

AppSec Automation at Scale

GitHub Advanced Security (GHAS)
with GitHub REST API

Comprehensive Implementation Guide

Document Type: Technical Reference & Implementation Guide
Platform: GitHub Enterprise Cloud / GHES / GHE.com
Integration: Harness CD Pipeline Orchestration
Version: 1.0

Enterprise Security Engineering

January 10, 2026

Contents

1	Introduction	3
1.1	Scope and Objectives	3
1.2	Platform Compatibility	3
2	Reference Automation Architecture	4
2.1	Core Architecture Pattern	4
2.2	GitHub REST API Best Practices	4
2.3	Authentication Strategy	4
3	Python Building Blocks	5
3.1	REST Client with Pagination and Rate-Limit Handling	5
3.2	GitHub App Authentication	7
4	Enablement & Policy Automation	9
4.1	Security Baseline Enforcement	9
4.1.1	Per-Repository Security Toggles	9
4.1.2	Organization/Enterprise Security Configurations	9
4.2	Code Scanning Default Setup	10
4.3	Dependabot Configuration Management	11
4.4	Push Protection Governance	12
5	Detection Pipeline Automation	14
5.1	SARIF Upload from External Scanners	14
5.2	SBOM Export Automation	15
5.3	Secret Scanning Health Monitoring	16
6	Triage Automation	19
6.1	Secret Scanning Alert Management	19
6.2	Code Scanning Alert Management	21
6.3	Dependabot Alert Management	22
6.4	Unified Alert Ingestion	24
7	Webhook-Driven Event Processing	28
7.1	Webhook Receiver Implementation	28
7.2	Queue-Based Event Processing	30
8	Reporting & Metrics Automation	32
8.1	Security KPI Generation	32
8.2	Audit Log Integration	35
9	Harness CD Pipeline Integration	38
9.1	Integration Architecture	38
9.1.1	Recommended Pipeline Structure	38
9.2	Triggering Harness CD Pipelines	38
9.3	Security Gate Implementation	40
9.3.1	Harness Integration Instructions	44
10	Architecture Diagram	45

11 Implementation Backlog	47
11.1 Priority 1: Foundation	47
11.2 Priority 2: Detection & Normalization	47
11.3 Priority 3: Workflow Automation	47
11.4 Priority 4: Remediation Acceleration	47
11.5 Priority 5: Governance & Evidence	47
A GitHub REST API Endpoint Reference	47
B Environment Variables Reference	48
C Python Dependencies	49

1 Introduction

This document provides a comprehensive, production-ready catalog of Application Security (AppSec) automations implemented using GitHub Advanced Security (GHAS) integrated with the GitHub REST API. All examples include working Python implementations suitable for enterprise deployment.

The architecture presented here follows an event-driven design philosophy, prioritizing webhooks over polling, and centralizing security findings through a unified intake queue for normalization, deduplication, SLA tracking, and evidence collection.

1.1 Scope and Objectives

This guide addresses the following automation domains:

1. **Enablement & Policy Automation** — Consistent security baseline enforcement across repositories
2. **Detection Pipeline Automation** — Making security scans inevitable and comprehensive
3. **Triage Automation** — Reducing noise, routing to owners, and enforcing SLAs
4. **Remediation Automation** — Accelerating developer fix velocity
5. **Reporting & Metrics Automation** — Proving outcomes and maintaining audit evidence
6. **Release Gate Integration** — Harness CD pipeline orchestration with security gates

1.2 Platform Compatibility

The implementations in this document are compatible with:

- GitHub Enterprise Cloud (github.com)
- GitHub Enterprise Server (GHES) — swap API hostname accordingly
- GHE.com subdomain configurations

For GHES deployments, the GitHub REST API documentation explicitly addresses hostname configuration requirements.

2 Reference Automation Architecture

The recommended architecture follows an **event-driven first, polling second** approach to maximize efficiency and minimize API rate limit consumption.

2.1 Core Architecture Pattern

1. **GitHub Webhooks** (e.g., `secret_scanning_alert`, `code_scanning_alert`) → Intake service (API gateway / serverless function)
2. **Intake Service** → Message queue (SQS / Pub/Sub / Kafka)
3. **Workers** → GitHub REST API (triage/mutations) + Ticketing (Jira/ServiceNow) + Chat (Slack/Teams)
4. **Security Data Store** — Normalized findings for metrics, SLAs, and evidence

2.2 GitHub REST API Best Practices

GitHub publishes REST API usage best practices that should be followed:

- Avoid polling when webhooks are available
- Handle rate limits gracefully with exponential backoff
- Pause between mutating calls to prevent secondary rate limiting
- Use conditional requests (`If-None-Match`) where supported
- Implement pagination for large result sets

2.3 Authentication Strategy

For production deployments, GitHub App authentication is recommended over Personal Access Tokens (PATs):

- **GitHub Apps** — Scoped permissions, installation-based access, short-lived tokens
- **Fine-grained PATs** — User-bound, suitable for individual tooling
- **Classic PATs** — Legacy, avoid for new implementations

3 Python Building Blocks

This section provides foundational Python components for all subsequent automation implementations.

3.1 REST Client with Pagination and Rate-Limit Handling

The following client implements GitHub's recommended practices including proper headers, retry logic, and pagination support.

```
1 import os
2 import time
3 from typing import Any, Dict, Iterator, Optional
4 import requests
5
6 GITHUB_API = os.getenv("GITHUB_API", "https://api.github.com")
7
8 class GitHubClient:
9     """
10         Minimal GitHub REST client.
11         - Uses recommended Accept header
12         - Sends X-GitHub-Api-Version
13         - Basic retry/backoff on 429/5xx
14     """
15     def __init__(self, token: str, api_base: str = GITHUB_API) -> None:
16         self.api_base = api_base.rstrip("/")
17         self.session = requests.Session()
18         self.session.headers.update({
19             "Authorization": f"Bearer {token}",
20             "Accept": "application/vnd.github+json",
21             "X-GitHub-Api-Version": "2022-11-28",
22             "User-Agent": "appsec-automation/1.0",
23         })
24
25     def request(
26         self,
27         method: str,
28         path: str,
29         *,
30         params: Optional[dict] = None,
31         json: Optional[dict] = None
32     ) -> requests.Response:
33         url = f"{self.api_base}{path}"
34         for attempt in range(1, 7):
35             r = self.session.request(
36                 method, url, params=params, json=json, timeout=30
37             )
38
39             if r.status_code in (429, 500, 502, 503, 504):
40                 # Respect Retry-After when provided; otherwise exponential
41                 # backoff
42                 retry_after = r.headers.get("Retry-After")
43                 sleep_s = (
44                     int(retry_after)
45                     if retry_after and retry_after.isdigit()
46                     else min(2 ** attempt, 60)
47                 )
48                 time.sleep(sleep_s)
```

```

48         continue
49
50     if r.status_code >= 400:
51         raise RuntimeError(
52             f"GitHub API error {r.status_code}: {r.text}"
53         )
54
55     return r
56
57     raise RuntimeError(f"GitHub API failed after retries: {method} {path}")
58
59     def paginate(
60         self,
61         path: str,
62         *,
63         params: Optional[dict] = None
64     ) -> Iterator[Dict[str, Any]]:
65         page = 1
66         while True:
67             p = dict(params or {})
68             p.update({"per_page": 100, "page": page})
69             r = self.request("GET", path, params=p)
70             data = r.json()
71             if not isinstance(data, list):
72                 raise RuntimeError(
73                     f"Expected list response for pagination, got: {type(data)}"
74                 )
75             if not data:
76                 return
77             for item in data:
78                 yield item
79             page += 1

```

Listing 1: GitHub REST API Client

Implementation Notes:

- Uses Accept: application/vnd.github+json as recommended by GitHub
- Includes X-GitHub-Api-Version header for API stability
- Implements exponential backoff with configurable retry attempts
- Respects Retry-After header when provided by GitHub

3.2 GitHub App Authentication

GitHub's documented authentication flow for GitHub Apps: generate a JWT, then exchange it for an installation access token.

```

1 import time
2 import jwt  # PyJWT
3 import requests
4
5 def github_app_jwt(app_id: str, private_key_pem: str) -> str:
6     """Generate a JWT for GitHub App authentication."""
7     now = int(time.time())
8     payload = {
9         "iat": now - 60,                      # Issued at (with clock drift buffer)

```

```

10     "exp": now + (8 * 60),           # Expires in 8 minutes (under GitHub max)
11     "iss": app_id,                 # GitHub App ID
12 }
13 return jwt.encode(payload, private_key_pem, algorithm="RS256")
14
15
16 def github_installation_token(
17     api_base: str,
18     app_jwt: str,
19     installation_id: str
20 ) -> str:
21     """Exchange GitHub App JWT for installation access token."""
22     url = f"{api_base.rstrip('/')}app/installations/{installation_id}/
23         access_tokens"
24     r = requests.post(
25         url,
26         headers={
27             "Authorization": f"Bearer {app_jwt}",
28             "Accept": "application/vnd.github+json",
29             "X-GitHub-Api-Version": "2022-11-28",
30         },
31         timeout=30,
32     )
33     r.raise_for_status()
34     return r.json()["token"]
35
36 # Usage example
37 if __name__ == "__main__":
38     import os
39
40     app_id = os.environ["GITHUB_APP_ID"]
41     private_key = os.environ["GITHUB_APP_PRIVATE_KEY"]
42     installation_id = os.environ["GITHUB_INSTALLATION_ID"]
43     api_base = os.getenv("GITHUB_API", "https://api.github.com")
44
45     jwt_token = github_app_jwt(app_id, private_key)
46     install_token = github_installation_token(api_base, jwt_token,
47         installation_id)
48
49     # Use install_token with GitHubClient
50     client = GitHubClient(install_token, api_base)

```

Listing 2: GitHub App JWT and Installation Token Generation

Required Dependencies:

```
pip install PyJWT cryptography requests
```

4 Enablement & Policy Automation

Consistent security feature enablement across all repositories is foundational to a mature AppSec program.

