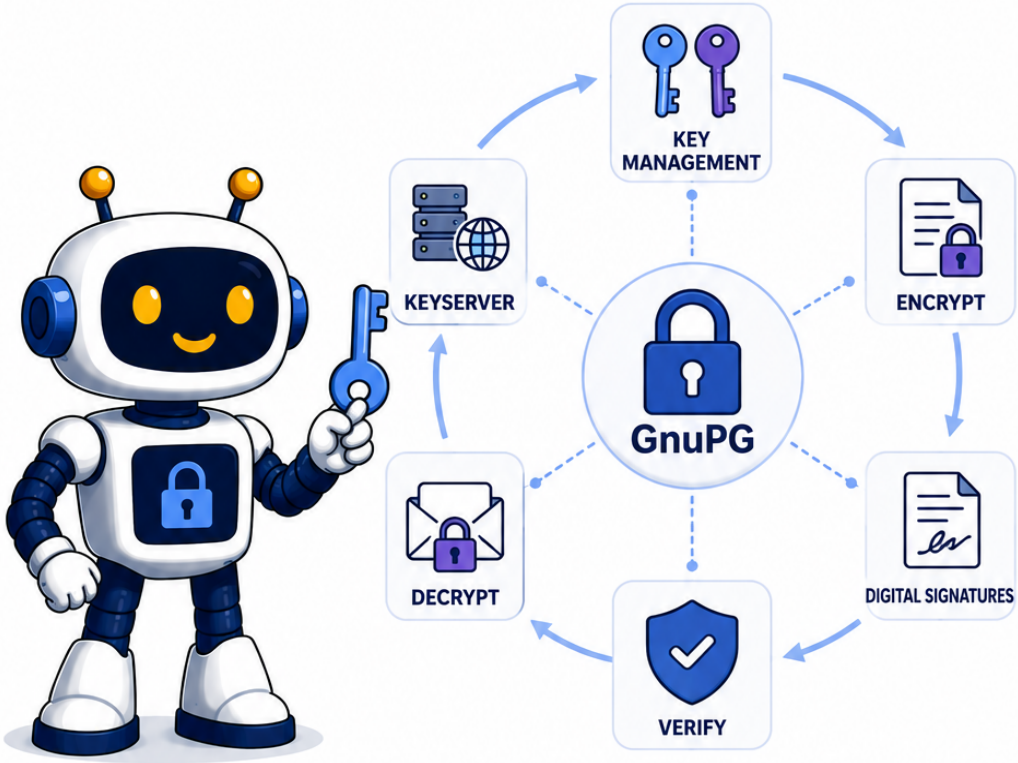
URADICAL

POCKET REFERENCE

GnuPG



Alan Bradley

GnuPG Pocket Reference

Encryption, Signing, and Key Management with GPG 2.x

Alan Bradley

uradical.io

GnuPG Pocket Reference

Alan Bradley

Table of Contents

Chapter 1: Concepts and Key Model

GnuPG (GNU Privacy Guard) is the canonical open-source implementation of the OpenPGP standard (RFC 4880). It provides asymmetric encryption, digital signatures, and a web-of-trust key certification model. The userspace tool is `gpg`; the agent daemon is `gpg-agent`; keyserver and WKD access goes through `dirmngr`.

The Key Hierarchy

Every GnuPG identity is structured as a primary key with one or more subkeys.

Component	Symbol	Purpose
Primary public key	pub	Certify other keys and UIDs; anchor of identity
Primary secret key	sec	Held offline in secure setups; rarely used directly
Subkey (public)	sub	Encryption or signing — separate from primary
Subkey (secret)	ssb	Private half of a subkey
User ID	uid	Name + email bound to the primary key by a self-signature
Fingerprint	fpr	40-character hex fingerprint identifying the primary key

Usage Flags

Each key or subkey carries capability flags shown in brackets after the algorithm.

Flag	Meaning
C	Certify — sign other keys (primary key only)
S	Sign — create data signatures
E	Encrypt — encrypt data to this key
A	Authenticate — SSH or TLS client auth

A typical setup has a primary key with `[C]` only, one `[S]` subkey, and one `[E]` subkey. The primary stays offline; day-to-day operations use subkeys only.

Trust Model

OpenPGP uses a web of trust rather than a certificate authority. Key validity depends on whether you have directly certified it, or whether a chain of trusted signers links you to it.

Trust Level	Meaning
Unknown	No information
Undefined	Not yet assessed
Marginal	Partially trusted to certify others
Full	Fully trusted to certify others
Ultimate	Your own key

A key gains full validity if certified by one fully-trusted key, or by three marginally-trusted keys.

