

Lab — Scoped-Token Demo

~20 minutes · local only, no network

This lab is a small, offline simulation of the blast-radius difference between a broad token and a scoped token, plus a working pre-send secret scanner. Every token in the lab is fictional and every command runs locally — the lab never touches a real API and never leaves your machine.

Prerequisites

- `python3` on the host (any 3.8+ will do)
- A shell (bash, zsh, fish — whichever you prefer)
- No network connection required; nothing is fetched, nothing is sent

All tokens in this lab are obviously fake (`tok_admin_ALL_ACCESS_...`, `tok_readonly_repo42_...`). They match no real provider's token shape. The point is to reason about scope and blast radius, not to exercise a real credential.

Exercise 1 — Create two fictional tokens

Set up a scratch directory and write two `.env` files: one with a broad "admin everything" token, one with a tightly scoped read-only token. These are the two credentials whose blast radius we are going to compare.

```
mkdir -p ~/work/ai-opsec-lab09 && cd ~/work/ai-opsec-lab09
cat > .env.broad <<'EOF'
# Broad token: fictional "admin" scope
FAKE_TOKEN=tok_admin_ALL_ACCESS_0000000000000000
FAKE_TOKEN_SCOPE=admin:*,repo:*,user:*,billing:*
EOF
cat > .env.scoped <<'EOF'
# Scoped token: fictional "read only, single repo, 1h TTL"
```

```
FAKE_TOKEN=tok_readonly_repo42_11111111111111
FAKE_TOKEN_SCOPE=repo:42:read
FAKE_TOKEN_TTL=3600
EOF
```

Expected: both files exist in `~/work/ai-opsec-lab09/`. Run `ls -la .env.*` to confirm.

✓ **Self-check:** `.env.broad` and `.env.scoped` both exist, and their contents match what you wrote above.

Exercise 2 — Simulate a prompt that leaks the token

Write a small simulator that reads an `.env` file and prints the "blast radius" of the token it finds. This stands in for the reasoning step you should be doing every time a credential is about to enter an agent's context: what does the world look like if this token leaks?

Save the following as `simulator.py` in `~/work/ai-opsec-lab09`:

```
import os, sys

def load_env(path):
    env = {}
    with open(path) as f:
        for line in f:
            line = line.strip()
            if not line or line.startswith('#'): continue
            k, _, v = line.partition('=')
            env[k.strip()] = v.strip()
    return env

def describe_leak(env):
    print(f" token          = {env.get('FAKE_TOKEN', '<none>')}")
    print(f" scope          = {env.get('FAKE_TOKEN_SCOPE', '<none>')}")
    print(f" ttl            = {env.get('FAKE_TOKEN_TTL', 'no ttl (long-live)')}")
    print(f" blast radius = ", end='')
    scope = env.get('FAKE_TOKEN_SCOPE', '')
    if '*' in scope or 'admin' in scope:
        print("CATASTROPHIC — full account compromise")
    elif 'write' in scope:
        print("WRITE to scoped resource only")
```

```
else:
    print("READ to scoped resource only — recoverable")
```

```
for path in sys.argv[1:]:
```

Run it against both env files:

```
python3 simulator.py .env.broad .env.scoped
```

Expected output:

```
[.env.broad]
token          = tok_admin_ALL_ACCESS_000000000000000
scope          = admin:*,repo:*,user:*,billing:*
ttl            = no ttl (long-lived)
blast radius   = CATASTROPHIC — full account compromise

[.env.scoped]
token          = tok_readonly_repo42_111111111111111
scope          = repo:42:read
ttl            = 3600
blast radius   = READ to scoped resource only — recoverable
```

✓ Self-check: both sections print, the broad file is flagged **CATASTROPHIC**, and the scoped file is flagged **READ to scoped resource only — recoverable**. Internalise the difference — that is the entire point of scoping.

Exercise 3 — Add a pre-send scanner

Write a minimal secret scanner that reads text on stdin, matches it against a list of known-bad patterns, and refuses to pass the text through if it finds one. This is the shape of the pre-send hook every prompt-sending tool should have.

Save the following as `scan.py` in `~/work/ai-opsec-lab09`:

```
import re, sys
PATTERNS = [
    (r'tok_admin_\w+', 'fictional admin token'),
    (r'AKIA[0-9A-Z]{16}', 'AWS access key ID'),
    (r'ghp_[A-Za-z0-9]{20,}', 'GitHub PAT'),
    (r'-----BEGIN [A-Z ]+PRIVATE KEY-----', 'private key'),
```

```

    (r'oxb-[A-Za-z0-9-]+' , 'Slack bot token'),
]
def scan(text):
    hits = []
    for pat, name in PATTERNS:
        for m in re.finditer(pat, text):
            hits.append((name, m.group(0)))
    return hits

text = sys.stdin.read()
hits = scan(text)
if hits:
    print("REFUSE TO SEND — secrets detected:")
    for name, val in hits:
        print(f"  {name}: {val[:12]}...")
    sys.exit(1)
print("OK: no secrets detected")

```

Run it over both env files:

```

python3 scan.py < .env.broad
python3 scan.py < .env.scoped

```

Expected:

- `.env.broad` → `REFUSE TO SEND — secrets detected:` followed by `fictional admin token: tok_admin_ALL...`, exit code 1
- `.env.scoped` → `OK: no secrets detected`, exit code 0

Check the exit codes explicitly with `echo $?` after each run if your shell doesn't surface them automatically.

Why is the scoped file "clean" to the scanner? The scoped token starts with `tok_readonly_`, which does not match the `tok_admin_\w+` pattern. This is deliberate — the scanner is tuned to catch the dangerous shape, not every shape. A scoped read-only token is not what you are worried about sending; an admin token is. Real scanners work the same way: they optimise for the high-blast patterns and tolerate the low-blast ones passing through.

✓ Self-check: `.env.broad` was refused with exit 1, `.env.scoped` passed with exit 0, and you understand why the scoped file was allowed through.

Exercise 4 — Clean up

This lab wrote fake tokens to disk. Even though they are not real, the habit of cleaning up scratch directories with credential-shaped strings in them is worth building.

```
cd ~  
rm -rf ~/work/ai-opsec-lab09
```

✓ Self-check: `ls ~/work/ai-opsec-lab09` reports "No such file or directory." The lab leaves no artifacts behind.

The takeaway from this lab is not the scripts themselves — they are twenty lines each. The takeaway is the mental model: before a credential enters any context that might be logged, you should be able to answer "what does the blast radius look like if this leaks?" in one sentence. If the sentence contains the word "catastrophic," the credential does not belong in that context. Mint a narrower one.