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 19, 2026

Contents

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.

```python
import os
import time
from typing import Any, Dict, Iterator, Optional
import requests

GITHUB_API = os.getenv("GITHUB_API", "https://api.github.com")

class GitHubClient:
    """
    Minimal GitHub REST client.
    - Uses recommended Accept header
    - Sends X-GitHub-Api-Version
    - Basic retry/backoff on 429/5xx
    """
    def __init__(self, token: str, api_base: str = GITHUB_API) -> None:
        self.api_base = api_base.rstrip("/")
        self.session = requests.Session()
        self.session.headers.update({
            "Authorization": f"Bearer {token}",
            "Accept": "application/vnd.github+json",
            "X-GitHub-Api-Version": "2022-11-28",
            "User-Agent": "appsec-automation/1.0",
        })

    def request(
        self,
        method: str,
        path: str,
        *,
        params: Optional[dict] = None,
        json: Optional[dict] = None
    ) -> requests.Response:
        url = f"{self.api_base}{path}"
        for attempt in range(1, 7):
            r = self.session.request(
                method, url, params=params, json=json, timeout=30
            )

            if r.status_code in (429, 500, 502, 503, 504):
                # Respect Retry-After when provided; otherwise exponential
                    backoff
                retry_after = r.headers.get("Retry-After")
                sleep_s = (
                    int(retry_after)
                    if retry_after and retry_after.isdigit()
                    else min(2 ** attempt, 60)
                )
                time.sleep(sleep_s)
```

```python
                    continue

                if r.status_code >= 400:
                    raise RuntimeError(
                        f"GitHub API error {r.status_code}: {r.text}"
                    )

                return r

        raise RuntimeError(f"GitHub API failed after retries: {method} {path}")

    def paginate(
        self,
        path: str,
        *,
        params: Optional[dict] = None
    ) → Iterator[Dict[str, Any]]:
        page = 1
        while True:
            p = dict(params or {})
            p.update({"per_page": 100, "page": page})
            r = self.request("GET", path, params=p)
            data = r.json()
            if not isinstance(data, list):
                raise RuntimeError(
                    f"Expected list response for pagination, got: {type(data)}"
                )
            if not data:
                return
            for item in data:
                yield item
            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.

```python
import time
import jwt  # PyJWT
import requests

def github_app_jwt(app_id: str, private_key_pem: str) → str:
    """Generate a JWT for GitHub App authentication."""
    now = int(time.time())
    payload = {
        "iat": now - 60,              # Issued at (with clock drift buffer)
        "exp": now + (8 * 60),         # Expires in 8 minutes (under GitHub max)
        "iss": app_id,                 # GitHub App ID
    }
    return jwt.encode(payload, private_key_pem, algorithm="RS256")


def github_installation_token(
    api_base: str,
    app_jwt: str,
    installation_id: str
) -> str:
    """Exchange GitHub App JWT for installation access token."""
    url = f"{api_base.rstrip('/')}/app/installations/{installation_id}/
        access_tokens"
    r = requests.post(
        url,
        headers={
            "Authorization": f"Bearer {app_jwt}",
            "Accept": "application/vnd.github+json",
            "X-GitHub-Api-Version": "2022-11-28",
        },
        timeout=30,
    )
    r.raise_for_status()
    return r.json()["token"]


# Usage example
if __name__ == "__main__":
    import os

    app_id = os.environ["GITHUB_APP_ID"]
    private_key = os.environ["GITHUB_APP_PRIVATE_KEY"]
    installation_id = os.environ["GITHUB_INSTALLATION_ID"]
    api_base = os.getenv("GITHUB_API", "https://api.github.com")

    jwt_token = github_app_jwt(app_id, private_key)
    install_token = github_installation_token(api_base, jwt_token,
        installation_id)

    # Use install_token with GitHubClient
    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.

```python
def enable_repo_security_baseline(
    gh: GitHubClient,
    owner: str,
    repo: str
) → dict:
    """
    Enable GHAS security features on a single repository.

    Enables:
    - Advanced Security (required for GHAS features)
    - Secret Scanning
    - Push Protection
    - Non-Provider Pattern detection
    """
    path = f"/repos/{owner}/{repo}"
    payload = {
        "security_and_analysis": {
            "advanced_security": {"status": "enabled"},
            "secret_scanning": {"status": "enabled"},
            "secret_scanning_push_protection": {"status": "enabled"},
            "secret_scanning_non_provider_patterns": {"status": "enabled"},
        }
    }
    return gh.request("PATCH", path, json=payload).json()


def verify_repo_security_posture(
    gh: GitHubClient,
    owner: str,
    repo: str
) → dict:
    """Retrieve current security configuration for a repository."""
    path = f"/repos/{owner}/{repo}"
    response = gh.request("GET", path).json()
    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.

```python
def list_security_configurations(
    gh: GitHubClient,
    org: str
) → list:
    """List available security configurations for an organization."""
    path = f"/orgs/{org}/code-security/configurations"
    return list(gh.paginate(path))


def apply_security_configuration(
    gh: GitHubClient,
    org: str,
    config_id: int,
    repo_ids: list[int]
) → dict:
    """Apply a security configuration to specified repositories."""
    path = f"/orgs/{org}/code-security/configurations/{config_id}/attach"
    payload = {
        "scope": "selected",
        "selected_repository_ids": repo_ids
    }
    return gh.request("POST", path, json=payload).json()


def get_github_recommended_config(
    gh: GitHubClient,
    org: str
) → dict | None:
    """Find the GitHub-recommended security configuration."""
    configs = list_security_configurations(gh, org)
    for config in configs:
        if config.get("name") == "GitHub recommended":
            return config
    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.

```python
def get_code_scanning_default_setup(
    gh: GitHubClient,
    owner: str,
    repo: str
) → dict:
    """Get current code scanning default setup configuration."""
    path = f"/repos/{owner}/{repo}/code-scanning/default-setup"
    return gh.request("GET", path).json()


def enable_code_scanning_default_setup(
    gh: GitHubClient,
    owner: str,
    repo: str,
    *,
    query_suite: str = "default",
    languages: list[str] | None = None
) -> dict:
    """
    Enable code scanning default setup.

    Args:
        gh: GitHub client
        owner: Repository owner
        repo: Repository name
        query_suite: Query suite to use ('default', 'extended')
        languages: Languages to scan (auto-detected if None)
    """
    path = f"/repos/{owner}/{repo}/code-scanning/default-setup"
    payload = {
        "state": "configured",
        "query_suite": query_suite,
    }
    if languages:
        payload["languages"] = languages

    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.