Key ID Formats

Format	Length	Example	Risk
Short	8 hex chars	DEADBEEF	Trivially spoofable — never use
Long	16 hex chars	CAFEBABE DEADBEEF	Collision attacks exist — avoid
Fingerprint	40 hex chars	4F2A...9C1E	Safe — always prefer this

Always identify keys by full fingerprint in scripts, configuration, and communication. Short IDs were demonstrated spoofable at scale in the Evil 32 attack (2016).

Chapter 2: Key Generation and Management

Generate a Key

```
# Interactive — choose algorithm, size, expiry, passphrase
gpg --full-generate-key

# Non-interactive — Ed25519 primary, default subkeys, 1-year
expiry
gpg --quick-generate-key 'Alice Example <alice@example.com>'
ed25519 default 1y

# Add a subkey to an existing primary
gpg --quick-add-key <fingerprint> cv25519 encr 1y
gpg --quick-add-key <fingerprint> ed25519 sign 1y
gpg --quick-add-key <fingerprint> ed25519 auth 1y
```

Recommended algorithm choices:

Use	Algorithm	Notes
Primary (certify)	Ed25519	Fast, small, modern
Signing subkey	Ed25519	Same curve, separate subkey
Encryption subkey	Cv25519	X25519 ECDH — recommended
Auth subkey	Ed25519	For SSH via gpg-agent
Legacy compatibility	RSA 4096	Only when required by recipient tooling

List Keys

```
# Public keyring
gpg --list-keys
gpg --list-keys --with-fingerprint
gpg --list-keys --with-colons          # machine-readable,
colon-delimited

# Secret keyring
gpg --list-secret-keys
gpg --list-secret-keys --with-keygrip  # include gpg-agent
keygrip
```

Edit a Key

```
gpg --edit-key <fingerprint>
```

Common interactive commands inside `--edit-key`:

Command	Action
expire	Change expiry on selected key or subkey
passwd	Change passphrase
adduid	Add a new user ID
deluid	Delete a user ID
addkey	Add a subkey
revkey	Revoke a subkey
trust	Set owner trust level
save	Write changes and exit
quit	Exit without saving

Use `key N` to select subkey N before `expire` or `revkey`.

Non-interactive Expiry Extension

```
gpg --quick-set-expire <fingerprint> 1y
gpg --quick-set-expire <fingerprint> 0   # remove expiry
```

Delete Keys

```
# Delete public key (secret must be deleted first)
gpg --delete-key <fingerprint>

# Delete secret key only
gpg --delete-secret-key <fingerprint>

# Delete both in one step
gpg --delete-secret-and-public-key <fingerprint>
```

Chapter 3: Revocation

Generate a revocation certificate immediately after key creation — before you need it.

Generate a Revocation Certificate

```
gpg --gen-revoke <fingerprint> > revoke.asc
```

Store `revoke.asc` offline, separately from the private key. If your key is compromised or lost, this is your only mechanism to invalidate it publicly.

Apply a Revocation

```
# Import revocation into your local keyring
gpg --import revoke.asc

# Publish to keyserver
gpg --keyserver keys.openpgp.org --send-keys <fingerprint>
```

Once published, revocation propagates through keyserver federation. It cannot be undone. Recipients who refresh their keyring will see the key as revoked.

Revoke a Subkey Only

```
gpg --edit-key <fingerprint>
key 1        # select subkey 1
revkey       # revoke it
save
```

Subkey revocation is the right response when a subkey is compromised but the primary is intact — revoke and replace the subkey without changing your identity.

Chapter 4: Encrypt, Decrypt, and Sign

Encrypt to a Recipient

```
# Binary output (.gpg)
gpg -e -r alice@example.com file.txt

# ASCII-armored output (.asc) — use for email or text channels
gpg -ea -r alice@example.com file.txt

# Multiple recipients
gpg -e -r alice@example.com -r bob@example.com file.txt

# Explicit output filename
gpg -e -r alice@example.com -o encrypted.gpg file.txt

# Encrypt from stdin
cat secret.txt | gpg -e -r alice@example.com -o encrypted.gpg
```

Symmetric Encryption

Symmetric encryption uses a passphrase only — no recipient key required. Useful for backups or sharing with non-GPG users who receive the passphrase via a separate channel.

```
gpg --symmetric file.txt                                    #
AES-128 by default
gpg -c file.txt                                             # short
form
gpg -ca file.txt                                            #
ASCII-armored
gpg --symmetric --cipher-algo AES256 file.txt               #
explicit AES-256
gpg --symmetric --s2k-mode 3 --s2k-count 65011712 \
    --cipher-algo AES256 file.txt                           #
maximum KDF iterations
```

Encrypt and Sign Together

```
gpg -se -r alice@example.com file.txt
gpg -sea -r alice@example.com file.txt  # ASCII-armored
```

The recipient decrypts and verifies authorship in one operation.

