# /sentry-security-review

**Source:** `~/.claude/skills/sentry-security-review/SKILL.md`

---

name: security-review description: Security code review for vulnerabilities. Use when asked to "security review", "find vulnerabilities", "check for security issues", "audit security", "OWASP review", or review code for injection, XSS, authentication, authorization, cryptography issues. Provides systematic review with confidence-based reporting. allowed-tools: Read Grep Glob Bash Task license: LICENSE

## Security Review Skill

Identify exploitable security vulnerabilities in code. Report only **HIGH CONFIDENCE** findings—clear vulnerable patterns with attacker-controlled input.

# Scope: Research vs. Reporting

**CRITICAL DISTINCTION:**

- **Report on**: Only the specific file, diff, or code provided by the user
- **Research**: The ENTIRE codebase to build confidence before reporting

Before flagging any issue, you MUST research the codebase to understand:

- Where does this input actually come from? (Trace data flow)
- Is there validation/sanitization elsewhere?
- How is this configured? (Check settings, config files, middleware)
- What framework protections exist?

**Do NOT report issues based solely on pattern matching.** Investigate first, then report only what you're confident is exploitable.

# Confidence Levels

| Level | Criteria | Action |
|---|---|---|
| **HIGH** | Vulnerable pattern + attacker-controlled input confirmed | **Report** with severity |
| **MEDIUM** | Vulnerable pattern, input source unclear | **Note** as "Needs verification" |
| **LOW** | Theoretical, best practice, defense-in-depth | **Do not report** |

# Do Not Flag

## General Rules

- Test files (unless explicitly reviewing test security)
- Dead code, commented code, documentation strings
- Patterns using **constants** or **server-controlled configuration**
- Code paths that require prior authentication to reach (note the auth requirement instead)

# Server-Controlled Values (NOT Attacker-Controlled)

These are configured by operators, not controlled by attackers:

| Source | Example | Why It's Safe |
|---|---|---|
| Django settings | `settings.API_URL`, `settings.ALLOWED_HOSTS` | Set via config/env at deployment |
| Environment variables | `os.environ.get('DATABASE_URL')` | Deployment configuration |
| Config files | `config.yaml`, `app.config['KEY']` | Server-side files |
| Framework constants | `django.conf.settings.*` | Not user-modifiable |
| Hardcoded values | `BASE_URL = "https://api.internal"` | Compile-time constants |

**SSRF Example - NOT a vulnerability:**

```
# SAFE: URL comes from Django settings (server-controlled)
response = requests.get(f"{settings.SEER_AUTOFIX_URL}{path}")
```

**SSRF Example - IS a vulnerability:**

```
# VULNERABLE: URL comes from request (attacker-controlled)
response = requests.get(request.GET.get('url'))
```

# Framework-Mitigated Patterns

Check language guides before flagging. Common false positives:

| Pattern | Why It's Usually Safe |
|---|---|
| Django `{{ variable }}` | Auto-escaped by default |
| React `{variable}` | Auto-escaped by default |
| Vue `{{ variable }}` | Auto-escaped by default |
| `User.objects.filter(id=input)` | ORM parameterizes queries |
| `cursor.execute("...%s", (input,))` | Parameterized query |
| `innerHTML = "<b>Loading...</b>"` | Constant string, no user input |

**Only flag these when:**

- Django: `{{ var|safe }}`, `{% autoescape off %}`, `mark_safe(user_input)`

- React: `dangerouslySetInnerHTML={{__html: userInput}}`
- Vue: `v-html="userInput"`
- ORM: `.raw()`, `.extra()`, `RawSQL()` with string interpolation

# Review Process

## 1. Detect Context

What type of code am I reviewing?

