Setting Security Policies in GitHub

Preventive Controls, Documentation, Enforcement, Incident Response, and Auditing

Note

This document is a practical guide for administrators and security owners onboarding collaborators and scaling secure development on GitHub. It focuses on: preventive measures, vulnerability documentation, policy enforcement at repository/organization/enterprise levels, scrubbing sensitive data, publishing security advisories, and auditing security-relevant activity.

Contents

1 Unit Objectives and Operating Model

1.1 What you will be able to do

By the end of this unit, you should be able to:

- Define **repository, organization, and enterprise** policy boundaries and inheritance rules.
- Establish **security documentation** standards (e.g., `SECURITY.md`) and community health baseline files.
- Enable **security tooling** (Dependabot, advisories, code scanning, secret scanning) and select the correct control profile for your risk posture.
- Execute a **secret leak response** including rotation, history rewrite, and cache invalidation requests when required.
- Publish **security advisories** with consistent severity, impact scope, and remediation guidance.
- Audit activity using **audit logs**, policy change tracking, and automation/webhooks.

1.2 Mental model: Security policy as "guardrails" + "evidence"

A strong GitHubsecurity program combines:

- **Guardrails** (preventive controls): rulesets, branch protection, authentication enforcement, least privilege, and scanning.
- **Evidence** (detective controls): alerts, audit logs, advisories, and consistent documentation for triage and compliance.

2 Why Security Policies Matter

Security policies maintain the integrity of your GitHubecosystem by:

- **Guiding workflows:** secure, standardized development and release processes.
- **Reporting clarity:** consistent, actionable steps for vulnerability disclosure and triage.
- **Access control:** least-privilege permissions to limit blast radius and reduce risk.

2.1 Where policies apply

Policies can apply at multiple scopes:

- **Repository-level:** fine-grained control for a specific codebase.
- **Organization-level:** consistent defaults and governance across teams.
- **Enterprise-level:** uniform, enforceable rules across many organizations; enterprise rules can override and lock organization settings.

Note

A good operating principle is: **Enterprise enforces invariants**; **Organizations tailor within constraints**; **Repositories implement specifics**.

2.2 Visual placeholder: onboarding/security policy screens (stacked)

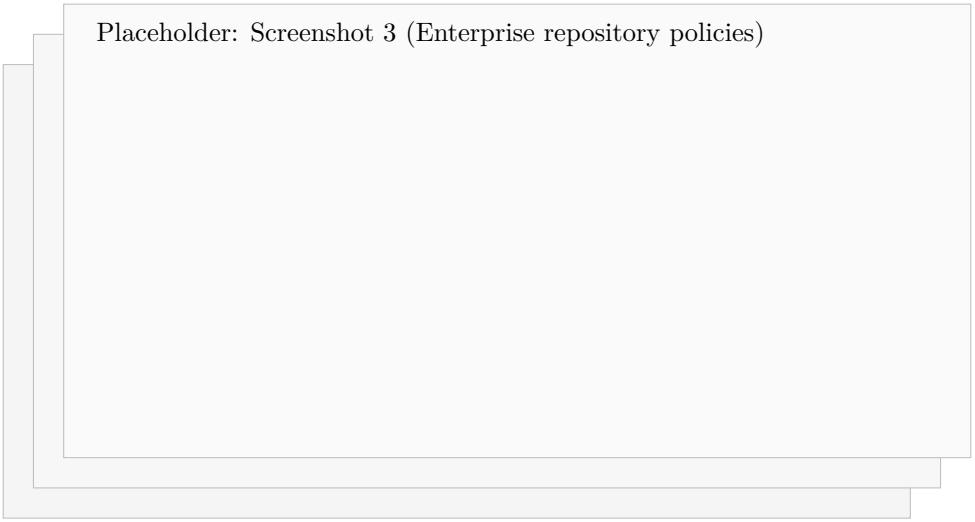

Figure 1: Illustrative "stacked" screenshots: repository community health files and organization/enterprise settings.

