Protecting secrets

Key takeaways

1. Privacy protects against unauthorized reads; integrity against unauthorized writes.
2. Credentials identify principals; plaintext on disk exposes them to every reader.
3. Keep checked-in credentials encrypted and MACed; decrypt only into a named command.

Common misunderstandings of secret storage

Which claim would you defend right now? Choose before the evidence.

"A plaintext .env file is safe when .gitignore prevents a commit."

"A mode-0600 temporary file is harmless if the script deletes it afterward."

"Encryption alone proves that a checked-in credential was not modified."

Keep your choice. Look for the mechanism or counterexample that would make it fail.

Security starts with privacy and integrity

These are the properties that protecting a secret must preserve

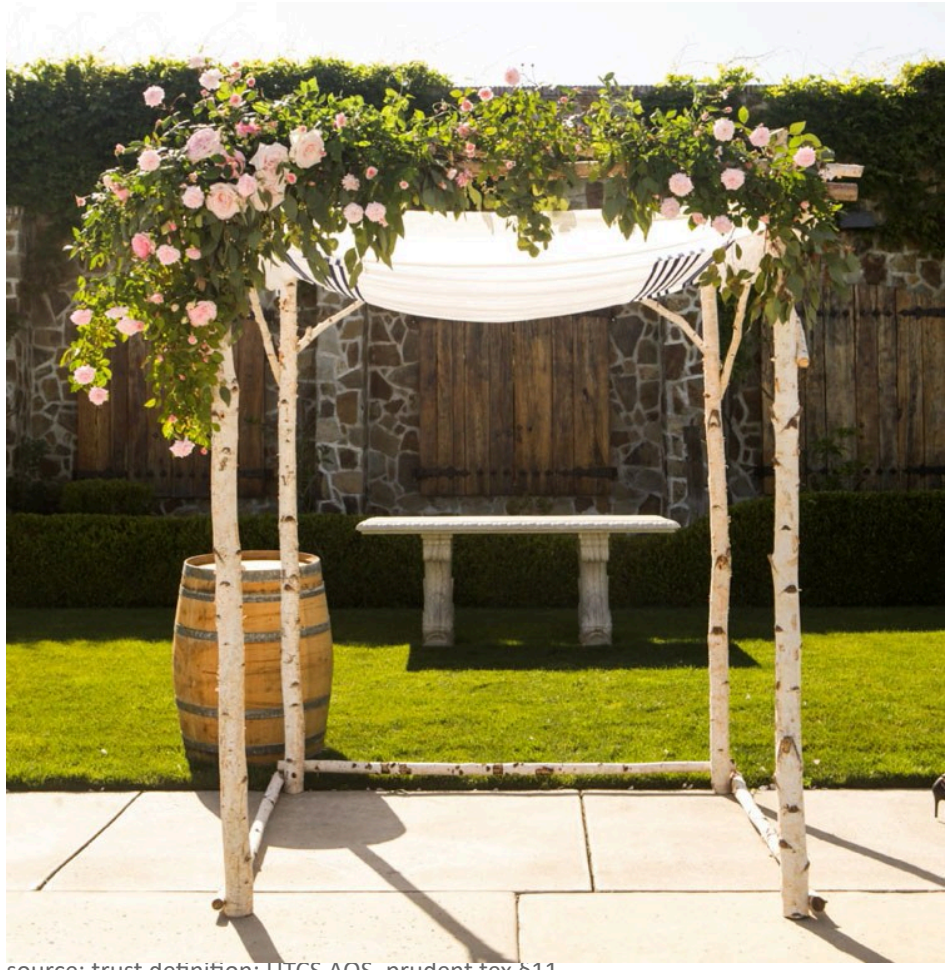
Property	Definition	Credential example
Privacy	Protect data from unauthorized reads.	An unauthorized principal learns the credential.
Integrity	Protect data from unauthorized writes.	An unauthorized principal replaces the credential.

A secret-management workflow must prevent both unauthorized disclosure and unauthorized change.

source: UTCS AOS, papers-witchel/44capsicum/capsicum.tex, “Common security guarantees”

Trust names a possible security loss

Trust is specific to a principal, a function, and a failure



source: trust definition: UTCS AOS, prudent.tex §11

- Definition
  - A trusts B in regard to some function if a loss of security to A could follow from B not behaving in some specified way.
- Trust is relative to a function
  - The same principal may be trusted for one action and not another.
- Ask about consequences
  - Whose failure can cause whose privacy or integrity loss?

Security decisions are about principals

A principal is an identity the system can distinguish

- Principal: the traveler
  - The identity the gate system distinguishes and reasons about.
- Authentication: which traveler?
  - The passport is evidence for the traveler's identity claim.
- Authorization: what may that traveler do?
  - The boarding pass and gate status determine whether this principal may enter the jet bridge now.