Decrypt

```
# Decrypt to stdout
gpg -d file.gpg

# Decrypt to file
gpg -d -o plaintext.txt file.gpg

# Non-interactive symmetric
gpg --batch --passphrase-fd 0 -d file.gpg < passphrase.txt

# Force terminal passphrase prompt (useful when pinentry is
suppressed)
gpg --pinentry-mode loopback -d file.gpg
```

When a file was signed before encryption, `gpg -d` verifies the signature automatically.

Sign Data

```
# Binary signature — produces file.txt.gpg (signature +
content)
gpg -s file.txt

# Clearsign — inline, human-readable, suitable for email
gpg --clearsign file.txt              # produces file.txt.asc

# Detached signature — separate .sig, original file untouched
gpg -b file.txt                       # produces file.txt.sig
gpg -ab file.txt                      # produces file.txt.asc
(ASCII-armored)
```

```
# Specify signing key explicitly
gpg -u <fingerprint> -ab file.txt
```

When to Use Each Signing Format

Format	Command	Use Case
Binary	-s	Signed + compressed archives
Clearsign	--clearsign	Email body text, commit messages
Detached	-b	Release artifacts — sign without modifying the file
Detached ASCII	-ab	Release artifacts on text channels

For software releases, detached ASCII-armored signatures are the standard. Distribute `software-1.0.tar.gz` and `software-1.0.tar.gz.asc` as a pair.

Chapter 5: Verification

```
# Verify a clearsign or inline signature
gpg --verify file.txt.asc

# Verify a detached signature against the original file
gpg --verify file.txt.sig file.txt
gpg --verify file.txt.asc file.txt

# Machine-readable status output — for scripts
gpg --verify --status-fd 1 file.txt.sig file.txt 2>&1
```

Exit Codes

Code	Meaning
0	Signature valid
1	Signature bad — tampered or wrong key
2	Error — key not found, file missing, or other failure

status-fd Tokens

Using `--status-fd 1` emits structured tokens on stdout.

Token	Meaning
GOODSIG	Signature cryptographically valid
BADSIG	Signature invalid
ERRSIG	Signature could not be verified
VALIDSIG	Fingerprint, timestamp, and hash confirmed
KEY_NOT_TRUSTED	Key found but not in your trust model

NO_PUBKEY	Signing key not in your keyring

`VALIDSIG` includes the signing key fingerprint. Matching against it is stricter than `GOODSIG` alone — it confirms the signature came from the exact key you trust, not just any valid key in your ring.

Signature Verification in CI

```
#!/usr/bin/env bash
set -euo pipefail

ARTIFACT="release-1.0.tar.gz"
SIGNATURE="release-1.0.tar.gz.asc"
TRUSTED_FPR="AABBCCDDEEFF00112233445566778899AABBCCDD"

gpg --keyserver keys.openpgp.org --recv-keys "$TRUSTED_FPR"

gpg --verify --status-fd 1 "$SIGNATURE" "$ARTIFACT" 2>&1 \
  | grep -q "VALIDSIG $TRUSTED_FPR" \
  || { echo "Signature verification failed"; exit 1; }

echo "Signature valid"
```

Chapter 6: Key Exchange and Keyservers

Import and Export

```
# Import a public key from file
gpg --import pubkey.asc

# Import a full backup (restores trust database entries)
gpg --import-options restore --import backup.gpg

# Export public key
gpg --export -a <fingerprint> > pubkey.asc

# Export secret key — handle with extreme care
gpg --export-secret-keys -a <fingerprint> > secret.asc

# Export subkeys only (for distribution to daily machines)
gpg --export-secret-subkeys -a <fingerprint> > subkeys.asc

# Full backup including trust data
gpg --export-options backup --export-secret-keys -a >
full-backup.gpg
```

Keyservers

```
# Publish a key
gpg --keyserver keys.openpgp.org --send-keys <fingerprint>

# Fetch a specific key by fingerprint
gpg --keyserver keys.openpgp.org --recv-keys <fingerprint>

# Search by email
gpg --keyserver keys.openpgp.org --search-keys
alice@example.com

# Refresh all keys in your keyring
gpg --refresh-keys
gpg --keyserver keys.openpgp.org --refresh-keys
```

Preferred Keyservers

Keyserver	Notes
keys.openpgp.org	Verifies email before publishing UIDs — preferred
keyserver.ubuntu.com	High availability; no email verification
pgp.mit.edu	Legacy; increasingly unreliable

Avoid `pool.sks-keyservers.net` — deprecated in 2021 following certificate flooding attacks and persistent availability failures.

