Security in Converge: What, Why, and How

> This document explains how Converge uses the Security Gate (Gate 4) to detect and block security vulnerabilities before a change can be integrated. It is written for an external audience — no prior knowledge of the Converge internals is assumed.

---

1. What Problem Does Security Solve?

Code changes can introduce security vulnerabilities at multiple levels: insecure coding patterns (SAST), vulnerable dependencies (SCA), and leaked secrets (hardcoded API keys, tokens, passwords). A single missed vulnerability in a merge can compromise the entire system.

The Security Gate ensures that no change with unacceptable security findings can proceed through the merge pipeline. It coordinates multiple scanner tools, each specialized for a different class of vulnerability, and enforces severity-based thresholds that tighten with risk level.

---

2. The Security Finding Model

Every vulnerability detected by any scanner is normalized into a common SecurityFinding structure:

```python
@dataclass
class SecurityFinding:
    id: str                          # unique finding identifier
    scanner: str                     # "bandit", "pip-audit", "gitleaks"
    category: FindingCategory        # sast, sca, secrets
    severity: FindingSeverity        # critical, high, medium, low, info
    file: str                        # file path (or "dependency:name==version")
    line: int                        # line number (0 if not applicable)
    rule: str                        # scanner-specific rule ID
    evidence: str                    # human-readable description
    confidence: str                  # "high", "medium", "low"
    intent_id: str | None            # linked intent (if scan is per-change)
    tenant_id: str | None            # tenant context
    timestamp: str                   # ISO 8601
```

Source: `security_models.py`, class `SecurityFinding`

2.1 Categories

Category	Enum Value	What It Covers
SAST	`FindingCategory.SAST`	Static Application Security Testing — insecure code patterns
SCA	`FindingCategory.SCA`	Software Composition Analysis — vulnerable dependencies
Secrets	`FindingCategory.SECRETS`	Hardcoded secrets, API keys, tokens

2.2 Severity Levels

Severity	Enum Value	Meaning
Critical	`FindingSeverity.CRITICAL`	Exploitable vulnerability, immediate risk
High	`FindingSeverity.HIGH`	Significant vulnerability, needs prompt fix
Medium	`FindingSeverity.MEDIUM`	Moderate issue, should be addressed
Low	`FindingSeverity.LOW`	Minor issue, low exploitability
Info	`FindingSeverity.INFO`	Informational, no direct risk

---

3. Scanner Adapters

Converge uses a port/adapter architecture for security scanning. The `SecurityScannerPort` protocol defines the interface, and three concrete adapters implement it:

3.1 Bandit (SAST)

What it detects: Insecure Python code patterns — hardcoded passwords, use of `eval()`, insecure random number generation, SQL injection risks, unsafe deserialization, and ~100 other rules.

How it works:

```
1. Check availability: shutil.which("bandit") → binary exists?
2. Execute: bandit -r <path> -f json -q
3. Parse JSON output → extract results array
4. Map severity: HIGH → high, MEDIUM → medium, LOW → low, UNDEFINED → info
5. Map confidence: HIGH → high, MEDIUM → medium, LOW → low
6. Create SecurityFinding per result
```

Timeout: 120 seconds (configurable via `options["timeout"]`)

Source: `adapters/security/bandit_adapter.py`, class `BanditScanner`

3.2 pip-audit (SCA)

What it detects: Known vulnerabilities in Python dependencies by checking installed packages against the OSV and PyPI vulnerability databases.

How it works:

```
1. Check availability: shutil.which("pip-audit") → binary exists?
2. Execute: pip-audit --format json --desc [--r <requirements-file>]
3. Parse JSON output → extract dependencies with vulns
4. Map CVSS score to severity using thresholds:
```

```
   - CVSS >= 9.0 → Critical
   - CVSS >= 7.0 → High
   - CVSS >= 4.0 → Medium
   - CVSS >= 0.1 → Low
   - CVSS < 0.1  → Info
5. Create SecurityFinding per vulnerability
```

CVSS-to-Severity Mapping: This follows the standard CVSS v3 severity classification, ensuring consistency with industry-standard vulnerability scoring.

Timeout: 180 seconds (longer because pip-audit needs to resolve and check all transitive dependencies)

Source: `adapters/security/pip_audit_adapter.py`, class `PipAuditScanner`

3.3 Gitleaks (Secrets)

What it detects: Hardcoded secrets — API keys, tokens, passwords, private keys, AWS credentials, and other sensitive data committed to source code.

How it works:

```
1. Check availability: shutil.which("gitleaks") → binary exists?
2. Execute: gitleaks detect --source <path> --report-format json
                            --report-path /dev/stdout --no-git
3. Parse JSON output → extract leak array
4. Every finding gets severity = HIGH (always)
5. Evidence is redacted: "Rule: <rule> | Match: <first 8 chars>****"
6. Create SecurityFinding per leak
```

Why always HIGH severity? A leaked secret is always a high-severity finding, regardless of the type of secret. Even a "low-importance" API key can be used for lateral movement or to access systems that the original developer didn't intend to expose.