4.1 Security Baseline Enforcement

4.1.1 Per-Repository Security Toggles

GitHub exposes repository-level security settings through the `security_and_analysis` properties. This approach is suitable for targeted enablement or drift remediation.

```

1 def enable_repo_security_baseline(
2     gh: GitHubClient,
3     owner: str,
4     repo: str
5 ) -> dict:
6     """
7         Enable GHAS security features on a single repository.
8
9     Enables:
10    - Advanced Security (required for GHAS features)
11    - Secret Scanning
12    - Push Protection
13    - Non-Provider Pattern detection
14 """
15 path = f"/repos/{owner}/{repo}"
16 payload = {
17     "security_and_analysis": {
18         "advanced_security": {"status": "enabled"},
19         "secret_scanning": {"status": "enabled"},
20         "secret_scanning_push_protection": {"status": "enabled"},
21         "secret_scanning_non_provider_patterns": {"status": "enabled"},
22     }
23 }
24 return gh.request("PATCH", path, json=payload).json()
25
26
27 def verify_repo_security_posture(
28     gh: GitHubClient,
29     owner: str,
30     repo: str
31 ) -> dict:
32     """
33         Retrieve current security configuration for a repository.
34     """
35     path = f"/repos/{owner}/{repo}"
36     response = gh.request("GET", path).json()
37     return response.get("security_and_analysis", {})

```

Listing 3: Enable Repository Security Baseline

4.1.2 Organization/Enterprise Security Configurations

For enterprise-scale deployments, security configurations provide centralized policy management. These configurations bundle multiple security features (Dependabot, Secret Scanning, Code Scanning default setup) into reusable policy objects.

```

1 def list_security_configurations(

```

```

2     gh: GitHubClient,
3     org: str
4 ) → list:
5     """List available security configurations for an organization."""
6     path = f"/orgs/{org}/code-security/configurations"
7     return list(gh.paginate(path))
8
9
10 def apply_security_configuration(
11     gh: GitHubClient,
12     org: str,
13     config_id: int,
14     repo_ids: list[int]
15 ) → dict:
16     """Apply a security configuration to specified repositories."""
17     path = f"/orgs/{org}/code-security/configurations/{config_id}/attach"
18     payload = {
19         "scope": "selected",
20         "selected_repository_ids": repo_ids
21     }
22     return gh.request("POST", path, json=payload).json()
23
24
25 def get_github_recommended_config(
26     gh: GitHubClient,
27     org: str
28 ) → dict | None:
29     """Find the GitHub-recommended security configuration."""
30     configs = list_security_configurations(gh, org)
31     for config in configs:
32         if config.get("name") == "GitHub recommended":
33             return config
34     return None

```

Listing 4: Apply Security Configuration at Scale

4.2 Code Scanning Default Setup

Code Scanning default setup standardizes CodeQL enablement and reduces per-repository configuration drift.

```

1 def get_code_scanning_default_setup(
2     gh: GitHubClient,
3     owner: str,
4     repo: str
5 ) → dict:
6     """Get current code scanning default setup configuration."""
7     path = f"/repos/{owner}/{repo}/code-scanning/default-setup"
8     return gh.request("GET", path).json()
9
10
11 def enable_code_scanning_default_setup(
12     gh: GitHubClient,
13     owner: str,
14     repo: str,
15     *,
16     query_suite: str = "default",
17     languages: list[str] | None = None

```

```

18     ) → dict:
19     """
20     Enable code scanning default setup.
21
22     Args:
23         gh: GitHub client
24         owner: Repository owner
25         repo: Repository name
26         query_suite: Query suite to use ('default', 'extended')
27         languages: Languages to scan (auto-detected if None)
28     """
29     path = f"/repos/{owner}/{repo}/code-scanning/default-setup"
30     payload = {
31         "state": "configured",
32         "query_suite": query_suite,
33     }
34     if languages:
35         payload["languages"] = languages
36
37     return gh.request("PATCH", path, json=payload).json()

```

Listing 5: Code Scanning Default Setup Management

4.3 Dependabot Configuration Management

Automate presence and quality of Dependabot configuration across repositories.

```

1 import yaml
2
3 def check_dependabot_config(
4     gh: GitHubClient,
5     owner: str,
6     repo: str
7 ) → dict:
8     """
9     Check if repository has valid dependabot.yml configuration.
10
11     Returns:
12         dict with 'exists', 'valid', 'ecosystems', and 'issues' keys
13     """
14     path = f"/repos/{owner}/{repo}/contents/.github/dependabot.yml"
15     result = {
16         "exists": False,
17         "valid": False,
18         "ecosystems": [],
19         "issues": []
20     }
21
22     try:
23         response = gh.request("GET", path)
24         content = response.json()
25
26         if content.get("encoding") == "base64":
27             import base64
28             yaml_content = base64.b64decode(content["content"]).decode("utf-8")
29             result["exists"] = True
30
31         try:

```

```

32     config = yaml.safe_load(yaml_content)
33     if config and "updates" in config:
34         result["valid"] = True
35         result["ecosystems"] = [
36             u.get("package-ecosystem")
37             for u in config.get("updates", [])
38         ]
39     else:
40         result["issues"].append("Missing 'updates' key")
41     except yaml.YAMLError as e:
42         result["issues"].append(f"YAML parse error: {e}")
43
44     except RuntimeError:
45         result["issues"].append("File not found")
46
47     return result
48
49
50 def list_dependabot_secrets(
51     gh: GitHubClient,
52     owner: str,
53     repo: str
54 ) -> list:
55     """List Dependabot secrets configured for private registry access."""
56     path = f"/repos/{owner}/{repo}/dependabot/secrets"
57     return list(gh.paginate(path))

```

Listing 6: Dependabot Configuration Validation

4.4 Push Protection Governance

Monitor and govern push protection bypasses for security oversight.

```

1 from datetime import datetime, timedelta
2
3
4 def list_push_protection_bypasses(
5     gh: GitHubClient,
6     owner: str,
7     repo: str,
8     *,
9     since: datetime | None = None
10 ) -> list:
11     """
12         List secret scanning alerts that were bypassed during push.
13
14         Bypassed secrets have 'push_protection_bypassed' = True
15     """
16     path = f"/repos/{owner}/{repo}/secret-scanning/alerts"
17     params = {"state": "open"}
18
19     alerts = list(gh.paginate(path, params=params))
20
21     bypassed = []
22     for alert in alerts:
23         if alert.get("push_protection_bypassed"):
24             bypassed_at = alert.get("push_protection_bypassed_at")
25             if bypassed_at and since:

```

```

26         alert_time = datetime.fromisoformat(
27             bypassed_at.replace("Z", "+00:00")
28         )
29         if alert_time < since:
30             continue
31         bypassed.append(alert)
32
33     return bypassed
34
35
36 def generate_bypass_report(
37     gh: GitHubClient,
38     org: str,
39     *,
40     days_back: int = 7
41 ) -> list[dict]:
42     """Generate a report of push protection bypasses across an org."""
43     since = datetime.utcnow() - timedelta(days=days_back)
44     report = []
45
46     # Get all org repos
47     repos_path = f"/orgs/{org}/repos"
48     for repo in gh.paginate(repos_path, params={"type": "all"}):
49         repo_name = repo["name"]
50         bypasses = list_push_protection_bypasses(
51             gh, org, repo_name, since=since
52         )
53
54         for bypass in bypasses:
55             report.append({
56                 "repository": f"{org}/{repo_name}",
57                 "alert_number": bypass.get("number"),
58                 "secret_type": bypass.get("secret_type"),
59                 "bypassed_by": bypass.get("push_protection_bypassed_by", {}).get("login"),
60                 "bypassed_at": bypass.get("push_protection_bypassed_at"),
61                 "reason": bypass.get("resolution_comment"),
62                 "url": bypass.get("html_url"),
63             })
64
65     return report

```

Listing 7: Push Protection Bypass Monitoring

5 Detection Pipeline Automation

Making security scans inevitable requires automation of workflow presence, external tool integration, and pipeline health monitoring.

