

Set up and secure a new GitHub account and organization

Configuring GitHub securely for a small team. Covers account and commit signing, organization policies, third-party and token access, branch rulesets, and security scanning.

GITHUB · CIS GITHUB BENCHMARK v1.2.0 · ORGANIZATION SETUP · HOW-TO GUIDE

General educational guidance, not affiliated with or endorsed by the Center for Internet Security or GitHub.

Recommendation numbers refer to the CIS GitHub Benchmark v1.2.0 (18 February 2026). Published 2026-10-01.

12 steps, account to verification

one sitting, while the organization is still empty

50 recommendations in Section 1

of the 120 in CIS v1.2.0, where almost all the GitHub settings live

SCOPE What this guide covers, and what it does not

This guide takes a small team from a GitHub account, new or existing, to a secured organization in one sitting, while the organization is still empty. It follows the CIS GitHub Benchmark v1.2.0, the security baseline for GitHub published by the Center for Internet Security (CIS). Each setting the benchmark asks for carries its recommendation number, such as 1.3.8, so you can show an assessor where it came from. The benchmark is written as an audit of a live organization, followed by fixes. Applying the same settings to an empty organization creates no exceptions to unwind later.

The benchmark holds 120 recommendations across five sections. Section 1, Source Code, holds 50 of them and is where almost all the GitHub-specific settings live; this guide implements the ones you can put in place while the organization is empty. Sections 2 to 5 cover build pipelines, dependencies, artifacts, and deployment, mostly in product-neutral language. From Sections 2, 3, and 4 the guide takes the Actions and attestation controls that apply before you have written any code.

If you have only an hour, do these five

RANKED BY RISK REDUCED PER MINUTE SPENT

1. Require 2FA for everyone (Step 4). A stolen password is then not enough to sign in.
2. Set base permissions to Read or No permission (Step 5). Membership alone gives no write access to any repository.
3. Restrict personal access tokens (Step 7). A leaked token cannot reach your private code.
4. Put a ruleset on the default branch, with required review and an empty bypass list (Step 10). No commit reaches the default branch without a review.
5. Turn on push protection (Step 11). A commit that contains a credential is rejected at push, before it lands in the repository. This one is free on public repositories and metered on private ones, so check the PLAN table first.

LIMITS lists what the full setup still leaves open.

The guide refers to its sections by these labels:

- **PLAN** and **PREP**: which GitHub plan to choose, and what to have ready.
- **Steps 1 to 12**: the setup, in order, ending with an API check of the settings.
- **NEXT**: signing release artifacts, once you ship software.

- **SMALL:** what changes for a team of one or two.
- **HELP:** the break-glass route around your own rules, common problems, and offboarding.
- **REVIEW:** the checks to put on a calendar.
- **EXCEPT** and **LIMITS:** what you cannot do below GitHub Enterprise Cloud, and what the setup does not protect against.
- **NOTES** and **DONE:** how the guide maps to the benchmark, what you achieved, and what to tell an assessor.

PLAN What each GitHub plan includes

Decide this first, because it determines how much of the benchmark you can satisfy. GitHub Free covers the organization-wide policy settings but gives you rulesets on public repositories only, so a team with private code needs Team.

CAPABILITY	CIS	FREE	TEAM	TEAM + PAID ADD-ONS
Require 2FA for everyone	1.3.4, 1.3.5	Yes	Yes	Yes
Base permissions and member privileges	1.2.2-1.2.4, 1.2.6, 1.3.2, 1.3.8	Yes	Yes	Yes
OAuth app and GitHub App restrictions	1.4.1	Yes	Yes	Yes
Personal access token policy	-	Yes	Yes	Yes
Rulesets on public repositories	most of 1.1.3-1.1.20	Yes	Yes	Yes
Rulesets on private repositories	most of 1.1.3-1.1.20	No	Yes	Yes
Organization-wide rulesets	most of 1.1.3-1.1.20	No	Yes	Yes
Security configurations	1.5.1, 1.5.2, 1.5.4-1.5.6	Yes	Yes	Yes
Dependabot alerts and updates	1.5.5	Yes	Yes	Yes
Secret scanning and push protection, public repos	1.5.1	Yes	Yes	Yes
Secret scanning and push protection, private repos	1.5.1	No	No	Yes, \$19 per active committer per month
Code scanning, public repos	1.5.4	Yes	Yes	Yes
Code scanning, private repos	1.5.4	No	No	Yes, \$30 per active committer per month
Domain verification and the verified badge	1.3.9	Yes	Yes	Yes
Artifact attestations, private repos	2.4.1, 4.1.1	No	No	No, Enterprise Cloud only

CAPABILITY	CIS	FREE	TEAM	TEAM + PAID ADD-ONS
IP allowlist, SSH CA, restricted email domains, dormant-user report	1.3.1, 1.3.10- 1.3.12	No	No	No, Enterprise Cloud only

GitHub Secret Protection covers secret scanning and push protection. GitHub Code Security covers code scanning, dependency review, and the premium Dependabot features. GitHub groups the two as its Advanced Security products, which is also the name of the settings section Step 11 uses. Both became available to Team organizations on 1 April 2025 and are priced per active committer per month (\$19 and \$30 at publication). On public repositories the features are free on every plan, which covers five of the six Section 1.5 recommendations at no cost; the sixth, 1.5.3, needs a scanner you supply on any plan and any visibility.

A committer counts as active when one of their commits has been pushed in the last 90 days to a private or internal repository with the feature switched on, whenever the commit was originally written. Only members, outside collaborators, and people with a pending invitation count. The license count follows those committers across every such repository, so the scope you apply the Step 11 configuration to is what sets the bill.

PREP

Before you start

- A **domain you control** and a **role mailbox** on it, such as `github@example.com`. The billing contact and the organization's public email should never be one person's inbox.
- A **password manager**, shared with whoever the second owner will be. Shared break-glass credentials live here. Each person's own recovery codes do not, for the reason given in Step 1.
- A **hardware security key or an authenticator app**. Register two methods, so losing one does not lock you out (Step 1).
- A **second person** lined up as the other organization owner. GitHub's guidance is no fewer than two, and SMALL covers what to do when there is only one of you.
- The **GitHub CLI** (`gh`) installed, for the verification in Step 12.

01

Create the personal account

Organizations are owned by personal accounts, so the account you use here becomes your most powerful credential. Sign up at `github.com` with a company-domain address, and use a long unique password generated by your password manager. If you already have a GitHub account, use it rather than open a second one: GitHub's terms allow one free account per person. Add the company address to it under **Settings > Emails**, and apply the rest of this step to that account.

Turn on two-factor authentication immediately, from **your profile picture > Settings > Password and authentication > Enable two-factor authentication**. Since March 2023 GitHub has required 2FA of accounts it selects as contributors, and owning an organization that holds repositories or other users makes yours eligible. The methods, strongest first:

Passkeys

Sign in without a password at all, and phishing-resistant by design. The best default where your devices support it.

Security keys

A hardware token. GitHub recommends registering one as a backup even after another method is configured.

TOTP applications

GitHub strongly recommends a time-based one-time password app over SMS, with security keys as the backup. Store the setup key in a password manager that syncs, so a lost phone does not cost you the second factor. The setup key is the secret behind the QR code GitHub shows during setup.

SMS

The weakest option, and the one GitHub explicitly steers you away from. Do not set it up: Step 4 blocks anyone with SMS configured from the organization's resources, even alongside a stronger method.

- **Set up a TOTP app before a security key or passkey**

GitHub's 2FA setup starts with a TOTP app or SMS. A security key can be registered only after one of those is configured, and GitHub documents the same prerequisite for a passkey used as 2FA. A passkey added on its own from the passkey settings signs you in, but complete the 2FA setup as well, because GitHub issues the recovery codes below during that setup. Set up TOTP, add the passkey or security key straight after, and keep the TOTP entry as your second method.