WKD — Web Key Directory

WKD enables key discovery via HTTPS from the key owner's domain, without a keyserver.

```
# Fetch via WKD automatically
gpg --locate-key alice@example.com

# Explicit WKD fetch
gpg --auto-key-locate wkd --locate-key alice@example.com
```

To publish via WKD, export your key and serve it at the well-known path under your domain as defined by the WKD specification.

Chapter 7: Trust and Certification

Setting Owner Trust

Owner trust governs how much weight you give a keyholder when they certify other keys. It does not directly affect the validity of their key itself.

```
gpg --edit-key <fingerprint>
# then: trust
```

Level	Value	Meaning
Unknown	1	No information
Undefined	2	Not assessed
Marginal	3	Partially trusted to certify others
Full	4	Fully trusted to certify others
Ultimate	5	Your own key

Always set your own primary key to Ultimate after generating.

Certifying Another Key

```
# Local certification — not exportable (certify for your own use)
gpg --lsign-key <fingerprint>

# Exportable certification — can be distributed via keyserver
gpg --sign-key <fingerprint>
```

Only certify after verifying the fingerprint out-of-band — in person, over a verified call, or via a second authenticated channel. A key appearing on a keyserver is not evidence it belongs to the named person.

Revoke Your Certification

```
gpg --edit-key <fingerprint>
# then: revsig
```

Chapter 8: Scripting and Automation

Passing Passphrases

```
# Via a dedicated file descriptor (preferred)
gpg --batch --passphrase-fd 3 -d secret.gpg 3< passphrase.txt

# Via stdin
echo "my passphrase" | gpg --batch --passphrase-fd 0 -d
secret.gpg

# Via environment (least secure — visible in /proc on Linux)
gpg --batch --passphrase "$GPG_PASSPHRASE" -d secret.gpg
```

Prefer `--passphrase-fd` over environment variables. The environment is readable by other processes on the same system via `/proc/<pid>/environ`.

Fixing TTY Errors in Scripts

```
export GPG_TTY=$(tty)
```

Add this to your shell profile and to any script that calls `gpg`. Without it, pinentry cannot attach to the terminal and passphrase prompts silently fail — often producing a misleading "No secret key" error downstream.

Extracting Data from Key Listings

```
# All fingerprints in keyring
gpg --list-keys --with-colons | awk -F: '/^fpr/{print $10}'

# All secret key fingerprints
gpg --list-secret-keys --with-colons | awk -F: '/^fpr/{print
$10}'

# Key ID and UID pairs
```

```
gpg --list-keys --with-colons \
  | awk -F: '/^pub/{id=$5} /^uid/{print id, $10}'

# Check if a fingerprint is present
gpg --list-keys <fingerprint> &>/dev/null && echo "found" ||
echo "not found"
```

Encrypt a Directory

```
# Archive and encrypt
tar -czf - ./secrets/ | gpg -e -r alice@example.com -o
secrets.tar.gz.gpg

# Decrypt and extract
gpg -d secrets.tar.gz.gpg | tar -xzf -
```

Chapter 9: gpg-agent

`gpg-agent` caches passphrases and private key material between operations. It starts automatically on first use and persists across sessions.

```
# Check if agent is running
gpg-connect-agent /bye

# Start agent manually
gpgconf --launch gpg-agent

# Kill agent
gpgconf --kill gpg-agent

# Reload configuration without restart
gpg-connect-agent reloadagent /bye

# List keys cached in agent
gpg-connect-agent "keyinfo --list" /bye

# Flush all cached passphrases
gpg-connect-agent "clear_passphrase --all" /bye
```

SSH Authentication via gpg-agent

`gpg-agent` can replace `ssh-agent` using an `[A]` authentication subkey.

```
# 1. Enable SSH support in ~/.gnupg/gpg-agent.conf
echo "enable-ssh-support" >> ~/.gnupg/gpg-agent.conf
gpg-connect-agent reloadagent /bye

# 2. Get the keygrip of your auth subkey
gpg --list-secret-keys --with-keygrip

# 3. Add keygrip to the sshcontrol file
echo <keygrip> >> ~/.gnupg/sshcontrol

# 4. Point SSH at gpg-agent socket
```

```
export SSH_AUTH_SOCK=$(gpgconf --list-dirs agent-ssh-socket)

# 5. Export GPG auth subkey as SSH public key
gpg --export-ssh-key <fingerprint>
```

Add the `SSH_AUTH_SOCK` export to your shell profile. Add the SSH public key output to `~/.ssh/authorized_keys` on the target host.

