anoBytes | TECHNOLOGY SOLUTIONS

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 2026-08-07.

12 steps, account to verification

one sitting, before the first repository exists

50 benchmark recommendations

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

SCOPE

What this guide covers, and what it does not

The CIS GitHub Benchmark v1.2.0 holds 120 recommendations across five sections. Section 1, Source Code, holds 50 of them and is where almost all the GitHub-specific settings live. Sections 2 to 5 cover build pipelines, dependencies, artifacts, and deployment mostly in product-neutral language, with eight exceptions that do 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. This guide implements the Section 1 settings you can put in place while the organization is empty, plus the Actions and attestation controls from Sections 2, 3, and 4 that apply before you have written any code.

Where Steps 1 and 2 come from

The benchmark is scoped to organizations and repositories and says almost nothing about the individual account. Password strength, passkeys, recovery-code custody, and signing-key setup come from GitHub's documentation. The CIS citations start at Step 3, so every framework reference in this guide is one you can look up.

Three of the benchmark's product claims are out of date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. The benchmark's branch-protection steps also predate rulesets, which is the mechanism Step 10 uses.

An empty organization is the easy case

The benchmark is written as an audit-and-remediate exercise against a live organization. Applying the same settings to an empty organization costs one sitting and creates no exceptions to unwind later.

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. The profiles track implementation burden more than risk, though: 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 hour-long list below is ranked by risk instead, so it opens with two Level 2 items on purpose.

PLAN What each plan buys you

Decide this first, because it determines how much of the benchmark you can satisfy. The short version: 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 + ADVANCED SECURITY
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 restrictions, token policy	1.4.1, 1.4.3	Yes	Yes	Yes
Rulesets on public repositories	1.1.3-1.1.20	Yes	Yes	Yes
Rulesets on private repositories	1.1.3-1.1.20	No	Yes	Yes
Organization-wide rulesets	1.1.3-1.1.20	No	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
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. Both became available to Team organizations on 1 April 2025 and are priced per active committer per month. A committer counts as active when one of their commits has been pushed to such a repository in the last 90 days, whenever it was originally written. The license count follows those committers across every repository with the feature switched on, so the scope you apply the Step 11 configuration to is what sets the bill. 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.

If you only get an hour, do these five

RANKED BY RISK REDUCED PER MINUTE SPENT

Require 2FA for everyone (Step 4). Set base permissions to Read or No permission (Step 5). Restrict access by personal access tokens (Step 7). Put a ruleset on the default branch with required review and no bypass (Step 10). Turn on push protection (Step 11). These five remove the paths that get small organizations breached: a stolen password, a member who could always push everywhere, a leaked token carrying its owner's full rights, an unreviewed commit, and a credential in a public commit.

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. GitHub states that Support cannot restore access to an account whose owner has lost their 2FA credentials, and the documented fallbacks are slow and conditional, so treat a second registered method as the real backup.
- A **second person** lined up as the other organization owner. GitHub's guidance is no fewer than two, and Step 3 explains what to do when there is genuinely 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 create 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.

Turn on two-factor authentication immediately, from **your profile photo > Settings > Password and authentication > Enable two-factor authentication**. GitHub has required 2FA of everyone who contributes code on github.com since March 2023, so this is already required of you. 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 seed in a password manager that syncs, so a lost phone does not cost you the second factor.

SMS

The weakest option and the one GitHub explicitly steers you away from. Treat it as a last resort in regions where nothing else works.

- **TOTP first, then the stronger method**
  GitHub's 2FA setup still starts with a TOTP app or SMS: a security key can only be registered once one of those is configured, and the documented passkey-as-2FA flow lists the same prerequisite, so you cannot begin at the top of that list. A passkey added on its own from the passkey settings does sign you in, but complete the 2FA setup as well; the enrollment is what issues the recovery codes below. Set up TOTP, add the passkey or key straight after, and keep the TOTP entry as your second method.