```python
import yaml

def check_dependabot_config(
    gh: GitHubClient,
    owner: str,
    repo: str
) -> dict:
    """
    Check if repository has valid dependabot.yml configuration.

    Returns:
        dict with 'exists', 'valid', 'ecosystems', and 'issues' keys
    """
    path = f"/repos/{owner}/{repo}/contents/.github/dependabot.yml"
    result = {
        "exists": False,
        "valid": False,
        "ecosystems": [],
        "issues": []
    }

    try:
        response = gh.request("GET", path)
        content = response.json()

        if content.get("encoding") == "base64":
            import base64
            yaml_content = base64.b64decode(content["content"]).decode("utf-8")
            result["exists"] = True

            try:
                config = yaml.safe_load(yaml_content)
                if config and "updates" in config:
                    result["valid"] = True
                    result["ecosystems"] = [
                        u.get("package-ecosystem")
                        for u in config.get("updates", [])
                    ]
                else:
                    result["issues"].append("Missing 'updates' key")
            except yaml.YAMLError as e:
                result["issues"].append(f"YAML parse error: {e}")

    except RuntimeError:
        result["issues"].append("File not found")

    return result


def list_dependabot_secrets(
    gh: GitHubClient,
    owner: str,
    repo: str
) -> list:
    """List Dependabot secrets configured for private registry access."""
    path = f"/repos/{owner}/{repo}/dependabot/secrets"
    return list(gh.paginate(path))
```

Listing 6: Dependabot Configuration Validation

4.4 Push Protection Governance

Monitor and govern push protection bypasses for security oversight.

```
from datetime import datetime, timedelta


def list_push_protection_bypasses(
    gh: GitHubClient,
    owner: str,
    repo: str,
    *,
    since: datetime | None = None
) -> list:
    """
    List secret scanning alerts that were bypassed during push.

    Bypassed secrets have 'push_protection_bypassed' = True
    """
    path = f"/repos/{owner}/{repo}/secret-scanning/alerts"
    params = {"state": "open"}

    alerts = list(gh.paginate(path, params=params))

    bypassed = []
    for alert in alerts:
        if alert.get("push_protection_bypassed"):
            bypassed_at = alert.get("push_protection_bypassed_at")
            if bypassed_at and since:
```

```python
                alert_time = datetime.fromisoformat(
                    bypassed_at.replace("Z", "+00:00")
                )
                if alert_time < since:
                    continue
            bypassed.append(alert)

    return bypassed


def generate_bypass_report(
    gh: GitHubClient,
    org: str,
    *,
    days_back: int = 7
) -> list[dict]:
    """Generate a report of push protection bypasses across an org."""
    since = datetime.utcnow() - timedelta(days=days_back)
    report = []

    # Get all org repos
    repos_path = f"/orgs/{org}/repos"
    for repo in gh.paginate(repos_path, params={"type": "all"}):
        repo_name = repo["name"]
        bypasses = list_push_protection_bypasses(
            gh, org, repo_name, since=since
        )

        for bypass in bypasses:
            report.append({
                "repository": f"{org}/{repo_name}",
                "alert_number": bypass.get("number"),
                "secret_type": bypass.get("secret_type"),
                "bypassed_by": bypass.get("push_protection_bypassed_by", {}).
                    get("login"),
                "bypassed_at": bypass.get("push_protection_bypassed_at"),
                "reason": bypass.get("resolution_comment"),
                "url": bypass.get("html_url"),
            })

    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.

```python
import base64
import gzip
import json
from io import BytesIO


def _gzip_base64(data: bytes) -> str:
    """Compress data with gzip and encode as base64."""
    buf = BytesIO()
    with gzip.GzipFile(fileobj=buf, mode="wb") as gz:
        gz.write(data)
    return base64.b64encode(buf.getvalue()).decode("utf-8")


def upload_sarif(
    gh: GitHubClient,
    owner: str,
    repo: str,
    *,
    commit_sha: str,
    ref: str,
    sarif_dict: dict,
    tool_name: str = "external-scanner"
) -> dict:
    """
    Upload SARIF analysis results to GitHub Code Scanning.

    Args:
        gh: GitHub client
        owner: Repository owner
        repo: Repository name
        commit_sha: Full commit SHA the analysis was run against
        ref: Git ref (e.g., "refs/heads/main")
        sarif_dict: SARIF document as Python dict
        tool_name: Name of the scanning tool

    Returns:
        Upload response with 'id' and 'url' for status tracking
    """
    path = f"/repos/{owner}/{repo}/code-scanning/sarifs"
    sarif_bytes = json.dumps(sarif_dict).encode("utf-8")

    payload = {
        "commit_sha": commit_sha,
        "ref": ref,
        "sarif": _gzip_base64(sarif_bytes),
        "tool_name": tool_name,
    }

    return gh.request("POST", path, json=payload).json()


def check_sarif_upload_status(
    gh: GitHubClient,
    owner: str,
    repo: str,
    sarif_id: str
) -> dict:
    """Check the processing status of a SARIF upload."""
    path = f"/repos/{owner}/{repo}/code-scanning/sarifs/{sarif_id}"
    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 GRC/CMDB integration, release attachment, or risk engine feeds.

```
import json
from datetime import datetime


def export_sbom_spdx(
    gh: GitHubClient,
    owner: str,
    repo: str
) -> dict:
    """
    Export SBOM in SPDX format from GitHub Dependency Graph.

    Returns:
        SPDX-formatted SBOM document
    """
    path = f"/repos/{owner}/{repo}/dependency-graph/sbom"
    return gh.request("GET", path).json()


def save_sbom_for_release(
    gh: GitHubClient,
    owner: str,
    repo: str,
    version: str,
    output_dir: str = "."
) -> str:
    """
    Export SBOM and save to file with version metadata.

    Returns:
        Path to saved SBOM file
    """
    sbom = export_sbom_spdx(gh, owner, repo)

    # Add export metadata
    sbom["_metadata"] = {
        "exported_at": datetime.utcnow().isoformat() + "Z",
```

```python
        "repository": f"{owner}/{repo}",
        "version": version,
    }

    filename = f"{output_dir}/{repo}-{version}-sbom.json"
    with open(filename, "w") as f:
        json.dump(sbom, f, indent=2)

    return filename


def analyze_sbom_dependencies(sbom: dict) -> dict:
    """
    Analyze SBOM for dependency statistics.