| Code Type | Load These References |
|---|---|
| API endpoints, routes | `authorization.md`, `authentication.md`, `injection.md` |
| Frontend, templates | `xss.md`, `csrf.md` |
| File handling, uploads | `file-security.md` |
| Crypto, secrets, tokens | `cryptography.md`, `data-protection.md` |
| Data serialization | `deserialization.md` |
| External requests | `ssrf.md` |
| Business workflows | `business-logic.md` |
| GraphQL, REST design | `api-security.md` |
| Config, headers, CORS | `misconfiguration.md` |
| CI/CD, dependencies | `supply-chain.md` |
| Error handling | `error-handling.md` |
| Audit, logging | `logging.md` |

## 2. Load Language Guide

Based on file extension or imports:

| Indicators | Guide |
|---|---|
| `.py`, `django`, `flask`, `fastapi` | `languages/python.md` |
| `.js`, `.ts`, `express`, `react`, `vue`, `next` | `languages/javascript.md` |
| `.go`, `go.mod` | `languages/go.md` |
| `.rs`, `Cargo.toml` | `languages/rust.md` |
| `.java`, `spring`, `@Controller` | `languages/java.md` |

# 3. Load Infrastructure Guide (if applicable)

| File Type | Guide |
|---|---|
| `Dockerfile`, `.dockerignore` | `infrastructure/docker.md` |
| K8s manifests, Helm charts | `infrastructure/kubernetes.md` |
| `.tf`, Terraform | `infrastructure/terraform.md` |
| GitHub Actions, `.gitlab-ci.yml` | `infrastructure/ci-cd.md` |
| AWS/GCP/Azure configs, IAM | `infrastructure/cloud.md` |

# 4. Research Before Flagging

**For each potential issue, research the codebase to build confidence:**

- Where does this value actually come from? Trace the data flow.
- Is it configured at deployment (settings, env vars) or from user input?
- Is there validation, sanitization, or allowlisting elsewhere?
- What framework protections apply?

Only report issues where you have HIGH confidence after understanding the broader context.

# 5. Verify Exploitability

For each potential finding, confirm:

**Is the input attacker-controlled?**

| Attacker-Controlled (Investigate) | Server-Controlled (Usually Safe) |
|---|---|
| `request.GET`, `request.POST`, `request.args` | `settings.X`, `app.config['X']` |
| `request.json`, `request.data`, `request.body` | `os.environ.get('X')` |
| `request.headers` (most headers) | Hardcoded constants |
| `request.cookies` (unsigned) | Internal service URLs from config |
| URL path segments: `/users/<id>/` | Database content from admin/system |
| File uploads (content and names) | Signed session data |
| Database content from other users | Framework settings |
| WebSocket messages | |

**Does the framework mitigate this?**

- Check language guide for auto-escaping, parameterization

- Check for middleware/decorators that sanitize

**Is there validation upstream?**

- Input validation before this code
- Sanitization libraries (DOMPurify, bleach, etc.)

# 6. Report HIGH Confidence Only

Skip theoretical issues. Report only what you've confirmed is exploitable after research.

---

# Severity Classification

| Severity | Impact | Examples |
|---|---|---|
| **Critical** | Direct exploit, severe impact, no auth required | RCE, SQL injection to data, auth bypass, hardcoded secrets |
| **High** | Exploitable with conditions, significant impact | Stored XSS, SSRF to metadata, IDOR to sensitive data |
| **Medium** | Specific conditions required, moderate impact | Reflected XSS, CSRF on state-changing actions, path traversal |
| **Low** | Defense-in-depth, minimal direct impact | Missing headers, verbose errors, weak algorithms in non-critical context |

---

# Quick Patterns Reference

## Always Flag (Critical)

```
eval(user_input)           # Any language
exec(user_input)           # Any language
pickle.loads(user_data)    # Python
yaml.load(user_data)       # Python (not safe_load)
unserialize($user_data)    # PHP
deserialize(user_data)     # Java ObjectInputStream
shell=True + user_input    # Python subprocess
child_process.exec(user)   # Node.js
```

# Always Flag (High)

```
innerHTML = userInput              # DOM XSS
dangerouslySetInnerHTML={user}     # React XSS
v-html="userInput"                 # Vue XSS
f"SELECT * FROM x WHERE {user}"    # SQL injection
`SELECT * FROM x WHERE ${user}`    # SQL injection
os.system(f"cmd {user_input}")     # Command injection
```

