

Secrets Management & Cryptographic Hygiene

Learn to find, secure, and manage secrets across your entire stack — and apply AI-driven analysis to detect weak cryptographic implementations before attackers do.

Secrets Scanning

Vault

TLS/SSL

Key Rotation

AI-Assisted Audit

CTF Blue Team

⚠ Educational Use Only

Secrets scanning and cryptographic analysis techniques are for authorized security auditing and CTF competitions. Never exploit discovered credentials without authorization.

Why Secrets Management Matters

Leaked secrets are the number-one cause of preventable breaches. A single exposed API key can compromise an entire organization in minutes.

- **70%+ of breaches** involve compromised credentials (Verizon DBIR)
- **Automated bots** scan GitHub for new commits containing keys within seconds of push
- **Average cost** of a credentials-related breach: \$4.5M+ (IBM Cost of a Data Breach 2024)
- **Mean time to exploit** a leaked AWS key on GitHub: under 5 minutes

Notes: Emphasize that secrets management is not optional — it is a fundamental security control, not an afterthought.

Real Breach Case Studies

Uber (2022)

Hardcoded admin credentials in a PowerShell script on a network share. Attacker gained access to Uber's AWS, GCP, and internal Slack.

Samsung (2022)

Lapsus\$ group extracted 190 GB of source code including private keys, API keys, and secret keys for Samsung Knox authentication.

Toyota (2022)

Access key to customer data server was accidentally committed to a public GitHub repo — exposed for 5 years before discovery.

CircleCI (2023)

Compromised engineer laptop led to stolen session tokens. All customer secrets stored in CircleCI needed rotation.

Anatomy of a Leaked Secret

Here is a typical `.env` file accidentally committed to a public repository:

```
# .env — DO NOT COMMIT THIS FILE
DATABASE_URL=postgresql://admin:S3cur3P@ss!@prod-db.company.io:5432/maindb
AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
STRIPE_SECRET_KEY=sk_live_4eC39HqLyjWDarjtT1zdp7dc
OPENAI_API_KEY=sk-proj-abc123def456ghi789jkl012mno345pqr678
GITHUB_TOKEN=ghp_XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
JWT_SECRET=my-super-secret-jwt-key-dont-tell-anyone
SENDGRID_API_KEY=SG.XXXXXXXXXXXXXXXXXXXXXXXXXX.YYYYYYYYYYYYYYYY
```

Every line is a potential breach vector

Database credentials, cloud provider keys, payment processor secrets, AI API keys, and authentication tokens — all in plain text, all one `git push` away from public exposure.

Secret Sprawl: Where Secrets Hide

Secrets do not stay in one place. They spread across your infrastructure like weeds.

Location	Risk Level	Example
Source code	Critical	Hardcoded connection strings
Git history	Critical	Removed from HEAD but still in commits
.env files	High	Missing <code>.gitignore</code> entry
CI/CD pipelines	High	GitHub Actions secrets in logs
Container images	High	Baked into Docker layers
Config files	Medium	<code>application.yml</code> with passwords
Wikis / docs	Medium	Shared credentials in Confluence

Location	Risk Level	Example
----------	------------	---------

Chat / email

Medium

API keys shared via Slack DMs

Hardcoded Credentials in Code

The most dangerous pattern — secrets embedded directly in application source:

```
# BAD: Hardcoded credentials in Python
import boto3
import requests

# Database connection with embedded password
conn = psycopg2.connect(
    host="prod-db.internal",
    user="admin",
    password="Pr0d_P@ssw0rd_2024!", # NEVER DO THIS
    database="customers"
)

# AWS client with hardcoded keys
s3 = boto3.client(
    "s3",
```

Why this persists

Developers hardcode secrets during prototyping, then forget to remove them. Code review catches some, but git history preserves all.