Download the recovery codes now

THIS IS THE STEP PEOPLE SKIP

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. The one documented fallback is an account recovery request proved with a previously verified device, SSH key, or personal access token, which Support reviews within three business days and does not guarantee; configuring a fallback SMS number is no longer supported. GitHub's instruction on the codes themselves is plain: do not share or distribute them, and 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. That is what the floor of two is for.

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

Recommendation 1.1.12 asks that every commit in a pull request be signed and verified before merging, and Step 10 turns that into an enforced rule. Set the signing up now, on your own machine, so the rule never blocks you. GitHub supports GPG, SSH, and S/MIME signatures. SSH is the simplest to generate and can reuse a key you already have.

```
# 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 in the **Key type** dropdown choose **Signing Key**. 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 gpg.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 reads as ordinary. With it, anything you did not sign is labeled Unverified, which is what makes the signal worth anything.

03 Create the organization

Create it from **your profile photo > Your 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 is survivable but leaky: 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 those matter once Advanced Security is metered per 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.

Two owners is the number

RECONCILING CIS 1.3.3 WITH GITHUB'S GUIDANCE

The benchmark pushes the owner count down and GitHub puts the floor at two. Two is the answer that satisfies both at this size: one owner is a single point of failure for account recovery and offboarding, and at a founding team each owner past the second widens the blast radius of a compromised account while the resilience is already there. 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 they 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

Recommendations 1.3.4 and 1.3.5 both ask for this, one for outside contributors and one for members. One screen covers both: **Settings > Security > Authentication security**, then select **Require two-factor authentication for everyone in your organization** and save. 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, which Step 1 ranked last for the same reason. 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** without 2FA are removed from the organization outright and lose repository access. 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

Seven benchmark recommendations live on this one screen. Go to **Settings > Access > Member privileges**.

SETTING	SET IT TO	CIS	WHY
Base permissions	`Read, or No permission`	`1.3.8`	The level every member gets on every repository automatically. The dropdown offers No permission, Read, Write, and Admin, and Read is GitHub's default, so a new organization already passes. Confirm it anyway. Anything at Write or above hands the whole organization push access to every repository, whatever their actual role. 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 tidy 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. Stopping the change beats spotting it afterwards, and the REVIEW table below still tracks it as the benchmark asks. 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. Clearing it closes that route, which is why it is worth doing rather than relying on the review alone.
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 hole in the approval gate Step 6 describes. 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 open team creation is a quiet route to privilege escalation.
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.

Recommendation 1.3.6 asks that new members be invited using a company-approved email address. On Team there is no setting that enforces this, so it is a process control: invite by company address, and check the Invitations tab under People before each invite goes out. Verifying a domain does not close this gap either. It earns the verified badge, which is recommendation 1.3.9 and is available to a Team organization, so do it as its own task at **Settings > Security > Verified and approved domains**, where you add the domain and publish the TXT record it gives you; it does not gate who can be invited. Restricting email notifications to verified domains (1.3.10) is a separate setting and needs Enterprise Cloud. Enterprise Managed Users is the tier where identity is provisioned rather than invited.

06 Restrict third-party application access

Recommendation 1.4.1 wants administrator approval before any application is installed. GitHub splits this across two application types that behave differently.

GitHub Apps: compliant once you 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 exists out of the box. The exception is the repository admin who can install an app for a repository they administer whenever it asks for neither organization permissions nor repository administration permissions. Step 5 turns that off. What is left is 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: restricted by default, so keep it that way

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 application policy**: an organization where the restriction was ever lifted stays open until an owner restores it. Be aware of the documented limit: even with restrictions on, users can still authorize privileged OAuth apps and use them to access data from the organization, so this reduces exposure rather than eliminating it.

Recommendation 1.4.4 asks that webhooks use HTTPS. There is nothing to pre-configure in an empty organization, so make it a standing rule: every webhook payload URL starts `https://` with SSL verification enabled, at both organization and repository level. HTTPS satisfies the benchmark and leaves the endpoint open to anyone who guesses the URL, so set a secret on every webhook as well, and have the receiver compute an HMAC-SHA256 over the raw body and compare it against the `X-Hub-Signature-256` header before acting on the payload. Reject anything that fails. Check both whenever you add one.

07 Set a personal access token policy

The benchmark does not cover this, and it is the largest gap in Section 1. A leaked classic token carries the full access of the person who issued it, and 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. Whoever holds the token holds that person's reach into your private code. Configure it at **Settings > Personal access tokens > Settings**.

Restrict access via personal access tokens

Set this twice. The screen has a **Fine-grained tokens** tab and a **Tokens (classic)** tab, and the restriction applies only to the family whose tab you are on, so setting one and stopping leaves the other family reaching your private code. Choose the restrictive option on both. That is the right starting posture 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

Available for fine-grained tokens only, and already the default, so confirm it and move on. An owner reviews each request, which gives you a record of what automation exists and what it can touch. 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; classic tokens carry no expiry requirement at all until you impose one. Shorten both. An expired token is no use to whoever finds it in an old CI log. Setting the policy blocks non-compliant tokens from reaching the organization without revoking them, so the owner learns about it when their API calls start failing.

08 Set the GitHub Actions policy

Set this before anyone writes a workflow, at **Settings > Actions > General**. Four controls carry recommendations from Sections 1, 2, and 3 of the benchmark.

Which actions can run (3.1.1, verifying third-party artifacts). Under Policies choose **Allow OWNER, and select non-OWNER, actions and reusable workflows**, 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. This is the enforcement behind the card below.

Workflow permissions (2.1.7, minimal secret scope). A new organization already defaults `GITHUB_TOKEN` 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
```

