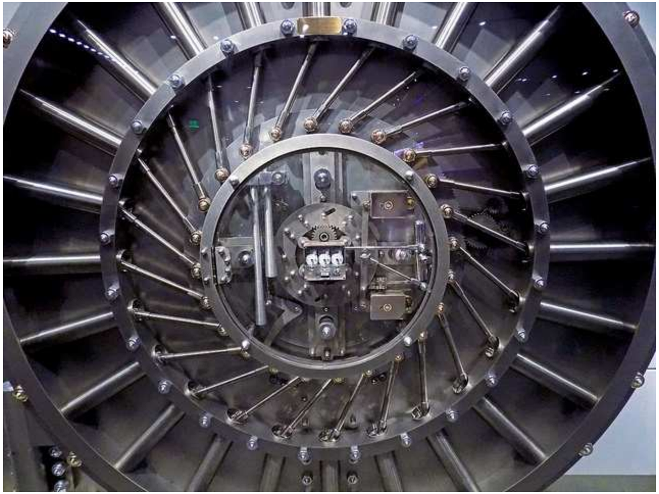
HSM: https://security.stackexchange.com/questions/36664/criteria-for-selecting-an-hsm
https://www.sans.org/reading-room/whitepapers/vpns/overview-hardware-security-modules-757

Encryption Locations

	Encryption Provided		
Encryption Location[1]	Offline Offline Storage[2]	Online Storage/WAL/ Replicas/Backups	Data in Queries/ Logs/Memory
Client-side column encryption	✓	✓	✓
Server-side column encryption	✓	✓	
File system encryption	✓		

1 Assumes secure key storage
2 If the storage is remote and online, it is also effectively encrypted to anyone without access to the server containing the decryption key.

Conclusion



https://momjian.us/presentations

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