    Returns:
        Summary of dependency counts by ecosystem
    """
    packages = sbom.get("sbom", {}).get("packages", [])

    ecosystems = {}
    for pkg in packages:
        # SPDX uses externalRefs for package manager info
        for ref in pkg.get("externalRefs", []):
            if ref.get("referenceCategory") == "PACKAGE-MANAGER":
                ecosystem = ref.get("referenceType", "unknown")
                ecosystems[ecosystem] = ecosystems.get(ecosystem, 0) + 1
                break
        else:
            ecosystems["unknown"] = ecosystems.get("unknown", 0) + 1

    return {
        "total_packages": len(packages),
        "by_ecosystem": ecosystems,
    }
```

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.

```python
from datetime import datetime, timedelta


def get_secret_scanning_status(
    gh: GitHubClient,
    owner: str,
    repo: str
) -> dict:
    """
    Get secret scanning enablement and recent activity status.

    Returns:
        dict with 'enabled', 'last_scan', 'alert_count' keys
    """
    # Check if enabled
    repo_path = f"/repos/{owner}/{repo}"
    repo_info = gh.request("GET", repo_path).json()

    security_analysis = repo_info.get("security_and_analysis", {})
    secret_scanning = security_analysis.get("secret_scanning", {})

    status = {
        "enabled": secret_scanning.get("status") == "enabled",
        "push_protection_enabled": security_analysis.get(
            "secret_scanning_push_protection", {}
        ).get("status") == "enabled",
        "alert_count": 0,
        "open_alerts": 0,
    }

    if status["enabled"]:
        # Count alerts
        alerts_path = f"/repos/{owner}/{repo}/secret-scanning/alerts"
        all_alerts = list(gh.paginate(alerts_path))
        status["alert_count"] = len(all_alerts)
        status["open_alerts"] = len(
            [a for a in all_alerts if a.get("state") == "open"]
        )

    return status


def org_secret_scanning_coverage(
    gh: GitHubClient,
    org: str
) → dict:
    """
    Generate secret scanning coverage report for an organization.

    Returns:
        dict with 'total_repos', 'enabled_count', 'coverage_pct'
    """
    repos_path = f"/orgs/{org}/repos"

    total = 0
    enabled = 0
    push_protection_enabled = 0

    for repo in gh.paginate(repos_path, params={"type": "all"}):
        total += 1
        security = repo.get("security_and_analysis", {})

        if security.get("secret_scanning", {}).get("status") == "enabled":
            enabled += 1
        if security.get("secret_scanning_push_protection", {}).get("status") ==
             "enabled":
            push_protection_enabled += 1

    return {
        "total_repos": total,
        "secret_scanning_enabled": enabled,
        "push_protection_enabled": push_protection_enabled,
        "coverage_pct": round(enabled / total * 100, 2) if total > 0 else 0,
        "push_protection_pct": round(push_protection_enabled / total * 100, 2)
           if total > 0 else 0,
    }
```

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

```python
def list_open_secret_alerts(
    gh: GitHubClient,
    owner: str,
    repo: str
) -> list:
    """List all open secret scanning alerts for a repository."""
    path = f"/repos/{owner}/{repo}/secret-scanning/alerts"
    return list(gh.paginate(path, params={"state": "open"}))


def update_secret_alert(
    gh: GitHubClient,
    owner: str,
    repo: str,
    alert_number: int,
    *,
    state: str,
    resolution: str | None = None,
    resolution_comment: str | None = None
) -> dict:
    """
    Update a secret scanning alert.

    Args:
        state: "open" or "resolved"
        resolution: Required if resolving: "false_positive", "wont_fix",
                    "revoked", "pattern_edited", "pattern_deleted", "
                        used_in_tests"
        resolution_comment: Optional comment explaining resolution
    """
    path = f"/repos/{owner}/{repo}/secret-scanning/alerts/{alert_number}"
    payload = {"state": state}

    if state == "resolved" and resolution:
        payload["resolution"] = resolution
    if resolution_comment:
        payload["resolution_comment"] = resolution_comment

    return gh.request("PATCH", path, json=payload).json()


def create_issue_from_secret_alert(
    gh: GitHubClient,
    owner: str,
    repo: str,
    alert: dict
) -> dict:
    """Create a GitHub Issue from a secret scanning alert."""
    alert_number = alert.get("number")
    secret_type = alert.get("secret_type", "secret")
    url = alert.get("html_url", "")

    title = f"[Secret Scanning] {secret_type} exposed (alert #{alert_number})"
    body = f"""## Secret Scanning Alert

GitHub Secret Scanning detected a potential secret.

**Alert Details:**
- **Type:** {secret_type}
- **Alert URL:** {url}
- **State:** {alert.get('state', 'unknown')}

## Triage Checklist

- [ ] Confirm validity of the detected secret
- [ ] Revoke/rotate the secret if valid
- [ ] Remove from git history if required
- [ ] Update secret storage location
- [ ] Document remediation in this issue

## References

- [GitHub Secret Scanning Docs](https://docs.github.com/en/code-security/secret
   -scanning)
"""

    path = f"/repos/{owner}/{repo}/issues"
    payload = {
        "title": title,
        "body": body,
        "labels": ["security", "secret-scanning", "triage-needed"]
    }

    return gh.request("POST", path, json=payload).json()


def secret_scanning_to_issues(
    gh: GitHubClient,
    owner: str,
    repo: str,
    max_items: int = 20
) -> list:
    """
    Create issues for open secret scanning alerts.

    Returns:
        List of created issues
    """
    alerts = list_open_secret_alerts(gh, owner, repo)[:max_items]
    created_issues = []

    for alert in alerts:
        issue = create_issue_from_secret_alert(gh, owner, repo, alert)
        created_issues.append(issue)

    return created_issues
```

Listing 11: Secret Scanning Alert Operations

6.2 Code Scanning Alert Management

```python
def list_code_scanning_alerts(
    gh: GitHubClient,
    owner: str,
    repo: str,
    state: str = "open",
    *,
    severity: str | None = None,
    tool_name: str | None = None
) -> list:
    """
    List code scanning alerts with optional filtering.

    Args:
        state: "open", "closed", "dismissed", or "fixed"
        severity: Filter by severity (critical, high, medium, low, warning,
            note)
        tool_name: Filter by scanning tool name
    """
    path = f"/repos/{owner}/{repo}/code-scanning/alerts"
    params = {"state": state}

    if severity:
        params["severity"] = severity
    if tool_name:
        params["tool_name"] = tool_name

    return list(gh.paginate(path, params=params))


def dismiss_code_scanning_alert(
    gh: GitHubClient,
    owner: str,
    repo: str,
    alert_number: int,
    reason: str,
    comment: str
) -> dict:
    """
    Dismiss a code scanning alert.