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.

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.

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

  `pull_request_target` runs with repository write permissions and access to secrets. GitHub's hardening guidance names it alongside `workflow_run`: both are dangerous in the same way, so 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. Separately, never interpolate 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 is a script injection waiting to happen.

Where a workflow needs cloud credentials, use OpenID Connect 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. Repositories created after July 15, 2026 present an immutable subject that 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 actually 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)

The benchmark requires this on public repositories (1.2.1). Write it into your private repository anyway, because the day it goes public is the wrong day to be drafting a disclosure process. It tells a finder how to report a vulnerability and which versions you support. Once a repository is public, pair it with private vulnerability reporting under **Settings > Security > Advanced Security**, which gives researchers a private form and is a public-repository feature.

`CODEOWNERS` (1.1.6, 1.1.7)

Put it in `.github/`, which is also the most defensible location because the file can then 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)

Same logic as organization owners. Set it under **Settings > Access > Collaborators and teams**.

```
# .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 the history before you push**
  Push protection from Step 11 blocks the next secret; it does nothing about the ones already in the commits you are about to import. An empty organization is the easy case only while it stays empty. Before the first `git push` of an existing repository, run a history scan, rotate anything it finds, and treat the old credentials as burned. Rewriting history does not un-leak a secret that has already been cloned.

10 Protect the default branch with a ruleset

This single 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: they can be turned off without being deleted, they are readable by anyone with read access where classic protection was 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, and missing the first is the usual reason a ruleset appears

to do nothing. Under **Target repositories** choose **All repositories**, then under **Target branches** add your default branch. Set enforcement to Active and 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 from three engineers up, since an author plus two reviewers is possible at three. 1 at two, 0 for a solo founder. Anything below 2 is a recorded exception to 1.1.3; see SMALL TEAMS.	1.1.3
Dismiss stale pull request approvals	On. New commits clear earlier approvals.	1.1.4
Require review from Code Owners	On, which is what makes Step 9's file do anything.	1.1.7
Restrict who can dismiss pull request reviews	On, naming as few people as the team allows.	1.1.5
Require an approval from someone other than the last person to push	On. Newer than the benchmark, with no equivalent recommendation.	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, then add each check by name. The rule gates nothing until at least one check is named, and 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.	1.1.16
Restrict deletions	On, and on by default.	1.1.17
Restrict updates	Off. It cancels out the empty bypass list; see the card below.	1.1.15
Require code scanning results	On once Step 11 is done, with a severity threshold.	1.1.18

Leave the bypass list empty

CIS 1.1.14, THE ONE MOST ORGANIZATIONS GET WRONG

Classic branch protection lets administrators straight past 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, and administrator accounts are the ones worth stealing. 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.

- **Restrict updates is the one rule to leave 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 and every merge, including yours, 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 together close both web merge buttons**