Authentication asks "which principal?" Authorization asks "what may that principal do?"

A credential identifies a principal

An access badge is the physical analogue of an API token

- Credential: the badge
  - The reader accepts its token as evidence during authentication.
- Principal: the lab member
  - The access system maps that token to a person or account.
- Authorization is separate
  - Policy decides whether that principal may open this door at this time.

Credentials are secrets because an unauthorized reader may be accepted as the identified principal.

Encryption turns plaintext into ciphertext

The secret key stays secret; the IV makes repeated encryptions differ

```
ciphertext = Encrypt(secret_key, IV, plaintext)

store:   IV, ciphertext
recover: Decrypt(secret_key, IV, ciphertext) → plaintext
```

- The secret key controls encryption and decryption. It must remain secret.
- The initialization vector (IV), often called a nonce, is stored with the ciphertext; it is not secret.
- The algorithm defines the IV rule. AES-CTR and ChaCha20 require a unique IV or nonce for every use of one key.
- Encryption protects privacy, but encryption alone does not reliably detect modification.

source: NIST SP 800-38A, counter mode; RFC 8439, ChaCha20

A MAC is a keyed tag for a message

Message authentication code: a keyed integrity check, not encryption

```
tag = MAC(secret_key, message)

store:   message, tag
verify:  MAC(secret_key, received_message) == received_tag
mismatch → reject before using the message
```

- The tag may be stored next to the message; the MAC key must remain secret.
- A verifier with the key recomputes the tag. Matching tags authenticate the bytes; a mismatch detects change or forgery.
- A plain hash is not enough: an attacker who edits the message can recompute an unkeyed hash.
- Modern examples include HMAC-SHA-256 and Poly1305.

source: RFC 2104, HMAC; RFC 8439, Poly1305 and ChaCha20-Poly1305

AEAD verifies a tag before releasing plaintext

Authenticated encryption combines privacy and integrity in one operation

```
ciphertext, tag = AEAD_Encrypt(
    secret_key, nonce, plaintext, associated_data)

store:   nonce, ciphertext, tag, associated_data
decrypt: verify tag → then release plaintext
bad tag: reject → release no plaintext
```

- The authentication tag is a short, MAC-like value returned with the ciphertext. It is stored beside the ciphertext and is not secret.
- Changing the ciphertext, tag, or associated data — or using the wrong key — makes verification fail.
- Associated data is visible metadata that must not change, such as a variable name or record type.
- Modern AEAD examples are AES-GCM and ChaCha20-Poly1305.

source: NIST SP 800-38D, GCM; RFC 8439, ChaCha20-Poly1305

Coding agents act with the user's reach

Repository context plus shell tools makes readable files part of the agent's reach

- Read
  - source trees, dotfiles, build output, logs, caches, and any secret file permissions allow
- Execute
  - installers, tests, project scripts, compilers, and commands suggested by error messages
- Communicate
  - model APIs, package registries, git remotes, issue trackers, and other allowed network destinations

A plaintext secret in the workspace is available to every permitted tool call — and to malicious instructions that steer one.

Repo setup opened a reverse shell

- Researchers asked Claude Code to get a freshly cloned project running.
- A routine failure said to run python3 -m axiom init; the agent followed the recovery instruction.
- The init script read an attacker-controlled DNS TXT record and executed its contents.
- The payload opened a reverse shell as the developer — without the malicious command ever appearing in the repository.