3 Documenting Security: Your First Line of Defense

3.1 `SECURITY.md`: required content and conventions

`SECURITY.md` communicates (1) which versions you support, (2) how to report vulnerabilities, (3) what to expect (timelines, response SLAs), and (4) any legal/safe-harbor guidance.

Example: Production-grade `SECURITY.md`

```
# Security Policy

## Supported Versions
We provide security fixes for the following versions:

| Version | Supported |
|--------:|:---------:|
| 2.x     |  |
| 1.x     |  (critical fixes only) |
| 0.x     |  |

## Reporting a Vulnerability
Please report security issues privately.
```

```
**Preferred:** GitHub Security Advisories (Private Vulnerability Report)
1. Open a new security advisory in this repository.
2. Include reproduction steps, impact, and affected versions.
3. If applicable, include a proposed fix or mitigation.

**Alternative:** Email security@example.com (PGP available upon request)

## What to Include
- Product/component name and version
- Clear reproduction steps (PoC if safe)
- Expected vs. actual behavior
- Potential impact (data exposure, RCE, privilege escalation, etc.)
- Any relevant logs, stack traces, screenshots

## Response Timeline
- Acknowledgement: within 2 business days
- Triage complete: within 5 business days
- Fix or mitigation plan: within 10 business days (or communicate constraints)

## Coordinated Disclosure
We support coordinated disclosure. Please do not publicly disclose details
until we have confirmed a fix and coordinated a release timeline.

## Safe Harbor
We will not pursue legal action for good-faith security research that:
- avoids privacy violations and service disruption,
- uses test accounts where possible,
- does not exfiltrate or retain sensitive data.
```

3.2 Security documentation beyond `SECURITY.md`

Security documentation should also include:

- **Triage SOPs:** severity criteria, escalation steps, ownership, and timelines.
- **Exception handling:** how policy exceptions are requested, reviewed, approved, and time-boxed.
- **Evidence guides:** how to prove controls are operating (audit logs, scanning configurations, rulesets).

3.3 Visual placeholder: SECURITY.md snippet screenshot

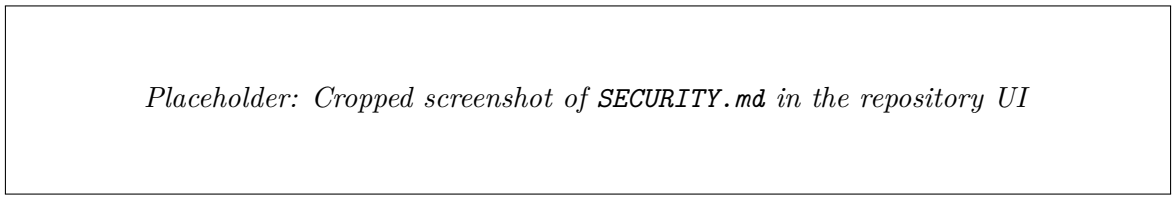

Figure 2: Example location and visibility of `SECURITY.md` in a repository.

4 Community Health Files and Standardization

4.1 Recognized community files

GitHubrecognizes key community files that improve transparency and reduce friction:

File	Purpose
`CODE_OF_CONDUCT.md`	Community behavior standards and enforcement steps.
`CONTRIBUTING.md`	Contribution guidelines: branching, style, tests, review expectations.
Discussion category forms	Templates that standardize discussions and reduce back-and-forth.
`FUNDING.yml`	Sponsor options and funding metadata (as applicable).
`GOVERNANCE.md`	Decision-making structure and escalation paths.
Issue/PR templates + `config.yml`	Standardize contributor input (repro steps, risk, evidence).
`README.md`	Canonical project overview, build/test instructions, support links.
`SUPPORT.md`	Help resources and support channels (internal or external).