5.1 SARIF Upload from External Scanners

When running scanners outside GitHub Actions, normalize results to SARIF and upload via the Code Scanning API.

```

1 import base64
2 import gzip
3 import json
4 from io import BytesIO
5
6
7 def _gzip_base64(data: bytes) -> str:
8     """Compress data with gzip and encode as base64."""
9     buf = BytesIO()
10    with gzip.GzipFile(fileobj=buf, mode="wb") as gz:
11        gz.write(data)
12    return base64.b64encode(buf.getvalue()).decode("utf-8")
13
14
15 def upload_sarif(
16     gh: GitHubClient,
17     owner: str,
18     repo: str,
19     *,
20     commit_sha: str,
21     ref: str,
22     sarif_dict: dict,
23     tool_name: str = "external-scanner"
24 ) -> dict:
25     """
26     Upload SARIF analysis results to GitHub Code Scanning.
27
28     Args:
29         gh: GitHub client
30         owner: Repository owner
31         repo: Repository name
32         commit_sha: Full commit SHA the analysis was run against
33         ref: Git ref (e.g., "refs/heads/main")
34         sarif_dict: SARIF document as Python dict
35         tool_name: Name of the scanning tool
36
37     Returns:
38         Upload response with 'id' and 'url' for status tracking
39     """
40     path = f"/repos/{owner}/{repo}/code-scanning/sarifs"
41     sarif_bytes = json.dumps(sarif_dict).encode("utf-8")
42
43     payload = {
44         "commit_sha": commit_sha,
45         "ref": ref,
46         "sarif": _gzip_base64(sarif_bytes),
47         "tool_name": tool_name,
48     }

```

```

49     return gh.request("POST", path, json=payload).json()
50
51
52
53 def check_sarif_upload_status(
54     gh: GitHubClient,
55     owner: str,
56     repo: str,
57     sarif_id: str
58 ) -> dict:
59     """Check the processing status of a SARIF upload."""
60     path = f"/repos/{owner}/{repo}/code-scanning/sarifs/{sarif_id}"
61     return gh.request("GET", path).json()

```

Listing 8: SARIF Upload to GitHub Code Scanning

5.2 SBOM Export Automation

Generate and export Software Bill of Materials (SBOM) from the dependency graph for GR-C/CMDB integration, release attachment, or risk engine feeds.

```

1 import json
2 from datetime import datetime
3
4
5 def export_sbom_spdx(
6     gh: GitHubClient,
7     owner: str,
8     repo: str
9 ) -> dict:
10     """
11     Export SBOM in SPDX format from GitHub Dependency Graph.
12
13     Returns:
14         SPDX-formatted SBOM document
15     """
16     path = f"/repos/{owner}/{repo}/dependency-graph/sbom"
17     return gh.request("GET", path).json()
18
19
20 def save_sbom_for_release(
21     gh: GitHubClient,
22     owner: str,
23     repo: str,
24     version: str,
25     output_dir: str = "."
26 ) -> str:
27     """
28     Export SBOM and save to file with version metadata.
29
30     Returns:
31         Path to saved SBOM file
32     """
33     sbom = export_sbom_spdx(gh, owner, repo)
34
35     # Add export metadata
36     sbom["_metadata"] = {
37         "exported_at": datetime.utcnow().isoformat() + "Z",

```

```

38     "repository": f"{owner}/{repo}",
39     "version": version,
40 }
41
42 filename = f"{output_dir}/{repo}-{version}-sbom.json"
43 with open(filename, "w") as f:
44     json.dump(sbom, f, indent=2)
45
46 return filename
47
48
49 def analyze_sbom_dependencies(sbom: dict) -> dict:
50     """
51     Analyze SBOM for dependency statistics.
52
53     Returns:
54         Summary of dependency counts by ecosystem
55     """
56     packages = sbom.get("sbom", {}).get("packages", [])
57
58     ecosystems = {}
59     for pkg in packages:
60         # SPDX uses externalRefs for package manager info
61         for ref in pkg.get("externalRefs", []):
62             if ref.get("referenceCategory") == "PACKAGE-MANAGER":
63                 ecosystem = ref.get("referenceType", "unknown")
64                 ecosystems[ecosystem] = ecosystems.get(ecosystem, 0) + 1
65                 break
66             else:
67                 ecosystems["unknown"] = ecosystems.get("unknown", 0) + 1
68
69     return {
70         "total_packages": len(packages),
71         "by_ecosystem": ecosystems,
72     }

```

Listing 9: SBOM Export in SPDX Format

5.3 Secret Scanning Health Monitoring

Use Secret Scanning scan history to detect lag or failures and alert teams proactively.

```

1 from datetime import datetime, timedelta
2
3
4 def get_secret_scanning_status(
5     gh: GitHubClient,
6     owner: str,
7     repo: str
8 ) -> dict:
9     """
10     Get secret scanning enablement and recent activity status.
11
12     Returns:
13         dict with 'enabled', 'last_scan', 'alert_count' keys
14     """
15     # Check if enabled
16     repo_path = f"/repos/{owner}/{repo}"

```

```

17     repo_info = gh.request("GET", repo_path).json()
18
19     security_analysis = repo_info.get("security_and_analysis", {})
20     secret_scanning = security_analysis.get("secret_scanning", {})
21
22     status = {
23         "enabled": secret_scanning.get("status") == "enabled",
24         "push_protection_enabled": security_analysis.get(
25             "secret_scanning_push_protection", {}
26         ).get("status") == "enabled",
27         "alert_count": 0,
28         "open_alerts": 0,
29     }
30
31     if status["enabled"]:
32         # Count alerts
33         alerts_path = f"/repos/{owner}/{repo}/secret-scanning/alerts"
34         all_alerts = list(gh.paginate(alerts_path))
35         status["alert_count"] = len(all_alerts)
36         status["open_alerts"] = len(
37             [a for a in all_alerts if a.get("state") == "open"]
38         )
39
40     return status
41
42
43 def org_secret_scanning_coverage(
44     gh: GitHubClient,
45     org: str
46 ) -> dict:
47     """
48     Generate secret scanning coverage report for an organization.
49
50     Returns:
51         dict with 'total_repos', 'enabled_count', 'coverage_pct'
52     """
53     repos_path = f"/orgs/{org}/repos"
54
55     total = 0
56     enabled = 0
57     push_protection_enabled = 0
58
59     for repo in gh.paginate(repos_path, params={"type": "all"}):
60         total += 1
61         security = repo.get("security_and_analysis", {})
62
63         if security.get("secret_scanning", {}).get("status") == "enabled":
64             enabled += 1
65         if security.get("secret_scanning_push_protection", {}).get("status") ==
66             "enabled":
67             push_protection_enabled += 1
68
69     return {
70         "total_repos": total,
71         "secret_scanning_enabled": enabled,
72         "push_protection_enabled": push_protection_enabled,
73         "coverage_pct": round(enabled / total * 100, 2) if total > 0 else 0,
74         "push_protection_pct": round(push_protection_enabled / total * 100, 2)
75             if total > 0 else 0,
76     }

```

74 | }

Listing 10: Secret Scanning Health Check

6 Triage Automation

Effective triage automation reduces noise, routes findings to appropriate owners, and enforces SLA compliance.

6.1 Secret Scanning Alert Management

```

1 def list_open_secret_alerts(
2     gh: GitHubClient,
3     owner: str,
4     repo: str
5 ) -> list:
6     """List all open secret scanning alerts for a repository."""
7     path = f"/repos/{owner}/{repo}/secret-scanning/alerts"
8     return list(gh.paginate(path, params={"state": "open"}))
9
10
11 def update_secret_alert(
12     gh: GitHubClient,
13     owner: str,
14     repo: str,
15     alert_number: int,
16     *,
17     state: str,
18     resolution: str | None = None,
19     resolution_comment: str | None = None
20 ) -> dict:
21     """
22     Update a secret scanning alert.
23
24     Args:
25         state: "open" or "resolved"
26         resolution: Required if resolving: "false_positive", "wont_fix",
27                     "revoked", "pattern_edited", "pattern_deleted", "
28                         used_in_tests"
29         resolution_comment: Optional comment explaining resolution
30     """
31     path = f"/repos/{owner}/{repo}/secret-scanning/alerts/{alert_number}"
32     payload = {"state": state}
33
34     if state == "resolved" and resolution:
35         payload["resolution"] = resolution
36     if resolution_comment:
37         payload["resolution_comment"] = resolution_comment
38
39     return gh.request("PATCH", path, json=payload).json()
40
41 def create_issue_from_secret_alert(
42     gh: GitHubClient,
43     owner: str,
44     repo: str,
45     alert: dict
46 ) -> dict:
47     """Create a GitHub Issue from a secret scanning alert."""
48     alert_number = alert.get("number")
49     secret_type = alert.get("secret_type", "secret")
50     url = alert.get("html_url", "")
```

```

51     title = f"[Secret Scanning] {secret_type} exposed (alert #{alert_number})"
52     body = f"""## Secret Scanning Alert
53
54 GitHub Secret Scanning detected a potential secret.
55
56 **Alert Details:***
57 - **Type:** {secret_type}
58 - **Alert URL:** {url}
59 - **State:** {alert.get('state', 'unknown')}
60
61 ## Triage Checklist
62
63 - [ ] Confirm validity of the detected secret
64 - [ ] Revoke/rotate the secret if valid
65 - [ ] Remove from git history if required
66 - [ ] Update secret storage location
67 - [ ] Document remediation in this issue
68
69 ## References
70
71 - [GitHub Secret Scanning Docs](https://docs.github.com/en/code-security/secret-scanning)
72 """
73
74
75     path = f"/repos/{owner}/{repo}/issues"
76     payload = {
77         "title": title,
78         "body": body,
79         "labels": ["security", "secret-scanning", "triage-needed"]
80     }
81
82     return gh.request("POST", path, json=payload).json()
83
84
85 def secret_scanning_to_issues(
86     gh: GitHubClient,
87     owner: str,
88     repo: str,
89     max_items: int = 20
90 ) -> list:
91 """
92 Create issues for open secret scanning alerts.
93
94 Returns:
95     List of created issues
96 """
97
98     alerts = list_open_secret_alerts(gh, owner, repo)[:max_items]
99     created_issues = []
100
101     for alert in alerts:
102         issue = create_issue_from_secret_alert(gh, owner, repo, alert)
103         created_issues.append(issue)
104
105     return created_issues