    Args:
        reason: "false_positive", "won't_fix", or "used_in_tests"
        comment: Explanation for dismissal (required for audit)
    """
    path = f"/repos/{owner}/{repo}/code-scanning/alerts/{alert_number}"
    payload = {
        "state": "dismissed",
        "dismissed_reason": reason,
        "dismissed_comment": comment,
    }
    return gh.request("PATCH", path, json=payload).json()


def get_code_scanning_alert_instances(
    gh: GitHubClient,
    owner: str,
    repo: str,
    alert_number: int
) -> list:
    """Get all instances of a code scanning alert across branches."""
    path = f"/repos/{owner}/{repo}/code-scanning/alerts/{alert_number}/
        instances"
    return list(gh.paginate(path))


def trigger_code_scanning_autofix(
    gh: GitHubClient,
    owner: str,
    repo: str,
    alert_number: int
) -> dict:
    """
    Trigger autofix generation for an eligible code scanning alert.

    Note: Not all alerts are eligible for autofix.
    """
    path = f"/repos/{owner}/{repo}/code-scanning/alerts/{alert_number}/autofix"
    return gh.request("POST", path).json()


def commit_code_scanning_autofix(
    gh: GitHubClient,
    owner: str,
    repo: str,
    alert_number: int,
    *,
    message: str | None = None
) -> dict:
    """
    Commit an autofix for a code scanning alert.

    Args:
        message: Custom commit message (optional)
    """
    path = f"/repos/{owner}/{repo}/code-scanning/alerts/{alert_number}/autofix/
        commits"
    payload = {}
    if message:
        payload["message"] = message

    return gh.request("POST", path, json=payload).json()
```

Listing 12: Code Scanning Alert Operations

6.3 Dependabot Alert Management

```python
def list_dependabot_alerts(
    gh: GitHubClient,
    owner: str,
    repo: str,
    state: str = "open",
    *,
    severity: str | None = None,
    ecosystem: str | None = None
) -> list:
    """
    List Dependabot alerts with optional filtering.

    Args:
        state: "open", "dismissed", or "fixed"
        severity: Filter by severity (critical, high, medium, low)
        ecosystem: Filter by package ecosystem (npm, pip, maven, etc.)
    """
    path = f"/repos/{owner}/{repo}/dependabot/alerts"
    params = {"state": state}

    if severity:
        params["severity"] = severity
    if ecosystem:
        params["ecosystem"] = ecosystem

    return list(gh.paginate(path, params=params))


def dismiss_dependabot_alert(
    gh: GitHubClient,
    owner: str,
    repo: str,
    alert_number: int,
    reason: str,
    comment: str
) -> dict:
    """
    Dismiss a Dependabot alert.

    Args:
        reason: "fix_started", "inaccurate", "no_bandwidth",
                "not_used", "tolerable_risk"
        comment: Explanation for dismissal
    """
    path = f"/repos/{owner}/{repo}/dependabot/alerts/{alert_number}"
    payload = {
        "state": "dismissed",
        "dismissed_reason": reason,
        "dismissed_comment": comment,
    }
    return gh.request("PATCH", path, json=payload).json()


def get_dependabot_alert_details(
    gh: GitHubClient,
    owner: str,
    repo: str,
    alert_number: int
) -> dict:
    """Get detailed information about a Dependabot alert."""
    path = f"/repos/{owner}/{repo}/dependabot/alerts/{alert_number}"
    return gh.request("GET", path).json()


def prioritize_dependabot_alerts(
    gh: GitHubClient,
    owner: str,
    repo: str
```

```python
) → dict:
    """
    Analyze and prioritize Dependabot alerts.

    Returns:
        Categorized alerts by priority
    """
    alerts = list_dependabot_alerts(gh, owner, repo)

    prioritized = {
        "critical": [],
        "high": [],
        "medium": [],
        "low": [],
    }

    for alert in alerts:
        severity = alert.get("security_advisory", {}).get("severity", "low")
        severity = severity.lower()

        if severity in prioritized:
            prioritized[severity].append({
                "number": alert.get("number"),
                "package": alert.get("dependency", {}).get("package", {}).get("
                    name"),
                "ecosystem": alert.get("dependency", {}).get("package", {}).get
                    ("ecosystem"),
                "vulnerable_version": alert.get("security_vulnerability", {}).
                    get("vulnerable_version_range"),
                "fixed_in": alert.get("security_vulnerability", {}).get("
                    first_patched_version", {}).get("identifier"),
                "cvss_score": alert.get("security_advisory", {}).get("cvss",
                    {}).get("score"),
                "url": alert.get("html_url"),
            })

    return prioritized
```

Listing 13: Dependabot Alert Operations

6.4 Unified Alert Ingestion

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

```python
from dataclasses import dataclass, asdict
from datetime import datetime
from enum import Enum
from typing import Optional


class AlertSource(Enum):
    SECRET_SCANNING = "secret_scanning"
    CODE_SCANNING = "code_scanning"
    DEPENDABOT = "dependabot"


class AlertSeverity(Enum):
    CRITICAL = "critical"
    HIGH = "high"
    MEDIUM = "medium"
    LOW = "low"
    NOTE = "note"


@dataclass
class UnifiedAlert:
    """Normalized alert schema for cross-tool processing."""
    id: str                             # Unique identifier
    source: AlertSource                 # Origin tool
    repository: str                     # owner/repo
    severity: AlertSeverity             # Normalized severity
    title: str                          # Human-readable title
    description: str                    # Alert details
    state: str                          # open/resolved/dismissed
    created_at: datetime
    url: str                            # Link to alert

    # Optional fields
    cwe_id: Optional[str] = None
    cvss_score: Optional[float] = None
    rule_id: Optional[str] = None
    file_path: Optional[str] = None
    line_number: Optional[int] = None