4.2 Examples: templates that reduce security triage time

Example: Security-focused bug report issue template (Markdown)

```
---
name: "Security Bug Report"
about: "Report a security-relevant defect (non-sensitive)."
title: "[SECURITY] <short summary>"
labels: ["security", "triage"]
assignees: []
---

## Summary
Describe the security issue at a high level.

## Impact
- [ ] Confidentiality
- [ ] Integrity
- [ ] Availability
- [ ] Privilege escalation
- [ ] Other: ____

## Affected Component(s)
List services, modules, endpoints, workflows, etc.

## Reproduction Steps
1.
2.
3.

## Expected vs. Actual Behavior
**Expected:**
**Actual:**
```

```
## Environment
- Version/commit:
- OS:
- Deployment (prod/stage/dev):

## Evidence
Logs, traces, screenshots (redact sensitive info).
```

Example: `CONTRIBUTING.md` security addendum

```
## Security Requirements (Applies to all contributions)

- All changes must include tests for security-relevant logic.
- Do not commit secrets. Use environment variables and secret managers.
- Ensure dependencies are updated and vulnerable packages are not introduced.
- Follow secure coding guidelines (input validation, authz checks, logging hygiene).
- Pull Requests require review by CODEOWNERS for critical paths.
```

5 Security Settings and Enforcement: Trust vs. Control

5.1 Choosing a control posture

Small teams often start with broader permissions and evolve toward stricter controls as the organization scales. Large teams generally require:

- enforced authentication (SSO/2FA),
- standardized branch rules,
- mandatory reviews for sensitive code,
- consistent scanning and alert handling,
- auditable changes to policies and permissions.

5.2 Configuration levels and inheritance

- **Organization-level:** *Settings* → *Member privileges*
- **Enterprise-level:** *Your enterprises* → *Policies* → *Repository policies*

Warning

Enterprise rules may **override** organization rules. If a setting is locked at the enterprise level, organization owners cannot change it.

5.3 Quadrant model: availability vs. interaction required

The following conceptual model helps categorize security settings by:

- **X-axis:** availability to regular users (limited → broadly available)

- **Y-axis:** required level of interaction (low → high)