  GitHub documents two restrictions, and combining these rules leaves a reviewer no web route to merge someone else's pull request. Squash and merge on the web is blocked unless you are the pull request's author. Rebase and merge is worse: GitHub rewrites the commits and cannot sign them on your behalf, so the result carries no signature verification and the required-signature rule rejects it. Linear history removes the ordinary merge commit, so those two are all that is left. GitHub's documented answer is to rebase and merge locally and push the result, which works in every case. Decide this before you enable the rules, write the local 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

Recommendations 1.5.1 through 1.5.6 ask for six kinds of scanner. The benchmark has you enabling them repository by repository. GitHub now ships security configurations, which are named bundles of these settings that apply across an organization in one action.

Go to **Settings > Security > Advanced Security > Configurations** and create a configuration that switches on the features in the table below. Apply it to every repository from the **Repositories** tab (select them, then **Apply configuration**). Set it as the default for all newly created repositories too, so every repository you make from here inherits it without a decision.

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 analyses 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 above covers it.	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 starts the meter**

  The configuration you just built includes Code Security and Secret Protection features. On public repositories they are free. On private and internal repositories, applying them consumes licenses or incurs usage costs. Decide the budget before you click Apply.

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. Write four lines somewhere durable and put them in the owning policy. Who triages, by name, with a named deputy. How fast, by severity, counting from when the alert appears: a leaked live credential is 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. Set the same four for Dependabot, code scanning, and secret scanning, and revisit them the first time the volume surprises you.

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 organization endpoint also needs the `admin:org` scope, which `gh auth login` does not request. Add it once, or the first command below returns `null` for every field and reads like a failed configuration.

```
# Grant the scope the organization endpoint needs
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
}'
```

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
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}'
gh api /orgs/YOUR-ORG/rulesets/RULESET-ID --jq '{name, enforcement, bypass_actors}'

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

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 the output into your evidence store with the date in the filename. A dated pass file from the day the organization was built is solid evidence for an assessor. The token policy from Step 7 has no readback a user token can call, so keep a dated screenshot of that one screen alongside the file. Re-run the set each quarter and diff it against the setup-day file.

Export the audit log before it ages out

`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 and caps each run at 100 MB or ten minutes. Two more limits to plan around: only owners can open the log at all, and the view shows the past three months unless you widen it with a `created` date range, so the oldest three months are invisible until you ask for them. Export quarterly into the same evidence store as the readback. An annual audit asking about March will not find it in November.

- **Git events are the exception, and they are not in the log you can see**
  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, reaches them only through the REST API, audit log streaming, or an export, and the API is Enterprise Cloud only. They never appear in the web interface on any plan. A quarterly export cannot preserve a seven-day window, so treat push history as something you do not retain below Enterprise, and do not design a control that depends on reading it later.

NEXT When you start shipping: sign the artifacts

Sections 2 and 4 of the benchmark ask for signed release artifacts (2.4.1), a Software Bill of Materials produced and signed on every pipeline run (2.4.5, 2.4.6), and artifacts signed by the pipeline that built them (4.1.1). GitHub artifact attestations cover three of the four, and they need no key management because the signing identity comes from the workflow itself. 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.

- **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.