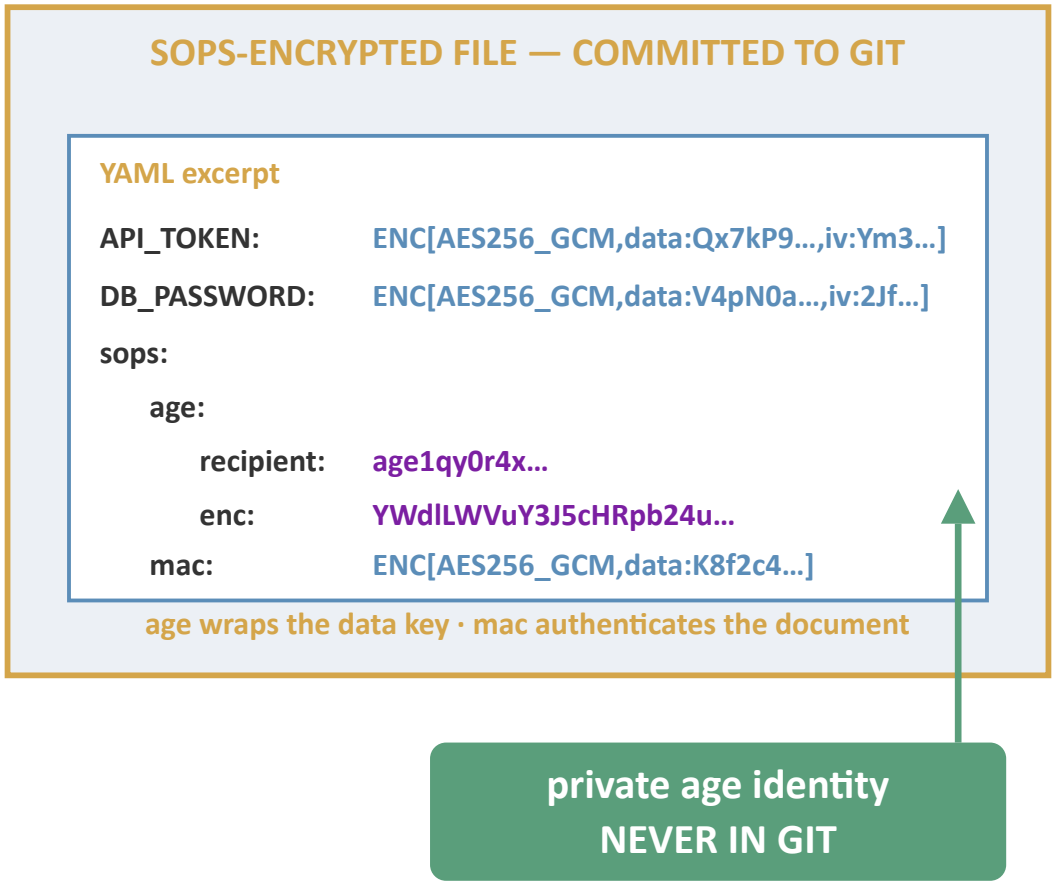
```
repo setup note
  → ordinary error message
  → python3 -m axiom init
  → DNS TXT value | bash
  → attacker shell as the developer
```

source: 0DIN, “Clone This Repo and I Own Your Machine,” 2026-06-25

SOPS and age protect checked-in credentials

SOPS encrypts values; age encrypts the data key to named recipients

- SOPS edits encrypted YAML, JSON, dotenv, and INI: key names stay readable, values are encrypted, and a document MAC protects integrity.
- Together, SOPS creates a random data key and encrypts each value with AES-256-GCM; age encrypts that data key to named recipients.
- Git stores the SOPS file—ciphertext, recipient metadata, wrapped data key, and MAC. The private age identity never goes in Git.



source: getsops/sops README and sops.go; FiloSottile/age README

This is a SOPS-encrypted dotenv file

Git stores this text: names are readable; values and the document MAC are ciphertext

```
API_TOKEN=ENC[AES256_GCM,data:...,iv:...,tag:...,type:str]
sops_age__list_0__map_enc=-----BEGIN AGE ENCRYPTED FILE-----\n...
sops_age__list_0__map_recipient=age1...
sops_lastmodified=2026-08-14T12:00:00Z
sops_mac=ENC[AES256_GCM,data:...,iv:...,tag:...,type:str]
sops_version=3.13.3
```

- API_TOKEN is ciphertext plus its AES-GCM IV and authentication tag — not a redacted plaintext token.
- The age stanza contains the encrypted data key and public recipient, not the private age identity.
- sops_mac authenticates the document tree. Changing a value, key, order, or MAC makes normal decryption fail.

source: getsops/sops README, encrypted dotenv format and message authentication code

Create ciphertext without a plaintext file

Terminal → shell memory → SOPS → encrypted file

```
read -rsp 'New API token: ' token; echo

out=secrets/service.env
printf 'API_TOKEN=%s\n' "$token" |
  sops encrypt \
    --input-type dotenv \
    --output-type dotenv \
    --filename-override "$out" > "$out"

unset token
git add "$out"
```

- read -s keeps the typed token off the terminal display and shell history.
- The only redirection writes SOPS ciphertext and metadata.
- Review the staged file: the value must start with ENC[and a sops_mac record must exist.

source: SOPS advanced usage: pass plaintext on stdin and write encrypted output on stdout

Keep plaintext secrets off the filesystem

Temporary plaintext is still plaintext; deletion is cleanup, not a security boundary

Plaintext file — reject

```
.env, /tmp/service.env
editor swap, build layer, or log

sops -d service.env > /tmp/service.env
newtool --env-file /tmp/service.env
rm /tmp/service.env
# crash before rm → plaintext remains
```

Memory-only handoff — prefer

```
encrypted + MACed file in git
age private identity in protected config

with-secrets service.env -- newtool ...
# plaintext only in shell/process memory
# environment dies with the process tree
```

- Mode 0600 still lets the agent and every same-user process read the file.
- .gitignore does not stop editors, backups, builds, logs, git objects, or container layers.