Chapter 10: Hardware Tokens and Smartcards

Hardware tokens (Yubikey, Nitrokey, OpenPGP Card) store private key material in tamper-resistant hardware. The private key never leaves the device; GPG operations are performed on-card.

Prerequisites — pcscd

GPG communicates with smartcards via the PC/SC daemon. If `gpg --card-status` returns "No such device" or "card not present", pcscd is either not installed or not running.

```
# Debian/Ubuntu
sudo apt install pcscd
sudo systemctl enable --now pcscd

# Arch
sudo pacman -S ccid
sudo systemctl enable --now pcscd

# macOS — pcscd runs as a system service; if the card is not
detected:
sudo killall -SIGTERM pcscd   # force restart

# Verify the card is visible before proceeding
gpg --card-status
```

On some systems, the Yubikey is claimed by a kernel HID driver before pcscd can access it. If `gpg --card-status` fails despite pcscd running, unplug and replug the token after pcscd is started.

Move Subkeys to a Yubikey

`keytocard` is a destructive, one-way operation — the local private subkey is deleted and replaced with a stub. Back up the full private key

before proceeding.

```
# Back up first
gpg --export-secret-keys -a <fingerprint> > secret-backup.asc

# Enter key editor
gpg --edit-key <fingerprint>

# Move signing subkey
key 1          # select subkey 1 (signing)
keytocard      # choose slot: 1 = signature
key 1          # deselect

# Move encryption subkey
key 2          # select subkey 2 (encryption)
keytocard      # choose slot: 2 = encryption
key 2

# Move authentication subkey
key 3          # select subkey 3 (authentication)
keytocard      # choose slot: 3 = authentication
save
```

After `save`, the local private subkeys become stubs pointing to the card. The primary key is unaffected and remains offline.

If a Yubikey slot already holds a key, `keytocard` will overwrite it without warning. There is no recovery — the previous key on the card is gone.

Card Status and PIN Management

```
# View card info — serial number, key slots, PIN retry counter
gpg --card-status

# Change PINs interactively
gpg --card-edit
# then: passwd
# PIN (default 123456) — required for signing/decryption
# Admin PIN (default 12345678) — required for key management
# Reset Code — used to unblock a locked PIN (optional, not set
by default)
```

Always change default PINs immediately after first use. The PIN retry counter is shown in `gpg --card-status` as `PIN retry counter`. When it reaches 0, the PIN is blocked; use the Admin PIN to unblock it via `gpg --card-edit` . `passwd`. When the Admin PIN counter reaches 0, the card must be factory reset — all keys on the card are destroyed.

Move Subkeys to a Second Machine

After moving subkeys to a Yubikey, setting up a second machine is straightforward — import the public key and the card handles the rest.

```
# On second machine — import your public key
gpg --keyserver keys.openpgp.org --recv-keys <fingerprint>
# or: gpg --import pubkey.asc

# Set trust
gpg --edit-key <fingerprint>
# then: trust -> 5 (ultimate) -> save

# Insert Yubikey — GPG auto-discovers the card and links stubs
gpg --card-status

# Verify subkeys are linked
gpg --list-secret-keys
# sec#  ed25519 ... [C]    <- primary offline
# ssb>  ed25519 ... [S]    <- > means key is on card
# ssb>  cv25519 ... [E]
# ssb>  ed25519 ... [A]
```

The > suffix in `ssb>` confirms the subkey is on the hardware token.

Without a Hardware Token — Subkey Distribution

If you are not using a hardware token, export subkeys only and import on secondary machines. The primary key remains on the airgap.

```
# On primary (airgap) machine
gpg --export-secret-subkeys -a <fingerprint> > subkeys.asc

# On secondary machine
gpg --import subkeys.asc
gpg --edit-key <fingerprint>
# then: trust -> 5 -> save

# Verify — primary is a stub
gpg --list-secret-keys
# sec#  ed25519 ... [C]    <- # means primary is absent
# ssb   ed25519 ... [S]
# ssb   cv25519 ... [E]
```

