Secrets Management for the Workflow Construct

Ronald ‘Ryy’ G. Thomas

2026-05-17

2026-05-17 17:08 PDT

Secrets are state. They deserve the same backup, audit, and recovery discipline as any other state in the workflow construct.

Introduction

A working data scientist's laptop accumulates secrets the way a cluttered desk accumulates pens: silently, in multiple drawers, with no inventory of what is where. By the time the laptop is two years old it holds an AWS access-key pair in `~/.aws/credentials`, an Anthropic API key in `~/.zshrc` exported as a shell variable, an OpenAI key in a project-specific `.env` file, a Zotero API token in a Quarto YAML, a GitHub personal-access token in the keyring, an institutional VPN certificate in `~/.vpn/`, and one or two SSH private keys in `~/.ssh/`. Each was placed there in good faith by a past version of the operator who had a specific task to complete and no inclination to engineer a generic solution. The accumulated state has no inventory, no rotation schedule, no recovery plan if the laptop is lost, and no shared abstraction across the construct's other layers.

The cost of this state of affairs is paid in three predictable failure modes. First, secrets leak: an API key exported in `.zshrc` is sourced into every subshell, including the ones that scrape `env` for diagnostics, and eventually a stack trace or a verbose-logging script prints the variable verbatim. Second, secrets rot: a long-lived AWS access key issued in 2023 is still in `credentials` in 2026, with no automation reminding the operator to rotate it, and no audit trail of which scripts have used it in the meantime. Third, secrets are lost when the laptop is: a fresh machine has no `~/.zshrc`, no `~/.aws/credentials`, and no `~/.env` until the operator manually reconstructs each from memory or from another machine that has not yet been wiped.

We document an open-source three-tier secrets scheme that addresses all three failure modes. The first tier is a single encrypted store (`pass`, the Unix password store) for long-lived API keys and tokens, synchronised across machines via a private git remote. The second tier is short-lived credentials for AWS, obtained via AWS IAM Identity Center (formerly SSO), expiring automatically without operator action. The third tier is the residual safety net of `gitleaks`-at- staging (documented in post 49), which catches the cases where the first two tiers have already failed.

More formally, we document here an extension to the Cloud and File-system layers of the Workflow Construct described in post 52, specifically the secrets-management concern that cuts across both. The recommendation is deliberately open-source: the construct's other layers are auditable down to the source, and the secrets layer should be too.

Motivations

- The laptop's `~/.aws/credentials` file holds long-lived access keys for an IAM user with substantial permissions. If the laptop is stolen, the recovery procedure is not 'log into AWS and rotate' but rather 'log into AWS from a different machine, panic, rotate everything, audit the logs for unauthorised use.'
- API keys exported in `.zshrc` are visible to every process the user starts. A diagnostic script that runs `env | sort` will print the keys to its log; a collaborator who joins a screen-shared session will see them in `printenv` output.
- Multi-machine work (laptop, ThinkPad, EC2 instance, HPC cluster) creates pressure to copy `.aws/credentials` across machines, which doubles or triples the blast-radius of a single key compromise.
- The `zzgit` secret scanner (post 49) catches accidentally-committed secrets at staging time, which is necessary but not sufficient.

- The construct's other layers are open-source and auditable. A proprietary CLI for the secrets layer would be a foreign body in an otherwise transparent stack.

Objectives

The post sets out to deliver:

1. A reproducible installation and configuration of `pass` (the Unix password store) for non-AWS API keys and tokens, synchronised across machines via a private git remote.
2. A walk-through of AWS IAM Identity Center (SSO) configuration that replaces the long-lived `~/.aws/credentials` file with short-lived tokens obtained via `aws sso login`.
3. A small set of secret-injection patterns that move secrets from environment variables to per-process injection.
4. A failure-mode catalogue covering GPG-key loss, sync-conflict resolution, and the macOS Keychain / gpg-agent interaction, plus a documented recovery procedure for a lost laptop.

The reader should be able to install and configure the scheme in approximately two hours: thirty minutes for `pass` and GPG, thirty minutes for AWS IAM Identity Center configuration (one-time, requires admin access), and an hour for per-tool migrations and verification.

Figure 1: Placeholder ambiance image. To be replaced with a screenshot of `pass ls` output showing a small organised hierarchy of API keys under category headings.

What Is the Scheme?