Secrets in CI/CD Pipelines

CI/CD pipelines are a major attack surface for secret leakage:

```
# BAD: GitHub Actions workflow leaking secrets
name: Deploy
on: push
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to production
        run: |
          # Secret printed to logs via debug output
          echo "Deploying with key: ${ secrets.AWS_SECRET_KEY }"
          # Secret leaked through environment variable dump
          env
          # Secret baked into Docker image layer
          docker build --build-arg API_KEY=${ secrets.API_KEY } -t app .
```

```
# GOOD: Secure secret handling in GitHub Actions
name: Deploy
on: push
jobs:
  deploy:
```

```
runs-on: ubuntu-latest
steps:
  - name: Configure AWS Credentials
    uses: aws-actions/configure-aws-credentials@v4
    with:
      role-to-assume: arn:aws:iam::123456789:role/deploy-role
      aws-region: us-east-1
  - name: Deploy
    run: |
```

Secrets in Docker Images

Docker layer caching preserves secrets even if you delete them in later layers:

```
# BAD: Secret persists in intermediate layer
FROM python:3.11
COPY .env /app/.env
RUN pip install -r requirements.txt
RUN python setup.py --config /app/.env
RUN rm /app/.env    # TOO LATE - .env is in a previous layer!
```

```
# GOOD: Use multi-stage builds and runtime secrets
FROM python:3.11 AS builder
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

FROM python:3.11-slim
COPY --from=builder /usr/local/lib/python3.11/site-packages /usr/local/lib/python3.11/site-packages
COPY app/ /app/
# Secrets injected at runtime via environment or mounted volumes
CMD ["python", "/app/main.py"]
```

Extraction is trivial

`docker history --no-trunc` and `docker save | tar -x` can recover secrets from any layer in the image.

Secret Scanning Tools

Tool	Type	Strengths
TruffleHog	Git history scanner	Scans entire git history, high-entropy detection, verified secrets
git-secrets	Pre-commit hook	Prevents commits containing secrets, AWS-focused patterns
Gitleaks	Static analysis	Fast, configurable rules, CI/CD integration
detect-secrets	Baseline scanner	Yelp's tool, maintains baseline of known secrets
GitHub Secret Scanning	Platform native	Automatic alerts, partner program for key revocation
GitGuardian	SaaS platform	Real-time monitoring, remediation workflows

Notes: No single tool catches everything. A layered approach — pre-commit hooks + CI scanning + monitoring — provides the best coverage.

TruffleHog in Action

TruffleHog scans git history for high-entropy strings and known secret patterns:

```
# Install TruffleHog
pip install trufflehog3
# or use the Go version
brew install trufflehog

# Scan a local repository (entire git history)
trufflehog git file:///path/to/repo --only-verified

# Scan a GitHub org
trufflehog github --org=mycompany --only-verified

# Scan with JSON output for CI integration
trufflehog git file:///path/to/repo --json | jq '.SourceMetadata'

# Example output:
```

Verified vs. Unverified

TruffleHog can actively verify if discovered credentials are still valid by making test API calls. The

`--only-verified` flag reduces false positives dramatically.

git-secrets: Prevent Before You Push

The best secret leak is the one that never happens. Pre-commit hooks stop secrets at the source:

```
# Install git-secrets
brew install git-secrets

# Initialize in your repository
cd /path/to/repo
git secrets --install

# Register AWS patterns
git secrets --register-aws

# Add custom patterns
git secrets --add 'sk_live_[a-zA-Z0-9]{24}'          # Stripe live keys
git secrets --add 'sk-proj-[a-zA-Z0-9]{48}'          # OpenAI keys
git secrets --add 'ghp_[a-zA-Z0-9]{36}'              # GitHub PATs
git secrets --add 'password\s*=\s*.*+'                # Generic password assignments
```

```
# .pre-commit-config.yaml - team-wide enforcement
repos:
  - repo: https://github.com/awslabs/git-secrets
    rev: master
```

```
hooks:
```

```
- id: git-secrets
```