# Always Flag (Secrets)

```
password = "hardcoded"
api_key = "sk-..."
AWS_SECRET_ACCESS_KEY = "..."
private_key = "-----BEGIN"
```

# Check Context First (MUST Investigate Before Flagging)

```
# SSRF - ONLY if URL is from user input, NOT from settings/config
requests.get(request.GET['url'])     # FLAG: User-controlled URL
requests.get(settings.API_URL)       # SAFE: Server-controlled config
requests.get(f"{settings.BASE}/{x}") # CHECK: Is 'x' user input?

# Path traversal - ONLY if path is from user input
open(request.GET['file'])            # FLAG: User-controlled path
open(settings.LOG_PATH)              # SAFE: Server-controlled config
open(f"{BASE_DIR}/{filename}")       # CHECK: Is 'filename' user input?

# Open redirect - ONLY if URL is from user input
redirect(request.GET['next'])        # FLAG: User-controlled redirect
redirect(settings.LOGIN_URL)         # SAFE: Server-controlled config

# Weak crypto - ONLY if used for security purposes
hashlib.md5(file_content)            # SAFE: File checksums, caching
hashlib.md5(password)                # FLAG: Password hashing
random.random()                      # SAFE: Non-security uses (UI, sampling)
random.random() for token            # FLAG: Security tokens need secrets module
```

# Output Format

````
## Security Review: [File/Component Name]

### Summary
- **Findings**: X (Y Critical, Z High, ...)
- **Risk Level**: Critical/High/Medium/Low
- **Confidence**: High/Mixed

### Findings

#### [VULN-001] [Vulnerability Type] (Severity)
- **Location**: `file.py:123`
- **Confidence**: High
- **Issue**: [What the vulnerability is]
- **Impact**: [What an attacker could do]
- **Evidence**:
  ```python
  [Vulnerable code snippet]
````

- **Fix**: [How to remediate]

# Needs Verification

## [VERIFY-001] [Potential Issue]

- **Location**: `file.py:456`
- **Question**: [What needs to be verified]

````
If no vulnerabilities found, state: "No high-confidence vulnerabilities identified."

---

## Reference Files

### Core Vulnerabilities (`references/`)
| File | Covers |
|------|--------|
````
````

```
| `injection.md` | SQL, NoSQL, OS command, LDAP, template injection |
| `xss.md` | Reflected, stored, DOM-based XSS |
| `authorization.md` | Authorization, IDOR, privilege escalation |
| `authentication.md` | Sessions, credentials, password storage |
| `cryptography.md` | Algorithms, key management, randomness |
| `deserialization.md` | Pickle, YAML, Java, PHP deserialization |
| `file-security.md` | Path traversal, uploads, XXE |
| `ssrf.md` | Server-side request forgery |
| `csrf.md` | Cross-site request forgery |
| `data-protection.md` | Secrets exposure, PII, logging |
| `api-security.md` | REST, GraphQL, mass assignment |
| `business-logic.md` | Race conditions, workflow bypass |
| `modern-threats.md` | Prototype pollution, LLM injection, WebSocket |
| `misconfiguration.md` | Headers, CORS, debug mode, defaults |
| `error-handling.md` | Fail-open, information disclosure |
| `supply-chain.md` | Dependencies, build security |
| `logging.md` | Audit failures, log injection |

### Language Guides (`languages/`)
- `python.md` - Django, Flask, FastAPI patterns
- `javascript.md` - Node, Express, React, Vue, Next.js
- `go.md` - Go-specific security patterns
- `rust.md` - Rust unsafe blocks, FFI security
- `java.md` - Spring, Java EE patterns

### Infrastructure (`infrastructure/`)
- `docker.md` - Container security
- `kubernetes.md` - K8s RBAC, secrets, policies
- `terraform.md` - IaC security
- `ci-cd.md` - Pipeline security
- `cloud.md` - AWS/GCP/Azure security
```

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