The scheme has three tiers, distinguished by the characteristics of the secrets they hold.

Tier 1: Long-Lived API Keys and Tokens (`pass`)

Most non-AWS secrets are long-lived API keys: the Anthropic key issued at signup, the OpenAI key minted from the dashboard, the GitHub personal-access token created last quarter for a CI pipeline, the Zotero API token copied from the user-preferences page. They have no automatic expiration, are reissued infrequently, and are reused across multiple invocations.

The `pass` store, also known as the Unix password store, is the canonical open-source home for them. It is a thin shell wrapper around GPG: each secret is a small text file encrypted with a user GPG key, stored in `~/.password-store/`, and optionally synchronised to a private git remote. The interface is a small set of verbs: `pass insert anthropic/api-key` to add, `pass anthropic/api-key` to retrieve, `pass git push` to sync. The implementation is approximately 700 lines of bash; the audit surface is small and the behaviour is predictable.

Tier 2: Short-Lived Cloud Credentials (AWS IAM Identity Center)

AWS credentials are different in kind from API keys: they authorise broad permissions and are best replaced rather than guarded. AWS IAM Identity Center (the service formerly known as AWS SSO) issues short-lived tokens by default, configurable per-user, with eight-hour sessions as the conventional default. The tokens are obtained via `aws sso login` and stored under `~/.aws/sso/cache/` rather than in `~/.aws/credentials`; they expire automatically and re-authentication is a single command.

Tier 3: Residual Safety Net (`gitleaks` at Staging)

Tiers 1 and 2 prevent the most common ways secrets reach the working tree. Despite this, secrets occasionally slip through: a credential pasted into a notebook output cell, an SSH private key copied into a project directory for testing, a `.env` file that was supposed to be gitignored but was not. The `gitleaks`-at-staging integration documented in post 49 is the residual safety net. It is a separate post and is not documented here beyond cross-reference.

Prerequisites

The scheme assumes:

- **GPG key.** A user GPG key is required for `pass`. Users who already sign git commits with GPG can reuse the existing key; users without one can generate a fresh key during installation. The key is the single point of failure for the entire encrypted store and warrants careful backup (see Things to Watch Out For below).

- **AWS account with IAM Identity Center enabled.** The identity-center configuration is a one-time, AWS account-level setup that requires admin permissions. For personal AWS accounts the setup is straightforward; for institutional accounts the user may need to coordinate with an administrator.
- **Existing construct setup.** The dotfiles repository (post 24) and the zsh setup (post 01) should be in place. The scheme adds a small block to `.zshrc` and a `~/.password-store/` directory.
- **Private git remote for `pass` sync.** A private repository on a self-hosted forge or a private GitHub/GitLab repository. The pass-store contents are GPG-encrypted before being committed, but the existence of an inventory of encrypted secrets is itself sometimes a privacy concern worth keeping behind authentication.
- **Time investment.** Approximately two hours for the full setup, dominated by the AWS IAM Identity Center configuration (one-time, ~30 minutes) and the migration of existing secrets from `.zshrc` and `.env` files into `pass` (~60 minutes for a moderate-sized inventory).

Installation

Tier 1: `pass` and GPG

The `pass` binary is small (a single bash script plus a manpage) and available in every major package manager.

```
# macOS
brew install pass gnupg pinentry-mac

# Debian-derived Linux
sudo apt install pass gnupg

# Confirm both are present
pass version
gpg --version | head -1
```

Generating a GPG Key

`pass` requires a GPG key to encrypt and decrypt entries. Generate one interactively:

```
gpg --full-generate-key
```

At the prompts, select:

- **Key type:** ECC (sign and encrypt). Ed25519 / Curve 25519 is the correct choice for new keys in 2026: constant-time operations, smaller keys, and equivalent or stronger security than RSA 4096. RSA remains an option but is not recommended for new keys.
- **Curve:** Curve 25519 (default when ECC is selected).

- **Expiration:** 2y (two years) rather than 0. A key that never expires remains valid indefinitely if compromised; a two-year expiration creates a forced review point. Add the renewal date to a calendar immediately.
- **Real name and email:** as one wants them associated with the key.
- **Passphrase:** a minimum of six random words (diceware) or a 25-character random string. The passphrase protects every credential in the store.