```

Listing 11: Secret Scanning Alert Operations

6.2 Code Scanning Alert Management

```

1 def list_code_scanning_alerts(
2     gh: GitHubClient,
3     owner: str,
4     repo: str,
5     state: str = "open",
6     *,
7     severity: str | None = None,
8     tool_name: str | None = None
9 ) -> list:
10     """
11     List code scanning alerts with optional filtering.
12
13     Args:
14         state: "open", "closed", "dismissed", or "fixed"
15         severity: Filter by severity (critical, high, medium, low, warning,
16                     note)
17         tool_name: Filter by scanning tool name
18     """
19     path = f"/repos/{owner}/{repo}/code-scanning/alerts"
20     params = {"state": state}
21
22     if severity:
23         params["severity"] = severity
24     if tool_name:
25         params["tool_name"] = tool_name
26
27     return list(gh.paginate(path, params=params))
28
29 def dismiss_code_scanning_alert(
30     gh: GitHubClient,
31     owner: str,
32     repo: str,
33     alert_number: int,
34     reason: str,
35     comment: str
36 ) -> dict:
37     """
38     Dismiss a code scanning alert.
39
40     Args:
41         reason: "false_positive", "won't_fix", or "used_in_tests"
42         comment: Explanation for dismissal (required for audit)
43     """
44     path = f"/repos/{owner}/{repo}/code-scanning/alerts/{alert_number}"
45     payload = {
46         "state": "dismissed",
47         "dismissed_reason": reason,
48         "dismissed_comment": comment,
49     }
50     return gh.request("PATCH", path, json=payload).json()
51
52 def get_code_scanning_alert_instances(
53     gh: GitHubClient,
54     owner: str,
55     repo: str,
56

```

```

57     alert_number: int
58 ) → list:
59     """Get all instances of a code scanning alert across branches."""
60     path = f"/repos/{owner}/{repo}/code-scanning/alerts/{alert_number}/
61         instances"
62     return list(gh.paginate(path))
63
64 def trigger_code_scanning_autofix(
65     gh: GitHubClient,
66     owner: str,
67     repo: str,
68     alert_number: int
69 ) → dict:
70     """
71     Trigger autofix generation for an eligible code scanning alert.
72
73     Note: Not all alerts are eligible for autofix.
74     """
75     path = f"/repos/{owner}/{repo}/code-scanning/alerts/{alert_number}/autofix"
76     return gh.request("POST", path).json()
77
78 def commit_code_scanning_autofix(
79     gh: GitHubClient,
80     owner: str,
81     repo: str,
82     alert_number: int,
83     *,
84     message: str | None = None
85 ) → dict:
86     """
87     Commit an autofix for a code scanning alert.
88
89     Args:
90         message: Custom commit message (optional)
91     """
92     path = f"/repos/{owner}/{repo}/code-scanning/alerts/{alert_number}/autofix/
93         commits"
94     payload = {}
95     if message:
96         payload["message"] = message
97
98     return gh.request("POST", path, json=payload).json()

```

Listing 12: Code Scanning Alert Operations

6.3 Dependabot Alert Management

```

1 def list_dependabot_alerts(
2     gh: GitHubClient,
3     owner: str,
4     repo: str,
5     state: str = "open",
6     *,
7     severity: str | None = None,
8     ecosystem: str | None = None
9 ) → list:

```

```

10     """
11     List Dependabot alerts with optional filtering.
12
13     Args:
14         state: "open", "dismissed", or "fixed"
15         severity: Filter by severity (critical, high, medium, low)
16         ecosystem: Filter by package ecosystem (npm, pip, maven, etc.)
17     """
18     path = f"/repos/{owner}/{repo}/dependabot/alerts"
19     params = {"state": state}
20
21     if severity:
22         params["severity"] = severity
23     if ecosystem:
24         params["ecosystem"] = ecosystem
25
26     return list(gh.paginate(path, params=params))
27
28
29 def dismiss_dependabot_alert(
30     gh: GitHubClient,
31     owner: str,
32     repo: str,
33     alert_number: int,
34     reason: str,
35     comment: str
36 ) -> dict:
37     """
38     Dismiss a Dependabot alert.
39
40     Args:
41         reason: "fix_started", "inaccurate", "no_bandwidth",
42                 "not_used", "tolerable_risk"
43         comment: Explanation for dismissal
44     """
45     path = f"/repos/{owner}/{repo}/dependabot/alerts/{alert_number}"
46     payload = {
47         "state": "dismissed",
48         "dismissed_reason": reason,
49         "dismissed_comment": comment,
50     }
51     return gh.request("PATCH", path, json=payload).json()
52
53
54 def get_dependabot_alert_details(
55     gh: GitHubClient,
56     owner: str,
57     repo: str,
58     alert_number: int
59 ) -> dict:
60     """Get detailed information about a Dependabot alert."""
61     path = f"/repos/{owner}/{repo}/dependabot/alerts/{alert_number}"
62     return gh.request("GET", path).json()
63
64
65 def prioritize_dependabot_alerts(
66     gh: GitHubClient,
67     owner: str,
68     repo: str

```

```

69     ) → dict:
70     """
71     Analyze and prioritize Dependabot alerts.
72
73     Returns:
74         Categorized alerts by priority
75     """
76
77     alerts = list_dependabot_alerts(gh, owner, repo)
78
79     prioritized = {
80         "critical": [],
81         "high": [],
82         "medium": [],
83         "low": []
84     }
85
86     for alert in alerts:
87         severity = alert.get("security_advisory", {}).get("severity", "low")
88         severity = severity.lower()
89
90         if severity in prioritized:
91             prioritized[severity].append({
92                 "number": alert.get("number"),
93                 "package": alert.get("dependency", {}).get("package", {}).get(
94                     "name"),
95                 "ecosystem": alert.get("dependency", {}).get("package", {}).get(
96                     "ecosystem"),
97                 "vulnerable_version": alert.get("security_vulnerability", {}).get(
98                     "vulnerable_version_range"),
99                 "fixed_in": alert.get("security_vulnerability", {}).get(
100                     "first_patched_version", {}).get("identifier"),
101                 "cvss_score": alert.get("security_advisory", {}).get("cvss",
102                     {}).get("score"),
103                 "url": alert.get("html_url"),
104             })
105
106     return prioritized

```

Listing 13: Dependabot Alert Operations

6.4 Unified Alert Ingestion

Normalize alerts from multiple sources into a unified schema for consistent processing.

```

1  from dataclasses import dataclass, astype
2  from datetime import datetime
3  from enum import Enum
4  from typing import Optional
5
6
7  class AlertSource(Enum):
8      SECRET_SCANNING = "secret_scanning"
9      CODE_SCANNING = "code_scanning"
10     DEPENDABOT = "dependabot"
11
12
13 class AlertSeverity(Enum):
14     CRITICAL = "critical"

```

```

15     HIGH = "high"
16     MEDIUM = "medium"
17     LOW = "low"
18     NOTE = "note"
19
20
21 @dataclass
22 class UnifiedAlert:
23     """Normalized alert schema for cross-tool processing."""
24     id: str                                     # Unique identifier
25     source: AlertSource                         # Origin tool
26     repository: str                            # owner/repo
27     severity: AlertSeverity                   # Normalized severity
28     title: str                                  # Human-readable title
29     description: str                           # Alert details
30     state: str                                 # open/resolved/dismissed
31     created_at: datetime                       # When created
32     url: str                                   # Link to alert
33
34     # Optional fields
35     cwe_id: Optional[str] = None
36     cvss_score: Optional[float] = None
37     rule_id: Optional[str] = None
38     file_path: Optional[str] = None
39     line_number: Optional[int] = None
40
41     # SLA tracking
42     sla_deadline: Optional[datetime] = None
43     assigned_to: Optional[str] = None
44     ticket_id: Optional[str] = None
45
46
47 def normalize_secret_alert(
48     alert: dict,
49     owner: str,
50     repo: str
51 ) -> UnifiedAlert:
52     """Convert secret scanning alert to unified schema."""
53     return UnifiedAlert(
54         id=f"secret-{owner}-{repo}-{alert['number']}",
55         source=AlertSource.SECRET_SCANNING,
56         repository=f"{owner}/{repo}",
57         severity=AlertSeverity.CRITICAL, # Secrets always critical
58         title=f"Exposed {alert.get('secret_type', 'secret')}",
59         description=f"Secret type: {alert.get('secret_type_display_name', 'Unknown')}",
60         state=alert.get("state", "open"),
61         created_at=datetime.fromisoformat(
62             alert["created_at"].replace("Z", "+00:00")
63         ),
64         url=alert.get("html_url", ""),
65         file_path=alert.get("secret", {}).get("path"),
66     )
67
68
69 def normalize_code_alert(
70     alert: dict,
71     owner: str,
72     repo: str

```

```

73     ) -> UnifiedAlert:
74         """Convert code scanning alert to unified schema."""
75         severity_map = {
76             "critical": AlertSeverity.CRITICAL,
77             "high": AlertSeverity.HIGH,
78             "medium": AlertSeverity.MEDIUM,
79             "low": AlertSeverity.LOW,
80             "warning": AlertSeverity.MEDIUM,
81             "note": AlertSeverity.NOTE,
82         }
83
84         rule = alert.get("rule", {})
85         location = alert.get("most_recent_instance", {}).get("location", {})
86
87         return UnifiedAlert(
88             id=f"code-{owner}-{repo}-{alert['number']}",
89             source=AlertSource.CODE_SCANNING,
90             repository=f"{owner}/{repo}",
91             severity=severity_map.get(
92                 alert.get("rule", {}).get("severity", "medium").lower(),
93                 AlertSeverity.MEDIUM
94             ),
95             title=rule.get("description", "Code scanning alert"),
96             description=rule.get("full_description", ""),
97             state=alert.get("state", "open"),
98             created_at=datetime.fromisoformat(
99                 alert["created_at"].replace("Z", "+00:00")
100            ),
101            url=alert.get("html_url", ""),
102            cwe_id=next((t["name"] for t in rule.get("tags", []))
103                        if t.startswith("CWE-"), None),
104            rule_id=rule.get("id"),
105            file_path=location.get("path"),
106            line_number=location.get("start_line"),
107        )
108
109
110    def normalize_dependabot_alert(
111        alert: dict,
112        owner: str,
113        repo: str
114    ) -> UnifiedAlert:
115        """Convert Dependabot alert to unified schema."""
116        severity_map = {
117            "critical": AlertSeverity.CRITICAL,
118            "high": AlertSeverity.HIGH,
119            "medium": AlertSeverity.MEDIUM,
120            "low": AlertSeverity.LOW,
121        }
122
123        advisory = alert.get("security_advisory", {})
124        dependency = alert.get("dependency", {})
125        package = dependency.get("package", {})
126
127        return UnifiedAlert(
128            id=f"dep-{owner}-{repo}-{alert['number']}",
129            source=AlertSource.DEPENDABOT,
130            repository=f"{owner}/{repo}",
131            severity=severity_map.get(