Chapter 11: Configuration

~/.gnupg/gpg.conf

```
# Always show full fingerprints and long key IDs
keyid-format long
with-fingerprint

# Strong cipher and digest preferences
personal-cipher-preferences AES256 AES192
personal-digest-preferences SHA512 SHA384 SHA256
personal-compress-preferences ZLIB BZIP2 ZIP Uncompressed

# Default preference list written into new self-signatures
default-preference-list SHA512 SHA384 SHA256 AES256 AES192
ZLIB BZIP2 ZIP Uncompressed

# Use SHA-512 for key certifications
cert-digest-algo SHA512

# Suppress version and comment headers in ASCII output
no-comments
no-emit-version

# Cross-certify subkeys (default in 2.1+, but be explicit)
require-cross-certification
```

~/.gnupg/gpg-agent.conf

```
# Cache passphrase for 8 hours; maximum 24 hours
default-cache-ttl 28800
max-cache-ttl 86400

# Pinentry program (adjust path to your system)
pinentry-program /usr/bin/pinentry-curses

# Enable SSH agent emulation
enable-ssh-support
```

~/.gnupg/dirmngr.conf

`dirmngr` handles all keyserver and WKD network access. Configure it when you need proxy support, custom keyservers, or HKPS enforcement.

```
# Prefer HKPS (keyserver over TLS) — never fall back to plain
HKP
hkp-cacert /etc/ssl/certs/ca-certificates.crt

# Set default keyserver
keyserver hkps://keys.openpgp.org

# HTTP proxy for keyserver access (if behind a corporate
proxy)
# http-proxy http://proxy.example.com:3128

# Increase timeout for slow keyservers (seconds)
connect-timeout 30
```

Restart dirmngr after changes:

```
gpgconf --kill dirmngr
```

File Permissions

GPG will refuse to operate if `~/.gnupg` permissions are too open.

```
chmod 700 ~/.gnupg
chmod 600 ~/.gnupg/*
```

Chapter 12: Common Flags Reference

Operational Flags

Flag	Long Form	Description
-e	--encrypt	Encrypt
-d	--decrypt	Decrypt
-s	--sign	Sign (attached)
-b	--detach-sign	Sign (detached)
-a	--armor	ASCII-armored output
-r	--recipient	Specify recipient key
-u	--local-user	Specify signing key
-o	--output	Specify output file
-c	--symmetric	Passphrase-only encryption

Behaviour Flags

Flag	Description
--batch	Non-interactive mode
--yes	Automatically answer yes
--no-tty	Suppress tty interaction
--quiet	Suppress informational output
--status-fd N	Write status tokens to file descriptor N
--passphrase-fd N	Read passphrase from file descriptor N
--pinentry-mode loopback	Force terminal passphrase entry

--keyid-format long	64-bit key IDs in output
--with-fingerprint	Include fingerprint in listings
--with-colons	Machine-parseable colon-delimited output
--with-keygrip	Include keygrip in key listings
--trust-model always	Skip trust checks — dangerous, see Chapter 13

Chapter 13: Operational Gotchas

These are the failure modes that catch practitioners off guard. They are documented here because they are either silent, produce misleading errors, or are hard to find in the official documentation.

The --trust-model always Footgun

`--trust-model always` disables all trust checking and treats every key as fully valid. It is sometimes recommended in tutorials to silence the "no assurance this key belongs to the named user" warning.

Do not use it in production. It silently bypasses the entire validity model and will encrypt to an attacker-supplied key if one is imported. The correct fix is to certify the key with `--lsign-key` and set appropriate owner trust.

gpg-agent Caches Stale Key Material

After replacing a subkey (expiry renewal, compromise recovery), `gpg-agent` can serve the old cached subkey material for the duration of `max-cache-ttl`. Operations will succeed but use the old key.

```
# Force agent to reload key material
gpg-connect-agent reloadagent /bye
# or kill and restart
gpgconf --kill gpg-agent
```

Pinentry Silent Failure in CI

In non-interactive environments, `gpg` silently fails to prompt for a passphrase if `gpg-agent` cannot find a pinentry program. The error manifests downstream as "bad passphrase" or "no secret key" rather than a pinentry error.

Fix: use `--pinentry-mode loopback` combined with `--passphrase-fd` to bypass pinentry entirely in scripts.

```
gpg --batch --pinentry-mode loopback --passphrase-fd 0 -d
file.gpg < passphrase.txt
```

Encrypting to an Untrusted Key Succeeds Without Warning (by Default)

GPG encrypts to any key in your keyring, trusted or not. The output is valid ciphertext. There is no error — only a warning that is easy to miss in script output.

```
# Capture warnings in scripts
gpg -e -r alice@example.com file.txt 2>&1 | tee gpg.log
grep -i "no assurance\|untrusted\|invalid" gpg.log && exit 1
```

Or certify all keys before use and run with `--trust-model pgp` (the default) and rely on the validity model.