If a GPG key already exists (for git commit signing or otherwise), reuse it; the long-form key ID is the input to `pass init`:

```
gpg --list-secret-keys --keyid-format=long
# sec   rsa4096/ABCD1234EFGH5678 2024-01-15 [SC]

pass init ABCD1234EFGH5678
# mkdir: created directory '/Users/zenn/.password-store/'
# Password store initialized for ABCD1234EFGH5678
```

Initialising the git Remote

Initialise the git remote for cross-machine sync. The remote should be a private repository:

```
cd ~/.password-store
pass git init
pass git remote add origin git@github.com:mygit/password-store.git
pass git push -u origin master
```

> ⚠ Warning
>
> Do not use a public git service as the remote for the password store. Even a 'private' repository places credential metadata (entry names, creation dates) and ciphertext under a third party's access controls. Prefer a self-hosted Gitea or Forgejo instance, or a VPS accessed over SSH. See Appendix A for further guidance.

Guard the Store Against Accidental Commits

Before the first `git add` in any dotfiles repository, add `.password-store` and `.gnupg` to its `.gitignore`. After a first commit these entries are non-trivial to reverse (a full `git filter-repo` rewrite plus credential rotation):

```
# In ~/dotfiles/.gitignore, before git init:
.password-store
.gnupg
.ssh
.aws
secrets/*
!secrets/README.md
```

Tier 2: AWS IAM Identity Center

The IAM Identity Center setup is a one-time AWS-account operation. The high-level steps are:

1. In the AWS console, enable IAM Identity Center for the account.
2. Create a permission set (the named bundle of permissions a session will receive).
3. Assign the permission set to the user for the appropriate AWS account.

These three steps are AWS-console-only; AWS's documentation (see See Also) is the canonical reference. After the AWS-side setup is complete, the laptop-side configuration is a single command:

```
aws configure sso
# SSO start URL: https://my-org.awsapps.com/start
# SSO Region: us-west-1
# CLI default client Region [None]: us-west-1
# CLI default output format [None]: json
# CLI profile name [123456789012_PowerUser]: my-profile
```

Subsequent re-authentication is one command:

```
aws sso login --profile my-profile
```

The legacy `~/.aws/credentials` file should be deleted after IAM Identity Center is configured and verified:

```
mv ~/.aws/credentials ~/.aws/credentials.legacy.bak
# After verifying that all scripts work via the SSO profile,
# rotate the legacy keys via the AWS console and remove the backup.
```

Configuration

`pass` Daily Use

The `.zshrc` block to make `pass`-stored secrets reachable on demand is small. The construct convention is to inject secrets at process-start time rather than to export them into the shell environment.

```
# ============================================================
# Secrets injection helpers (post 55)
# ============================================================

# Inject a single pass-stored secret as an environment variable
# for the duration of one command.
# Usage: with-secret VAR_NAME pass/path command...
# Example: with-secret ANTHROPIC_API_KEY anthropic/api-key claude
with-secret() {
```

```
  local var_name="$1"
  local pass_path="$2"
  shift 2
  local value
  value="$(pass "$pass_path")" || {
    echo "with-secret: pass lookup failed for '$pass_path'" >&2
    return 1
  }
  env "$var_name=$value" "$@"
}

# Convenience wrappers for common secret-bearing tools.
claude() {
  with-secret ANTHROPIC_API_KEY anthropic/api-key \
    command claude "$@"
}
```

The `with-secret` pattern is preferred over exporting:

```
# WRONG: visible to every subprocess for the rest of the session
export ANTHROPIC_API_KEY=$(pass anthropic/api-key)
claude

# RIGHT: visible only to the wrapped command
with-secret ANTHROPIC_API_KEY anthropic/api-key claude
```

GPG Agent Caching

`pass` invokes `gpg` for each retrieval, which prompts for the GPG-key passphrase. Caching the passphrase in `gpg-agent` avoids the repeated prompt while limiting exposure to a configurable cache lifetime.

The `~/.gnupg/gpg-agent.conf` file:

```
default-cache-ttl 3600
max-cache-ttl 28800
pinentry-program /opt/homebrew/bin/pinentry-mac   # macOS Apple Silicon
# pinentry-program /usr/local/bin/pinentry-mac    # macOS Intel
# pinentry-program /usr/bin/pinentry-curses       # Linux
```

After editing the config, reload the agent:

```
gpg-connect-agent reloadagent /bye
```

AWS IAM Identity Center: Profile Usage

Once configured, the SSO profile is used the same way as any AWS profile:

```
aws s3 ls --profile my-profile
# Or set for the session:
export AWS_PROFILE=my-profile
```

The token is cached at `~/.aws/sso/cache/` and refreshed automatically by the SDK as long as the cached token is not yet expired. When it expires, run `aws sso login --profile my-profile` to refresh.