Why redact evidence? The evidence field is stored in the event log and may be visible in dashboards. Storing the actual secret value would create a secondary exposure vector. The redaction shows enough to identify the finding (rule + first 8 chars) without revealing the full secret.

Source: `adapters/security/gitleaks_adapter.py`, class `GitleaksScanner`

3.4 Scanner Availability

Each scanner uses `shutil.which()` to check if its binary is installed. If a scanner is not available:

- It is skipped (not an error)
- The scanner result is recorded as `{"status": "skipped", "reason": "not installed"}`
- The gate evaluation proceeds with whatever findings the available scanners produced

Why not fail on missing scanners? Not every environment has every scanner installed. A development machine might have bandit but not gitleaks. The system degrades gracefully — fewer scanners means fewer findings, which means the gate is more likely to pass. Teams that need strict scanning can enforce scanner availability through their infrastructure.

---

4. Scan Orchestration

4.1 The Scan Flow

The `run_scan` function coordinates all scanners:

```
1. Generate scan_id (unique identifier for this scan run)
2. Emit SECURITY_SCAN_STARTED event
3. For each scanner:
   a. Check is_available()
   b. If not available → record as "skipped"
   c. If available → scanner.scan(path) → list of SecurityFinding
4. Persist each finding to the event log (upsert_security_finding)
5. For CRITICAL and HIGH findings → emit SECURITY_FINDING_DETECTED event
6. Count findings by severity
7. Emit SECURITY_SCAN_COMPLETED event with summary
8. Return summary dict
```

Source: `security.py`, function `run_scan`

4.2 Event Recording

The scan process emits three types of events:

Event Type	When	Purpose
`SECURITY_SCAN_STARTED`	Before running scanners	Records which scanners will run, the target path, and the scan_id
`SECURITY_FINDING_DETECTED`	For each CRITICAL/HIGH finding	Immediate alerting for severe vulnerabilities
`SECURITY_SCAN_COMPLETED`	After all scanners finish	Summary with total counts, per-scanner results, severity breakdown

Why emit per-finding events only for CRITICAL/HIGH? Medium and low findings are stored but don't need per-event alerting. Critical and high findings may trigger immediate notifications or review escalation. This keeps the event log focused on actionable items.

---

5. The Security Gate

5.1 Gate Logic

The Security Gate (Gate 4) evaluates two severity counters against configured thresholds:

```
critical_count = count of findings where severity == "critical"
high_count     = count of findings where severity == "high"

max_critical = profile[risk_level].security.max_critical
max_high     = profile[risk_level].security.max_high

Gate passes if: critical_count <= max_critical AND high_count <= max_high
Gate blocks if: either count exceeds its threshold
```

Source: `policy.py`, function `_evaluate_security_gate`

5.2 Why Only Critical and High?

Medium, low, and info findings are informational — they should be addressed but don't warrant blocking a merge. The gate focuses on findings that represent active, exploitable risks:

- Critical: A vulnerability that can be exploited immediately (e.g., SQL injection, remote code execution)
- High: A significant vulnerability that requires attention (e.g., leaked secret, use of known-vulnerable dependency)

Blocking on medium findings would create excessive friction and slow down the merge pipeline without proportionate security benefit.

5.3 Thresholds by Risk Level

Risk Level	max_critical	max_high	Rationale
Low	0	5	No critical findings ever. Up to 5 high findings tolerated — low-risk changes have limited blast radius
Medium	0	2	Tighter high threshold — medium-risk changes get less tolerance
High	0	0	Zero tolerance for both critical and high — high-risk changes must be clean
Critical	0	0	Same — any high/critical finding blocks a critical-risk change

Source: `defaults.py`, constant `SECURITY_GATE_DEFAULTS`

5.4 Why max_critical Is Always 0

Critical findings represent immediate, exploitable vulnerabilities. No risk level tolerates them. Even a low-risk change with a critical SQL injection finding should be blocked — the finding isn't less dangerous because the change is "low risk."

5.5 The Gate Value

The gate reports a composite value for dashboard display:

```
value     = critical_count × 10 + high_count
threshold = max_critical × 10 + max_high
```

This creates a single comparable number: 23 means "2 critical + 3 high." The ×10 multiplier ensures critical findings always dominate the number, making it easy to see at a glance whether the issue is critical or high findings.

---

6. The GateResult

The security gate produces a `GateResult` with machine-readable fields:

Field	Value
`gate`	`GateName.SECURITY`
`passed`	`True` if both counts within limits
`reason`	`"Security: 0 critical, 1 high (max critical=0, max high=2)"`
`value`	Composite value (critical×10 + high)
`threshold`	Composite threshold (max_critical×10 + max_high)

Example — passing (medium risk):

```
Gate: security
Passed: true
Reason: "Security: 0 critical, 1 high (max critical=0, max high=2)"
Value: 1 (0 critical, 1 high)
Threshold: 2 (max critical=0, max high=2)
```

Example — failing (high risk):

```
Gate: security
Passed: false
Reason: "Security: 0 critical, 2 high (max critical=0, max high=0)"
Value: 2 (0 critical, 2 high)
Threshold: 0 (max critical=0, max high=0)
```

