/sharp-edges

Source: `~/.claude/skills/tob-sharp-edges/skills/sharp-edges/SKILL.md`

---

name: sharp-edges description: "Identifies error-prone APIs, dangerous configurations, and footgun designs that enable security mistakes. Use when reviewing API designs, configuration schemas, cryptographic library ergonomics, or evaluating whether code follows 'secure by default' and 'pit of success' principles. Triggers: footgun, misuse-resistant, secure defaults, API usability, dangerous configuration." allowed-tools:

- Read
- Grep
- Glob

Sharp Edges Analysis

Evaluates whether APIs, configurations, and interfaces are resistant to developer misuse. Identifies designs where the "easy path" leads to insecurity.

When to Use

- Reviewing API or library design decisions
- Auditing configuration schemas for dangerous options
- Evaluating cryptographic API ergonomics
- Assessing authentication/authorization interfaces
- Reviewing any code that exposes security-relevant choices to developers

When NOT to Use

- Implementation bugs (use standard code review)
- Business logic flaws (use domain-specific analysis)
- Performance optimization (different concern)

Core Principle

The pit of success: Secure usage should be the path of least resistance. If developers must understand cryptography, read documentation carefully, or remember special rules to avoid vulnerabilities, the API has failed.

Rationalizations to Reject

Rationalization	Why It's Wrong	Required Action
"It's documented"	Developers don't read docs under deadline pressure	Make the secure choice the default or only option
"Advanced users need flexibility"	Flexibility creates footguns; most "advanced" usage is copy-paste	Provide safe high-level APIs; hide primitives
"It's the developer's responsibility"	Blame-shifting; you designed the footgun	Remove the footgun or make it impossible to misuse
"Nobody would actually do that"	Developers do everything imaginable under pressure	Assume maximum developer confusion
"It's just a configuration option"	Config is code; wrong configs ship to production	Validate configs; reject dangerous combinations
"We need backwards compatibility"	Insecure defaults can't be grandfather-claused	Deprecate loudly; force migration

Sharp Edge Categories

1. Algorithm/Mode Selection Footguns

APIs that let developers choose algorithms invite choosing wrong ones.

The JWT Pattern (canonical example):

- Header specifies algorithm: attacker can set `"alg": "none"` to bypass signatures
- Algorithm confusion: RSA public key used as HMAC secret when switching RS256→HS256
- Root cause: Letting untrusted input control security-critical decisions

Detection patterns:

- Function parameters like `algorithm`, `mode`, `cipher`, `hash_type`
- Enums/strings selecting cryptographic primitives
- Configuration options for security mechanisms

Example - PHP password_hash allowing weak algorithms:

```
// DANGEROUS: allows crc32, md5, sha1
password_hash($password, PASSWORD_DEFAULT); // Good - no choice
hash($algorithm, $password); // BAD: accepts "crc32"
```

2. Dangerous Defaults

Defaults that are insecure, or zero/empty values that disable security.

The OTP Lifetime Pattern:

```
# What happens when lifetime=0?
def verify_otp(code, lifetime=300):  # 300 seconds default
    if lifetime == 0:
        return True  # OOPS: 0 means "accept all"?
        # Or does it mean "expired immediately"?
```

Detection patterns:

- Timeouts/lifetimes that accept 0 (infinite? immediate expiry?)
- Empty strings that bypass checks
- Null values that skip validation
- Boolean defaults that disable security features
- Negative values with undefined semantics

Questions to ask:

- What happens with `timeout=0`? `max_attempts=0`? `key=""`?
- Is the default the most secure option?
- Can any default value disable security entirely?

3. Primitive vs. Semantic APIs

APIs that expose raw bytes instead of meaningful types invite type confusion.

The Libsodium vs. Halite Pattern:

```
// Libsodium (primitives): bytes are bytes
sodium_crypto_box($message, $nonce, $keypair);
// Easy to: swap nonce/keypair, reuse nonces, use wrong key type

// Halite (semantic): types enforce correct usage
Crypto::seal($message, new EncryptionPublicKey($key));
// Wrong key type = type error, not silent failure
```

Detection patterns:

- Functions taking `bytes`, `string`, `[]byte` for distinct security concepts
- Parameters that could be swapped without type errors
- Same type used for keys, nonces, ciphertexts, signatures

The comparison footgun:

```
// Timing-safe comparison looks identical to unsafe
if hmac == expected { }           // BAD: timing attack
if hmac.Equal(mac, expected) { }  // Good: constant-time
// Same types, different security properties
```

4. Configuration Cliffs

One wrong setting creates catastrophic failure, with no warning.