```

```

132     advisory.get("severity", "medium").lower(),
133     AlertSeverity.MEDIUM
134 ),
135     title=f"Vulnerable dependency: {package.get('name', 'unknown')}",
136     description=advisory.get("summary", ""),
137     state=alert.get("state", "open"),
138     created_at=datetime.fromisoformat(
139         alert["created_at"].replace("Z", "+00:00")
140     ),
141     url=alert.get("html_url", ""),
142     cvss_score=advisory.get("cvss", {}).get("score"),
143     cwe_id=next((cwe.get("cwe_id") for cwe in advisory.get("cves", []))),
144         None),
145     file_path=dependency.get("manifest_path"),
146 )
147
148 def ingest_all_alerts(
149     gh: GitHubClient,
150     owner: str,
151     repo: str
152 ) -> list[UnifiedAlert]:
153     """
154     Ingest all security alerts from a repository into unified format.
155
156     Returns:
157         List of UnifiedAlert objects from all sources
158     """
159     unified = []
160
161     # Secret scanning
162     secrets = list_open_secret_alerts(gh, owner, repo)
163     unified.extend(normalize_secret_alert(a, owner, repo) for a in secrets)
164
165     # Code scanning
166     code_alerts = list_code_scanning_alerts(gh, owner, repo)
167     unified.extend(normalize_code_alert(a, owner, repo) for a in code_alerts)
168
169     # Dependabot
170     dep_alerts = list_dependabot_alerts(gh, owner, repo)
171     unified.extend(normalize_dependabot_alert(a, owner, repo) for a in
172         dep_alerts)
173
174     return unified

```

Listing 14: Unified Alert Schema and Ingestion

7 Webhook-Driven Event Processing

Event-driven architecture using GitHub webhooks eliminates polling overhead and enables real-time response to security events.

7.1 Webhook Receiver Implementation

```

1 import hmac
2 import hashlib
3 import os
4 import json
5 from flask import Flask, request, abort, jsonify
6 from typing import Callable
7
8 app = Flask(__name__)
9 WEBHOOK_SECRET = os.environ["GITHUB_WEBHOOK_SECRET"].encode("utf-8")
10
11 # Event handlers registry
12 _handlers: dict[str, list[Callable]] = {}
13
14
15 def register_handler(event_type: str):
16     """Decorator to register event handlers."""
17     def decorator(func: Callable):
18         if event_type not in _handlers:
19             _handlers[event_type] = []
20         _handlers[event_type].append(func)
21         return func
22     return decorator
23
24
25 def verify_signature(req) → None:
26     """Verify GitHub webhook signature using HMAC-SHA256."""
27     sig = req.headers.get("X-Hub-Signature-256", "")
28     if not sig.startswith("sha256="):
29         abort(401, "Missing signature")
30
31     expected = hmac.new(
32         WEBHOOK_SECRET,
33         req.data,
34         hashlib.sha256
35     ).hexdigest()
36
37     if not hmac.compare_digest(sig, f"sha256={expected}"):
38         abort(401, "Invalid signature")
39
40
41 @app.post("/webhook")
42 def webhook():
43     """Main webhook endpoint."""
44     verify_signature(request)
45
46     event = request.headers.get("X-GitHub-Event", "")
47     delivery_id = request.headers.get("X-GitHub-Delivery", "")
48     payload = request.get_json(silent=True) or {}
49
50     app.logger.info(f"Received {event} event (delivery: {delivery_id})")
51

```

```

52     # Dispatch to registered handlers
53     handlers = _handlers.get(event, [])
54     results = []
55
56     for handler in handlers:
57         try:
58             result = handler(payload)
59             results.append({"handler": handler.__name__, "result": result})
60         except Exception as e:
61             app.logger.error(f"Handler {handler.__name__} failed: {e}")
62             results.append({"handler": handler.__name__, "error": str(e)})
63
64     return jsonify({
65         "ok": True,
66         "event": event,
67         "delivery_id": delivery_id,
68         "handlers_executed": len(handlers),
69         "results": results,
70     })
71
72
73     # Example event handlers
74
75 @register_handler("secret_scanning_alert")
76 def handle_secret_alert(payload: dict) -> dict:
77     """Process secret scanning alert events."""
78     action = payload.get("action")
79     alert = payload.get("alert", {})
80     repo = payload.get("repository", {})
81
82     if action == "created":
83         # New secret detected - trigger incident workflow
84         return {
85             "action": "incident_created",
86             "alert_number": alert.get("number"),
87             "secret_type": alert.get("secret_type"),
88             "repository": repo.get("full_name"),
89         }
90
91     elif action == "resolved":
92         # Secret resolved - update ticket
93         return {
94             "action": "incident_resolved",
95             "alert_number": alert.get("number"),
96             "resolution": alert.get("resolution"),
97         }
98
99     return {"action": action, "handled": True}
100
101
102 @register_handler("code_scanning_alert")
103 def handle_code_alert(payload: dict) -> dict:
104     """Process code scanning alert events."""
105     action = payload.get("action")
106     alert = payload.get("alert", {})
107
108     return {
109         "action": action,
110         "alert_number": alert.get("number"),

```

```

111     "rule_id": alert.get("rule", {}).get("id"),
112     "severity": alert.get("rule", {}).get("severity"),
113 }
114
115
116 @register_handler("dependabot_alert")
117 def handle_dependabot_alert(payload: dict) → dict:
118     """Process Dependabot alert events."""
119     action = payload.get("action")
120     alert = payload.get("alert", {})
121
122     return {
123         "action": action,
124         "alert_number": alert.get("number"),
125         "package": alert.get("dependency", {}).get("package", {}).get("name"),
126         "severity": alert.get("security_advisory", {}).get("severity"),
127     }
128
129
130 if __name__ == "__main__":
131     app.run(host="0.0.0.0", port=8080, debug=False)

```

Listing 15: Flask-based Webhook Receiver

7.2 Queue-Based Event Processing

For production deployments, decouple webhook receipt from processing using a message queue.

```

1 import json
2 import boto3
3 from dataclasses import dataclass, astype
4 from datetime import datetime
5
6
7 @dataclass
8 class SecurityEvent:
9     """Standardized security event for queue dispatch."""
10    event_id: str
11    event_type: str
12    action: str
13    repository: str
14    alert_number: int | None
15    severity: str | None
16    timestamp: str
17    payload: dict
18
19
20 class EventDispatcher:
21     """Dispatch security events to SQS queue for async processing."""
22
23     def __init__(self, queue_url: str, region: str = "us-east-1"):
24         self.sqs = boto3.client("sns", region_name=region)
25         self.queue_url = queue_url
26
27     def dispatch(self, event: SecurityEvent) → str:
28         """Send event to queue. Returns message ID."""
29         response = self.sqs.send_message(
30             QueueUrl=self.queue_url,

```

```

31     MessageBody=json.dumps(asdict(event)),
32     MessageAttributes={
33         "EventType": {
34             "StringValue": event.event_type,
35             "DataType": "String"
36         },
37         "Severity": {
38             "StringValue": event.severity or "unknown",
39             "DataType": "String"
40         },
41     }
42 )
43     return response["MessageId"]
44
45
46 def webhook_to_event(
47     event_type: str,
48     payload: dict,
49     delivery_id: str
50 ) -> SecurityEvent:
51     """Convert webhook payload to SecurityEvent."""
52     repo = payload.get("repository", {})
53     alert = payload.get("alert", {})
54
55     # Determine severity based on event type
56     severity = None
57     if event_type == "secret_scanning_alert":
58         severity = "critical"
59     elif event_type == "code_scanning_alert":
60         severity = alert.get("rule", {}).get("severity")
61     elif event_type == "dependabot_alert":
62         severity = alert.get("security_advisory", {}).get("severity")
63
64     return SecurityEvent(
65         event_id=delivery_id,
66         event_type=event_type,
67         action=payload.get("action", "unknown"),
68         repository=repo.get("full_name", ""),
69         alert_number=alert.get("number"),
70         severity=severity,
71         timestamp=datetime.utcnow().isoformat() + "Z",
72         payload=payload,
73     )

```

Listing 16: Queue-Based Event Dispatcher

8 Reporting & Metrics Automation

Automated reporting proves program outcomes and maintains audit evidence for compliance.