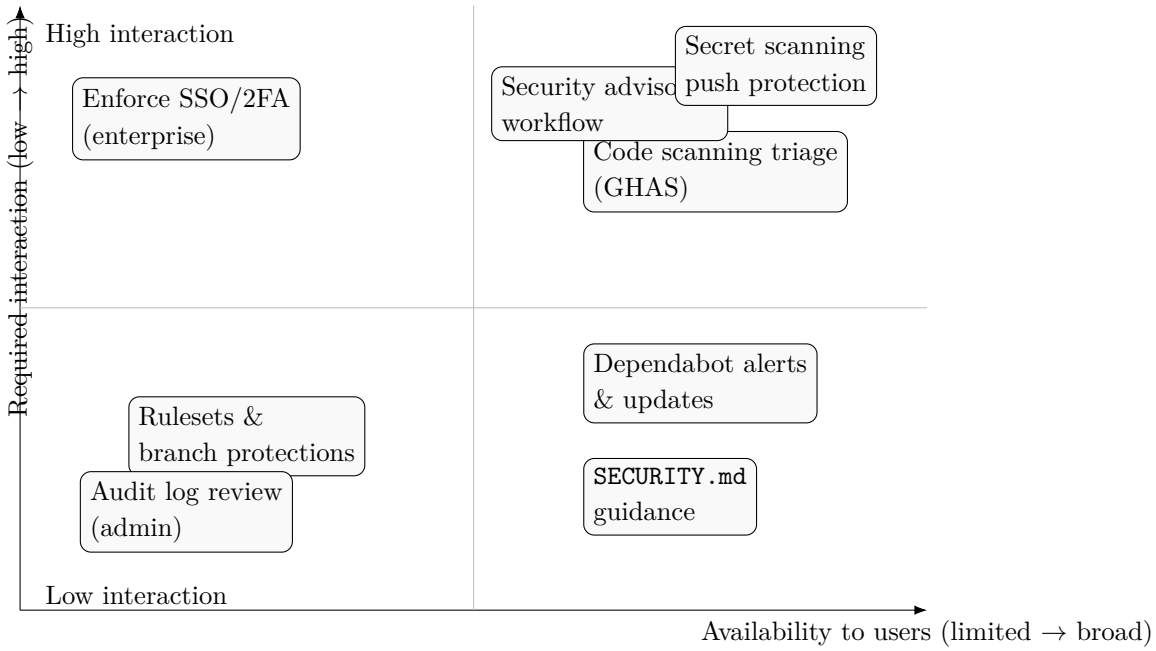

Figure 3: Quadrant model for thinking about how "available" a control is vs. how much human interaction it requires.

Note

Use this model to prioritize adoption: start with **high-leverage, low-interaction guardrails** (rulesets, baseline scanning, defaults), then expand to **high-interaction workflows** (advisories, deep triage, exception management).

6 Available Security Tools and What They Enable

6.1 Capabilities by repository tier

Repo Tier	Typical Security Capabilities
All repositories	Access controls, `SECURITY.md`, Dependabot alerts/updates, advisories
With GitHub Advanced Security (GHAS)	Code scanning, secret scanning, dependency review (and broader security reporting)

Table 2: Security capabilities vary by plan and enabled features.

6.2 Recommended baseline: "minimum viable security"

A practical baseline for most teams:

Checklist

- **Documentation:** `SECURITY.md` + issue/PR templates
- **Dependencies:** Dependabot alerts + automated update PRs
- **Branch safety:** rulesets / branch protection with required checks
- **Secrets:** secret scanning (and push protection if available)
- **Code:** code scanning with a documented triage process (GHAS)
- **Auditability:** audit log access + policy change review cadence

7 Enhancing Enterprise Security and Compliance with GitHub

7.1 Enterprise security features

- GitHub Advanced Security (GHAS): code scanning, secret scanning, dependency review.
- **Security configurations:** enforce consistent settings across repositories.
- **Centralized policy enforcement:** enterprise rulesets and locked organization settings.

7.2 Compliance support (audit readiness)

For regulated environments, align policies to evidence:

- **Compliance reports:** provide standardized attestations useful for audit and regulatory needs.
- **Controls as evidence:** branch protection + review enforcement, signed commits, SSO/2FA, audit logging, and security scanning configurations.

Example: "Control-to-evidence" mapping excerpt

Control Objective	GitHub Feature	Evidence Artifact
Change approval	Required PR reviews	Ruleset config + PR review logs
Identity assurance	SSO + 2FA enforcement	Auth policy + audit events
Secure code	Code scanning	Alert exports + triage records
Secret hygiene	Secret scanning	Alert history + remediation commits

8 Scrubbing Sensitive Data from GitHub Repositories

When secrets leak, remediation typically has three parallel tracks:

1. **Containment:** revoke/rotate exposed credentials immediately.
2. **Eradication:** remove sensitive data from repository history if required.
3. **Recovery and prevention:** reissue credentials, strengthen controls (push protection, scanning, training).

Warning

Rotating the secret is non-negotiable. Even if you rewrite history, assume the secret was compromised the moment it was committed and pushed.

8.1 Option A: Legacy history rewrite using `git filter-branch`

Remove a sensitive file using `git filter-branch` (legacy)

```
git filter-branch --force --index-filter \
  'git rm --cached --ignore-unmatch path/to/sensitive_file' \
  --prune-empty --tag-name-filter cat -- --all
```

8.2 Option B: BFG Repo-Cleaner (recommended for many history rewrite cases)

BFG: delete files or replace secret strings

```
# Delete a file across history
java -jar bfg.jar --delete-files path/to/sensitive_file

# Replace strings using a rules file (passwords.txt contains patterns to replace)
java -jar bfg.jar --replace-text passwords.txt
```

8.3 Option C: `git filter-repo` (modern, flexible, and scriptable)

Example: remove a file from all history using `git filter-repo`

```
# Install (one option): pipx install git-filter-repo
# Remove the file from history:
git filter-repo --path path/to/sensitive_file --invert-paths
```

8.4 Clean up and force-push

After rewriting history, clean and force push to overwrite remote history.