Gitleaks: Configurable Secret Detection

```
# Run Gitleaks against a repo
gitleaks detect --source /path/to/repo --verbose

# Scan only staged changes (pre-commit)
gitleaks protect --staged --verbose

# CI/CD mode with SARIF output
gitleaks detect --source . --report-format sarif --report-path gitleaks.sarif
```

```
# .gitleaks.toml - custom rules
title = "Custom Gitleaks Config"

[[rules]]
id = "internal-api-key"
description = "Internal API Key"
regex = '''INTERNAL_API_[A-Z]+=\\S+'''
tags = ["internal", "api"]

[[rules]]
id = "jwt-secret"
description = "JWT Secret"
regex = '''(?i)(jwt[_-]?secret|jwt[_-]?key)\\s*[=:]\\s*\\S+'''
tags = ["auth", "jwt"]
```


Using AI to Scan for Leaked Secrets

Traditional regex-based scanners miss context. AI models can understand *intent* and detect secrets that patterns cannot:

```
import anthropic

client = anthropic.Anthropic()

def ai_secret_scan(code_snippet: str) -> dict:
    """Use Claude to analyze code for potential secret leakage."""
    response = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=1024,
        messages=[{
            "role": "user",
            "content": f"""Analyze this code for leaked secrets, hardcoded credentials,
or sensitive data exposure. For each finding, provide:
1. The secret type (API key, password, token, etc.)
2. The exact line or value"""
```

AI advantages over regex

AI can detect obfuscated secrets, base64-encoded credentials, secrets split across variables, and contextually sensitive values (e.g., "this looks like a real key, not a placeholder").

AI Contextual Analysis: Beyond Pattern Matching

AI can distinguish real secrets from test fixtures, examples, and documentation:

```
SCAN_PROMPT = """You are a security auditor. Analyze this file for secrets.
Classify each finding as:
- TRUE_POSITIVE: Real secret that needs remediation
- FALSE_POSITIVE: Test data, example, or placeholder
- SUSPICIOUS: Needs human review

Consider:
- Is this a test file? Check filename and content patterns.
- Does the value look like a real key or a placeholder (e.g., 'xxx', 'your-key-here')?
- Is the entropy high enough to be a real secret?
- Is this in a comment explaining what NOT to do?

File: {filename}
Content:
```
```

# HashiCorp Vault: Centralized Secret Management

Vault provides a single source of truth for secrets with access control, audit logging, and automatic rotation:

```
Start Vault in dev mode (for learning only)
vault server -dev

Store a secret
vault kv put secret/myapp/database \
 username="app_user" \
 password="$(openssl rand -base64 32)" \
 host="prod-db.internal"

Read a secret
vault kv get secret/myapp/database

Read specific field
vault kv get -field=password secret/myapp/database
```

```
Vault policy — principle of least privilege
path "secret/data/myapp/*" {
```

```
 capabilities = ["read"]
}

path "secret/data/myapp/database" {
 capabilities = ["read"]
 allowed_parameters = {
 "version" = []
 }
}

Deny access to admin secrets
path "secret/data/admin/*" {
```

---

# Vault Dynamic Secrets

Dynamic secrets are generated on-demand and automatically revoked after a TTL. No more shared, long-lived credentials:

```
Enable database secrets engine
vault secrets enable database

Configure PostgreSQL connection
vault write database/config/mydb \
 plugin_name=postgresql-database-plugin \
 connection_url="postgresql://{{username}}:{{password}}@db:5432/app" \
 allowed_roles="app-readonly" \
 username="vault_admin" \
 password="vault_admin_pass"

Create a role with TTL
vault write database/roles/app-readonly \
 db_name=mydb \
 creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD '{{password}}' \
```

# AWS Secrets Manager & SOPS

## AWS Secrets Manager

```
import boto3
import json

client = boto3.client("secretsmanager")

Retrieve secret at runtime
response = client.get_secret_value(
 SecretId="prod/myapp/database"
)
secret = json.loads(
 response["SecretString"]
)
db_password = secret["password"]

Enable automatic rotation
```

## Mozilla SOPS

```
Encrypt a config file
sops --encrypt \
 --kms arn:aws:kms:us-east-1:123:key/abc \
 secrets.yaml > secrets.enc.yaml