8.1 Security KPI Generation

```

1  from collections import defaultdict
2  from datetime import datetime, timedelta
3  from dataclasses import dataclass
4
5
6  @dataclass
7  class SecurityKPIs:
8      """Key Performance Indicators for security program."""
9      total_open_critical: int
10     total_open_high: int
11     mttr_secrets_days: float
12     mttr_code_days: float
13     mttr_dependencies_days: float
14     top_cwes: list[tuple[str, int]]
15     reopen_rate: float
16     coverage_pct: float
17
18
19     def calculate_mttr(
20         alerts: list[dict],
21         created_field: str = "created_at",
22         resolved_field: str = "fixed_at"
23     ) -> float:
24         """
25             Calculate Mean Time To Remediation in days.
26
27             Only considers resolved alerts with both timestamps.
28         """
29         durations = []
30
31         for alert in alerts:
32             if alert.get("state") not in ("fixed", "resolved"):
33                 continue
34
35             created = alert.get(created_field)
36             resolved = alert.get(resolved_field) or alert.get("dismissed_at")
37
38             if created and resolved:
39                 created_dt = datetime.fromisoformat(created.replace("Z", "+00:00"))
40                 resolved_dt = datetime.fromisoformat(resolved.replace("Z", "+00:00"))
41                 duration = (resolved_dt - created_dt).total_seconds() / 86400
42                 durations.append(duration)
43
44         return sum(durations) / len(durations) if durations else 0.0
45
46
47     def count_by_severity(alerts: list[dict]) -> dict[str, int]:
48         """Count open alerts by severity."""
49         counts = defaultdict(int)
50
51         for alert in alerts:

```

```

52     if alert.get("state") != "open":
53         continue
54
55     # Handle different alert types
56     severity = (
57         alert.get("severity") or
58         alert.get("rule", {}).get("severity") or
59         alert.get("security_advisory", {}).get("severity") or
60         "unknown"
61     )
62     counts[severity.lower()] += 1
63
64 return dict(counts)
65
66
67 def top_cwes(alerts: list[dict], top_n: int = 10) → list[tuple[str, int]]:
68     """Extract top CWEs from code scanning alerts."""
69     cwe_counts = defaultdict(int)
70
71     for alert in alerts:
72         rule = alert.get("rule", {})
73         tags = rule.get("tags", [])
74
75         for tag in tags:
76             if isinstance(tag, str) and tag.startswith("CWE-"):
77                 cwe_counts[tag] += 1
78
79     return sorted(cwe_counts.items(), key=lambda x: x[1], reverse=True)[:top_n]
80
81
82 def generate_org_kpis(
83     gh: GitHubClient,
84     org: str,
85     *,
86     days_back: int = 30
87 ) → dict:
88     """
89     Generate organization-wide security KPIs.
90
91     Returns:
92         Comprehensive KPI report
93     """
94     since = datetime.utcnow() - timedelta(days=days_back)
95
96     all_secrets = []
97     all_code = []
98     all_deps = []
99
100    repos_path = f"/orgs/{org}/repos"
101
102    for repo in gh.paginate(repos_path, params={"type": "all"}):
103        repo_name = repo["name"]
104
105        try:
106            secrets = list(gh.paginate(
107                f"/repos/{org}/{repo_name}/secret-scanning/alerts"
108            ))
109            all_secrets.extend(secrets)
110        except RuntimeError:

```

```

111     pass
112
113     try:
114         code = list(gh.paginate(
115             f"/repos/{org}/{repo_name}/code-scanning/alerts"
116         ))
117         all_code.extend(code)
118     except RuntimeError:
119         pass
120
121     try:
122         deps = list(gh.paginate(
123             f"/repos/{org}/{repo_name}/dependabot/alerts"
124         ))
125         all_deps.extend(deps)
126     except RuntimeError:
127         pass
128
129     # Calculate metrics
130     secret_severity = count_by_severity(all_secrets)
131     code_severity = count_by_severity(all_code)
132     dep_severity = count_by_severity(all_deps)
133
134     return {
135         "period_days": days_back,
136         "generated_at": datetime.utcnow().isoformat() + "Z",
137         "summary": {
138             "total_open_critical": (
139                 secret_severity.get("critical", 0) +
140                 code_severity.get("critical", 0) +
141                 dep_severity.get("critical", 0)
142             ),
143             "total_open_high": (
144                 secret_severity.get("high", 0) +
145                 code_severity.get("high", 0) +
146                 dep_severity.get("high", 0)
147             ),
148         },
149         "mttr": {
150             "secrets_days": calculate_mttr(all_secrets),
151             "code_scanning_days": calculate_mttr(all_code),
152             "dependencies_days": calculate_mttr(all_deps),
153         },
154         "by_source": {
155             "secret_scanning": {
156                 "total": len(all_secrets),
157                 "by_severity": secret_severity,
158             },
159             "code_scanning": {
160                 "total": len(all_code),
161                 "by_severity": code_severity,
162                 "top_cwes": top_cwes(all_code),
163             },
164             "dependabot": {
165                 "total": len(all_deps),
166                 "by_severity": dep_severity,
167             },
168         },
169     }

```

Listing 17: Security KPI Calculation

8.2 Audit Log Integration

GitHub audit logs capture security-relevant events for compliance evidence.

```

1  from datetime import datetime, timedelta
2
3
4  def export_audit_log(
5      gh: GitHubClient,
6      enterprise: str,
7      *,
8      days_back: int = 30,
9      include_phrases: list[str] | None = None
10 ) -> list[dict]:
11     """
12         Export enterprise audit log entries.
13
14     Args:
15         enterprise: Enterprise slug
16         days_back: Number of days to export
17         include_phrases: Filter to entries containing these phrases
18
19     Returns:
20         List of audit log entries
21     """
22
23     # Calculate date range
24     end_date = datetime.utcnow()
25     start_date = end_date - timedelta(days=days_back)
26
27     path = f"/enterprises/{enterprise}/audit-log"
28     params = {
29         "phrase": " ".join(include_phrases) if include_phrases else "",
30         "include": "all",
31         "order": "desc",
32     }
33
34     entries = list(gh.paginate(path, params=params))
35
36     # Filter by date
37     filtered = []
38     for entry in entries:
39         created = entry.get("created_at") or entry.get("@timestamp")
40         if created:
41             entry_dt = datetime.fromisoformat(created.replace("Z", "+00:00"))
42             if entry_dt >= start_date.replace(tzinfo=entry_dt.tzinfo):
43                 filtered.append(entry)
44
45     return filtered
46
47 def extract_security_events(entries: list[dict]) -> list[dict]:
48     """
49         Filter audit log for security-relevant events.
50

```

```

51     Includes:
52     - Secret scanning events
53     - Code scanning configuration changes
54     - Security feature enablement/disablement
55     - Push protection bypasses
56     """
57
58     security_actions = {
59         "secret_scanning.disable",
60         "secret_scanning.enable",
61         "secret_scanning_alert.create",
62         "secret_scanning_alert.resolve",
63         "secret_scanning_push_protection.disable",
64         "secret_scanning_push_protection.enable",
65         "secret_scanning_push_protection.bypass",
66         "code_scanning.disable",
67         "code_scanning.enable",
68         "dependabot_alerts.disable",
69         "dependabot_alerts.enable",
70         "repository_vulnerability_alert.create",
71         "repository_vulnerability_alert.dismiss",
72     }
73
74     return [
75         entry for entry in entries
76         if entry.get("action") in security_actions
77     ]
78
79 def generate_compliance_report(
80     gh: GitHubClient,
81     enterprise: str,
82     *,
83     days_back: int = 30
84 ) -> dict:
85     """
86     Generate compliance report for audit purposes.
87
88     Includes:
89     - Security feature changes
90     - Alert activity summary
91     - Bypass events
92     """
93
94     entries = export_audit_log(gh, enterprise, days_back=days_back)
95     security_entries = extract_security_events(entries)
96
97     # Categorize events
98     report = {
99         "period_days": days_back,
100        "generated_at": datetime.utcnow().isoformat() + "Z",
101        "total_security_events": len(security_entries),
102        "feature_changes": [],
103        "bypasses": [],
104        "alerts_created": 0,
105        "alerts_resolved": 0,
106    }
107
108    for entry in security_entries:
109        action = entry.get("action", "")
```

```
110     if "bypass" in action:
111         report["bypasses"].append({
112             "action": action,
113             "actor": entry.get("actor"),
114             "repository": entry.get("repo"),
115             "timestamp": entry.get("created_at"),
116         })
117     elif "disable" in action or "enable" in action:
118         report["feature_changes"].append({
119             "action": action,
120             "actor": entry.get("actor"),
121             "repository": entry.get("repo"),
122             "timestamp": entry.get("created_at"),
123         })
124     elif "create" in action:
125         report["alerts_created"] += 1
126     elif "resolve" in action or "dismiss" in action:
127         report["alerts_resolved"] += 1
128
129 return report
```

Listing 18: Audit Log Export

9 Harness CD Pipeline Integration

Harness CD serves as the release orchestrator between artifact output and deployment targets, providing the optimal enforcement point for security gates.