Cleanup and force push

```
git reflog expire --expire=now --all
git gc --prune=now --aggressive

# Force push branches and tags
git push origin --force --all
git push origin --force --tags
```

8.5 Contact GitHub Support (cache invalidation and persistence considerations)

For public repositories (and in some cases mirrors/forks), cached indexes may persist. A typical escalation path:

1. Rewrite history & force-push.
2. Provide **repository name**, **commit(s)**, **file path(s)**, and confirmation of rewrite.
3. Request cache/index invalidation where applicable.
4. If risk is severe, consider temporarily deleting the repo (only if governance allows) and recreating after containment.

8.6 Prevention mechanisms to reduce recurrence

Checklist

- Use `.gitignore` for local secret files and environment overrides.
- Store secrets in GitHubActions Secrets, organization secrets, or an external secret manager.
- Use secret scanning and (if available) push protection to block secrets before they land.
- Adopt pre-commit hooks to detect high-risk patterns before commit.

Example: pre-commit hook for basic secret pattern detection (starter)

```
#!/usr/bin/env bash
set -euo pipefail

# Simple heuristic patterns (extend to your environment)
PATTERNS=(
  "AKIA[0-9A-Z]{16}"          # AWS Access Key ID pattern
  "-----BEGIN PRIVATE KEY-----"
  "xox[baprs]-"               # Slack tokens (approx)
)

FILES_CHANGED=$(git diff --cached --name-only)

for f in $FILES_CHANGED; do
  # Skip binary files
  if file "$f" | grep -qi "text"; then
    for p in "${PATTERNS[@]}"; do
      if grep -nE "$p" "$f" >/dev/null 2>&1; then
        echo "Potential secret detected in $f (pattern: $p). Commit blocked."
        exit 1
      fi
    done
  fi
done
```

9 Publishing Security Advisories

Security advisories help you:

- collaborate privately on fixes,
- communicate vulnerability details clearly,

- document mitigation steps and patch status,
- (optionally) request or reference CVEs where appropriate.

9.1 Workflow diagram (3-step)

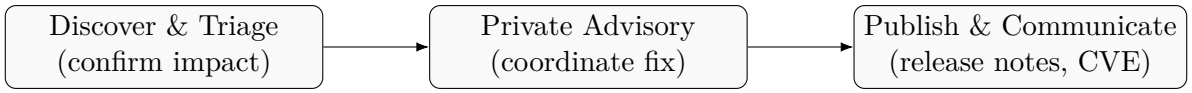

Figure 4: Typical end-to-end security advisory lifecycle.

9.2 What a "good advisory" contains

A strong advisory should include:

- **Affected versions:** exact ranges and deployment contexts.
- **Severity:** clear rating, with rationale (e.g., CVSS vector if used).
- **Impact:** confidentiality/integrity/availability and exploit prerequisites.
- **Patch status:** fixed in version(s) X, mitigation for Y, workaround for Z.
- **References:** CVE references or linked tracking items (internal IDs if private).

Example: Advisory template (copy/paste starter)

```
## Summary
A vulnerability in <component> allows <impact> when <condition>.

## Affected Versions
- Product: <name>
- Affected: >=1.2.0, <1.4.3
- Not affected: <1.2.0, >=1.4.3

## Impact
- CIA: Confidentiality (High), Integrity (Medium), Availability (Low)
- Exploitability: Remote / Authenticated / Requires user interaction (select as
↪  applicable)

## Details
Technical description, root cause, and how it can be triggered.

## Mitigation
- Upgrade to 1.4.3 or later.
- If upgrade is not possible: disable <feature>, apply <config>, restrict
↪  <endpoint>.

## Credits
Thanks to <reporter> for responsible disclosure.

## References
```

```
- CVE: (if assigned)
- Internal tracking: SEC-1234
```

10 Security and Compliance Profiles (Control Levels)

Each policy balances security and usability. The table below provides a practical control profile model.