Keyring Corruption on Import

Importing a key that was exported without `--export-options backup` loses trust database entries. Importing a revoked or corrupted key can produce inconsistent state.

```
# After any bulk import, check for problems
gpg --check-sigs 2>&1 | grep -E "^(pub|MISSING|ERROR|sig!)"
```

dirmngr Network Failures Are Silent

If `dirmngr` cannot reach a keyserver, `--refresh-keys` and `--recv-keys` fail without a useful error message when called non-interactively. Always check exit codes.

```
gpg --keyserver keys.openpgp.org --recv-keys "$FPR" \
  || { echo "Key fetch failed — check dirmngr and network";
exit 1; }
```

Short Key IDs Are Still in the Wild

Many tutorials, Makefiles, and CI scripts still use 8-character short IDs. These are trivially spoofable: an attacker can generate a key with a matching short ID. The Evil 32 dataset (2016) demonstrated this for all keys on the public keyserver network at the time.

If you find a short ID in your infrastructure, replace it with the full fingerprint.

Chapter 14: Git Integration

Configure Git Signing

```
# Set your signing key
git config --global user.signingkey <fingerprint>

# Sign all commits by default
git config --global commit.gpgsign true

# Sign all tags by default
git config --global tag.gpgsign true

# Sign a specific commit
git commit -S -m "signed commit"

# Sign a tag
git tag -s v1.0.0 -m "release v1.0.0"

# Verify a signed commit
git verify-commit HEAD

# Verify a signed tag
git verify-tag v1.0.0

# Show signature status in log
git log --show-signature
git log --pretty="format:%G? %GK %aN %s"
```

`%G?` shows `G` (good), `B` (bad), `U` (unknown key), `N` (no signature), or `E` (error).

Use SSH Keys for Git Signing

Git 2.34+ supports SSH key signing as an alternative to GPG.

```
git config --global gpg.format ssh
git config --global user.signingkey ~/.ssh/id_ed25519.pub

# Specify allowed signers for verification
git config --global gpg.ssh.allowedSignersFile
~/.ssh/allowed_signers
```

If you are using `gpg-agent` with SSH support enabled, the GPG authentication subkey can serve as the SSH signing key without a separate key pair.

Chapter 15: Common Recipes

Generate a Key with Offline Primary

The recommended production setup keeps the primary (certify-only) key on an air-gapped machine or hardware token, exposing only subkeys day-to-day.

```
# 1. On airgap — generate primary C-only key with no expiry
gpg --quick-generate-key 'Alice <alice@example.com>' ed25519
cert 0

# 2. Capture fingerprint
FPR=$(gpg --list-keys --with-colons alice@example.com | awk
-F: '/^fpr/{print $10; exit}')

# 3. Add subkeys with 1-year expiry
gpg --quick-add-key "$FPR" ed25519 sign 1y
gpg --quick-add-key "$FPR" cv25519 encr 1y
gpg --quick-add-key "$FPR" ed25519 auth 1y

# 4. Generate revocation certificate immediately
gpg --gen-revoke "$FPR" > revoke.asc

# 5. Export subkeys only for daily machine
gpg --export-secret-subkeys -a "$FPR" > subkeys.asc
gpg --export -a "$FPR" > pubkey.asc

# 6. On daily machine — import public key and subkeys
gpg --import pubkey.asc
gpg --import subkeys.asc
gpg --edit-key "$FPR"  # then: trust -> 5 -> save

# 7. Verify
gpg --list-secret-keys
# sec#  ed25519 ... [C]    <- # = primary offline (stub only)
# ssb   ed25519 ... [S]
# ssb   cv25519 ... [E]
# ssb   ed25519 ... [A]
```

Verify a Software Release

```
FPR="<publisher-fingerprint>"
gpg --keyserver keys.openpgp.org --recv-keys "$FPR"
gpg --verify software-1.0.tar.gz.asc software-1.0.tar.gz
```

Encrypt to Self for Backup

```
MY_FPR=$(gpg --list-keys --with-colons | awk -F: '/^fpr/{print
$10; exit}')
gpg -e -r "$MY_FPR" -o secrets-backup.gpg secrets.txt
```

Renew Expiring Subkeys

```
gpg --edit-key <fingerprint>
key 1                          # select signing subkey
expire                         # set new expiry
# enter: 1y
key 1                          # deselect
key 2                          # select encryption subkey
expire
key 2
save

# Re-publish updated key
gpg --keyserver keys.openpgp.org --send-keys <fingerprint>
```

Export Public Key for Publishing

```
gpg --export -a <fingerprint> > alan-bradley.asc
```

Publish this file on your website, in your README, or upload to a keyserver.