Figure 2: Placeholder ambiance image. To be replaced with a screenshot of `aws sso login` opening a browser, with the CLI showing the 'Successfully logged in' message.

Verification

```
# Tier 1: pass
pass version
pass init
pass ls
echo 'verify-only' | pass insert -m verify-temp/test
pass verify-temp/test
pass rm verify-temp/test
pass git status

# Tier 2: AWS SSO
aws sts get-caller-identity --profile my-profile
ls -la ~/.aws/credentials 2>&1
# Should error 'No such file or directory' after migration

# Tier 3: gitleaks (cross-reference post 49)
gitleaks version
```

A green pass on all three blocks indicates the scheme is materialised. A failure on `pass git status` localises to a sync issue; a failure on `aws sts get-caller-identity` localises to token expiration (run `aws sso login`); a failure on `gitleaks` is addressed in post 49.

Daily Workflow

Task	Command	Notes
Add a new API key	`pass insert anthropic/api-key`	Prompts for value (not echoed)
Add a multi-line secret	`pass insert -m ssh/key`	Use `-m` for multi-line input
Retrieve to stdout	`pass anthropic/api-key`	Avoid in pipelines that log
Inject into a command	`with-secret VAR pass/path cmd`	Visible only to `cmd`'s process tree
Copy to clipboard	`pass -c anthropic/api-key`	Clears after 45 seconds
List current store	`pass ls`	Shows hierarchy
Search by name	`pass find anthropic`	Greps the store names
Sync to remote	`pass git push`	Encrypted; safe over public network
Pull from remote	`pass git pull`	Requires GPG private key present
AWS session refresh	`aws sso login --profile my-profile`	Browser-based; fast
AWS-using command	`AWS_PROFILE=my-profile aws ...`	Token resolved automatically

Things to Watch Out For

Six gotchas dominate the failure modes of this scheme.

1. **The GPG private key is the single point of failure for the entire pass store.** If the key is lost, every encrypted secret in `~/.password-store/` is permanently unreadable. Back up the GPG key to two physical locations (one off-site) before adding any secret of consequence: `gpg --export-secret-keys --armor <key-id> > gpg-private-key.asc`. Store the resulting file on an encrypted USB drive.
2. **The `pass` git remote contains encrypted secrets but a plain-text inventory of names.** The contents are encrypted, but the existence of entries like `internal-systems/db-prod-password` is itself sensitive metadata. The remote should be a private repository, ideally on a self-hosted forge.
3. **macOS Keychain and `gpg-agent` can fight over `pinentry`.** Set `pinentry-program /opt/homebrew/bin/pinentry-mac` in `~/.gnupg/gpg-agent.conf`, restart the agent, and ensure `GPG_TTY=$(tty)` is exported in `.zshrc`.
4. **Clipboard managers bypass `pass -c`'s 45-second clear timer.** On a two-laptop setup with iCloud Universal Clipboard enabled, every `pass -c` on one machine appears in plain-text on the other. Any clipboard manager (Alfred, Raycast, ClipMenu, iCloud Universal Clipboard) captures the output before the timer fires and stores it permanently. Disable clipboard history, or exclude `pass` invocations, in any clipboard manager running on the system.
5. **AWS SSO sessions expire silently from the CLI's perspective.** A long-running script that issues an AWS call after the cached token has expired sees an authentication error rather than a refresh prompt. Scripts that run longer than the configured session duration should chunk their work into per-token batches.
6. **Secrets exported in `.zshrc` continue to work even after `pass` is set up.** A migration that adds `pass` but does not remove legacy `export FOO=...` lines results in two copies of each secret coexisting. Audit `.zshrc` for `export.*KEY|export.*TOKEN|export.*SECRET` patterns and replace each with a `with-secret` wrapper before considering the migration complete.