Control Level	Recommended Policies & Features	Use Case
Low Control (Guidance & Best Practices)	Security advisories & code scanning; Dependabot alerts; branch protection rules (optional reviews)	Teams needing flexibility with best practices
Moderate Control (Enforced Rules)	Required branch protection; commit signing; org-wide security policies; monitoring webhooks	Teams needing governance with developer autonomy
High Control (Strict Compliance & Governance)	Enforce SAML SSO & 2FA; restrict visibility & forking; mandatory PR approvals; prevent force pushes; CI/CD security checks	Strict compliance (e.g., SOC 2, ISO 27001)

Table 3: Security policies categorized by level of control.

10.1 When to use which profile

- **Startups & agile teams:** often operate effectively at **moderate control** (branch protections, Dependabot, secret scanning).
- **Enterprises & regulated industries:** typically require **high control** (SSO/2FA, audit logging, rulesets, strict repo controls).
- **Open source projects:** commonly **low to moderate control** with strong guidance, scanning, and clear community files.

11 Key Security and Compliance Features in GitHub Enterprise

11.1 Secure code development

- **Code scanning (GHAS):** detect vulnerabilities via CodeQL and third-party tools.
- **Secret scanning:** prevent hardcoded secrets and reduce exposure window.
- **Dependency review & Dependabot:** identify and update vulnerable dependencies.

11.2 Enforcing compliance policies

- **Branch protection rules:** require PR reviews, status checks, and signed commits.
- **Security rulesets:** apply policies across multiple repos consistently.

- **Audit logs & API monitoring:** track activity, permission changes, and policy modifications.

11.3 Controlling access and authentication

- **Enforce SAML SSO & 2FA:** strong authentication for all users.
- **Restrict repository visibility:** control who can view, fork, or clone.
- **Fine-grained access control:** assign roles per team or project; follow least privilege.

12 Defining Organization and Enterprise Policies

Organization and enterprise policies set governance, access, and workflow rules to ensure security and compliance.

12.1 Key policy domains

- **Security & access control:** SAML SSO, 2FA, RBAC, repository visibility.
- **Compliance & governance:** audit logging, branch protection, commit signing.
- **Development workflow & automation:** PR approvals, security rulesets, Actions policies.
- **Code & dependency security:** code scanning, secret scanning, Dependabot, action restrictions.

Warning

Any enterprise-level policy configured under *Your enterprises →Policies →Repository policies* can override organization-level settings under *Settings →Member privileges*.

12.2 Extensive examples: policy-as-configuration starters

12.2.1 Example: `CODEOWNERS` for sensitive paths

Require security review for high-risk code paths using `CODEOWNERS`

```
# Default owners
* @platform-team

# Security-sensitive paths
/auth/ @security-team
/infra/ @security-team @sre-team
/.github/workflows/ @security-team
```

12.2.2 Example: Dependabot configuration

Example: `.github/dependabot.yml`

```yaml
version: 2
updates:
  - package-ecosystem: "npm"
    directory: "/"
```

```yaml
    schedule:
      interval: "weekly"
    open-pull-requests-limit: 10
    labels:
      - "dependencies"
      - "security"

  - package-ecosystem: "pip"
    directory: "/"
    schedule:
      interval: "weekly"
    labels:
      - "dependencies"
      - "security"

  - package-ecosystem: "github-actions"
    directory: "/"
    schedule:
      interval: "weekly"
    labels:
      - "dependencies"
```