Key Transition

When replacing an old key with a new one — because of algorithm upgrade, compromise, or expiry — the correct procedure is a structured transition rather than an abrupt switch. Correspondents need time to update their keyrings.

```
# 1. Generate new key (see Chapter 2)
gpg --full-generate-key

NEW_FPR="<new key fingerprint>"
OLD_FPR="<old key fingerprint>"

# 2. Certify the new key with the old key — proves continuity
of identity
gpg -u "$OLD_FPR" --sign-key "$NEW_FPR"

# 3. Optionally certify the old key with the new key — mutual
cross-certification
gpg -u "$NEW_FPR" --sign-key "$OLD_FPR"

# 4. Publish both keys
gpg --keyserver keys.openpgp.org --send-keys "$OLD_FPR"
gpg --keyserver keys.openpgp.org --send-keys "$NEW_FPR"

# 5. Publish a transition statement — a signed document naming
both fingerprints,
#    signed by both keys, posted on your website and sent to
mailing lists.
#    Sign with old key:
gpg -u "$OLD_FPR" --clearsign transition.txt
#    Then sign the result with new key:
gpg -u "$NEW_FPR" --clearsign transition.txt.asc
```

A transition statement should include: old fingerprint, new fingerprint, reason for transition, date, and contact instructions. Sign it with both keys so correspondents can verify continuity regardless of which key they hold.

Run both keys in parallel for a transition period — typically 3–6 months — accepting messages encrypted to either, signing new messages with the new key only. Set a short expiry on the old key or revoke it once the transition period ends.

Physical Key Backup with paperkey

`paperkey` extracts the secret bytes from a GPG private key export and reduces them to the minimum data needed for reconstruction — suitable for printing and archiving offline. It is not a GPG tool; install it separately.

```
# Export private key and pipe to paperkey
gpg --export-secret-key <fingerprint> | paperkey --output
key-backup.txt

# Print key-backup.txt and store physically (safe deposit box,
etc.)
# To restore: you need the public key and the paperkey output
gpg --export <fingerprint> > pubkey.gpg
paperkey --pubring pubkey.gpg --secrets key-backup.txt | gpg
--import
```

The paperkey output is human-readable hex with checksums per line. It can be transcribed by hand if the printed copy is damaged. Store the public key separately — paperkey output alone is not sufficient for reconstruction.

Chapter 16: Security Notes

Algorithm Reference

Algorithm	Status	Notes
Ed25519 / Cv25519	Recommended	Modern, small, fast — use for new keys
RSA 4096	Acceptable	Legacy compatibility only
RSA 2048	Marginal	Minimum acceptable; prefer 4096
DSA / ElGamal	Avoid	Legacy only
RSA 1024	Never	Broken

Operational Security

Key fingerprint verification. A key on a keyserver is not evidence it belongs to the named person. Always verify fingerprints out-of-band before certifying or encrypting sensitive data. The minimum bar is a voice call; the right bar is in-person comparison or a signed transition statement from a key you already trust.

Revocation certificate handling. A revocation certificate imports and publishes silently. One accidental `gpg --import revoke.asc` followed by a keyserver push permanently invalidates your key with no undo. Store revocation certificates in a separate offline location from the private key — if both are in the same place, losing that location to an attacker means losing your identity and your ability to revoke it simultaneously.

Passphrase and agent exposure. `gpg-agent` caches the decrypted private key in memory. On a shared or multi-user system, the cache

window defined by `max-cache-ttl` is a live exposure window. Set it to the minimum that is operationally tolerable. On a machine that may be left unattended, consider `gpg-connect-agent "clear_passphrase --all" /bye` as part of a screen lock hook.

Export hygiene. A private key export is as sensitive as all the plaintext it has ever protected. Treat `gpg --export-secret-keys` output accordingly: encrypt it before writing to disk or transmitting, and prefer `--export-secret-subkeys` over full key export for day-to-day distribution to secondary machines.

Keyring Hygiene

```
# Check all signatures in keyring for validity
gpg --check-sigs

# Find keys with bad or missing self-signatures
gpg --check-sigs 2>&1 | grep -E "^(pub|MISSING|ERROR)"

# List expired keys
gpg --list-keys --with-colons \
  | awk -F: '/^pub/{split($0,a,":"); if(a[2]=="e") print
a[5]}'
```