---

7. Integration in the Validation Flow

Security is evaluated as part of the policy gates step (Step 5) in the full validation pipeline:

```
1. Simulation     — Can the change merge cleanly?
2. Verification   — Do required checks pass?
3. Risk Evaluation — Compute risk scores and signals
4. Coherence      — Does the change maintain system coherence?
5. Policy Gates   — All 5 gates evaluated together:
     Gate 1: Verification (checks passed)
     Gate 2: Containment (change is contained)
     Gate 3: Entropy (entropy within budget)
     Gate 4: Security (no critical findings)    ← THIS GATE
     Gate 5: Coherence (coherence score above threshold)
6. Risk Gate      — Composite risk score within limits
7. Verdict        — ALLOW or BLOCK
```

The security gate always evaluates, even if no findings exist (it passes with 0 findings). If `security_findings` is not provided to the policy evaluator, it defaults to an empty list — the gate passes with `"Security: 0 critical, 0 high"`.

---

8. Scanner Port Interface

Converge defines the scanner contract through a Protocol interface:

```python
class SecurityScannerPort(Protocol):
    scanner_name: str

    def is_available(self) -> bool:
        """Check if the scanner binary is installed."""
        ...

    def scan(self, path: str, **options: Any) -> list[SecurityFinding]:
        """Scan the path and return normalized findings."""
        ...
```

Source: `ports.py`, class `SecurityScannerPort`

This interface makes it easy to add new scanners (e.g., Semgrep, Trivy, custom internal tools) without modifying the orchestrator or gate logic. Any class that implements `scanner_name`, `is_available()`, and `scan()` can be plugged in.

---

9. Persistence and Querying

9.1 Finding Storage

Each finding is persisted via `event_log.upsert_security_finding()`. The upsert pattern means re-scanning the same code won't create duplicate findings — findings with the same scan-

ner, rule, file, and line are updated rather than duplicated.

9.2 Summary Queries

The `scan_summary` function provides dashboard-ready data:

```
def scan_summary(*, intent_id=None, tenant_id=None) -> dict:
    return {
        "finding_counts": event_log.count_security_findings(...),
        "recent_scans": event_log.query(event_type="SECURITY_SCAN_COMPLETED", limit=5),
    }
```

This returns both aggregated severity counts and the most recent scan events, enabling dashboard views like "5 critical / 12 high / 23 medium" and "Last scan: 2 minutes ago, 3 findings."

---

10. Configuration Override

Security thresholds can be customized per risk level in `.converge/policy.json`:

```
{
  "profiles": {
    "low": {
      "security": { "max_critical": 0, "max_high": 10 }
    },
    "medium": {
      "security": { "max_critical": 0, "max_high": 5 }
    },
    "high": {
      "security": { "max_critical": 0, "max_high": 1 }
    },
    "critical": {
      "security": { "max_critical": 0, "max_high": 0 }
    }
  }
}
```

Teams with legacy codebases that have many existing high-severity findings might temporarily raise thresholds while they remediate. Teams with strict security requirements might add custom scanners via the port interface.

---

11. Design Rationale Summary

Design Choice	Rationale
Port/adapter architecture	Scanners are pluggable. Adding a new scanner doesn't require modifying the gate or orchestrator.
Graceful degradation on missing scanners	Not every environment has every scanner. Skipping unavailable scanners lets the system work in development while enforcing full scanning in CI.
Normalized findings model	Different scanners produce different output formats. Normalizing to SecurityFinding enables unified counting, storage, and gate evaluation.
CVSS-to-severity mapping for SCA	Industry-standard mapping (CVSS v3) ensures consistent severity classification across sources.
Secrets always HIGH	A leaked secret is always serious, regardless of what type of secret it is. Fixed severity prevents underclassification.
Evidence redaction for secrets	Prevents secondary exposure through dashboards and event logs. Shows enough to identify the finding without revealing the actual secret.
Gate evaluates only critical/high	Medium and low findings don't warrant blocking merges. The gate focuses on findings that represent immediate, exploitable risk.
max_critical always 0	Critical vulnerabilities are never acceptable, regardless of risk level. This is a non-negotiable baseline.
Per-finding events only for critical/high	Keeps the event log focused on actionable items without flooding it with informational findings.
Composite gate value (critical×10 + high)	Single number for dashboard display. Critical dominates due to ×10 multiplier.

---

12. File Reference

File	Role
`security.py`	Scan orchestration, scanner coordination, event emission
`security_models.py`	`SecurityFinding`, `FindingSeverity`, `FindingCategory`

File	Role
`policy.py`	Gate 4 evaluation logic (`_evaluate_security_gate`)
`defaults.py`	`SECURITY_GATE_DEFAULTS` with per-level thresholds
`ports.py`	`SecurityScannerPort` protocol interface
`adapters/security/bandit_adapter.py`	Bandit SAST scanner implementation
`adapters/security/pip_audit_adapter.py`	pip-audit SCA scanner with CVSS mapping
`adapters/security/gitleaks_adapter.py`	Gitleaks secrets scanner with evidence redaction