9.1 Integration Architecture

The integration preserves existing shift-left controls while adding release-time enforcement:

- **Shift-left controls** remain anchored in GitHub events (GHAS, Polaris SAST, CodeQL)
- **Artifact controls** remain tied to built images/binaries (Trivy scans)
- **Runtime controls** remain in staging/test environments (IAST, DAST, manual testing)
- **Central Intake Queue** remains the normalization, deduplication, and SLA hub
- **Harness CD** becomes the release gate enforcement layer

9.1.1 Recommended Pipeline Structure

1. **Pre-Deploy Security Gate (blocking)** — Query GHAS and optionally the Intake Queue; fail stage if policy violated
2. **Deploy to Staging**
3. **Post-Deploy Security Tests** — Run DAST/IAST; publish results to Intake Queue
4. **Promotion Gate** — Approval step + “no new criticals since deploy” check
5. **Deploy to Production**

9.2 Triggering Harness CD Pipelines

Harness documents pipeline execution via the `/v1/orgs/{org}/projects/{project}/pipelines/{pipeline}/exec` endpoint.

```

1 import os
2 import requests
3
4
5 def trigger_harness_cd(
6     *,
7     harness_base: str,
8     account_id: str,
9     api_key: str,
10    org: str,
11    project: str,
12    pipeline_id: str,
13    inputs_yaml: str,
14 ) -> dict:
15     """
16         Trigger a Harness CD pipeline execution.
17
18     Args:
19         harness_base: Harness API base URL
20             (e.g., "https://app.harness.io/gateway/pipeline/api")
21         account_id: Harness account identifier

```

```
22     api_key: Harness API key
23     org: Harness organization identifier
24     project: Harness project identifier
25     pipeline_id: Pipeline to execute
26     inputs_yaml: YAML string with pipeline inputs
27
28     Returns:
29         Execution response with run details
30     """
31
32     url = (
33         f"{harness_base.rstrip('/')}v1/orgs/{org}/projects/{project}"
34         f"/pipelines/{pipeline_id}/execute"
35     )
36
37     response = requests.post(
38         url,
39         params={"module": "CD"},
40         headers={
41             "Harness-Account": account_id,
42             "x-api-key": api_key,
43             "Content-Type": "application/json",
44             "Accept": "application/json",
45         },
46         json={"inputs_yaml": inputs_yaml},
47         timeout=30,
48     )
49     response.raise_for_status()
50     return response.json()
51
52 def check_pipeline_status(
53     *,
54     harness_base: str,
55     account_id: str,
56     api_key: str,
57     org: str,
58     project: str,
59     pipeline_id: str,
60     execution_id: str,
61 ) -> dict:
62     """Check status of a pipeline execution."""
63     url = (
64         f"{harness_base.rstrip('/')}v1/orgs/{org}/projects/{project}"
65         f"/pipelines/{pipeline_id}/executions/{execution_id}"
66     )
67
68     response = requests.get(
69         url,
70         headers={
71             "Harness-Account": account_id,
72             "x-api-key": api_key,
73             "Accept": "application/json",
74         },
75         timeout=30,
76     )
77     response.raise_for_status()
78     return response.json()
```

```

81 if __name__ == "__main__":
82     # Example usage
83     resp = trigger_harness_cd(
84         harness_base=os.environ["HARNESS_BASE"],
85         account_id=os.environ["HARNESS_ACCOUNT"],
86         api_key=os.environ["HARNESS_API_KEY"],
87         org=os.environ["HARNESS_ORG"],
88         project=os.environ["HARNESS_PROJECT"],
89         pipeline_id=os.environ["HARNESS_PIPELINE_ID"],
90         inputs_yaml=os.environ["HARNESS_INPUTS_YAML"],
91     )
92     print(f"Pipeline execution started: {resp}")

```

Listing 19: Trigger Harness CD Pipeline Execution

9.3 Security Gate Implementation

The security gate queries GHAS alerts and blocks releases based on policy thresholds.

```

1  #!/usr/bin/env python3
2  """
3      Security Gate for Harness CD Pipeline
4
5      Checks GHAS alerts and blocks deployment if policy thresholds are exceeded.
6
7      Environment Variables:
8          GITHUB_TOKEN: GitHub token with security alert read access
9          GITHUB_OWNER: Repository owner
10         GITHUB_REPO: Repository name
11         GITHUB_API: GitHub API base URL (optional, defaults to github.com)
12         SECURITY_GATE_MIN_SEVERITY: Minimum severity to block (default: high)
13
14      Exit Codes:
15          0: Security gate passed
16          1: Configuration error
17          2: Blocked - open secret scanning alerts
18          3: Blocked - code scanning alerts at/above threshold
19          4: Blocked - dependabot alerts at/above threshold
20      """
21
22      import os
23      import sys
24      import requests
25      from typing import Any, Dict, List
26
27      GITHUB_API = os.getenv("GITHUB_API", "https://api.github.com")
28
29      SEV_ORDER = {"low": 1, "medium": 2, "high": 3, "critical": 4}
30
31
32      def gh_get(
33          token: str,
34          path: str,
35          params: Dict[str, Any] | None = None
36      ) -> Any:
37          """Make authenticated GET request to GitHub API."""
38          url = f"{GITHUB_API.rstrip('/')}/{path}"
39          response = requests.get(

```

```

40         url,
41         headers={
42             "Authorization": f"Bearer {token}",
43             "Accept": "application/vnd.github+json",
44             "X-GitHub-Api-Version": "2022-11-28",
45         },
46         params=params,
47         timeout=30,
48     )
49     response.raise_for_status()
50     return response.json()
51
52
53 def list_all_pages(
54     token: str,
55     path: str,
56     params: Dict[str, Any] | None = None
57 ) -> List[Dict[str, Any]]:
58     """Paginate through all results."""
59     out: List[Dict[str, Any]] = []
60     page = 1
61
62     while True:
63         p = dict(params or {})
64         p.update({"per_page": 100, "page": page})
65         data = gh_get(token, path, p)
66
67         if not isinstance(data, list) or not data:
68             break
69
70         out.extend(data)
71         page += 1
72
73     return out
74
75
76 def sev_at_least(item_sev: str | None, threshold: str) -> bool:
77     """Check if severity meets or exceeds threshold."""
78     if not item_sev:
79         return False
80     return SEV_ORDER.get(item_sev.lower(), 0) >= SEV_ORDER[threshold.lower()]
81
82
83 def get_alert_severity(alert: dict) -> str | None:
84     """Extract severity from different alert types."""
85     # Try direct severity field
86     if "severity" in alert:
87         return alert["severity"]
88
89     # Code scanning: rule.severity
90     if "rule" in alert:
91         return alert.get("rule", {}).get("severity")
92
93     # Dependabot: security_advisory.severity
94     if "security_advisory" in alert:
95         return alert.get("security_advisory", {}).get("severity")
96
97     return None
98

```

```
99
100 def main() -> int:
101     """Run security gate checks."""
102     # Validate configuration
103     token = os.environ.get("GITHUB_TOKEN")
104     owner = os.environ.get("GITHUB_OWNER")
105     repo = os.environ.get("GITHUB_REPO")
106     threshold = os.getenv("SECURITY_GATE_MIN_SEVERITY", "high").lower()
107
108     if not all([token, owner, repo]):
109         print("ERROR: Missing required environment variables")
110         print("Required: GITHUB_TOKEN, GITHUB_OWNER, GITHUB_REPO")
111         return 1
112
113     print("=" * 60)
114     print("SECURITY GATE CHECK")
115     print("=" * 60)
116     print(f"Repository: {owner}/{repo}")
117     print(f"Severity Threshold: {threshold}")
118     print()
119
120     # Fetch alerts
121     print("Fetching security alerts...")
122
123     try:
124         secrets = list_all_pages(
125             token,
126             f"/repos/{owner}/{repo}/secret-scanning/alerts",
127             {"state": "open"}
128         )
129     except Exception as e:
130         print(f"  Secret Scanning: Error - {e}")
131         secrets = []
132
133     try:
134         code_alerts = list_all_pages(
135             token,
136             f"/repos/{owner}/{repo}/code-scanning/alerts",
137             {"state": "open"}
138         )
139     except Exception as e:
140         print(f"  Code Scanning: Error - {e}")
141         code_alerts = []
142
143     try:
144         dep_alerts = list_all_pages(
145             token,
146             f"/repos/{owner}/{repo}/dependabot/alerts",
147             {"state": "open"}
148         )
149     except Exception as e:
150         print(f"  Dependabot: Error - {e}")
151         dep_alerts = []
152
153     # Filter by severity
154     code_blockers = [
155         a for a in code_alerts
156         if sev_at_least(get_alert_severity(a), threshold)
157     ]
```

```

158     dep_blockers = [
159         a for a in dep_alerts
160         if sev_at_least(get_alert_severity(a), threshold)
161     ]
162
163     # Report findings
164     print()
165     print("=" * 60)
166     print("FINDINGS SUMMARY")
167     print("=" * 60)
168     print(f"Secret Scanning Alerts (open): {len(secrets)}")
169     print(f"Code Scanning Alerts (>= {threshold}): "
170           f"{len(code_blockers)} / {len(code_alerts)} total")
171     print(f"Dependabot Alerts (>= {threshold}): "
172           f"{len(dep_blockers)} / {len(dep_alerts)} total")
173     print()
174
175     # Policy enforcement
176     print("=" * 60)
177     print("POLICY EVALUATION")
178     print("=" * 60)
179
180     # Rule 1: Any open secrets block deployment
181     if secrets:
182         print("BLOCKED: Open secret scanning alerts present")
183         print()
184         print("Blocking alerts:")
185         for alert in secrets[:5]: # Show first 5
186             print(f" - #{alert.get('number')}: {alert.get('secret_type')}")
187         if len(secrets) > 5:
188             print(f" ... and {len(secrets) - 5} more")
189     return 2
190
191     # Rule 2: High+ code scanning alerts block
192     if code_blockers:
193         print(f"BLOCKED: Code scanning alerts at/above {threshold}")
194         print()
195         print("Blocking alerts:")
196         for alert in code_blockers[:5]:
197             rule = alert.get("rule", {})
198             print(f" - #{alert.get('number')}: {rule.get('id')} "
199                   f"({get_alert_severity(alert)})")
200         if len(code_blockers) > 5:
201             print(f" ... and {len(code_blockers) - 5} more")
202     return 3
203
204     # Rule 3: High+ dependency alerts block
205     if dep_blockers:
206         print(f"BLOCKED: Dependabot alerts at/above {threshold}")
207         print()
208         print("Blocking alerts:")
209         for alert in dep_blockers[:5]:
210             pkg = alert.get("dependency", {}).get("package", {})
211             print(f" - #{alert.get('number')}: {pkg.get('name')} "
212                   f"({get_alert_severity(alert)})")
213         if len(dep_blockers) > 5:
214             print(f" ... and {len(dep_blockers) - 5} more")
215     return 4
216