Detection patterns:

- Boolean flags that disable security entirely
- String configs that aren't validated
- Combinations of settings that interact dangerously
- Environment variables that override security settings
- Constructor parameters with sensible defaults but no validation (callers can override with insecure values)

Examples:

```
# One typo = disaster
verify_ssl: fasle  # Typo silently accepted as truthy?

# Magic values
session_timeout: -1  # Does this mean "never expire"?
```

```
# Dangerous combinations accepted silently
auth_required: true
bypass_auth_for_health_checks: true
health_check_path: "/"  # Oops
```

```
// Sensible default doesn't protect against bad callers
public function __construct(
    public string $hashAlgo = 'sha256',  // Good default...
    public int $otpLifetime = 120,       // ...but accepts md5, 0, etc.
) {}
```

See config-patterns.md for detailed patterns.

5. Silent Failures

Errors that don't surface, or success that masks failure.

Detection patterns:

- Functions returning booleans instead of throwing on security failures
- Empty catch blocks around security operations
- Default values substituted on parse errors
- Verification functions that "succeed" on malformed input

Examples:

```
# Silent bypass
def verify_signature(sig, data, key):
    if not key:
        return True  # No key = skip verification?!

# Return value ignored
signature.verify(data, sig)  # Throws on failure
crypto.verify(data, sig)     # Returns False on failure
# Developer forgets to check return value
```

6. Stringly-Typed Security

Security-critical values as plain strings enable injection and confusion.

Detection patterns:

- SQL/commands built from string concatenation
- Permissions as comma-separated strings
- Roles/scopes as arbitrary strings instead of enums
- URLs constructed by joining strings

The permission accumulation footgun:

```
permissions = "read,write"
permissions += ",admin"  # Too easy to escalate

# vs. type-safe
permissions = {Permission.READ, Permission.WRITE}
permissions.add(Permission.ADMIN)  # At least it's explicit
```

Analysis Workflow

Phase 1: Surface Identification

1. **Map security-relevant APIs**: authentication, authorization, cryptography, session management, input validation
2. **Identify developer choice points**: Where can developers select algorithms, configure timeouts, choose modes?
3. **Find configuration schemas**: Environment variables, config files, constructor parameters

Phase 2: Edge Case Probing

For each choice point, ask:

- **Zero/empty/null**: What happens with `0`, `""`, `null`, `[]`?
- **Negative values**: What does `-1` mean? Infinite? Error?
- **Type confusion**: Can different security concepts be swapped?
- **Default values**: Is the default secure? Is it documented?
- **Error paths**: What happens on invalid input? Silent acceptance?

Phase 3: Threat Modeling

Consider three adversaries:

1. **The Scoundrel**: Actively malicious developer or attacker controlling config
   - Can they disable security via configuration?

    - Can they downgrade algorithms?
    - Can they inject malicious values?
2. **The Lazy Developer**: Copy-pastes examples, skips documentation
    - Will the first example they find be secure?
    - Is the path of least resistance secure?
    - Do error messages guide toward secure usage?
3. **The Confused Developer**: Misunderstands the API
    - Can they swap parameters without type errors?
    - Can they use the wrong key/algorithm/mode by accident?
    - Are failure modes obvious or silent?

Phase 4: Validate Findings

For each identified sharp edge:

1. **Reproduce the misuse**: Write minimal code demonstrating the footgun
2. **Verify exploitability**: Does the misuse create a real vulnerability?
3. **Check documentation**: Is the danger documented? (Documentation doesn't excuse bad design, but affects severity)
4. **Test mitigations**: Can the API be used safely with reasonable effort?

If a finding seems questionable, return to Phase 2 and probe more edge cases.

Severity Classification

Severity	Criteria	Examples
Critical	Default or obvious usage is insecure	`verify: false` default; empty password allowed
High	Easy misconfiguration breaks security	Algorithm parameter accepts "none"
Medium	Unusual but possible misconfiguration	Negative timeout has unexpected meaning
Low	Requires deliberate misuse	Obscure parameter combination

References

By category:

- **Cryptographic APIs**: See references/crypto-apis.md
- **Configuration Patterns**: See references/config-patterns.md

- **Authentication/Session**: See references/auth-patterns.md
- **Real-World Case Studies**: See references/case-studies.md (OpenSSL, GMP, etc.)

By language (general footguns, not crypto-specific):

Language	Guide
C/C++	references/lang-c.md
Go	references/lang-go.md
Rust	references/lang-rust.md
Swift	references/lang-swift.md
Java	references/lang-java.md
Kotlin	references/lang-kotlin.md
C#	references/lang-csharp.md
PHP	references/lang-php.md
JavaScript/TypeScript	references/lang-javascript.md
Python	references/lang-python.md
Ruby	references/lang-ruby.md

See also references/language-specific.md for a combined quick reference.

Quality Checklist

Before concluding analysis:

- ☐ Probed all zero/empty/null edge cases
- ☐ Verified defaults are secure
- ☐ Checked for algorithm/mode selection footguns
- ☐ Tested type confusion between security concepts
- ☐ Considered all three adversary types
- ☐ Verified error paths don't bypass security
- ☐ Checked configuration validation
- ☐ Constructor params validated (not just defaulted) - see config-patterns.md

---

Revision #5
Created 2026-02-18 08:40:08 UTC by John

Updated 2026-06-21 20:01:15 UTC by John