Download the recovery codes now

SUPPORT CANNOT RESTORE A LOCKED ACCOUNT

GitHub issues recovery codes during 2FA setup, and they are the documented way back into a locked account. GitHub's stated position is that Support cannot restore access to an account whose owner has lost their 2FA credentials. GitHub says not to share or distribute the codes, and to save them somewhere secure, such as a password manager. They stay with the person whose account they open. Organizational continuity comes from the second owner, who can keep running the organization while the first one recovers.

With the codes gone, the documented route is an account recovery request, proved with a previously verified device, an SSH key added as an authentication key, or a personal access token. The signing key from Step 2 does not count. GitHub says a recovery request can take up to three business days and may be denied. Configuring a fallback SMS number is no longer supported.

Add a second factor as well as a first. This also sets the example for the 2FA requirement you will impose on the whole organization in Step 4, where members who have not enrolled lose access.

02 Sign your commits

Set up commit signing now, on your own machine, so the signed-commits rule in Step 10 never blocks you. That rule enforces recommendation 1.1.12, which asks that every commit in a pull request be signed and verified before merging. GitHub supports GPG, SSH, and S/MIME signatures. SSH is the simplest to generate and can reuse a key you already have. GitHub rejects an RSA key shorter than 3072 bits if it is uploaded on or after 14 October 2026.

```
# Generate a key if you do not already have one
ssh-keygen -t ed25519 -C "you@example.com"
```

```
# Point Git at it for signing
git config --global gpg.format ssh
git config --global user.signingkey ~/.ssh/id_ed25519.pub
git config --global commit.gpgSign true
git config --global tag.gpgSign true
```

Upload the public key from **Settings > SSH and GPG keys > New SSH key**, and set the key type to signing rather than authentication. An authentication key and a signing key are separate entries even when the key material is identical, so upload the same key twice if you want it to do both jobs.

- **Git 2.34 or later**

SSH signature verification landed in Git 2.34. An older Git rejects the config outright with `error: unsupported value for pgp.format: ssh`, so you will know. Check with `git --version`.

- **The committer email must be verified on the account**

A signature over a commit whose email GitHub does not recognize shows as unverified. Confirm the address under **Settings > Emails** before you push.

Finish by turning on vigilant mode at **Settings > SSH and GPG keys > Vigilant mode**, ticking **Flag unsigned commits as unverified**. Without it, an unsigned commit carries no badge at all and looks like any other commit. With it, anything you did not sign is labeled Unverified, so a missing signature is visible to whoever reads the history. The signed-commits rule in Step 10 also accepts commits GitHub marks Partially verified. That label means the signature is verified, but the author is not the committer and has vigilant mode on.

03 Create the organization

Create it from **your profile picture > Settings > Organizations > New organization**, and choose the plan you settled on in the PLAN table. Three decisions are hard to reverse later.

The name

It appears in every clone URL your team will ever type. Renaming later works only partly: web links to existing repositories redirect and old remotes keep working, while API calls on the old name return 404, links to the old profile page break, and team @mentions using it stop resolving. Your old organization name also becomes available for someone else to claim.

The billing contact

Point it at the role mailbox, never a founder's personal address. Billing email is where seat and license warnings land, and you need to read them once Secret Protection or Code Security is billed per active committer.

The owners

Organization owners hold the highest permission level: they add and remove collaborators, create and delete repositories, change protection rules, and flip a private repository public. Choose them deliberately, and see the card below for how many.

Start with two owners

RECONCILING CIS 1.3.3 WITH GITHUB'S GUIDANCE

The benchmark pushes the owner count down and GitHub puts the floor at two. At a founding team's size, two owners satisfy both. One owner is a single point of failure for account recovery and offboarding. Each owner past the second adds another account whose compromise would hand over the organization, and two owners already remove the single point of failure. Larger or spread-out teams have their own reasons to go past two, such as time zones and on-call cover, so revisit the number as you grow instead of treating two as a permanent maximum.

Record who the owners are, and review the list whenever someone leaves. Give the two owners different second-factor types, and store at least one hardware key away from the laptop it unlocks, so a single theft cannot reach both accounts.

04 Require two-factor authentication

Go to **Settings > Security > Authentication security**, select **Require two-factor authentication for everyone in your organization**, and save. This one screen covers recommendations 1.3.4 and 1.3.5, which ask for multi-factor authentication (MFA) for outside collaborators and for members. Tick **Only allow secure two-factor methods** alongside it. That narrows the accepted methods to passkeys, security keys, authenticator apps, and the GitHub mobile app, and shuts out SMS, the method Step 1 told you not to set up. Anyone with SMS configured at all, even alongside a stronger method, is blocked from organization resources until they remove it.

Do it while the organization is empty and it costs nothing. Do it later and it removes access:

- **Members and billing managers** without 2FA lose access to organization resources until they enroll. They keep their membership and continue to consume a seat.
- **Outside collaborators**, people who can reach an organization repository without being members, are removed outright if they lack 2FA. They lose access to the organization's repositories, including their own forks of its private repositories. An owner can reinstate them within three months once they enroll, and after that the access is gone.

You have to have 2FA on your own account before you can impose it, which Step 1 handled. If you are turning this on for an organization that already has people in it, check the People page first to see who would be removed, and warn them.

05 Set member privileges and base permissions

Go to **Settings > Access > Member privileges** and set each row below. This one screen holds seven of the benchmark's recommendations.

SETTING	SET IT TO	CIS	WHY
Base permissions	Read, or No permission	1.3.8	The level every member gets on every repository automatically. GitHub documents Read as the default for public repositories and gives no default for private ones, so read the current value and set it. Anything at Write or above hands the whole organization push access to every repository, whatever their actual role. Read gives every member read access to every repository. No permission leaves members only the public repositories and the ones a team or a direct grant gives them, so set up team access before you choose it. Grant the rest per team.
Repository creation	Both boxes cleared	1.2.2	Leaves creation with owners. Keeps the set of repositories known and removes the path where a member creates a public repository and puts organization code in it.
Repository deletion and transfer	Off	1.2.3	Deletion and transfer are unrecoverable by the person who did them. Owners only.
Issue deletion	Off	1.2.4	Deleting an issue erases a record. It is also a way to hide activity.
Repository visibility change	Off	1.2.6	Clear Allow members to change repository visibilities for this organization and only owners can flip a repository public. Clearing the box removes the route; the benchmark only asks you to review visibility changes after the fact, and the REVIEW table below still does that. Leave this setting on and GitHub documents the gap it opens: anyone with repository admin access could then change an existing repository's visibility even where creating one with that visibility is blocked.

SETTING	SET IT TO	CIS	WHY
GitHub App installation	Owners only	1.4.1	Clear Allow repository admins to install GitHub Apps for their repositories . Without it, a repository admin can install any app that asks for neither organization permissions nor repository administration permissions, which is the one way an app gets installed without an owner approving it (Step 6). Repository admins then use the request flow instead. The App access requests setting on this same screen controls who may raise those requests.
Team creation rules	Members cannot create teams	1.3.2	Teams inherit permissions from parent teams, so a member who can create a team can give themselves the parent team's access.
Repository forking	Off unless you need it	-	Forks of private repositories multiply the copies of your code you have to account for. Turn it on deliberately, per need. The benchmark's ask is that you track forks (1.2.5), so if you leave forking on, add the review in the REVIEW table below.
Pages creation	Public cleared	-	A Pages site is public on every plan below Enterprise Cloud. On Team a repository admin or maintainer can publish one from a private repository; on Free, Pages works from public repositories only. Clear the box until a repository needs a site.

Recommendation 1.3.6 asks that new members be invited using a company-approved email address. On Team no setting enforces this, so it is a process control. Send each invitation to the person's company address rather than their username. Then check the pending list under **People > Invitations**, where an owner can cancel an invitation until it is accepted. Any repository admin can also invite an outside collaborator directly, and the setting that reserves that to owners is Enterprise Cloud only, which is one more reason Step 9 keeps repository admin to the two owners. Enterprise Managed Users is the tier where identity is provisioned rather than invited.