Uninstall / Rollback

```
# Tier 1: remove pass
brew uninstall pass gnupg pinentry-mac   # macOS
rm -rf ~/.password-store                 # CAUTION: irrecoverable

# Tier 2: revert to long-lived AWS credentials
mv ~/.aws/credentials.legacy.bak ~/.aws/credentials

# .zshrc: remove the with-secret block and wrapper functions.
# Restore any legacy 'export FOO=...' lines from version control.
```

```
# GPG: leave intact if used for git commit signing
```

The migration is reversible per tier, but Tier 1's removal step is destructive and irreversible without the GPG-key backup. If the goal is to stop using `pass` while preserving the secrets, export them first (`pass git push` to ensure the remote is current), then archive the local store (`mv ~/.password-store ~/password-store-archive-2026-05-17`) rather than deleting outright.

Figure 3: Placeholder ambiance image. To be replaced with a screenshot of a `with-secret` wrapper invocation showing the secret being injected and the command output, with no trace of the secret in `env` or shell history.

Lessons Learnt

The migration on three machines (the construct's MacBook, a ThinkPad with Linux Mint, and a clean EC2 Ubuntu instance) surfaced lessons in three buckets.

Conceptual

- **Secrets are state, with the same lifecycle concerns as any other state.** They have an authoritative source, they need backup, they require recovery procedures, and they should not be silently duplicated. Treating them as state rather than as configuration is the conceptual shift this scheme operationalises.
- **Short-lived credentials are strictly preferable to long-lived ones, where the issuer supports them.** An eight-hour token that re-authenticates from a SAML identity provider has a smaller blast-radius than a long-lived access key, and the user-experience cost of re-authentication is small once the workflow is habitual.
- **Open-source primitives are appropriate for the secrets layer.** The construct's other layers are auditable down to the source. The secrets layer holds the keys to every other system the operator relies on, which makes it the layer where source-code transparency matters most.
- **Per-process injection beats environment-variable export.** A secret exported in `.zshrc` is visible to every subprocess for the session's lifetime; a secret injected via `with-secret` is visible only to the wrapped command's process tree.

Technical

- **`pass` is a thin wrapper over GPG, and the limits of the wrapper are the limits of GPG.** Multi-recipient encryption works but requires both parties' GPG keys; key rotation (re-encrypting the store with a new key) is documented but non-trivial; offline access requires a GPG-agent cache, not just `pass` itself.
- **The pass-store git history is itself an audit log.** `pass git log` reveals when each secret was added or changed.
- **AWS IAM Identity Center sessions can be extended.** The default eight-hour session is configurable up to twelve hours. For long-running batch jobs, the right pattern is an assume-role chain rather than a single long session.
- **`gpg-agent`'s caching behaviour differs between macOS and Linux.** macOS with `pinentry-mac` integrates with the Keychain for the passphrase entry; Linux with `pinentry-curses` does not.

Gotcha-shaped

- **`pass git push` after `pass insert` does not happen automatically.** Each new secret must be explicitly pushed to the remote. Adding an alias `alias passp='pass git push'` is the simplest mitigation.
- **`with-secret` does not pass through `tty` settings.** Commands that detect 'is my stdin a terminal?' may behave differently inside the wrapper. The cases are rare but real; a `--no-pager` flag may be needed for affected tools.

- **AWS IAM Identity Center's session cache is per-CLI-install.** A second CLI installation (e.g., a Homebrew-managed `awscli` plus a Python venv `awscli`) maintains separate caches. Standardise on one CLI install per machine.

Limitations

- **It depends on GPG, which has its own complexity.** GPG-key generation, backup, rotation, and revocation are non-trivial topics, and `pass`'s usability is upper-bounded by GPG's.
- **AWS IAM Identity Center requires admin access to configure at the account level.** Personal AWS accounts are easy; institutional accounts may require a conversation with an unfamiliar administrator.
- **No hardware-token integration in this revision.** YubiKey-mediated GPG smart-card mode is supported by GPG and `pass` but warrants its own follow-up post. See Appendix B for an overview.
- **No coverage of TOTP / 2FA codes.** `pass-otp` is a popular extension that stores TOTP seeds alongside passwords; the present post does not cover it.
- **The recovery procedure for a lost laptop assumes the GPG-key backup is reachable.** If both the laptop and the USB-encrypted backup are lost simultaneously, the recovery path is long.
- **The scheme is single-user.** Team-scope secret sharing is supported by `pass` via multi-recipient encryption but is not covered here.