Encrypted file (safe to commit!)
myapp:
db_password: ENC[AES256_GCM,data:...,type:str]
api_key: ENC[AES256_GCM,data:...,type:str]

Decrypt at deploy time
sops --decrypt secrets.enc.yaml

Edit encrypted file in-place
sops secrets.enc.yaml
```

SOPS encrypts values but preserves keys, making diffs readable in git.

# Cryptographic Best Practices: Algorithm Selection

| Use Case               | Recommended                                            | Avoid                                                  |
|------------------------|--------------------------------------------------------|--------------------------------------------------------|
| Symmetric encryption   | AES-256-GCM, ChaCha20-Poly1305                         | DES, 3DES, RC4, AES-ECB                                |
| Password hashing       | Argon2id, bcrypt, scrypt                               | MD5, SHA-1, SHA-256 (unsalted)                         |
| Digital signatures     | Ed25519, ECDSA (P-256), RSA-4096                       | RSA-1024, DSA                                          |
| Key exchange           | X25519, ECDH (P-256)                                   | DH-1024, static RSA                                    |
| Hashing (non-password) | SHA-256, SHA-3, BLAKE3                                 | MD5, SHA-1                                             |
| Random generation      | <code>secrets</code> module, <code>/dev/urandom</code> | <code>random</code> module, <code>Math.random()</code> |

The golden rule

Never implement your own cryptographic algorithms. Use well-tested libraries: `cryptography` (Python), `libsodium`, or platform-provided APIs.

---

# Common Cryptographic Mistakes

```
BAD: Using ECB mode (patterns visible in ciphertext)
from Crypto.Cipher import AES
cipher = AES.new(key, AES.MODE_ECB) # NEVER use ECB

BAD: Using MD5 for password hashing
import hashlib
password_hash = hashlib.md5(password.encode()).hexdigest()

BAD: Predictable random for tokens
import random
token = ''.join(random.choices('abcdef0123456789', k=32))

BAD: Hardcoded IV / nonce reuse
cipher = AES.new(key, AES.MODE_GCM, nonce=b'\x00' * 12) # Static nonce!
```

```
GOOD: Proper implementations
from cryptography.fernet import Fernet
key = Fernet.generate_key()
f = Fernet(key)
ciphertext = f.encrypt(b"sensitive data")

GOOD: Password hashing with Argon2
from argon2 import PasswordHasher
```

```
ph = PasswordHasher(time_cost=3, memory_cost=65536, parallelism=4)
hash = ph.hash("user_password")
ph.verify(hash, "user_password") # Returns True or raises exception

GOOD: Cryptographic random
import secrets
```

---

# Key Management Lifecycle

Every cryptographic key must follow a defined lifecycle:

1. **Generation** — Use cryptographically secure random number generators; appropriate key length
2. **Distribution** — Secure transport (TLS, out-of-band); never email or chat keys
3. **Storage** — HSMs, KMS, or Vault; never in source code or plain-text config
4. **Usage** — Enforce least-privilege access; separate encryption and signing keys
5. **Rotation** — Regular schedule (30-90 days); automated via secrets manager
6. **Revocation** — Immediate on compromise; CRLs or OCSP for certificates
7. **Destruction** — Cryptographic erasure; verify all copies are removed

Rotation policy guidelines

API keys: 90 days. Database passwords: 30 days. TLS certificates: annually (or shorter with ACME). Signing keys: annually with overlap period for verification.

---

# Automated Key Rotation

```
"""Automated key rotation script using Vault + AWS Secrets Manager"""
import boto3
import hvac
import logging
from datetime import datetime, timedelta

logger = logging.getLogger(__name__)

def rotate_database_credentials(vault_client: hvac.Client, secret_path: str):
 """Rotate database credentials via Vault dynamic secrets."""
 # Read current lease
 current = vault_client.secrets.kv.v2.read_secret_version(path=secret_path)
 lease_id = current.get("lease_id")

 # Revoke old credentials
```

---

# TLS/SSL Configuration: Getting It Right

A misconfigured TLS setup can be worse than no encryption — it provides a false sense of security.