Verifying a domain does not gate who can be invited. It earns the verified badge, recommendation 1.3.9, which a Team organization can get: add the domain at **Settings > Security > Verified and approved domains** and publish the TXT record it gives you. The badge shows once the website and public email on the organization's profile use that domain. Restricting email notifications to verified domains (1.3.10) is a separate setting and needs Enterprise Cloud.

06 Restrict third-party application access

Require an owner's approval before any application is installed, as recommendation 1.4.1 asks. GitHub splits this across two application types that behave differently.

GitHub Apps: close the repository-admin route

By default organization owners install them, and everyone else sends a request an owner has to accept, so most of the approval gate is already in place. A repository admin can still install an app for a repository they administer whenever it asks for neither organization permissions nor repository administration permissions; Step 5 turns that off. Members can also authorize GitHub's own privileged GitHub Apps, Slack and Microsoft Teams for GitHub among them, and use them to reach organization data without approval.

What is left is recommendation 1.4.3, the principle of least privilege: an app gets the access it needs and no more. Under **Settings > Third-party Access > GitHub Apps**, open each installed app and confirm both its permission set and the list of repositories it can reach. Prefer selected repositories over all repositories.

OAuth apps: confirm the restriction is on

A new organization is created with OAuth app access restrictions enabled: members request approval and owners are notified. Confirm it anyway under **Settings > Third-party Access > OAuth app policy**: an organization where

the restriction was ever lifted stays open until an owner restores it. GitHub documents one limit: even with restrictions on, users can still authorize GitHub's own privileged OAuth apps, the GitHub CLI among them, and use them to access data from the organization.

An empty organization has no webhooks yet, so make this a standing rule: every webhook payload URL starts `https://` with SSL verification enabled, at both organization and repository level (1.4.4). HTTPS meets the benchmark and still leaves the endpoint open to anyone who guesses the URL, so set a secret on every webhook as well. Have the receiver hash the raw request body with HMAC-SHA256, using that secret as the key. Before acting on the payload, have it compare the result with the `X-Hub-Signature-256` header, using a constant-time comparison. Reject anything that fails. Check both whenever you add a webhook.

07 Set a personal access token policy

Set the token policy at **Settings > Personal access tokens > Settings**, before anyone creates a token. The benchmark has no recommendation for personal access tokens. A leaked classic personal access token (PAT) reaches every repository the account that created it can reach, within the scopes chosen when it was created. None of the approval gates from Steps 5 and 6 apply to it: classic tokens have no request flow, no per-repository scoping, and no owner review.

Restrict access via personal access tokens

Choose this option on both tabs. The screen has a **Fine-grained tokens** tab and a **Tokens (classic)** tab, and the restriction applies only to the kind of token whose tab you are on. Setting one tab and stopping leaves the other kind reaching your private code. Blocking both is the right starting point for an organization that has not yet decided which automation it needs. Public resources the organization owns stay reachable either way, so read this as a control over private code.

Require approval of fine-grained personal access tokens

This takes effect once you allow fine-grained tokens. Each token a member creates then needs an owner's approval before it can reach the organization, which gives you a record of what automation exists and what it can touch. Tokens an owner creates need no approval. Approval is already the default, so confirm it while you are on the **Fine-grained tokens** tab. Classic tokens have no equivalent approval flow, which is the practical argument for blocking them outright.

Maximum lifetime

Set a maximum on both tabs. Fine-grained tokens arrive with a 366-day maximum, and classic tokens carry no expiry requirement at all until you impose one. Shorten both; we suggest 90 days at most. An expired token is no use to whoever finds it in an old CI log. A member's token with a lifetime longer than the maximum is blocked from the organization rather than revoked, so its holder finds out when their API calls to the organization start failing.

08 Set the GitHub Actions policy

Set this before anyone writes a workflow, at **Settings > Actions > General**. Three of the five controls below carry recommendations from Sections 2 and 3 of the benchmark; the other two carry none and are worth setting anyway.

Which actions can run (3.1.1, verifying third-party artifacts). Under Policies, choose **Allow YOUR-ORG, and select non-YOUR-ORG, actions and reusable workflows**, where GitHub shows your organization's name in place of YOUR-ORG. Then tick **Allow actions created by GitHub** and, if you need them, **Allow Marketplace actions by verified creators**. The

allow list accepts up to 1000 entries with pattern matching, so you can name specific third-party actions and leave the rest of the Marketplace closed.

Require actions to be pinned to a full-length commit SHA (3.1.7). A checkbox sits under the policy options. Tick it and every action, including GitHub's own and your organization's, has to be referenced by a full commit SHA or the workflow fails. Reusable workflows can still be called by tag. The card below explains what pinning protects against and how to find an action's SHA.

Pin actions to a full-length commit SHA

CIS 3.1.7, AND GITHUB'S OWN HARDENING GUIDANCE

A tag is a movable pointer. Whoever controls the action's repository can repoint `v4` at new code, and every workflow that references it runs that code on the next execution with your secrets in scope. A full-length commit SHA is immutable in practice, because subverting it would require generating a SHA-1 collision for a valid Git object. Write `uses: actions/checkout@<40-character-sha>` and keep the version in a trailing comment for humans.

To find the SHA, list the action's tags and read the line for the version you want. Where a tag has a second line ending `^{}` , use that line's SHA: it is the commit the tag points to.

```
git ls-remote --tags https://github.com/actions/checkout
```

Workflow permissions (2.1.7, minimal secret scope). `GITHUB_TOKEN` is the token GitHub issues to each job in a workflow run, and it expires when the job finishes. A new organization already defaults it to read access for the `contents` and `packages` scopes, so confirm the restrictive option is selected and leave it. Individual workflows then request more through the `permissions` key. A job that only reads code should never hold a token that can push to it.

```
permissions:
  contents: read
  pull-requests: read
```

On the same screen, leave **Allow GitHub Actions to create and approve pull requests** cleared. A new organization starts that way. With it set, a workflow's own token can approve a pull request, and GitHub added the control so that a change could not reach a protected branch without a person reviewing it.

Fork pull request approval. On the same page, require approval for workflows from public forks. The options run from first-time contributors new to GitHub, through all first-time contributors, to all external contributors. Pick the strictest one your contribution volume can absorb. This setting does not reach `pull_request_target`. GitHub runs those workflows from the repository's default branch, which it treats as trusted, so they run whatever you select here.

Workflow execution protections. These live under **Settings > Actions > Policies**, a separate section from General. A policy there can hold two rules: an event rule lists the events that may start a workflow, and an actor rule lists who may trigger one. Each rule is an allowlist, so anything it leaves out is blocked. Create a policy that targets all repositories, set it to Active, and give it an event rule that allows the events your workflows use, such as `push` and `pull_request`, with `pull_request_target` left off. Add events as you write workflows that need them: a run started by an event that is not on the list fails with an error naming the event.

GitHub says actor rules do not apply to the built-in processes its own features run on Actions. It does not say whether event rules apply to them. After you activate the policy, open **Policy insights**, directly under Policies in the sidebar, and check that no code scanning, Dependabot, or Pages run shows as blocked. With `pull_request_target` left off the list, GitHub blocks the trigger the warning below describes, so stopping it no longer depends on review.

The protections cover every public repository, and private repositories on Team and Enterprise, so on Free they reach public repositories only. In a public repository with no applicable event policy, GitHub adds a default policy that blocks `pull_request_target`. It runs in evaluate mode for now, so workflows still run. GitHub will enforce it on 2 November 2026 for repositories that were using the default `pull_request_target` policy before the protections became generally

available on 17 September 2026. Choosing Evaluate for a policy you create, to see what it would block before you enforce it, is Enterprise Cloud only.