    # SLA tracking
    sla_deadline: Optional[datetime] = None
    assigned_to: Optional[str] = None
    ticket_id: Optional[str] = None


def normalize_secret_alert(
    alert: dict,
    owner: str,
    repo: str
) -> UnifiedAlert:
    """Convert secret scanning alert to unified schema."""
    return UnifiedAlert(
        id=f"secret-{owner}-{repo}-{alert['number']}",
        source=AlertSource.SECRET_SCANNING,
        repository=f"{owner}/{repo}",
        severity=AlertSeverity.CRITICAL,  # Secrets always critical
        title=f"Exposed {alert.get('secret_type', 'secret')}",
        description=f"Secret type: {alert.get('secret_type_display_name', '
            Unknown')}",
        state=alert.get("state", "open"),
        created_at=datetime.fromisoformat(
            alert["created_at"].replace("Z", "+00:00")
        ),
        url=alert.get("html_url", ""),
        file_path=alert.get("secret", {}).get("path"),
    )


def normalize_code_alert(
    alert: dict,
    owner: str,
    repo: str
```

```python
) → UnifiedAlert:
    """Convert code scanning alert to unified schema."""
    severity_map = {
        "critical": AlertSeverity.CRITICAL,
        "high": AlertSeverity.HIGH,
        "medium": AlertSeverity.MEDIUM,
        "low": AlertSeverity.LOW,
        "warning": AlertSeverity.MEDIUM,
        "note": AlertSeverity.NOTE,
    }

    rule = alert.get("rule", {})
    location = alert.get("most_recent_instance", {}).get("location", {})

    return UnifiedAlert(
        id=f"code-{owner}-{repo}-{alert['number']}",
        source=AlertSource.CODE_SCANNING,
        repository=f"{owner}/{repo}",
        severity=severity_map.get(
            alert.get("rule", {}).get("severity", "medium").lower(),
            AlertSeverity.MEDIUM
        ),
        title=rule.get("description", "Code scanning alert"),
        description=rule.get("full_description", ""),
        state=alert.get("state", "open"),
        created_at=datetime.fromisoformat(
            alert["created_at"].replace("Z", "+00:00")
        ),
        url=alert.get("html_url", ""),
        cwe_id=next((t["name"] for t in rule.get("tags", [])
                     if t.startswith("CWE-")), None),
        rule_id=rule.get("id"),
        file_path=location.get("path"),
        line_number=location.get("start_line"),
    )


def normalize_dependabot_alert(
    alert: dict,
    owner: str,
    repo: str
) → UnifiedAlert:
    """Convert Dependabot alert to unified schema."""
    severity_map = {
        "critical": AlertSeverity.CRITICAL,
        "high": AlertSeverity.HIGH,
        "medium": AlertSeverity.MEDIUM,
        "low": AlertSeverity.LOW,
    }

    advisory = alert.get("security_advisory", {})
    dependency = alert.get("dependency", {})
    package = dependency.get("package", {})

    return UnifiedAlert(
        id=f"dep-{owner}-{repo}-{alert['number']}",
        source=AlertSource.DEPENDABOT,
        repository=f"{owner}/{repo}",
        severity=severity_map.get(
                advisory.get("severity", "medium").lower(),
                AlertSeverity.MEDIUM
            ),
            title=f"Vulnerable dependency: {package.get('name', 'unknown')}",
            description=advisory.get("summary", ""),
            state=alert.get("state", "open"),
            created_at=datetime.fromisoformat(
                alert["created_at"].replace("Z", "+00:00")
            ),
            url=alert.get("html_url", ""),
            cvss_score=advisory.get("cvss", {}).get("score"),
            cwe_id=next((cwe.get("cwe_id") for cwe in advisory.get("cwes", [])),
                None),
            file_path=dependency.get("manifest_path"),
        )


def ingest_all_alerts(
        gh: GitHubClient,
        owner: str,
        repo: str
) -> list[UnifiedAlert]:
    """
    Ingest all security alerts from a repository into unified format.

    Returns:
        List of UnifiedAlert objects from all sources
    """
    unified = []

    # Secret scanning
    secrets = list_open_secret_alerts(gh, owner, repo)
    unified.extend(normalize_secret_alert(a, owner, repo) for a in secrets)

    # Code scanning
    code_alerts = list_code_scanning_alerts(gh, owner, repo)
    unified.extend(normalize_code_alert(a, owner, repo) for a in code_alerts)

    # Dependabot
    dep_alerts = list_dependabot_alerts(gh, owner, repo)
    unified.extend(normalize_dependabot_alert(a, owner, repo) for a in
        dep_alerts)

    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

```python
import hmac
import hashlib
import os
import json
from flask import Flask, request, abort, jsonify
from typing import Callable

app = Flask(__name__)
WEBHOOK_SECRET = os.environ["GITHUB_WEBHOOK_SECRET"].encode("utf-8")

# Event handlers registry
_handlers: dict[str, list[Callable]] = {}


def register_handler(event_type: str):
    """Decorator to register event handlers."""
    def decorator(func: Callable):
        if event_type not in _handlers:
            _handlers[event_type] = []
        _handlers[event_type].append(func)
        return func
    return decorator


def verify_signature(req) -> None:
    """Verify GitHub webhook signature using HMAC-SHA256."""
    sig = req.headers.get("X-Hub-Signature-256", "")
    if not sig.startswith("sha256="):
        abort(401, "Missing signature")

    expected = hmac.new(
        WEBHOOK_SECRET,
        req.data,
        hashlib.sha256
    ).hexdigest()

    if not hmac.compare_digest(sig, f"sha256={expected}"):
        abort(401, "Invalid signature")


@app.post("/webhook")
def webhook():
    """Main webhook endpoint."""
    verify_signature(request)

    event = request.headers.get("X-GitHub-Event", "")
    delivery_id = request.headers.get("X-GitHub-Delivery", "")
    payload = request.get_json(silent=True) or {}

    app.logger.info(f"Received {event} event (delivery: {delivery_id})")

    # Dispatch to registered handlers
    handlers = _handlers.get(event, [])
    results = []

    for handler in handlers:
        try:
            result = handler(payload)
            results.append({"handler": handler.__name__, "result": result})
        except Exception as e:
            app.logger.error(f"Handler {handler.__name__} failed: {e}")
            results.append({"handler": handler.__name__, "error": str(e)})

    return jsonify({
        "ok": True,
        "event": event,
        "delivery_id": delivery_id,
        "handlers_executed": len(handlers),
        "results": results,
    })


# Example event handlers

@register_handler("secret_scanning_alert")
def handle_secret_alert(payload: dict) → dict:
    """Process secret scanning alert events."""
    action = payload.get("action")
    alert = payload.get("alert", {})
    repo = payload.get("repository", {})

    if action == "created":
        # New secret detected - trigger incident workflow
        return {
            "action": "incident_created",
            "alert_number": alert.get("number"),
            "secret_type": alert.get("secret_type"),
            "repository": repo.get("full_name"),
        }

    elif action == "resolved":
        # Secret resolved - update ticket
        return {
            "action": "incident_resolved",
            "alert_number": alert.get("number"),
            "resolution": alert.get("resolution"),
        }

    return {"action": action, "handled": True}


@register_handler("code_scanning_alert")
def handle_code_alert(payload: dict) → dict:
    """Process code scanning alert events."""
    action = payload.get("action")
    alert = payload.get("alert", {})

    return {
        "action": action,
        "alert_number": alert.get("number"),
```

```python
        "rule_id": alert.get("rule", {}).get("id"),
        "severity": alert.get("rule", {}).get("severity"),
    }


@register_handler("dependabot_alert")
def handle_dependabot_alert(payload: dict) -> dict:
    """Process Dependabot alert events."""
    action = payload.get("action")
    alert = payload.get("alert", {})

    return {
        "action": action,
        "alert_number": alert.get("number"),
        "package": alert.get("dependency", {}).get("package", {}).get("name"),
        "severity": alert.get("security_advisory", {}).get("severity"),
    }


if __name__ == "__main__":
    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.