```

```
217     print("PASSED: Security gate checks passed")
218     print()
219     print("Deployment may proceed.")
220     return 0
221
222
223 if __name__ == "__main__":
224     sys.exit(main())
```

Listing 20: Harness Security Gate Script

9.3.1 Harness Integration Instructions

To integrate the security gate with Harness CD:

1. Add a **Shell Script** or **Run step** in the Pre-Deploy Security Gate stage
2. Configure environment variables:
 - **GITHUB_TOKEN** — Token with security alert read permissions
 - **GITHUB_OWNER** — Repository owner
 - **GITHUB_REPO** — Repository name
 - **SECURITY_GATE_MIN_SEVERITY** — Threshold (default: high)
3. Execute: `python security_gate.py`
4. Non-zero exit blocks the deployment stage

10 Architecture Diagram

The following PlantUML diagram illustrates the complete AppSec tooling design with Harness CD integration.

```

@startuml
title AppSec Program Tooling Design (High-Level) - Harness CD Integrated

skinparam shadowing false
skinparam componentStyle rectangle
skinparam wrapWidth 220
skinparam maxMessageSize 220
left to right direction

package "Developer & SCM" as DEV {
    component "Developer Workstations" as Dev
    component "GitHub Repositories" as GitHubRepo
    component "Pull Requests / Reviews" as PR
}

package "CI/CD Pipeline" as CICD {
    component "CI Orchestrator\n(GitHub Actions / Jenkins)" as CI
    component "Build & Test" as BuildTest
    component "Artifact Output\n(Container Image / Binary)" as Artifact
    component "Harness CD\n(Deploy Orchestration)" as HarnessCD
}

package "Deployment Targets" as TARGETS {
    component "Staging / Test Environment" as Staging
    component "Production Environment" as Prod
}

package "Static Code & Dependency Controls" as STATIC {
    component "SAST\nPolaris (Coverity)" as SAST
    component "SCA\nDependabot (GHAS)" as SCA
    component "Secret Scanning\nGHAS" as SecretScan
    component "Code Scanning\nGHAS (CodeQL)" as CodeScan
}

package "Build & Artifact Controls" as ARTIFACTCTL {
    component "Container Scanning\nTrivy" as Trivy
}

package "Runtime / Testing Controls" as RUNTIME {
    component "IAST\nSeeker" as IAST
    component "DAST\nRapid7 InsightAppSec" as DAST
    component "Manual Testing\nPen Test / Bug Bounty" as Manual
}

package "AppSec Triage & Governance" as GOV {
    component "Vulnerability Intake Queue\n(Aggregation + Dedup + SLA)" as Intake
    component "Ticketing / Work Tracking\n(ServiceNow / Jira)" as Ticketing
    component "Risk Acceptance / Exceptions\n(Approvals + Expiry + Evidence)" as Risk
    component "Metrics & Reporting\n(Dashboards / KPIs / Compliance)" as Metrics
}

Dev --> GitHubRepo : Push / Commit
Dev --> PR : Open PR

```

```

PR --> GitHubRepo : Merge
GitHubRepo --> CI : Trigger Pipeline

CI --> BuildTest
BuildTest --> Artifact
Artifact --> HarnessCD : Release input\n(image tag / version)
HarnessCD --> Staging : Deploy
HarnessCD --> Prod : Promote / Release

GitHubRepo --> SAST : PR / Push\n(SAST workflow)
GitHubRepo --> SCA : Dependency Graph\n(manifests)
GitHubRepo --> SecretScan : Repo Events\n(push/PR)
GitHubRepo --> CodeScan : PR / Push\n(CodeQL workflow)

SAST --> Intake : Findings\n(SAST results)
SCA --> Intake : Alerts\n(vulns + upgrades)
SecretScan --> Intake : Findings\n(secrets + validity)
CodeScan --> Intake : Findings\n(SAST-like results)

Artifact --> Trivy : Scan Image / FS
Trivy --> Intake : Findings\n(CVEs, misconfig)

Staging --> IAST : Instrument / Monitor
Staging --> DAST : Scan Target URLs/APIs
Staging --> Manual : Test target

IAST --> Intake : Findings\n(runtime traces)
DAST --> Intake : Findings\n(DAST vulns)
Manual --> Intake : Findings\n(report submissions)

Intake --> Ticketing : Create/Update Issues
Ticketing --> Risk : Route Exception Requests
Intake --> Metrics : Findings + Trends

Intake ..> HarnessCD : Policy decision\n(allow/block/approval)
HarnessCD ..> Intake : Deployment context\n(env, version, exec id)

legend right
<b>Legend</b>
- Rectangles: systems/tools
- Solid arrows: primary integration/data flow
- Dashed arrows: governance/policy loops (gates, context)
- "Intake Queue" is the logical hub for triage,
  ticketing, exceptions, and reporting
endlegend

@enduml

```

Listing 21: AppSec Program Tooling Design (PlantUML)

11 Implementation Backlog

The following prioritized backlog provides the fastest path to value for AppSec automation implementation.

11.1 Priority 1: Foundation

1. **Baseline Enforcement** — Security configuration (or repository toggles) with drift detection
2. **Webhook-Driven Triage** — Secret scanning event processing with automatic ticket creation
3. **GitHub App Authentication** — Production-grade authentication with scoped permissions

11.2 Priority 2: Detection & Normalization

1. **Unified Alert Ingestion** — Code Scanning + Dependabot normalization for metrics and SLAs
2. **SARIF Integration** — External scanner results uploaded to GitHub Code Scanning
3. **SBOM Pipeline** — Automated generation and export for release governance

11.3 Priority 3: Workflow Automation

1. **Noise Controls** — Deterministic dismissal workflows with required justification
2. **SLA Enforcement** — Automated escalation based on severity and age thresholds
3. **Owner Routing** — CODEOWNERS-based assignment and team routing

11.4 Priority 4: Remediation Acceleration

1. **Code Scanning Autofix** — Automated fix generation and PR creation
2. **Dependabot Auto-merge** — Policy-gated automatic merging for low-risk updates
3. **Secret Incident Playbooks** — Automated rotation tracking and evidence collection

11.5 Priority 5: Governance & Evidence

1. **Audit Log Exports** — Scheduled exports for ISO/SOC2 compliance evidence
2. **KPI Dashboards** — Automated metrics generation and trend reporting
3. **Release Gates** — Harness CD integration with GHAS-based policy enforcement

A GitHub REST API Endpoint Reference

Capability	Endpoint	Documentation
<i>Repository Configuration</i>		
Security Settings	PATCH /repos/{owner}/{repo}	REST API: Repositories
Security Configs	GET /orgs/{org}/code-security/configurations	REST API: Configurations

Capability	Endpoint	Documentation
<i>Secret Scanning</i>		
List Alerts	GET /repos/.../secret-scanning/alerts	REST API: Secret Scanning
Update Alert	PATCH /repos/.../secret-scanning/alerts	REST API: Secret Scanning
Push Protection	POST /repos/.../secret-scanning/push-protection	REST API: In-Synapse Scanning
<i>Code Scanning</i>		
List Alerts	GET /repos/.../code-scanning/alerts	REST API: Code Scanning
Update Alert	PATCH /repos/.../code-scanning/alerts	REST API: Code Scanning
Upload SARIF	POST /repos/.../code-scanning/sarif	REST API: Code Scanning
Default Setup	PATCH /repos/.../code-scanning/default-setup	REST API: Code Scanning
Autofix	POST /repos/.../code-scanning/alerts/fix	REST API: Code Scanning
<i>Dependabot</i>		
List Alerts	GET /repos/.../dependabot/alerts	REST API: Dependabot Alerts
Update Alert	PATCH /repos/.../dependabot/alerts	REST API: Dependabot Alerts
List Secrets	GET /repos/.../dependabot/secrets	REST API: Dependabot Secrets
<i>Dependency Graph</i>		
Export SBOM	GET /repos/.../dependency-graph/sbom	REST API: SBOM
<i>Authentication</i>		
Installation Token	POST /app/installations/{id}/access_tokens	GitHub Apps
<i>Audit</i>		
Audit Log	GET /enterprises/{enterprise}/audit	Audit Log API

B Environment Variables Reference

Variable	Description
<i>GitHub Configuration</i>	
GITHUB_API	GitHub API base URL (default: https://api.github.com)
GITHUB_TOKEN	Personal Access Token or Installation Token
GITHUB_APP_ID	GitHub App ID (for App authentication)
GITHUB_APP_PRIVATE_KEY	GitHub App private key (PEM format)
GITHUB_INSTALLATION_ID	GitHub App installation ID
GITHUB_WEBHOOK_SECRET	Webhook signature verification secret
<i>Harness Configuration</i>	

Variable	Description
HARNESS_BASE	Harness API base URL
HARNESS_ACCOUNT	Harness account identifier
HARNESS_API_KEY	Harness API key
HARNESS_ORG	Harness organization identifier
HARNESS_PROJECT	Harness project identifier
HARNESS_PIPELINE_ID	Target pipeline identifier
<i>Security Gate Configuration</i>	
GITHUB_OWNER	Repository owner for gate checks
GITHUB_REPO	Repository name for gate checks
SECURITY_GATE_MIN_SEVERITY	Minimum severity to block (default: high)

C Python Dependencies

```
# Core HTTP client
requests >=2.31.0

# GitHub App JWT authentication
PyJWT >=2.8.0
cryptography >=41.0.0

# YAML processing (Dependabot config)
PyYAML >=6.0.0

# Web framework (webhook receiver)
flask >=3.0.0

# AWS integration (optional, for SQS)
boto3 >=1.34.0

# Data analysis (optional, for reporting)
pandas >=2.1.0
```

Listing 22: requirements.txt