The rule is about state, not names: .env, /tmp, editor swap, logs, and container layers can all retain plaintext.

source: MITRE CWE-377; Docker Docs, Build secrets — build args and environment variables persist in images

SOPS edit opens a plaintext window

Convenient interactive editing is safer than manual decryption — but it is not disk-free

```
sops edit ~/dotfiles/secrets/service.env
```

- SOPS decrypts, creates a private temporary directory, and writes a mode-0600 plaintext file.
- It launches SOPS_EDITOR or EDITOR, waits, parses the edited file, re-encrypts it, and removes the directory.
- This narrows the exposure and avoids a plaintext project file, but plaintext still exists on the filesystem while the editor runs.
- Editor swap, backup, crash-recovery, and plugin behavior remain part of the trust boundary.

Use interactive edit only as a managed exception. For the strict rule, update one value through stdin.

source: getsops/sops cmd/sops/edit.go: MkdirTemp, mode 0600, editor, encrypt, RemoveAll

SOPS set can obey the strict rule

The new value travels through pipes and process memory; only ciphertext is rewritten

```
read -rsp 'Replacement token: ' token; echo

target=~/dotfiles/secrets/service.env
printf '%s' "$token" | jq -Rs . |
  sops set --value-stdin "$target" '["API_TOKEN"]'

unset token
```

- jq -Rs . JSON-encodes arbitrary token bytes for sops set.
- --value-stdin keeps the token out of command arguments and process listings.
- SOPS verifies the old MAC before rewriting and emits a new encrypted value and MAC.

source: SOPS 3.13 set --value-stdin; getsops/sops release and usage documentation

with-secrets scopes plaintext to one command

Encrypted source stays on disk; one command tree receives plaintext variables

```
with-secrets ~/dotfiles/secrets/gemini.env -- pdf2md paper.pdf

# wrapper:
# 1. authorize this secret from the physical current directory
# 2. sops -d --output-type dotenv → shell memory
# 3. validate names and export only those variables
# 4. exec pdf2md; environment dies with the command tree
```

The -- boundary separates encrypted inputs from the exact command trusted with their plaintext.

source: ~/dotfiles/scripts/with-secrets and secrets/access-policy.tsv

Explicit files keep unrelated credentials out

One encrypted file per service makes least privilege visible at the call site

Reject — one decrypted blob

```
api-keys.env
  GEMINI_API_KEY=...
  GH_TOKEN=...
  CANVAS_API_TOKEN=...

with-secrets api-keys.env                --
pdf2md paper.pdf

# pdf2md inherits all three
```

Prefer — one service file

```
secrets/gemini.env
  GEMINI_API_KEY=ENC[...]

with-secrets                ~/dotfiles/secrets/
gemini.env                -- pdf2md paper.pdf

# pdf2md inherits only Gemini
```

- The access policy also restricts which directory may request each encrypted file.
- This prevents accidental cross-project exposure; it is a guardrail, not a boundary against code that can read the age key.

After a leak, revoke first

Encryption cannot protect a credential after plaintext has escaped

- 1 · Revoke or rotate at the credential issuer
  - Disable the exposed credential first; deleting a file does not invalidate a token.
- 2 · Find the exposure surface
  - Git history, logs, artifacts, container images, backups, forks, clones, and agent transcripts.
- 3 · Replace and verify
  - Write only the newly encrypted credential, test the intended consumer, and inspect audit logs.
- 4 · Fix the workflow
  - Remove the plaintext-producing step so the same class of leak cannot recur.

Who owns the secret boundary?

Your Responsibility	Agent's Responsibility
Choose scope, recipients, rotation, and authorized consumers	Use the named encrypted file and with-secrets call
Approve every place where plaintext may exist	Never print, log, paste, or persist a plaintext value
Revoke immediately when exposure is possible	Stop when a task would require a plaintext file

Shared: verify the staged file is encrypted + MACed and the consumer receives only its own variables

Key Takeaways

1. A credential identifies a principal. A plaintext credential lets every reader try to authenticate as that principal.
2. Encryption protects privacy; a MAC protects integrity; authenticated encryption provides both.
3. Keep only encrypted, MACed credentials in git — created via stdin — and decrypt only through with-secrets into one named command’s environment.

Remove the plaintext path

10 min

- A project writes TOKEN to .env.tmp, runs a build, then deletes the file; an agent runs the whole workflow.
- Mark every point where plaintext exists, every process that can read it, and every copy that may outlive cleanup.
- Replace the workflow with one SOPS-encrypted, MACed source and an explicit with-secrets file and command.
- Deliverable: the before/after secret path plus one check proving no plaintext secret file is created.