```python
import json
import boto3
from dataclasses import dataclass, asdict
from datetime import datetime


@dataclass
class SecurityEvent:
    """Standardized security event for queue dispatch."""
    event_id: str
    event_type: str
    action: str
    repository: str
    alert_number: int | None
    severity: str | None
    timestamp: str
    payload: dict


class EventDispatcher:
    """Dispatch security events to SQS queue for async processing."""

    def __init__(self, queue_url: str, region: str = "us-east-1"):
        self.sqs = boto3.client("sqs", region_name=region)
        self.queue_url = queue_url

    def dispatch(self, event: SecurityEvent) -> str:
        """Send event to queue. Returns message ID."""
        response = self.sqs.send_message(
            QueueUrl=self.queue_url,
            MessageBody=json.dumps(asdict(event)),
            MessageAttributes={
                "EventType": {
                    "StringValue": event.event_type,
                    "DataType": "String"
                },
                "Severity": {
                    "StringValue": event.severity or "unknown",
                    "DataType": "String"
                },
            }
        )
        return response["MessageId"]


def webhook_to_event(
    event_type: str,
    payload: dict,
    delivery_id: str
) -> SecurityEvent:
    """Convert webhook payload to SecurityEvent."""
    repo = payload.get("repository", {})
    alert = payload.get("alert", {})

    # Determine severity based on event type
    severity = None
    if event_type == "secret_scanning_alert":
        severity = "critical"
    elif event_type == "code_scanning_alert":
        severity = alert.get("rule", {}).get("severity")
    elif event_type == "dependabot_alert":
        severity = alert.get("security_advisory", {}).get("severity")

    return SecurityEvent(
        event_id=delivery_id,
        event_type=event_type,
        action=payload.get("action", "unknown"),
        repository=repo.get("full_name", ""),
        alert_number=alert.get("number"),
        severity=severity,
        timestamp=datetime.utcnow().isoformat() + "Z",
        payload=payload,
    )
```

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

```python
from collections import defaultdict
from datetime import datetime, timedelta
from dataclasses import dataclass


@dataclass
class SecurityKPIs:
    """Key Performance Indicators for security program."""
    total_open_critical: int
    total_open_high: int
    mttr_secrets_days: float
    mttr_code_days: float
    mttr_dependencies_days: float
    top_cwes: list[tuple[str, int]]
    reopen_rate: float
    coverage_pct: float


def calculate_mttr(
    alerts: list[dict],
    created_field: str = "created_at",
    resolved_field: str = "fixed_at"
) -> float:
    """
    Calculate Mean Time To Remediation in days.

    Only considers resolved alerts with both timestamps.
    """
    durations = []

    for alert in alerts:
        if alert.get("state") not in ("fixed", "resolved"):
            continue

        created = alert.get(created_field)
        resolved = alert.get(resolved_field) or alert.get("dismissed_at")

        if created and resolved:
            created_dt = datetime.fromisoformat(created.replace("Z", "+00:00"))
            resolved_dt = datetime.fromisoformat(resolved.replace("Z", "+00:00"
                ))
            duration = (resolved_dt - created_dt).total_seconds() / 86400
            durations.append(duration)

    return sum(durations) / len(durations) if durations else 0.0


def count_by_severity(alerts: list[dict]) -> dict[str, int]:
    """Count open alerts by severity."""
    counts = defaultdict(int)

    for alert in alerts:
        if alert.get("state") != "open":
            continue

        # Handle different alert types
        severity = (
            alert.get("severity") or
            alert.get("rule", {}).get("severity") or
            alert.get("security_advisory", {}).get("severity") or
            "unknown"
        )
        counts[severity.lower()] += 1

    return dict(counts)


def top_cwes(alerts: list[dict], top_n: int = 10) → list[tuple[str, int]]:
    """Extract top CWEs from code scanning alerts."""
    cwe_counts = defaultdict(int)

    for alert in alerts:
        rule = alert.get("rule", {})
        tags = rule.get("tags", [])

        for tag in tags:
            if isinstance(tag, str) and tag.startswith("CWE-"):
                cwe_counts[tag] += 1

    return sorted(cwe_counts.items(), key=lambda x: x[1], reverse=True)[:top_n]


def generate_org_kpis(
    gh: GitHubClient,
    org: str,
    *,
    days_back: int = 30
) → dict:
    """
    Generate organization-wide security KPIs.

    Returns:
        Comprehensive KPI report
    """
    since = datetime.utcnow() - timedelta(days=days_back)

    all_secrets = []
    all_code = []
    all_deps = []

    repos_path = f"/orgs/{org}/repos"

    for repo in gh.paginate(repos_path, params={"type": "all"}):
        repo_name = repo["name"]

        try:
            secrets = list(gh.paginate(
                f"/repos/{org}/{repo_name}/secret-scanning/alerts"
            ))
            all_secrets.extend(secrets)
        except RuntimeError:
                pass

        try:
            code = list(gh.paginate(
                f"/repos/{org}/{repo_name}/code-scanning/alerts"
            ))
            all_code.extend(code)
        except RuntimeError:
            pass

        try:
            deps = list(gh.paginate(
                f"/repos/{org}/{repo_name}/dependabot/alerts"
            ))
            all_deps.extend(deps)
        except RuntimeError:
            pass

    # Calculate metrics
    secret_severity = count_by_severity(all_secrets)
    code_severity = count_by_severity(all_code)
    dep_severity = count_by_severity(all_deps)

    return {
        "period_days": days_back,
        "generated_at": datetime.utcnow().isoformat() + "Z",
        "summary": {
            "total_open_critical": (
                secret_severity.get("critical", 0) +
                code_severity.get("critical", 0) +
                dep_severity.get("critical", 0)
            ),
            "total_open_high": (
                secret_severity.get("high", 0) +
                code_severity.get("high", 0) +
                dep_severity.get("high", 0)
            ),
        },
        "mttr": {
            "secrets_days": calculate_mttr(all_secrets),
            "code_scanning_days": calculate_mttr(all_code),
            "dependencies_days": calculate_mttr(all_deps),
        },
        "by_source": {
            "secret_scanning": {
                "total": len(all_secrets),
                "by_severity": secret_severity,
            },
            "code_scanning": {
                "total": len(all_code),
                "by_severity": code_severity,
                "top_cwes": top_cwes(all_code),
            },
            "dependabot": {
                "total": len(all_deps),
                "by_severity": dep_severity,
            },
        },
    }
```

Listing 17: Security KPI Calculation

8.2 Audit Log Integration

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

```python
from datetime import datetime, timedelta


def export_audit_log(
    gh: GitHubClient,
    enterprise: str,
    *,
    days_back: int = 30,
    include_phrases: list[str] | None = None
) -> list[dict]:
    """
    Export enterprise audit log entries.