```yaml
permissions:
  id-token: write
  contents: read
  attestations: 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'
```

Two steps because the action attests one thing at a time: with only `subject-path` it signs build provenance, and with `sbom-path` it signs the SBOM you supply instead. Both 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.

```bash
# 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
```

Container images use `subject-name` and `subject-digest` instead of a path, and add `packages: write`. Verify them with an `oci://` reference. Leave the tag off the subject name and use the digest.

- **Provenance only counts where something checks it**
  Generating provenance changes nothing on its own. The security comes from the check at the other end, 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, remember that any workflow able to read that secret can sign anything, so 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.

SMALL Teams of one or two

Two recommendations assume more people than a founding team has. Handle them honestly, and write down what you did.

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 from someone other than the last pusher**. 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, switch those two rules back on with them, and go to 2 once there are three of you, which is the point where an author plus two reviewers becomes possible and the exception closes.

1.3.7 asks for two repository administrators

Same constraint, different fix. The organization must still have two owners for recovery, so the second owner is your second administrator even if they never write code. A co-founder, or a trusted contractor with an owner seat and a hardware key, is a real control. Pick someone who can actually judge what they are approving and who has a reason to hold the access; handing an owner seat to a bookkeeper to satisfy a count gives you the number without the control, and an assessor will say so.

Write the exceptions down on the day you make them

THE DIFFERENCE BETWEEN A GAP AND A FINDING

An undocumented deviation is a finding. A dated entry naming the recommendation, the reason, the compensating controls, the accepted risk, and a review date is a decision, and an assessor will treat it as one. Record the team-size exceptions here alongside the Enterprise-only items below and the two rules you switched off in Step 10, then review the list whenever you hire. The exception register template on our resources page already carries those fields. 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.

HELP What breaks, and the way back in

- **Plan the break-glass path before you need it**

  A ruleset with an empty bypass list will, one day, stand between you and a production fix at two in the morning. Decide now which it is: a named break-glass team added to the bypass list and reviewed monthly, or an owner who temporarily sets the ruleset to Disabled and records why. For two people, take the second: the disable and the re-enable both land in the audit log under `repository_ruleset`, and there is no standing bypass to forget about. Disabling leaves the branch unprotected for as long as it lasts, so agree the rules in advance and hold to them: the other owner is told before it happens rather than after, the window is time-boxed and closed in the same sitting, the reason and the time are written down, and the Step 12 readback runs afterwards to prove the ruleset came back as it was. Review the episode at the next check, since a break-glass that keeps happening means the design needs changing. Move to the named team once you have an on-call rotation that can review it monthly. Both options are defensible, so long as you pick one before you need it. Write the choice down next to the exception list.

- **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 genuinely cannot sign, a narrow recorded bypass for that one app beats turning the rule off.

- **Base permission of No permission hides repositories**
  Members lose the implicit access they had, and repositories they never explicitly joined disappear from view. Grant access per team before you tighten this, or expect a quiet morning of access requests. Read is the gentler starting point and still satisfies 1.3.8.

- **Restricting personal access tokens breaks existing automation**
  Anything already authenticating with a classic token stops working the moment the policy lands. On a new organization there is nothing to break, which is why this belongs in the first sitting.

- **Requiring 2FA removes outside collaborators**
  Covered in Step 4, and worth repeating because it is the one change that removes people rather than blocking them. You have three months to reinstate them.

- **Offboarding takes more than the People page**
  This is the most likely incident in the whole document, so write the list down before you need it. Removing someone from People ends their authorization, and their own tokens and SSH keys stop reaching your private code with it, because a token never carries more than the account behind it. Know what you can and cannot reach. You control organization-side credentials: revoke the fine-grained tokens you approved for them, remove deploy keys and app installations they set up, take them out of any ruleset bypass list and CODEOWNERS line, and rotate every secret they could read, since a copied secret leaves with the person. You do not control their personal account, so their own SSH, signing, and recovery credentials stay theirs and are not yours to remove or take custody of. When the leaver is an owner, promote the replacement before you remove the old one so the organization is never down to a single owner, and move the billing contact if it pointed anywhere near them. Export the audit log for their last months while it is still there.