- **Two workflow patterns to keep out of the repository from the start**

The first pattern is a workflow triggered by `pull_request_target` or `workflow_run`. A `pull_request_target` workflow runs with the base repository's `GITHUB_TOKEN`, its secrets, and the runner, and GitHub's hardening guidance names `workflow_run` as dangerous in the same way. A workflow using either must never check out code from the untrusted pull request. Where you need to act on a fork's contribution, keep the privileged job to reading metadata, and let the untrusted code run only under the unprivileged `pull_request` trigger.

`actions/checkout` now refuses to check out a fork's pull request code from a `pull_request_target` workflow, and from a `workflow_run` workflow whose original event was a pull request, unless the workflow sets `allow-unsafe-pr-checkout: true`. Treat that input as a review finding wherever it appears. Treat a `cache-mode` that grants write access in a `pull_request_target` workflow the same way, since GitHub says it reintroduces the cache-poisoning risk. The refusal arrived in v7.0.0 on 18 June 2026 and was backported to v2 through v6 on 20 July 2026. A SHA pin never moves, so a workflow pinned to an earlier commit keeps the old behavior until you move the pin.

The second is interpolating a context value, such as a pull request title, directly into a shell script. Pass it through an intermediate environment variable, because the raw form lets whoever wrote that title run commands inside your job.

Where a workflow needs cloud credentials, use OpenID Connect (OIDC) to authenticate directly to the provider rather than storing long-lived keys as secrets. That removes the standing credential you would otherwise have to rotate and would eventually find in a log. The workflow still receives a short-lived token, so scope the provider's trust policy to the exact repository, branch, and environment you mean to allow. Trust policies usually match the token's subject claim, the field that says where the workflow ran. For repositories created, renamed, or transferred after 15 July 2026 that subject is immutable: it embeds the owner and repository IDs (`repo:octo-org@123456/octo-repo@456789: ...`), so a recycled name cannot inherit the trust. Write the policy against the subject your workflow presents.

09 Create the first repository

Create it private unless it is meant to be public, then add two files and set the administrators before anyone else arrives.

SECURITY.md (1.2.1)

It tells a finder how to report a vulnerability and which versions you support. The benchmark requires it on public repositories; write it into your private repository anyway, so that a disclosure process exists before the repository becomes public. Once a repository is public, pair it with private vulnerability reporting under the repository's **Settings > Security and quality > Advanced Security**, which gives researchers a private form and is a public-repository feature.

CODEOWNERS (1.1.6, 1.1.7)

The file names the owners of each path, and GitHub requests their review on pull requests that touch it; Step 10 makes that review required. Put it in `.github/`: GitHub looks there before the root and `docs/` and uses the first file it finds, and owning that directory makes the file own itself. Owners must have write access to the repository, and the file must stay under 3 MB. At minimum, own the workflow directory, since anyone who can change a workflow can change what runs with your secrets.

Two repository administrators (1.3.7)

Keep admin access on each repository to the two organization owners, who have it on every repository already, and grant it to nobody else under **Settings > Access > Collaborators & teams**. The reasoning is the same as for the owner count.

```
# .github/CODEOWNERS: one line covers the workflows and this file itself
/.github/                @your-org/platform

# 1.1.6: add a line per sensitive path as the repository grows
/infra/                  @your-org/platform
```

CODEOWNERS follows gitignore-style patterns with three exceptions that catch people out: you cannot escape a leading `#`, negation with `!` does nothing, and character ranges in square brackets do nothing. A directory pattern already covers everything beneath it, so a separate line for `/.github/workflows/` adds nothing unless you want a different owner there. Owners matching the same pattern must all sit on one line. When the ruleset in Step 10 requires code owner review, an approval from any one of the listed owners satisfies it.

- **If you are importing existing code, scan its history before you push**

Old commits can hold credentials that still work. Before the first `git push` of an existing repository, scan every commit for secrets, including those no longer in the current files. The open-source scanner gitleaks does this: run `gitleaks git -v` inside the repository, and by default it reads every commit on every branch. Rotate anything it finds, and treat the old credentials as compromised: rewriting history does not undo a leak once the secret has been cloned.

Where you hold Secret Protection, or the repository is public, also apply the Step 11 configuration before that first push. Push protection then checks the push itself, though only for the most identifiable secret types. A very large push can time out without being blocked, and in a public repository a push larger than 50 MB is not scanned at all. Secret scanning still reads the whole history once it lands.

10 Protect the default branch with a ruleset

Protect the default branch of every repository with a ruleset, a named set of rules GitHub enforces on the branches it targets. This step carries most of recommendations 1.1.3 through 1.1.20, the largest block in the benchmark. Step 9 handled 1.1.6, and the note below the table picks up the ones that are process work.

The benchmark documents the older Settings > Branches path; use rulesets instead. GitHub keeps both, and where both apply to a branch, every rule from either is enforced. Rulesets can be turned off without being deleted, and anyone with read access can read them, where classic protection was visible to admins only. Several can apply at once, with the most restrictive winning, and one ruleset at organization level covers every repository you will ever create.

On GitHub Team, create it once for the whole organization: **Settings > Repository > Rulesets > New ruleset > New branch ruleset**. An organization ruleset needs two targets set: the repositories and the branches. Under **Target repositories** choose **All repositories**, then under **Target branches** add your default branch. Set enforcement to Active, as 1.1.20 asks, then enable the following.

RULE	SETTING	CIS
Require a pull request before merging	On. This is the parent of the next six.	1.1.3
Required approvals	2 once you have three or more engineers, 1 with two, 0 for a solo founder. Anything below 2 is a recorded exception to 1.1.3; see SMALL.	1.1.3
Dismiss stale pull request approvals when new commits are pushed	On. New commits that change the diff clear earlier approvals.	1.1.4
Require review from Code Owners	On. Without it, code owners are only asked to review; this makes their approval required. A solo founder turns it off; see SMALL.	1.1.7

RULE	SETTING	CIS
Restrict who can dismiss pull request reviews	On, naming as few people as the team allows.	1.1.5
Require approval of the most recent reviewable push	On. Someone other than the last person to push has to approve. Newer than the benchmark, with no equivalent recommendation. A solo founder turns it off; see SMALL.	beyond CIS
Require all comments on the pull request to be resolved	On. Open review comments block the merge.	1.1.11
Require status checks to pass	On. A status check is an automated test or scan that reports a result on the pull request. Add each check by name, because the rule blocks nothing until at least one is named. Require branches to be up to date has no effect on its own, so tick it after you have added a check. Come back and add the scanner checks from Step 11.	1.1.9, 1.1.10
Require signed commits	On, which Step 2 already prepared you for.	1.1.12
Require linear history	On. Merges become squash or rebase only, so the repository must allow at least one of those two merge methods first.	1.1.13
Block force pushes	On, and on by default. A force push rewrites the branch's history.	1.1.16
Restrict deletions	On, and on by default.	1.1.17
Restrict updates	Off. With the bypass list empty it makes the branch read-only; see the card below.	1.1.15
Require code scanning results	Not in this ruleset: add it, with a severity threshold, to a ruleset on each repository where code scanning runs. In an organization ruleset it blocks the merge in every repository where code scanning is not set up, and GitHub documents the organization-level rule only for Enterprise Cloud.	1.1.18
Require secret scanning alerts are resolved	On where you hold Secret Protection. Blocks the merge while the head commit's scan has not finished, or while a commit in the pull request has an open alert for a secret type the rule selects. Public preview.	beyond CIS

Two newer settings change what the approval count means, and both are in public preview. First, a ruleset requires one approval more than you configured on a pull request that Copilot opens under its own identity rather than for a person. That is on by default and has no effect where the count is 0, so the solo-founder setting is untouched, while a two-person team set to 1 will see 2 on such a pull request. Second, Copilot code review can submit an approval that counts toward the required-approvals rule. That one is off by default; leave it off wherever you rely on the approval count as a control.