```
GOOD: Strong Nginx TLS configuration
server {
 listen 443 ssl http2;
 server_name app.example.com;

 # Certificate and key
 ssl_certificate /etc/letsencrypt/live/app.example.com/fullchain.pem;
 ssl_certificate_key /etc/letsencrypt/live/app.example.com/privkey.pem;

 # Protocol versions – TLS 1.2 and 1.3 only
 ssl_protocols TLSv1.2 TLSv1.3;

 # Strong cipher suites (TLS 1.2)
 ssl_ciphers 'ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384';
 ssl_prefer_server_ciphers on;
```

# TLS Misconfigurations to Detect

| Misconfiguration              | Risk                               | Detection                                   |
|-------------------------------|------------------------------------|---------------------------------------------|
| SSLv3 / TLS 1.0 / 1.1 enabled | POODLE, BEAST attacks              | <code>nmap --script ssl-enum-ciphers</code> |
| Weak cipher suites            | RC4, DES, NULL ciphers             | <code>testssl.sh</code> , SSL Labs          |
| Expired certificates          | Browser warnings, MITM risk        | <code>openssl s_client</code>               |
| Self-signed in production     | No chain of trust                  | Certificate chain validation                |
| Missing HSTS                  | SSL stripping attacks              | HTTP response header check                  |
| Certificate key too small     | RSA < 2048 bits                    | <code>openssl x509 -text</code>             |
| Wildcard cert overuse         | Larger blast radius if compromised | Certificate inventory audit                 |

```
Quick TLS audit with testssl.sh
testssl.sh --severity HIGH https://app.example.com

Check certificate expiry
echo | openssl s_client -connect app.example.com:443 2>/dev/null | \
 openssl x509 -noout -dates -subject -issuer
```

```
Check supported protocols and ciphers
nmap --script ssl-enum-ciphers -p 443 app.example.com
```

---

# Certificate Lifecycle Management

Manual certificate management leads to outages. Automate the entire lifecycle:

```
Automated certificate issuance with Let's Encrypt + certbot
certbot certonly --nginx -d app.example.com --non-interactive --agree-tos

Auto-renewal (crontab)
0 0 1 * * certbot renew --post-hook "systemctl reload nginx"
```

```
"""AI-assisted certificate monitoring and alerting."""
import ssl
import socket
from datetime import datetime, timedelta

def check_cert_expiry(hostname: str, port: int = 443) -> dict:
 """Check certificate expiry and return status."""
 context = ssl.create_default_context()
 with socket.create_connection((hostname, port)) as sock:
 with context.wrap_socket(sock, server_hostname=hostname) as ssock:
 cert = ssock.getpeercert()
 expiry = datetime.strptime(cert['notAfter'], '%b %d %H:%M:%S %Y %Z')
 days_left = (expiry - datetime.utcnow()).days
 return {
 "hostname": hostname,
```

---

# AI-Assisted Certificate Automation

Use AI to automate certificate inventory, detect anomalies, and recommend actions:

```
CERT_AUDIT_PROMPT = """Analyze this certificate inventory and provide recommendations:

{cert_inventory}

For each certificate, evaluate:
1. Is the key size adequate (RSA >= 2048, ECDSA >= 256)?
2. Is the signature algorithm secure (not SHA-1)?
3. Is the certificate close to expiry (< 30 days)?
4. Are there any wildcard certs that should be replaced with specific ones?
5. Is the issuer a trusted CA?
6. Are Subject Alternative Names (SANs) appropriately scoped?