REVIEW The checks that need a recurring look

The nine checks below cover the Section 1 recommendations that ask for periodic review rather than a one-time setting, plus the drift check on everything you just configured, so none of them can be finished in the first sitting. Put them on a calendar now, while you remember why they matter. Where the benchmark or its mapped CIS Control names an interval, it is given below; the others are our suggested cadence for a small team.

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	Ruleset history, and the audit log `repository_ruleset` category	Quarterly, and after any incident	1.1.19
Repositories that changed visibility	Organization audit log, `repo` qualifier, `access` category	Monthly. Step 5 already stops members doing it; this catches an owner or repository admin who did.	1.2.6

WHAT TO REVIEW	WHERE	HOW OFTEN	CIS
Forks of your repositories	Repository > Insights > Forks	Quarterly, and whenever someone leaves	`1.2.5`
Inactive repositories	Organization > Repositories, then archive	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 Control v8 5.3 allows. The People page shows no last-active date below Enterprise, so the activity check is the real work here.	`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. Git push events themselves are Enterprise Cloud only, kept seven days, and never shown in the web log, so below Enterprise 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	Quarterly. The commands are already written; the control is comparing the output.	beyond CIS

These are calendar work below Enterprise, so name the person who owns each one before you leave this page. Three of them can be turned into alerts. An organization webhook delivers events as they happen, and three are worth an interrupt even in a two-person shop, because each one either removes a control or widens who holds it: the `repository` event with the `publicized` action, the `repository_ruleset` event, and the `organization` event when someone joins. 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. Two more events have no webhook and stay manual: an owner promotion appears in the audit log as `org.update_member`, and a push protection bypass as `secret_scanning_push_protection.bypass`. Put those two in the quarterly query along with everything else.

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 deliberately not on this list: domain verification is open to a Team organization, so treat it as a task rather than an exception. Two more items sit outside the plan question. Recommendation 1.3.13, tracking anomalous code behavior, has no setting on any tier, which is why it sits in the REVIEW table as a habit. Recommendation 1.1.15 has a setting you should leave off, for the reason given in Step 10, so it goes in the register as a decision 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 eight risks open.

- **A compromised owner account or live session.** Two factors raise the bar and do not remove the risk. This is why the owner count stays at two and why the bypass list stays empty.
- **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. Anything already in history needs finding and rotating, and rewriting history does not un-leak it.
- **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 has to ask one of them for approval before the push lands. Approval in advance beats a notification afterwards, so configure it rather than settling for the alert.
- **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.** Setting the organization up does not remove anyone. Removal ends a leaver's authorization, and it does not reach the secrets they already read, the clones already on their machine, or the credentials on their personal account. The checklist is in HELP above.
- **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, and 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 a bad day, how current it has to be, and test a restore once rather than assuming it.

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 the meter started. Infrastructure-as-code scanning (1.5.3) still needs a tool you supply.
- Third-party applications, OAuth authorizations, and personal access tokens gated on owner approval, closing the routes that bypass the settings above.
- A dated API readback of the organization 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, with a dated screenshot covering the token policy the API cannot return, and a written exception list covering the team-size and plan-tier gaps.

Where that leaves you against the benchmark

THE LINE TO PUT IN FRONT OF 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 and belong in the register with their condition written next to them: 1.1.3 if you reduced the approval count for team size, 1.1.9 and 1.1.10 until a status check actually exists to require, and 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 and 1.3.7 are process rather than settings, and 1.5.1 and 1.5.4 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 register on the day you finish. An assessor can only credit a configuration you can describe.

REF References

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

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

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

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

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

Rulesets, available rules, 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

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

Two-factor authentication recovery methods, account recovery, and commit signature verification · docs.github.com/authentication

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

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