    Args:
        enterprise: Enterprise slug
        days_back: Number of days to export
        include_phrases: Filter to entries containing these phrases

    Returns:
        List of audit log entries
    """
    # Calculate date range
    end_date = datetime.utcnow()
    start_date = end_date - timedelta(days=days_back)

    path = f"/enterprises/{enterprise}/audit-log"
    params = {
        "phrase": " ".join(include_phrases) if include_phrases else "",
        "include": "all",
        "order": "desc",
    }

    entries = list(gh.paginate(path, params=params))

    # Filter by date
    filtered = []
    for entry in entries:
        created = entry.get("created_at") or entry.get("@timestamp")
        if created:
            entry_dt = datetime.fromisoformat(created.replace("Z", "+00:00"))
            if entry_dt >= start_date.replace(tzinfo=entry_dt.tzinfo):
                filtered.append(entry)

    return filtered


def extract_security_events(entries: list[dict]) -> list[dict]:
    """
    Filter audit log for security-relevant events.

    Includes:
    - Secret scanning events
    - Code scanning configuration changes
    - Security feature enablement/disablement
    - Push protection bypasses
    """
    security_actions = {
        "secret_scanning.disable",
        "secret_scanning.enable",
        "secret_scanning_alert.create",
        "secret_scanning_alert.resolve",
        "secret_scanning_push_protection.disable",
        "secret_scanning_push_protection.enable",
        "secret_scanning_push_protection.bypass",
        "code_scanning.disable",
        "code_scanning.enable",
        "dependabot_alerts.disable",
        "dependabot_alerts.enable",
        "repository_vulnerability_alert.create",
        "repository_vulnerability_alert.dismiss",
    }

    return [
        entry for entry in entries
        if entry.get("action") in security_actions
    ]


def generate_compliance_report(
    gh: GitHubClient,
    enterprise: str,
    *,
    days_back: int = 30
) → dict:
    """
    Generate compliance report for audit purposes.

    Includes:
    - Security feature changes
    - Alert activity summary
    - Bypass events
    """
    entries = export_audit_log(gh, enterprise, days_back=days_back)
    security_entries = extract_security_events(entries)

    # Categorize events
    report = {
        "period_days": days_back,
        "generated_at": datetime.utcnow().isoformat() + "Z",
        "total_security_events": len(security_entries),
        "feature_changes": [],
        "bypasses": [],
        "alerts_created": 0,
        "alerts_resolved": 0,
    }

    for entry in security_entries:
        action = entry.get("action", "")

        if "bypass" in action:
            report["bypasses"].append({
                "action": action,
                "actor": entry.get("actor"),
                "repository": entry.get("repo"),
                "timestamp": entry.get("created_at"),
            })
        elif "disable" in action or "enable" in action:
            report["feature_changes"].append({
                "action": action,
                "actor": entry.get("actor"),
                "repository": entry.get("repo"),
                "timestamp": entry.get("created_at"),
            })
        elif "create" in action:
            report["alerts_created"] += 1
        elif "resolve" in action or "dismiss" in action:
            report["alerts_resolved"] += 1

    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}/exe` endpoint.

```python
import os
import requests


def trigger_harness_cd(
    *,
    harness_base: str,
    account_id: str,
    api_key: str,
    org: str,
    project: str,
    pipeline_id: str,
    inputs_yaml: str,
) -> dict:
    """
    Trigger a Harness CD pipeline execution.

    Args:
        harness_base: Harness API base URL
                      (e.g., "https://app.harness.io/gateway/pipeline/api")
        account_id: Harness account identifier
        api_key: Harness API key
        org: Harness organization identifier
        project: Harness project identifier
        pipeline_id: Pipeline to execute
        inputs_yaml: YAML string with pipeline inputs

    Returns:
        Execution response with run details
    """
    url = (
        f"{harness_base.rstrip('/')}/v1/orgs/{org}/projects/{project}"
        f"/pipelines/{pipeline_id}/execute"
    )

    response = requests.post(
        url,
        params={"module": "CD"},
        headers={
            "Harness-Account": account_id,
            "x-api-key": api_key,
            "Content-Type": "application/json",
            "Accept": "application/json",
        },
        json={"inputs_yaml": inputs_yaml},
        timeout=30,
    )
    response.raise_for_status()
    return response.json()


def check_pipeline_status(
    *,
    harness_base: str,
    account_id: str,
    api_key: str,
    org: str,
    project: str,
    pipeline_id: str,
    execution_id: str,
) -> dict:
    """Check status of a pipeline execution."""
    url = (
        f"{harness_base.rstrip('/')}/v1/orgs/{org}/projects/{project}"
        f"/pipelines/{pipeline_id}/executions/{execution_id}"
    )

    response = requests.get(
        url,
        headers={
            "Harness-Account": account_id,
            "x-api-key": api_key,
            "Accept": "application/json",
        },
        timeout=30,
    )
    response.raise_for_status()
    return response.json()


```

```python
if __name__ == "__main__":
    # Example usage
    resp = trigger_harness_cd(
        harness_base=os.environ["HARNESS_BASE"],
        account_id=os.environ["HARNESS_ACCOUNT"],
        api_key=os.environ["HARNESS_API_KEY"],
        org=os.environ["HARNESS_ORG"],
        project=os.environ["HARNESS_PROJECT"],
        pipeline_id=os.environ["HARNESS_PIPELINE_ID"],
        inputs_yaml=os.environ["HARNESS_INPUTS_YAML"],
    )
    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.

```python
#!/usr/bin/env python3
"""
Security Gate for Harness CD Pipeline

Checks GHAS alerts and blocks deployment if policy thresholds are exceeded.