12.2.3 Example: CodeQL workflow (starter)

Example: `.github/workflows/codeql.yml` (starter)

```yaml
name: "CodeQL"

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
  schedule:
    - cron: "0 5 * * 1"

jobs:
  analyze:
    name: Analyze
    runs-on: ubuntu-latest
    permissions:
      actions: read
      contents: read
      security-events: write

    strategy:
      fail-fast: false
      matrix:
        language: [ "javascript-typescript" ]  # add more languages as needed

    steps:
```

```yaml
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Initialize CodeQL
        uses: github/codeql-action/init@v3
        with:
          languages: ${{ matrix.language }}

      - name: Autobuild
        uses: github/codeql-action/autobuild@v3

      - name: Perform CodeQL Analysis
        uses: github/codeql-action/analyze@v3
```

12.2.4 Example: branch protection "intent" (rules you typically enforce)

Branch protection intent checklist (translate into rulesets/branch protections)

- Require pull request reviews (1–2 reviewers; require CODEOWNERS for sensitive paths).
- Require status checks (CI build, tests, lint, security scans) before merge.
- Restrict who can push to matching branches.
- Prevent force pushes and deletion on protected branches.
- Require linear history (optional) and signed commits (for higher assurance).

13 Auditing and Operational Governance

13.1 What you should audit

- Changes to authentication policies (SSO/2FA enforcement).
- Repository visibility changes (public/private/internal), forking policy changes.
- Modifications to rulesets/branch protection and required status checks.
- Addition/removal of administrators and elevated roles.
- Security alert lifecycle events (created, triaged, dismissed, fixed) and reasons.

13.2 Example: audit log queries using `gh` CLI + API

The following examples assume you have `gh` configured on your workstation.

Example: Retrieve organization audit log (illustrative)

```bash
# Replace ORG with your org name. Adjust include/phrase filters as needed.
gh api -X GET "/orgs/ORG/audit-log" \
  -f include=all \
  -f phrase="action:protected_branch.create OR action:protected_branch.update"
```

Example: Monitor changes to repository visibility or permissions (illustrative)

```
gh api -X GET "/orgs/ORG/audit-log" \
  -f include=all \
  -f phrase="action:repo.create OR action:repo.destroy OR action:repo.add_member OR
  ↪  action:repo.remove_member"
```

Note

Treat audit review as a recurring control: define a cadence (weekly/monthly), required reviewers, and evidence retention (export to a SIEM or ticketing system when appropriate).

14 Appendix A: "Day-1" Security Policy Pack (Quick Start)

14.1 Repository-level "Day-1" pack

Repository Baseline

- Add `SECURITY.md`, `README.md`, `CONTRIBUTING.md`, issue/PR templates.
- Enable Dependabot alerts and schedule Dependabot update PRs.
- Configure branch protections / rulesets for `main`.
- Add `CODEOWNERS` for sensitive code paths.
- Enable code scanning and secret scanning (if available).

14.2 Organization-level "Day-1" pack

Organization Baseline

- Standardize member privileges and default repository permissions.
- Define who can create repositories, manage Actions, and change visibility.
- Require 2FA (and SSO if enterprise-managed).
- Establish security ownership: escalation paths and triage responsibilities.

14.3 Enterprise-level "Day-1" pack

Enterprise Baseline

- Enforce SSO/2FA across organizations.
- Lock critical repository policies (visibility, forking, force push protections).
- Roll out security configurations for consistent scanning and alerting.
- Centralize audit logging and alert data retention requirements.

15 Appendix B: Reference File Tree (Suggested)

Example repository structure for community health + security configs

```
.
 .github
    CODEOWNERS
    dependabot.yml
    ISSUE_TEMPLATE
       security-bug-report.md
       bug_report.md
    PULL_REQUEST_TEMPLATE.md
    workflows
        ci.yml
        codeql.yml
 CONTRIBUTING.md
 GOVERNANCE.md
 README.md
 SECURITY.md
 SUPPORT.md
```