Provide a prioritized remediation plan."""

def ai_cert_audit(cert_data: list[dict]) -> str:
```

# AI-Assisted Crypto Audit

Use AI to identify weak cryptographic implementations in your codebase:

```
CRYPTO_AUDIT_PROMPT = """You are a cryptography security auditor. Analyze this code for:
```

1. **Weak algorithms**: MD5, SHA-1, DES, RC4, ECB mode
2. **Improper key management**: hardcoded keys, insufficient key length
3. **Nonce/IV issues**: reuse, predictable values, static nonces
4. **Random number generation**: using non-CSPRNG for security purposes
5. **Password storage**: unsalted hashes, fast hashing algorithms
6. **Padding oracle risks**: CBC without HMAC, custom padding
7. **Missing authentication**: encryption without integrity (AES-CBC without MAC)

```
For each finding, provide:
```

- Severity: CRITICAL / HIGH / MEDIUM / LOW
  - The vulnerable code line
  - The specific weakness
  - A corrected implementation
-

# Full Codebase Secrets & Crypto Audit

```
"""Comprehensive AI-powered security audit pipeline."""
import os
import subprocess
import json

def full_audit(repo_path: str) -> dict:
 results = {"secrets": [], "crypto": [], "tls": []}

 # Phase 1: Automated secret scanning with Gitleaks
 gitleaks_out = subprocess.run(
 ["gitleaks", "detect", "--source", repo_path, "--report-format", "json",
 "--report-path", "/tmp/gitleaks.json"],
 capture_output=True, text=True
)
 if os.path.exists("/tmp/gitleaks.json"):
```

# Proper .env and Configuration Management

## What to commit

```
.env.example (committed)
DATABASE_URL=
AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
STRIPE_SECRET_KEY=
OPENAI_API_KEY=
JWT_SECRET=
```

```
.gitignore (committed)
.env
.env.local
.env.production
*.pem
*.key
credentials.json
service-account.json
```

## What NOT to commit

```
.env (NEVER committed)
DATABASE_URL=postgresql://...
AWS_ACCESS_KEY_ID=AKIA...
AWS_SECRET_ACCESS_KEY=wJal...
STRIPE_SECRET_KEY=sk_live_...
OPENAI_API_KEY=sk-proj-...
JWT_SECRET=real-secret-here
```

```
Load at runtime
from dotenv import load_dotenv
import os

load_dotenv()
db_url = os.environ["DATABASE_URL"]
Fails fast if not set
```



# Defense in Depth: Layered Secrets Protection

No single control is sufficient. Layer your defenses:

1. **Prevention** — Pre-commit hooks ( `git-secrets`, `gitleaks protect` )
2. **Detection** — CI/CD pipeline scanning (TruffleHog, Gitleaks in CI)
3. **Monitoring** — Continuous scanning (GitGuardian, GitHub Secret Scanning)
4. **Centralization** — Vault / Secrets Manager for all secret storage
5. **Rotation** — Automated rotation with zero-downtime deployment
6. **AI augmentation** — Contextual analysis to reduce false positives and catch evasions
7. **Incident response** — Automated revocation playbooks when leaks are detected

The 5-minute rule

If a secret is leaked, you have approximately 5 minutes before automated scrapers find and exploit it. Your revocation process must be faster than that.

---

# Incident Response: When Secrets Leak

```
"""Automated incident response for leaked secrets."""

RESPONSE_PLAYBOOK = {
 "aws_key": {
 "revoke": "aws iam delete-access-key --access-key-id {key_id} --user-name {user}",
 "audit": "aws cloudtrail lookup-events --lookup-attributes AttributeKey=AccessKeyId,AttributeVa",
 "rotate": "aws iam create-access-key --user-name {user}",
 },
 "github_token": {
 "revoke": "gh api -X DELETE /user/installations/{installation_id}/tokens",
 "audit": "gh api /user/audit-log",
 },
 "stripe_key": {
 "revoke": "# Revoke via Stripe Dashboard > API Keys > Roll Key",
 "audit": "stripe_events_list --type=charge.succeeded --limit=100".
```

# HSTS: HTTP Strict Transport Security

HSTS tells browsers to only connect via HTTPS, preventing SSL stripping attacks:

```
Check if HSTS is set
curl -sI https://app.example.com | grep -i strict-transport

Proper HSTS header
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

| Directive         | Purpose                             | Recommendation                      |
|-------------------|-------------------------------------|-------------------------------------|
| max-age           | How long to enforce HTTPS (seconds) | 31536000 (1 year) minimum           |
| includeSubDomains | Apply to all subdomains             | Use if all subdomains support HTTPS |
| preload           | Submit to browser preload list      | Use for maximum protection          |

HSTS Preload warning

Once on the preload list, removal takes months. Ensure *all* subdomains support HTTPS before enabling `includeSubDomains` with `preload`.