Leave the bypass list empty

CIS 1.1.14, WRITTEN AGAINST CLASSIC BRANCH PROTECTION

Classic branch protection exempts administrators by default, which is the behavior 1.1.14 is written against. Rulesets start with an empty bypass list instead, so the work here is keeping it empty: anyone the list covers can bypass every rule in the ruleset. If you must grant a bypass, grant it to a named app or role with a reason recorded, review it on a schedule, and treat every entry as an exception you owe someone an explanation for.

• Leave Restrict updates off

Restrict updates lets only bypass-list members push to the branch. With the bypass list empty that is nobody, so it blocks every push, which reaches the merge button as well, and the branch goes read-only. Rulesets have no separate list of permitted pushers; the bypass list is that list. CIS 1.1.15 and 1.1.14 pull against each other here and the empty bypass list is the stronger control, so record 1.1.15 as an exception with that reason.

• Signed commits and linear history limit the web merge buttons

Linear history removes the ordinary merge commit, which leaves two merge buttons, and GitHub documents a restriction on each:

- **Squash and merge** checks the commits the pull request introduces, head branch included, so an unsigned commit there can block the merge even though GitHub would sign the squashed result. GitHub says this can block the pull request's own author too.
- **Rebase and merge** adds the head commits without signature verification: GitHub rewrites each commit and has no key to sign it with, so expect the rule to block that button as well.

Where a merge is blocked, rewrite and sign the commits on the head branch, then push it again:

```
# On the head branch, which must track its copy on GitHub. Replace main if your
# default branch has another name. --gpg-sign signs with the SSH key from Step 2.
git pull --ff-only
git rebase --force-rebase --gpg-sign origin/main
git push --force-with-lease --force-if-includes origin HEAD
```

If the pull stops, the branch on GitHub and your copy each have commits the other lacks; reconcile them first. Together, `--force-with-lease` and `--force-if-includes` make the push refuse, rather than overwrite, commits on the branch that are missing from your copy.

GitHub's other documented answer, merging locally and pushing the signed result, does not work here: the pull request rule blocks a direct push to the default branch, and the bypass list is empty. Where a merge has to happen another way, use the break-glass route in HELP and record it. Decide this before you enable the rules, write the rewrite-and-sign route into the contributing notes, and expect no on-screen explanation when a button greys out.

Recommendation 1.1.19 asks that changes to protection rules be audited. Rulesets keep their own version history, readable per ruleset through the API, and the organization audit log records every change to them under the `repository_ruleset` category. The older `protected_branch` category covers classic branch protection, so query both if any classic rules are still in place. Recommendations 1.1.1, 1.1.2, and 1.1.8 are process work: track work in a task system, link commits to tickets, and review stale branches. Turning on **Automatically delete head branches** in repository settings handles most of 1.1.8 without anyone remembering to; the rest sits in the REVIEW table.

11 Apply a security configuration

Switch on GitHub's scanners for every repository at once with a security configuration. A configuration is a named bundle of security settings that applies across the whole organization in one action, where the benchmark's own steps

go repository by repository. Recommendations 1.5.1 through 1.5.6 ask for six kinds of scanner, and GitHub provides five of them.

RECOMMENDATION	FEATURE	COST ON A PRIVATE REPO
1.5.1 Sensitive data in code	Secret scanning, plus push protection to block the commit before it lands	GitHub Secret Protection
1.5.4 Code vulnerabilities	Code scanning with CodeQL	GitHub Code Security
1.5.5 Vulnerable dependencies	Dependabot alerts and security updates	Free on all plans
1.5.6 License problems	Dependency review	GitHub Code Security
1.5.2 CI pipeline instructions	CodeQL analyzes workflow files themselves for script injection, missing permissions, and unvalidated inputs. Code scanning default setup switches this on when it finds workflow files on the default branch, so applying the configuration covers it. A repository on advanced setup, where you maintain the CodeQL workflow yourself, has to add the actions language to that workflow by hand.	GitHub Code Security
1.5.3 Infrastructure as Code	No native feature. Add a scanner to the pipeline and require it as a status check in Step 10.	Depends on the tool

- **Applying this to private repositories costs money**

The Code Security and Secret Protection features in the table are free on public repositories. On private and internal repositories, applying them consumes licenses or incurs usage costs, as the PLAN table shows. Decide the budget before you click Apply configuration.

Go to the organization's **Settings > Advanced Security > Configurations** and create a configuration that switches on the features in the table. In the configuration's **Policy** section, set **Use as default for newly created repositories** to **All repositories**, so every repository you make from here inherits it without a decision. Set **Enforce configuration** to **Enforce**, so repository admins cannot change the features it sets. Then apply it to every repository from the **Repositories** tab: select them, then **Apply configuration**.

The Configurations screen is there on Free and Team as well as Enterprise Cloud, though a Free organization cannot buy the metered features it switches on. A repository transferred into the organization does not pick up the default, so apply the configuration to that one by hand.

Push protection is the part worth paying for on a private repository. Secret scanning tells you a credential leaked and you then have to rotate it; push protection stops the commit from ever reaching the server.

- **Decide who answers the alerts before you switch them on**

Scanners produce findings, and findings that nobody owns become a backlog an assessor will read as evidence you are not managing them. For each of Dependabot, code scanning, and secret scanning, write these four lines somewhere durable, put them in the owning policy, and revisit them the first time the volume surprises you:

- Who triages, by name, with a named deputy.
- How fast, by severity, counting from when the alert appears. A leaked live credential is handled the same day, and the first action is rotating it.
- What closing an alert requires, and who may accept a risk instead of fixing it.
- Where the record lives, so you can show a year later that the process ran.

12 Verify with the API

Read the settings back instead of trusting the screens you just clicked through. Authenticate as an owner, because the full organization detail is returned only to owners. The ruleset and Actions endpoints also need the `admin:org` scope, which `gh auth login` does not request; without it they answer 403 or 404, which reads like a failed configuration. Add the scope before you start and remove it when you finish, because it lets `gh` fully manage the organization.

```
# Grant the scope the ruleset and Actions endpoints need
gh auth refresh -h github.com -s admin:org

# Organization policy: Steps 4 and 5
gh api /orgs/YOUR-ORG --jq '{
  two_factor_required:      .two_factor_requirement_enabled,
  base_permission:          .default_repository_permission,
  members_can_create_repos: .members_can_create_repositories,
  members_can_delete_repos: .members_can_delete_repositories,
  members_can_delete_issues: .members_can_delete_issues,
  members_can_create_teams: .members_can_create_teams,
  members_can_fork_private: .members_can_fork_private_repositories,
  members_can_change_visibility: .members_can_change_repo_visibility,
  members_can_publish_pages: .members_can_create_public_pages
}'
```

The result you want is `two_factor_required: true`, `base_permission` of `read` or `none`, and every `members_can_` field false.

```
# Owner count: expect exactly two (one page lists up to 30 members)
gh api '/orgs/YOUR-ORG/members?role=admin' --jq 'length'

# Every rule in force on the default branch, with its settings, from all levels
gh api /repos/YOUR-ORG/YOUR-REPO/rules/branches/main --jq '[.[] | {type, parameters}]'
```

```
# Rulesets and their bypass lists: expect bypass_actors to be empty
gh api /orgs/YOUR-ORG/rulesets --jq '.[] | {id, name, enforcement}'
# Run this for each id the line above prints
gh api /orgs/YOUR-ORG/rulesets/RULESET-ID --jq '{name, enforcement, bypass_actors}'

# Step 11: the configuration, its enforcement, and the default for new repositories
gh api /orgs/YOUR-ORG/code-security/configurations --jq '.[] | {id, name, enforcement}'
gh api /orgs/YOUR-ORG/code-security/configurations/defaults
```

```
# Actions policy: Step 8
gh api /orgs/YOUR-ORG/actions/permissions --jq '{allowed_actions, sha_pinning_required}'
gh api /orgs/YOUR-ORG/actions/permissions/selected-actions
gh api /orgs/YOUR-ORG/actions/permissions/workflow
gh api /orgs/YOUR-ORG/actions/permissions/fork-pr-contributor-approval
```

```
# Workflow execution protections: Step 8
gh api /orgs/YOUR-ORG/actions/policies --jq '.policies[] | {id, name, enforcement}'
# Run this for each id the line above prints
gh api /orgs/YOUR-ORG/actions/policies/POLICY-ID --jq '{name, enforcement, rules}'

# When you finish, remove the scope again
gh auth refresh -h github.com -r admin:org
```

From the Actions endpoints, expect `allowed_actions` of `selected`, `sha_pinning_required` `true`, `github_owned_allowed` `true`, `default_workflow_permissions` of `read`, `can_approve_pull_request_reviews` `false`, the `approval_policy` you chose, and a policy with `enforcement` of `active` whose `restrict_action_events` rule does not list `pull_request_target`. From the configurations endpoint, expect `enforcement` of `enforced`.