Alternatives

`gopass`

`gopass` is a drop-in replacement for `pass` written in Go. It accepts the same `~/.password-store/` layout and GPG-encryption pattern. The case for `gopass` over `pass` is multi-store support and slightly faster start-up time. For solo use, `pass` is sufficient.

`bitwarden-cli` Against Self-Hosted `vaultwarden`

The Bitwarden ecosystem includes an open-source CLI (`bw`) and an open-source server reimplementation (Vaultwarden). The combination gives a 1Password-like UX without the proprietary dependency. The operational cost is running a server (backups, TLS, updates) and the larger trust surface relative to `pass`'s small bash script.

1Password CLI (Proprietary)

The 1Password CLI (`op`) is a closed-source binary under a commercial subscription. For users already paying for 1Password, `op run` is structurally similar to `with-secret`. The construct does not recommend `op` as the primary because the binary is closed-source, the dependency is a paid subscription, and data is held in AgileBits's hosted vault by default.

Opportunities for Improvement

1. **`pass-otp` for TOTP integration.** The `pass-otp` extension stores TOTP seeds alongside passwords, producing current six-digit codes without a separate authenticator app.
2. **YubiKey-mediated GPG smart-card mode.** Moves the GPG private key from `~/.gnupg/` to the YubiKey, so that `pass` decryption requires the physical token.
3. **`pass-import` for migration from other stores.** Imports from 1Password, LastPass, KeePass, Bitwarden, and others.
4. **A construct-aware secrets recovery script.** A `~/bin/recover-secrets.sh` that walks a fresh laptop through GPG-key import, `pass git clone`, and the AWS SSO setup in sequence, with explicit checkpoints and rollback paths.
5. **A team-scope `gopass` follow-up.** Documents the multi-store, multi-recipient pattern for secrets that a small team needs to share, with explicit rotation procedures for when a member leaves.
6. **Hardware-token-backed AWS IAM Identity Center.** Configures IAM Identity Center to require a YubiKey-produced WebAuthn credential at login, reducing the credential-phishing blast-radius for the AWS layer.

Wrapping Up

A laptop accumulates secrets the way a desk accumulates pens: silently, in multiple drawers, with no inventory. The three-tier scheme documented here addresses the inventory problem directly: an encrypted, GPG-keyed, git-synchronised store (`pass`) for long-lived API keys; short-lived tokens via AWS IAM Identity Center for cloud credentials; and `gitleaks`-at-staging as the residual safety net. Each tier has a small, auditable implementation; each is built on open-source primitives; each is reversible without losing the underlying secrets.

In conclusion, four points merit emphasis. First, secrets should be treated as state: they require an inventory, backup, rotation schedule, and recovery procedure. Second, short-lived credentials are preferable to long-lived ones wherever the issuer supports them; AWS IAM Identity Center is the canonical example. Third, secrets should be injected per-process rather than exported: a secret in `.zshrc` is visible to every subprocess, whereas a secret injected via `with-secret` is visible only to the command that needs it. Fourth, open-source primitives are warranted at the secrets layer because the trust requirement is high enough that source visibility is load-bearing.

See Also

- post 22: the AWS CLI setup that this post complements, which requires revision once IAM Identity Center is in place.
- post 24: the workstation-IaC keystone where the `with-secret` helper and wrapper functions live.
- post 25: the version-control layer that GPG commit signing lives alongside.
- post 49: the staging-time secret scanner.
- post 52: the construct framing that names the secrets-management concern this post addresses.

- post 67: the threat-model audit for the full multi-laptop configuration.
- `pass`: https://www.passwordstore.org/
- `gopass`: https://github.com/gopasspw/gopass
- `pass-otp`: https://github.com/tadfisher/pass-otp
- GnuPG: https://gnupg.org/
- AWS IAM Identity Center user guide: https://docs.aws.amazon.com/singlesignon/

Reproducibility

Component	Version	Notes
Operating system	macOS 15 (Sequoia)	primary daily driver
Operating system	Linux Mint 22 (Wilma)	secondary verification
Operating system	Ubuntu 24.04 LTS (EC2)	clean-room verification
Shell	zsh 5.9	minimum 5.0
`pass`	1.7.4	minimum 1.7
GnuPG	2.4.5	minimum 2.2
`pinentry-mac`	1.3.0	macOS only
AWS CLI	2.18	minimum 2.13 for SSO token-cache changes
Optional	`gopass` 1.15	conditional alternative
Optional	`bitwarden-cli` 2024.10	conditional alternative
Recommended companion	`gitleaks` 8.21	post 49 staging scanner

Date of last verification: 2026-05-17.