---

# Strong Cipher Suite Selection

Choose cipher suites that provide forward secrecy and authenticated encryption:

```
TLS 1.3 cipher suites (all are strong — no configuration needed)
TLS_AES_256_GCM_SHA384
TLS_CHACHA20_POLY1305_SHA256
TLS_AES_128_GCM_SHA256

TLS 1.2 recommended cipher suites (order matters)
ECDHE-ECDSA-AES256-GCM-SHA384
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-ECDSA-CHACHA20-POLY1305
ECDHE-RSA-CHACHA20-POLY1305
ECDHE-ECDSA-AES128-GCM-SHA256
ECDHE-RSA-AES128-GCM-SHA256
```

## Required properties

- **ECDHE** — Forward secrecy
- **GCM/POLY1305** — AEAD (authenticated encryption)

## Reject these

- **RC4** — Biased output, broken
- **CBC** — Padding oracle risks
- **NULL** — No encryption at all

- **AES-128+** — Strong symmetric cipher

- **EXPORT** — Intentionally weak
-

# Secure TLS in Python Applications

```
"""Proper TLS configuration for Python applications."""
import ssl
import certifi

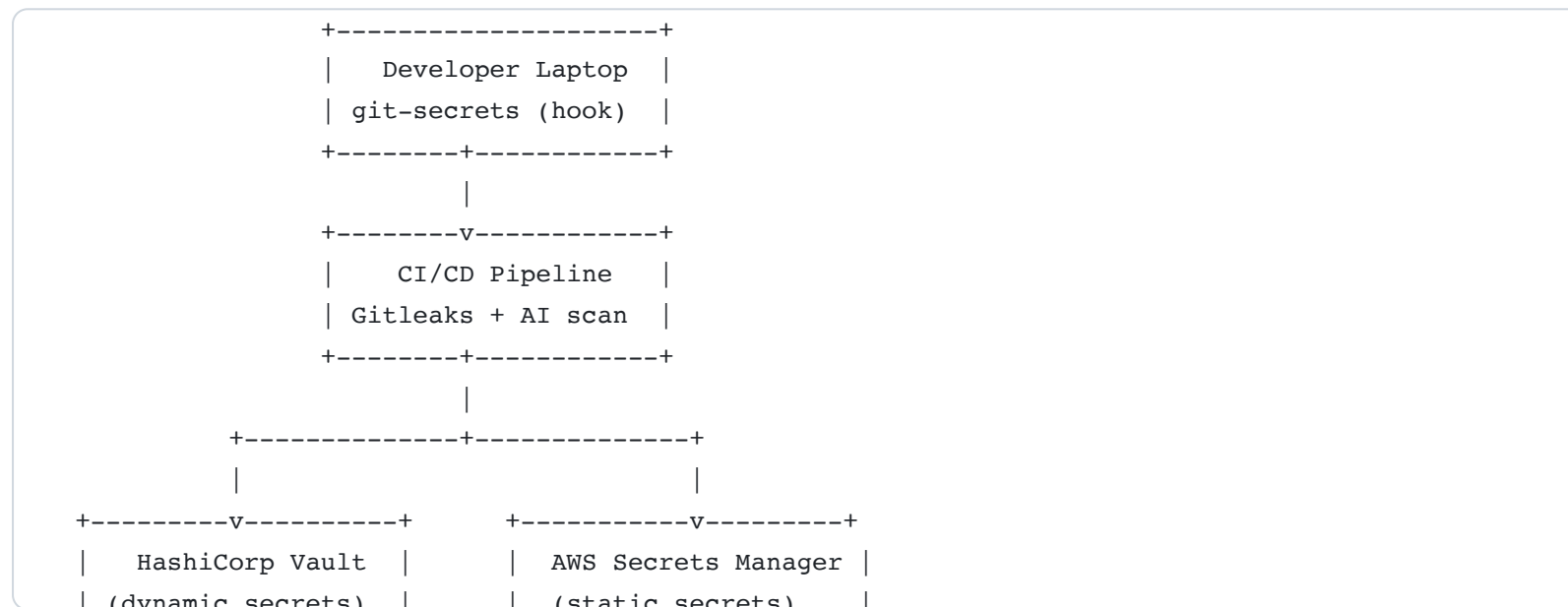
def create_secure_ssl_context() -> ssl.SSLContext:
 """Create a hardened SSL context."""
 ctx = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)

 # Minimum TLS 1.2
 ctx.minimum_version = ssl.TLSVersion.TLSv1_2

 # Load system CA certificates
 ctx.load_default_certs()
 # Or use certifi bundle
 ctx.load_verify_locations(certifi.where())
```