Ask for `parameters` as well as `type` on the branch endpoint, or you prove that a rule exists without proving it is the rule you set: the required approval count, the code-owner flag, and the severity threshold all live in there. `bypass_actors` is only returned to someone with write access to the ruleset, which is the other reason to run this as an owner.

The branch endpoint is the useful one, because it returns the active rules on that branch from every ruleset that targets it, whichever level the ruleset was created at. Save all the output in one text file with the date in its name, such as `readback-2026-10-01.txt`, and keep it in your evidence store: an assessor can check a dated file from the day the organization was built.

Two settings have no readback a user token can call: the token policy from Step 7, and the GitHub App installation restriction from Step 5, which the organization endpoint does not return. Keep a dated screenshot of those screens alongside the file. Re-run the set each quarter, save it the same way, and compare the two files with `diff`.

Export the audit log every quarter

180 DAYS OF HISTORY, AND NO API BELOW ENTERPRISE

Every detective control in this guide reads the organization audit log, and the log holds 180 days for ordinary events. The audit log API is an Enterprise Cloud feature, so on Free or Team the only way to keep the record is the **Export** dropdown on the log page, which writes JSON or CSV. Only owners can open the log at all, and it shows the past three months unless you search with a `created` date range, so the oldest three months are invisible until you ask for them.

Each quarter, search for the quarter just ended, such as `created:2026-07-01..2026-09-30`, and export that into the same evidence store as the readback. An export stops at 100 MB compressed or ten minutes of processing, so split the range if an export reaches either limit. Otherwise, anything older than 180 days is gone by the time an audit asks for it.

• Git events are kept for seven days and never shown in the web log

Pushes, clones, and fetches (`git.push`, `git.clone`, `git.fetch`) carry their own access and retention rules. GitHub keeps them for seven days rather than 180 and never shows them in the web interface. It lists three routes to them, the REST API, audit log streaming, and a JSON or CSV export, and its incident-investigation guidance says Free and Team plans cannot view Git events at all. Do not design a control that depends on reading push history later: on Free or Team you cannot read it, and a quarterly export could not cover a seven-day window anyway.

NEXT When you release software: sign the artifacts

• Attestations need a public repository below Enterprise

Artifact attestations work in public repositories on every current plan. In private and internal repositories they need GitHub Enterprise Cloud. Step 9 has you creating the repository private, so on Free or Team this section is a public-repository feature: 2.4.1, 2.4.5, 2.4.6, and 4.1.1 stay open, and the substitute is signing releases with your own key held as a repository secret. Record the gap with the plan-tier items in EXCEPT.

Once you publish releases, have the workflow that builds each one sign it with a GitHub artifact attestation. Attestations need no key management, because the signing identity comes from the workflow itself.

Sections 2 and 4 of the benchmark ask for signed release artifacts (2.4.1), artifacts signed by the pipeline that built them (4.1.1), and a Software Bill of Materials (SBOM) produced and signed on every pipeline run (2.4.5, 2.4.6). An SBOM lists the components a release contains, with their versions. Attestations cover three of the four: they sign the artifact (2.4.1),

sign the SBOM you hand them (2.4.6), and bind both to the pipeline that built them (4.1.1). They do not produce the SBOM, so 2.4.5 stays open until an earlier step in the pipeline generates one.

```
permissions:
  id-token: write
  contents: read
  attestations: write
  artifact-metadata: write

steps:
  - name: Attest build provenance
    uses: actions/attest@1e69f48acb82d1966a394da916b4c1698aa569d6 # v4.2.2
    with:
      subject-path: 'path/to/artifact'

  - name: Attest the SBOM
    uses: actions/attest@1e69f48acb82d1966a394da916b4c1698aa569d6 # v4.2.2
    with:
      subject-path: 'path/to/artifact'
      sbom-path: 'path/to/sbom'
```

`id-token` lets the action mint the OIDC token it exchanges for a signing certificate, and `attestations` lets it persist the result. `artifact-metadata` is what the action's README lists for creating an artifact storage record, which the action writes only when `push-to-registry` is true, the setting that pushes the attestation to the image registry. Naming a `permissions` key drops every permission you leave out, so keep `contents: read` for the checkout step.

There are two steps because the action attests one thing at a time. With only `subject-path` it signs build provenance, a record of the workflow, repository, and commit that built the artifact. With `sbom-path` it signs the SBOM you supply instead. Both steps are pinned the way Step 8 asks for, with the version in a trailing comment. Resolve the SHA yourself against the release you intend to run; this one was current when the guide was written.

```
# Anyone downstream can verify provenance
gh attestation verify path/to/artifact -R YOUR-ORG/YOUR-REPO

# The SBOM attestation is separate: name its predicate to check it
gh attestation verify path/to/artifact -R YOUR-ORG/YOUR-REPO \
  --predicate-type https://spdx.dev/Document/v2.3
```

That predicate type is the SPDX one, and the action takes the version from the document's own `spdxVersion` field, so name the version your SBOM carries. The action accepts SPDX 2.x JSON and CycloneDX; a CycloneDX SBOM carries its own type, `https://cyclonedx.org/bom`. Container images use `subject-name` and `subject-digest` instead of a path, and add `packages: write` to the permissions above. Verify them with an `oci://` reference. Leave the tag off the subject name and use the digest.

• Make the verify step a gate before anything deploys

An attestation protects you only when something checks it, so make `gh attestation verify` a gate that has to pass before anything deploys or gets promoted, and fail the step when it does not. Write down what you require of an artifact before it ships: whose repository built it, which workflow, and which branch. Where you fall back to signing releases with your own key in a repository secret, any workflow able to read that secret can sign anything. Keep it out of pull-request-triggered workflows, put it behind a protected environment with a required reviewer, and write down who rotates it and when.

Teams of one or two

Two recommendations assume more people than a founding team has. Write down what you did about each.

1.1.3 asks for two approvals

A solo founder cannot approve their own pull request. Set required approvals to 0, then turn off the two rules that demand a second human whatever that number says: **Require review from Code Owners** and **Require approval of the most recent reviewable push**. Both are enforced independently of the approval count, so leaving them on with 0 approvals stops your own pull requests from merging at all.

Everything else in the Step 10 table stays on, so changes still flow through a pull request, still run status checks, still require signed commits, and still cannot be force-pushed. You keep the audit trail and the automated gates, and lose only the human review that does not exist. Raise approvals to 1 the day the second engineer starts, and switch those two rules back on with them. Go to 2 once there are three engineers, the point where an author plus two reviewers becomes possible and the exception closes.

1.3.7 asks for two repository administrators

The constraint is the same and the fix is different. The organization must still have two owners for recovery, so the second owner is your second administrator even if they never write code. Choose someone who can judge what they are approving and has a reason to hold the access, such as a co-founder or a trusted contractor with an owner seat and a hardware key. A second owner chosen only to make the count, such as a bookkeeper, holds full control of the organization without being able to review what they approve.

