/insecure-defaults

Source: `~/.claude/skills/tob-insecure-defaults/skills/insecure-defaults/SKILL.md`

---

name: insecure-defaults description: "Detects fail-open insecure defaults (hardcoded secrets, weak auth, permissive security) that allow apps to run insecurely in production. Use when auditing security, reviewing config management, or analyzing environment variable handling." allowed-tools:

- Read
- Grep
- Glob
- Bash

Insecure Defaults Detection

Finds **fail-open** vulnerabilities where apps run insecurely with missing configuration. Distinguishes exploitable defaults from fail-secure patterns that crash safely.

- **Fail-open (CRITICAL):** `SECRET = env.get('KEY') or 'default'` → App runs with weak secret
- **Fail-secure (SAFE):** `SECRET = env['KEY']` → App crashes if missing

When to Use

- **Security audits** of production applications (auth, crypto, API security)
- **Configuration review** of deployment files, IaC templates, Docker configs
- **Code review** of environment variable handling and secrets management
- **Pre-deployment checks** for hardcoded credentials or weak defaults

When NOT to Use

Do not use this skill for:

- **Test fixtures** explicitly scoped to test environments (files in `test/`, `spec/`, `__tests__/`)
- **Example/template files** (`.example`, `.template`, `.sample` suffixes)
- **Development-only tools** (local Docker Compose for dev, debug scripts)
- **Documentation examples** in README.md or docs/ directories
- **Build-time configuration** that gets replaced during deployment
- **Crash-on-missing behavior** where app won't start without proper config (fail-secure)

When in doubt: trace the code path to determine if the app runs with the default or crashes.

Rationalizations to Reject

- **"It's just a development default"** → If it reaches production code, it's a finding
- **"The production config overrides it"** → Verify prod config exists; code-level vulnerability remains if not
- **"This would never run without proper config"** → Prove it with code trace; many apps fail silently
- **"It's behind authentication"** → Defense in depth; compromised session still exploits weak defaults
- **"We'll fix it before release"** → Document now; "later" rarely comes

Workflow

Follow this workflow for every potential finding:

1. SEARCH: Perform Project Discovery and Find Insecure Defaults

Determine language, framework, and project conventions. Use this information to further discover things like secret storage locations, secret usage patterns, credentialed third-party integrations, cryptography, and any other relevant configuration. Further use information to analyze insecure default configurations.

Example Search for patterns in `**/config/`, `**/auth/`, `**/database/`, and env files:

- **Fallback secrets:** `getenv.*\) or ['"]`, `process\.env\.[A-Z_]+ \|\| ['"]`, `ENV\.fetch.*default:`

- **Hardcoded credentials:** `password.*=.*['"][^'"]{8,}['"]`, `api[_-]?key.*=.*['"][^'"]+['"]`
- **Weak defaults:** `DEBUG.*=.*true`, `AUTH.*=.*false`, `CORS.*=.*\*`
- **Crypto algorithms:** `MD5|SHA1|DES|RC4|ECB` in security contexts

Tailor search approach based on discovery results.

Focus on production-reachable code, not test fixtures or example files.

2. VERIFY: Actual Behavior

For each match, trace the code path to understand runtime behavior.

Questions to answer:

- When is this code executed? (Startup vs. runtime)
- What happens if a configuration variable is missing?
- Is there validation that enforces secure configuration?

3. CONFIRM: Production Impact

Determine if this issue reaches production:

If production config provides the variable → Lower severity (but still a code-level vulnerability) If production config missing or uses default → CRITICAL

4. REPORT: with Evidence

Example report:

```
Finding: Hardcoded JWT Secret Fallback
Location: src/auth/jwt.ts:15
Pattern: const secret = process.env.JWT_SECRET || 'default';

Verification: App starts without JWT_SECRET; secret used in jwt.sign() at line 42
Production Impact: Dockerfile missing JWT_SECRET
Exploitation: Attacker forges JWTs using 'default', gains unauthorized access
```

Quick Verification Checklist

Fallback Secrets: `SECRET = env.get(X) or Y` → Verify: App starts without env var? Secret used in crypto/auth? → Skip: Test fixtures, example files

Default Credentials: Hardcoded `username`/`password` pairs → Verify: Active in deployed config? No runtime override? → Skip: Disabled accounts, documentation examples

Fail-Open Security: `AUTH_REQUIRED = env.get(X, 'false')` → Verify: Default is insecure (false/disabled/permissive)? → Safe: App crashes or default is secure (true/enabled/restricted)

Weak Crypto: MD5/SHA1/DES/RC4/ECB in security contexts → Verify: Used for passwords, encryption, or tokens? → Skip: Checksums, non-security hashing

Permissive Access: CORS `*`, permissions `0777`, public-by-default → Verify: Default allows unauthorized access? → Skip: Explicitly configured permissiveness with justification

Debug Features: Stack traces, introspection, verbose errors → Verify: Enabled by default? Exposed in responses? → Skip: Logging-only, not user-facing

For detailed examples and counter-examples, see examples.md.

Revision #5
Created 2026-02-18 08:40:07 UTC by John
Updated 2026-06-21 20:01:13 UTC by John