# Secrets Management Architecture

A production-ready secrets architecture integrates multiple components:



**Notes:** Environment variables are better than hardcoded secrets, but Vault sidecar injection or SDK-based retrieval is the gold standard.

---

# CTF Challenge: Secrets Scavenger Hunt

## Mission: Find All Leaked Secrets & Implement Proper Management

You have been given access to a vulnerable codebase. Your objectives:

1. **Discovery** — Use TruffleHog, Gitleaks, and AI scanning to find ALL leaked secrets in the repository (including git history)
2. **Classification** — Categorize each secret: type, severity, potential blast radius
3. **Triage** — Determine which secrets are still valid/active (use `--only-verified`)
4. **Remediation** — Implement proper secrets management:
  - Move all secrets to HashiCorp Vault or equivalent
  - Update application code to fetch secrets from Vault at runtime
  - Add pre-commit hooks to prevent future leaks
  - Configure CI/CD pipeline scanning
5. **Crypto audit** — Find and fix all weak cryptographic implementations

## 6. **TLS hardening** — Fix the Nginx configuration to achieve an A+ SSL Labs rating

### **Scoring**

Points awarded for: secrets found (10 pts each), proper Vault integration (25 pts), pre-commit hooks (15 pts), crypto fixes (20 pts), TLS A+ rating (20 pts), AI-augmented report (10 pts).

---

# CTF Challenge: Hints & Toolkit

```
Step 1: Clone and scan the challenge repository
git clone https://ctf-challenge.example.com/secrets-hunt.git
cd secrets-hunt

Step 2: Run automated scanners
trufflehog git file://. --json > trufflehog_results.json
gitleaks detect --source . --report-format json --report-path gitleaks_results.json

Step 3: Check git history for removed secrets
git log --all --diff-filter=D -- "*.env" "*.pem" "*.key"
git log --all -p -S "password" --source --all

Step 4: Scan Docker images
docker history --no-trunc challenge-app:latest
docker save challenge-app:latest | tar -xf - -C /tmp/layers/
```

**Notes:** Remind students that git history contains deleted secrets. A file removed from HEAD is NOT removed from the repository. Use `git filter-branch` or BFG Repo-Cleaner for true removal.

# Key Takeaways

- **Secrets sprawl is inevitable** — assume they will leak and build your defenses accordingly
- **Layer your controls** — pre-commit hooks, CI scanning, runtime monitoring, and AI augmentation
- **Centralize with Vault** — use dynamic secrets with short TTLs; eliminate long-lived credentials
- **Never roll your own crypto** — use established libraries and follow algorithm selection guides
- **TLS 1.2+ with AEAD ciphers** — enforce HSTS, OCSP stapling, and certificate automation
- **AI amplifies human auditing** — contextual analysis catches what regex misses and reduces false positives
- **Automate rotation and response** — the 5-minute window means manual processes are too slow
- **Git history never forgets** — removing a secret from HEAD does not remove it from the repository

Next lesson

Lesson 07 will cover **network traffic analysis and intrusion detection** — using AI to identify malicious patterns in real-time network data.

---