Write the exceptions down on the day you make them

WHAT AN ASSESSOR READS AS A DECISION

An assessor treats an undocumented deviation as a finding, and a dated entry naming the recommendation, the reason, the compensating controls, the accepted risk, and a review date as a decision. Record the team-size exceptions and the two rules you switched off in Step 10 in an exception register, with an end date you revisit whenever you hire, and log the Enterprise-only gaps below as accepted risks in a risk register. Our exception register and risk register templates on the resources page cover these fields between them. Give these settings an owning policy while you are at it, so there is a document above them saying why they are set this way.

Troubleshooting and offboarding

• Plan the break-glass path before you need it

At some point a ruleset with an empty bypass list will block an urgent production fix, so decide the route now. For a team of one or two, add the organization owner role to the Step 10 ruleset's bypass list for the window, and remove it in the same sitting. Leave its mode at Always allow, so the fix can be pushed directly. While the entry stands, owners can bypass the ruleset and nobody else can. The add and the remove both land in the audit log under `repository_ruleset`.

Disabling the ruleset would also unblock the fix, but it lifts the protection from every repository's default branch, for everyone, until you switch it back on. Once you have an on-call rotation that can review it monthly, move to a named break-glass team kept on the bypass list. If you need the route often, change the ruleset so routine work does not need it. Write the route you chose next to the exception list, and agree its rules in advance:

- Before: tell the other owner.
- During: set a time limit on the window and close it in the same sitting.
- After: write down the reason and the time, run the Step 12 readback to prove the ruleset came back as it was, and review the episode at the next check.

- **Required signed commits also applies to bots**

The rule covers contributors and bots alike, so automation that pushes commits has to sign them too. Enable it, then check that your dependency updates, release automation, and any Actions workflow that commits still work. Where something cannot sign, a narrow recorded bypass for that one app beats turning the rule off.

- **Restricting personal access tokens breaks existing automation**

Anything already authenticating with a kind of token you restrict stops working the moment the policy lands. On a new organization there is nothing to break, so it belongs in the first sitting.

- **Offboarding checklist**

Write this list down before you need it. Removing someone from People ends their authorization. Their own tokens and SSH keys stop reaching your private code with it, because a token never carries more than the account behind it. You control the organization-side credentials, so on the day they leave:

- Revoke the fine-grained tokens you approved for them.
- Remove the deploy keys (SSH keys that grant access to a single repository) and app installations they set up.
- Take them out of any ruleset bypass list and CODEOWNERS line.
- Rotate every secret they could read, since a copied secret leaves with the person.
- If they were an owner, promote the replacement before you remove them, so the organization is never down to a single owner, and move the billing contact if it pointed at them or at a mailbox they could read.
- Export the audit log for their last months while it is still there.

You do not control their personal account. Their own SSH, signing, and recovery credentials stay theirs and are not yours to remove.

REVIEW

Recurring checks

Below Enterprise Cloud these checks are manual work, so put each of the nine on a calendar and name its owner before you leave this page. They cover the Section 1 recommendations that ask for periodic review rather than a one-time setting, plus a check for configuration drift, so none of them can be finished in the first sitting. Where the benchmark or its mapped CIS Critical Security Controls Safeguard names an interval, the table gives it; the rest are our suggested cadence.

WHAT TO REVIEW	WHERE	HOW OFTEN	CIS
Stale branches	Repository > Branches > Stale	Quarterly. GitHub's Stale view lists branches with no commits in three months.	1.1.8
Changes to protection rules	The ruleset's history through the API, and the audit log <code>repository_ruleset</code> category	Quarterly, and after any incident	1.1.19
Repositories that changed visibility or left	Organization audit log: <code>repo.access</code> , <code>repo.transfer_outgoing</code> , and <code>repo.destroy</code>	Monthly. After Step 5 only an owner can do any of these; this catches one who did.	1.2.6
Forks of your repositories	Repository > Insights > Forks	Quarterly, and whenever someone leaves	1.2.5

WHAT TO REVIEW	WHERE	HOW OFTEN	CIS
Inactive repositories	Organization > Repositories, then archive	Quarterly. The benchmark's threshold is no activity in three to six months.	1.2.7
Inactive members	Organization > People, cross-checked against recent commits, reviews, and issue activity	Monthly, so nobody sits dormant much past the 45 days the mapped CIS Critical Security Controls v8.1 Safeguard 5.3 allows. GitHub's dormant-users report exists only for enterprise accounts, so below Enterprise this review is the activity cross-check itself.	1.3.1
Installed applications still in use	Settings > Third-party Access	Twice a year, checking permissions as well as need	1.4.2, 1.4.3
Unusual code activity	Repository Insights > Network and the commit history, plus the force-push and ruleset events in the audit log	Monthly. GitHub says Free and Team plans cannot view Git push events at all, so this is a read of the code history rather than a log query.	1.3.13
Configuration drift	The Step 12 readback, diffed against the setup-day file, and an audit log search for <code>org.update_member</code>	Quarterly. The commands are already written; the control is comparing the output.	beyond CIS

Some events are worth an alert as they happen, even in a two-person team, because each one removes a protection or widens who has access. An organization webhook can deliver four of them:

- The `repository` event with the `publicized` or `deleted` action.
- The `repository_ruleset` event.
- The `organization` event when someone joins.
- The `secret_scanning_alert` event for an alert whose `push_protection_bypassed` field is true, which the audit log records as `secret_scanning_push_protection.bypass`.

Add the webhook at the organization's **Settings > Webhooks > Add webhook**, choose **Let me select individual events**, and tick those four events. The webhook delivers every action of each event it subscribes to, so have the receiver keep only the ones above. A webhook is an HTTPS POST, so it needs a receiver: a chat platform's own GitHub integration, or a small endpoint you run that formats the payload and forwards it to the role mailbox. Pointing a webhook straight at a mailbox does nothing.

A repository that leaves by transfer sends nothing, because GitHub delivers the `transferred` action only to the account receiving it, so `repo.transfer_outgoing` stays in the monthly check. An owner promotion has no webhook at all. It appears in the audit log as `org.update_member`, so search for that event in the quarterly drift check.

EXCEPT What you cannot do below Enterprise

These four recommendations have no path on Free or Team. Log them as accepted risks with the plan tier as the reason, and revisit if you ever upgrade.

- **1.3.1** automated identification of inactive users, which relies on the Enterprise dormant-users report. The manual substitute is in the REVIEW table above.
- **1.3.10** restricting email notifications to verified domains, which is an Enterprise Cloud setting even once the domain is verified.
- **1.3.11** organization-provided SSH certificates, which needs an SSH certificate authority.
- **1.3.12** IP allow lists for Git access.

Recommendation 1.3.9, the verified badge, is a Step 5 task rather than an exception. Recommendation 1.3.13, tracking anomalous code behavior, has no setting on any tier, so it sits in the REVIEW table as a habit. Recommendation 1.1.15 is the decision Step 10 records, with the empty bypass list as its compensating control. Add the artifact attestation items from the NEXT section if your repositories are private.

LIMITS

What this setup does not protect against

The settings above leave these nine risks open.