Environment Variables:
    GITHUB_TOKEN: GitHub token with security alert read access
    GITHUB_OWNER: Repository owner
    GITHUB_REPO: Repository name
    GITHUB_API: GitHub API base URL (optional, defaults to github.com)
    SECURITY_GATE_MIN_SEVERITY: Minimum severity to block (default: high)

Exit Codes:
    0: Security gate passed
    1: Configuration error
    2: Blocked - open secret scanning alerts
    3: Blocked - code scanning alerts at/above threshold
    4: Blocked - dependabot alerts at/above threshold
"""

import os
import sys
import requests
from typing import Any, Dict, List

GITHUB_API = os.getenv("GITHUB_API", "https://api.github.com")

SEV_ORDER = {"low": 1, "medium": 2, "high": 3, "critical": 4}


def gh_get(
    token: str,
    path: str,
    params: Dict[str, Any] | None = None
) -> Any:
    """Make authenticated GET request to GitHub API."""
    url = f"{GITHUB_API.rstrip('/')}{path}"
    response = requests.get(
        url,
        headers={
            "Authorization": f"Bearer {token}",
            "Accept": "application/vnd.github+json",
            "X-GitHub-Api-Version": "2022-11-28",
        },
        params=params,
        timeout=30,
    )
    response.raise_for_status()
    return response.json()


def list_all_pages(
    token: str,
    path: str,
    params: Dict[str, Any] | None = None
) → List[Dict[str, Any]]:
    """Paginate through all results."""
    out: List[Dict[str, Any]] = []
    page = 1

    while True:
        p = dict(params or {})
        p.update({"per_page": 100, "page": page})
        data = gh_get(token, path, p)

        if not isinstance(data, list) or not data:
            break

        out.extend(data)
        page += 1

    return out


def sev_at_least(item_sev: str | None, threshold: str) → bool:
    """Check if severity meets or exceeds threshold."""
    if not item_sev:
        return False
    return SEV_ORDER.get(item_sev.lower(), 0) >= SEV_ORDER[threshold.lower()]


def get_alert_severity(alert: dict) → str | None:
    """Extract severity from different alert types."""
    # Try direct severity field
    if "severity" in alert:
        return alert["severity"]

    # Code scanning: rule.severity
    if "rule" in alert:
        return alert.get("rule", {}).get("severity")

    # Dependabot: security_advisory.severity
    if "security_advisory" in alert:
        return alert.get("security_advisory", {}).get("severity")

    return None

```

```python

def main() -> int:
    """Run security gate checks."""
    # Validate configuration
    token = os.environ.get("GITHUB_TOKEN")
    owner = os.environ.get("GITHUB_OWNER")
    repo = os.environ.get("GITHUB_REPO")
    threshold = os.getenv("SECURITY_GATE_MIN_SEVERITY", "high").lower()

    if not all([token, owner, repo]):
        print("ERROR: Missing required environment variables")
        print("Required: GITHUB_TOKEN, GITHUB_OWNER, GITHUB_REPO")
        return 1

    print("=" * 60)
    print("SECURITY GATE CHECK")
    print("=" * 60)
    print(f"Repository: {owner}/{repo}")
    print(f"Severity Threshold: {threshold}")
    print()

    # Fetch alerts
    print("Fetching security alerts...")

    try:
        secrets = list_all_pages(
            token,
            f"/repos/{owner}/{repo}/secret-scanning/alerts",
            {"state": "open"}
        )
    except Exception as e:
        print(f"  Secret Scanning: Error - {e}")
        secrets = []

    try:
        code_alerts = list_all_pages(
            token,
            f"/repos/{owner}/{repo}/code-scanning/alerts",
            {"state": "open"}
        )
    except Exception as e:
        print(f"  Code Scanning: Error - {e}")
        code_alerts = []

    try:
        dep_alerts = list_all_pages(
            token,
            f"/repos/{owner}/{repo}/dependabot/alerts",
            {"state": "open"}
        )
    except Exception as e:
        print(f"  Dependabot: Error - {e}")
        dep_alerts = []

    # Filter by severity
    code_blockers = [
        a for a in code_alerts
        if sev_at_least(get_alert_severity(a), threshold)
    ]
    dep_blockers = [
        a for a in dep_alerts
        if sev_at_least(get_alert_severity(a), threshold)
    ]

    # Report findings
    print()
    print("=" * 60)
    print("FINDINGS SUMMARY")
    print("=" * 60)
    print(f"Secret Scanning Alerts (open):      {len(secrets)}")
    print(f"Code Scanning Alerts (>= {threshold}):   "
          f"{len(code_blockers)} / {len(code_alerts)} total")
    print(f"Dependabot Alerts (>= {threshold}):      "
          f"{len(dep_blockers)} / {len(dep_alerts)} total")
    print()

    # Policy enforcement
    print("=" * 60)
    print("POLICY EVALUATION")
    print("=" * 60)

    # Rule 1: Any open secrets block deployment
    if secrets:
        print("BLOCKED: Open secret scanning alerts present")
        print()
        print("Blocking alerts:")
        for alert in secrets[:5]:  # Show first 5
            print(f"  - #{alert.get('number')}: {alert.get('secret_type')}")
        if len(secrets) > 5:
            print(f"  ... and {len(secrets) - 5} more")
        return 2

    # Rule 2: High+ code scanning alerts block
    if code_blockers:
        print(f"BLOCKED: Code scanning alerts at/above {threshold}")
        print()
        print("Blocking alerts:")
        for alert in code_blockers[:5]:
            rule = alert.get("rule", {})
            print(f"  - #{alert.get('number')}: {rule.get('id')} "
                  f"({get_alert_severity(alert)})")
        if len(code_blockers) > 5:
            print(f"  ... and {len(code_blockers) - 5} more")
        return 3

    # Rule 3: High+ dependency alerts block
    if dep_blockers:
        print(f"BLOCKED: Dependabot alerts at/above {threshold}")
        print()
        print("Blocking alerts:")
        for alert in dep_blockers[:5]:
            pkg = alert.get("dependency", {}).get("package", {})
            print(f"  - #{alert.get('number')}: {pkg.get('name')} "
                  f"({get_alert_severity(alert)})")
        if len(dep_blockers) > 5:
            print(f"  ... and {len(dep_blockers) - 5} more")
        return 4

    print("PASSED: Security gate checks passed")
    print()
    print("Deployment may proceed.")
    return 0


if __name__ == "__main__":
    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**
Repository Configuration		
Security Settings	`PATCH /repos/{owner}/{repo}`	REST API: Repositories
Security Configs	`GET /orgs/{org}/code-security/configurations`	REST API: Configurations

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

B Environment Variables Reference

Variable	Description
GitHub Configuration	
`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
Harness Configuration	

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
Security Gate Configuration	
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