Appendix A: iOS Sync with Pass for iOS

The Pass for iOS app reads the same GPG-encrypted `.gpg` files that the desktop `pass` command produces. Once the store is in a private git remote, the iOS app can pull and decrypt entries on the device.

Step 1: Confirm the remote is set up:

```
pass git remote -v
pass git push
```

Step 2: Export the GPG key:

```
gpg --export-armor YOUR_KEY_ID > public.asc
gpg --export-secret-keys --armor YOUR_KEY_ID > private.asc
```

Transfer `private.asc` to the iPhone securely (AirDrop is acceptable). Delete the `.asc` files from the desktop immediately after transfer.

> ⚠ Warning
>
> Do not store the exported private key in Dropbox or iCloud unencrypted. Delete it from the Files app on the iPhone after importing into Pass.

Step 3: Configure the iOS app. Install 'Pass - Password Store' from the App Store. In Settings, set the Git Repository URL to your self-hosted remote.

For SSH: the app generates its own SSH key pair. Go to Settings > SSH Key, copy the displayed public key, and add it to `~/.ssh/authorized_keys` on your self-hosted git server. Do not copy your desktop key; the iOS app generates a separate key internally.

Step 4: Import the GPG key and sync. In the app, go to Settings > PGP Key > Import Key, select the `private.asc` file from Files, enter your GPG passphrase when prompted, then tap Sync.

Common snags:

- 'Permission denied (publickey)' on sync: the app's SSH key has not been added to the server's `authorized_keys`. Copy the public key from Settings > SSH Key in the app.
- 'No secret key' decryption error: the fingerprint in the app does not match the `.gpg-id` in the repository. Check both: `cat ~/.password-store/.gpg-id` on the desktop, Settings > PGP Key in the app.

Appendix B: GPG Subkey Architecture and Hardware Tokens

The setup above creates a single key pair used for both certification and encryption. The correct operational model for daily use separates these roles:

- **Master key (certify only):** signs new subkeys and revocation certificates. Generated once, then moved to offline storage or a hardware security key (YubiKey). Never used in daily operation.
- **Encryption subkey:** the only key needed by `pass` for day-to-day operations. Shorter expiry than the master.
- **Signing subkey (optional):** used for `git commit --gpg-sign`.

With this architecture, the master private key never exists on the laptop's disk during normal operation. A compromised laptop exposes only the encryption subkey, which can be revoked by signing a new revocation certificate with the (offline) master key. Implementing the full subkey architecture is beyond this post's scope but is the recommended next step after the basic setup is stable.

FileVault prerequisite: Full-disk encryption (FileVault on macOS) is a prerequisite for any local credential store. The GPG agent caches the decrypted private key in memory; if the machine is lost while unlocked, the cached key is readable without FileVault.

```
fdesetup status
# Expected: FileVault is On.
```

SSH key policy for multi-laptop setups: The correct multi-machine policy is NOT to copy SSH private keys between machines but to generate a new key pair on each machine:

```
ssh-keygen -t ed25519 -C "$(hostname)-$(date +%Y%m)"
```

Then add the new public key to GitHub and servers. A compromised laptop exposes only that machine's key, and revocation is surgical. The GPG private key is the one exception: it is a single identity and must be the same across machines by design. The subkey architecture above limits the blast radius of any single machine compromise.

Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

Feedback is welcome in the following cases:

- An error or a better approach to any of the code in this post.
- Suggestions for topics to cover in future posts.
- Discussion of R programming, data science, or reproducible research.
- Questions about anything in this tutorial.
- A general connection is of interest.

Rendered on 2026-05-17 at 17:08 PDT. Source: ~/prj/qblog/posts/55-secrets-management/secrets-management/analysis/report/index.qmd

---

Related posts in this cluster

This post is part of the *Security, Backup, and Sync* series. Recommended reading order:

1. Post 31: Research Backup Architecture
2. Post 32: Migrating Off Dropbox: Beyond Dotfiles
3. Post 33: Setting Up pass: a Unix Password Manager
4. **Post 34: Secrets Management for the Workflow Construct** (this post)
5. Post 35: Security Foundations for a Multi-Laptop Research Cluster