- **A compromised owner account or live session.** Two factors make the account harder to take over and do not remove the risk. That is the reason for two owners and an empty bypass list.
- **A malicious or careless insider with legitimate write access.** Review catches mistakes and slows deliberate abuse; it stops neither on its own. The audit log is what you will investigate with.
- **Secrets that leaked before push protection was on.** Push protection stops the next one; Step 9 covers scanning the history and rotating what it finds.
- **Push protection bypasses.** By default a developer can push past a block by picking one of three reasons, and the reason is self-declared, so alert on those events. Where you hold Secret Protection, delegated bypass turns this into a request: name a small group as reviewers, and everyone else except organization owners and security managers then has to ask one of them before the push lands. A request expires after seven days, so name enough reviewers that one is always reachable. That gets the approval in before the push lands, where the alert only reaches you afterwards, so configure it where you can.
- **Self-hosted runners.** Out of scope here. If you add them, treat the runner as production infrastructure, because a workflow is arbitrary code execution on it.
- **Offboarding.** Removing someone does not reach the clones already on their machine or the secrets they already read; the checklist in HELP covers what you can do.
- **Machine identities.** Everything here governs people. Bot accounts, deploy keys, app installations, and the tokens your pipelines use answer to nobody unless you give each one a named human owner, a reason to exist, and a place on the same review calendar. Inventory them the day you create the first one.
- **Losing GitHub itself.** Every control here assumes you can reach the account. An outage, a billing lapse, or a suspended organization takes every repository and record with it, and a locked-out sole owner does the same. Keep GitHub on your vendor register with the rest of your suppliers.
- **What a clone does not keep.** Keep a copy of the repositories somewhere you control, with the same care applied to whatever mirrors them. A clone preserves the code and its history and nothing else. Issues, pull requests and their review history, repository settings, rulesets, Actions history, releases, packages, security alerts, and the audit log all live only on GitHub. Decide how much of that you would need after an outage or a lost organization, how current it has to be, and test a restore once rather than assuming it.

NOTES

How the guide maps to the benchmark

Where the benchmark names GitHub outside Section 1

Eight recommendations in Sections 2 to 5 carry GitHub steps or documentation links: 2.1.1, 3.1.1, 4.1.1, 4.2.5, 4.2.6, 4.3.1, 4.3.2, and 4.3.3. The rest of those sections are written in product-neutral language.

Where Steps 1 and 2 come from

Steps 1 and 2 set up your personal account. The benchmark has no recommendation for that, so both steps follow GitHub's documentation for the password, the two-factor methods, the recovery codes, and the signing key. The closest recommendations are 1.3.4 and 1.3.5, which require two-factor authentication for the whole organization. Step 4 sets them. Step 2 cites 1.1.12 because Step 10 makes it an enforced rule. The benchmark settings start at Step 3.

Three of the benchmark's product claims are out of date

v1.2.0 states that secret scanning is enterprise only (1.5.1) and that only GitHub Enterprise can run code scanning on private repositories (1.5.4). Since 1 April 2025 both are sold to GitHub Team organizations as add-ons. Its 1.3.9 verifies a domain only from an enterprise account, while the organization-level setting Step 5 uses is documented for every plan.

Level 1, Level 2, and where this guide differs

Every recommendation carries a profile. Section 1 holds 31 at Level 1 and 19 at Level 2, and Level 1 is the minimum defensible set if you are working to a compliance date. CIS sets the profiles by impact on utility rather than by how much risk each item removes, so requiring 2FA (1.3.4, 1.3.5) and secret scanning (1.5.1) are both Level 2 while reviewing stale branches (1.1.8) is Level 1. The list of five in SCOPE is ranked by risk instead, so its first item is the pair of Level 2 recommendations that require 2FA.

DONE What you achieved

- An organization where every member carries a strong second factor, where the base permission grants read at most and write nowhere, and where repository creation, deletion, visibility changes, issue deletion, team creation, and app installation stay with two named owners.
- A default branch that requires reviewed, signed, linear, status-checked pull requests, with no bypass for administrators and one organization-wide rule in place of a per-repository copy.
- GitHub's own scanners applied across every repository and set as the default for repositories that do not exist yet, with the license cost understood before any charge began. Infrastructure-as-code scanning (1.5.3) still needs a tool you supply.
- Third-party applications and OAuth authorizations gated on owner approval, apart from GitHub's own privileged apps, and personal access tokens of both kinds blocked from your private code, with owner approval waiting for fine-grained tokens if you later allow them.
- A dated API readback of the organization policy, the Actions policy, the owner count, the rules in force on the default branch and their settings, and the security configuration, taken on the day the organization was built. Beside it, dated screenshots of the settings the API cannot return, and a written exception list covering the team-size and plan-tier gaps.

Benchmark coverage after these steps

WHAT TO TELL AN ASSESSOR

All 31 Level 1 recommendations in Section 1 are addressed: 21 as settings you made today, and 10 as process or the recurring checks in REVIEW (1.1.1, 1.1.2, 1.1.8, 1.1.19, 1.2.5, 1.2.7, 1.3.13, 1.4.2, 1.4.3, 1.4.4). Four of the 21 are conditional, so record each in the register with its condition beside it:

- 1.1.3, if you reduced the approval count for team size.
- 1.1.9 and 1.1.10, until a status check exists to require.
- 1.1.18, until code scanning is running and paid for on a private repository.

At Level 2, four have no path below Enterprise Cloud (1.3.1, 1.3.10, 1.3.11, 1.3.12), 1.1.15 is a recorded decision, and 1.5.3 needs a scanner you supply, which leaves 13. Of those, 1.3.6 is process, and 1.3.7 is a per-repository access grant rather than an organization setting. Recommendations 1.5.1, 1.5.2, 1.5.4, and 1.5.6 are free on public repositories and metered on private ones, so on Team your coverage of them follows your budget. Write those numbers and their conditions into the exception and risk registers on the day you finish.

REF References

CIS GitHub Benchmark v1.2.0, 18 February 2026 · cisecurity.org/cis-benchmarks

CIS Critical Security Controls v8.1, Safeguard 5.3 · cisecurity.org/controls. Refer to cisecurity.org/controls for the current version of the CIS Controls.

GitHub security features and plan availability · docs.github.com/code-security/getting-started/github-security-features

Security configurations for an organization, including enforcement · docs.github.com/code-security/securing-your-organization

Secret scanning, push protection, and delegated bypass · docs.github.com/code-security/concepts/secret-security

Push protection limits: the secret types it blocks, and large pushes · docs.github.com/code-security/reference/secret-security/secret-scanning-scope

Code scanning with CodeQL, including Actions workflow analysis · docs.github.com/code-security/code-scanning

Rulesets, their bypass lists, available rules, code scanning merge protection, and merge methods including rebase and merge signing · docs.github.com/repositories/configuring-branches-and-merges-in-your-repository

Requiring two-factor authentication; reviewing and exporting the organization audit log, its event list, and the Git event retention rules · docs.github.com/organizations/keeping-your-organization-secure

Inviting members, and cancelling a pending invitation · docs.github.com/organizations/managing-membership-in-your-organization

Roles in an organization; setting base permissions · docs.github.com/organizations/managing-peoples-access-to-your-organization-with-roles

Personal access token policy for your organization · docs.github.com/organizations/managing-programmatic-access-to-your-organization

Actions policy, the allow list, and SHA pinning; repository visibility changes; GitHub App installations; renaming; domain verification · docs.github.com/organizations/managing-organization-settings

Security hardening for GitHub Actions; artifact attestations · docs.github.com/actions/security-for-github-actions

Actions policies: workflow execution protections, their event and actor rules, and the default `pull_request_target` policy · docs.github.com/actions/concepts/about-actions-policies

REST endpoints for Actions policies · docs.github.com/rest/actions/policies

Privileged OAuth apps and GitHub Apps, the GitHub CLI among them · docs.github.com/apps

Mandatory two-factor authentication and who it reaches, recovery methods, account recovery, and commit signature verification · docs.github.com/authentication

Creating webhooks; webhook events and payloads; validating webhook deliveries · docs.github.com/webhooks

Investigation tools: the Git events Free and Team plans cannot view · docs.github.com/code-security/reference/security-incident-response/investigation-tools

GitHub Terms of Service: one free account per person · docs.github.com/site-policy/github-terms/github-terms-of-service

gitleaks: scanning every commit in a repository for secrets · github.com/gitleaks/gitleaks

git rebase: `--force-rebase` and `--gpg-sign` · git-scm.com/docs/git-rebase

Security improvements for SSH: the 3072-bit RSA minimum from 14 October 2026, 22 September 2026 · github.blog/changelog

GitHub Advanced Security for Team organizations, 1 April 2025 · github.blog/changelog