CHECKLIST

Secure Git Branching Strategies for Protected Releases Checklist

A practical checklist for evaluating and implementing protected release branching strategies, merge policies, CI gates, and hotfix controls before production use.

Secure Git Branching Strategies for Protected Releases

Checklist

Use this checklist to verify that your Git branching model supports safe, auditable releases.

---

1) Branch model and release flow

- [ ] The team has a clearly defined branch model for feature work, integration, release candidates, and production releases.
- [ ] Release branches are created from a known stable point in time.
- [ ] Short-lived feature branches are used for work in progress.
- [ ] A dedicated integration or release-candidate branch exists if the team needs a stabilization stage.
- [ ] The release branch is treated as the source of truth for what was shipped.
- [ ] The team can explain how code moves from feature branch to production without ambiguity.

2) Branch protection rules

- [ ] Direct pushes to protected release branches are disabled.
- [ ] Changes to protected branches require pull requests or an equivalent review workflow.
- [ ] Required reviewers are defined and consistently enforced.
- [ ] Administrative bypass is limited, monitored, and justified by policy.
- [ ] Protected branches have clear rules for who can merge and under what conditions.
- [ ] Branch protection settings are documented and reviewed periodically.

3) Review and approval controls

- [ ] Code review is mandatory before merge.
- [ ] Review requirements match the risk of the release path.
- [ ] High-risk changes require additional approval or separation of duties where appropriate.
- [ ] Reviewers have enough context to evaluate release impact, not just code style.
- [ ] Merge approvals are recorded and traceable.
- [ ] The team has a defined process for urgent approvals without weakening controls.

4) CI and validation gates

- [ ] CI runs against the exact commit set proposed for release.
- [ ] Build, test, and packaging checks must pass before merge or deployment.
- [ ] Security checks are included where relevant to the environment.
- [ ] Failed checks block release progression.
- [ ] CI results are visible to reviewers and release managers.
- [ ] The pipeline validates the same artifact or commit that will be promoted.

5) Merge policy and history management

- [ ] The team has chosen a merge strategy intentionally, not by default.
- [ ] The merge strategy preserves an audit trail suitable for release tracking.
- [ ] Developers understand when to merge and when to rebase.
- [ ] Shared branches are not rewritten in ways that confuse release history.
- [ ] The chosen strategy supports incident investigation and rollback.
- [ ] History cleanup does not override traceability requirements.

6) Hotfix and exception handling

- [ ] There is a documented hotfix path for urgent production issues.
- [ ] Hotfixes follow a controlled branch and review process.
- [ ] Emergency changes still produce traceable evidence of approval and validation.
- [ ] Hotfixes are limited in scope to the issue being addressed.
- [ ] Unapproved direct production changes are prohibited.
- [ ] Post-hotfix reconciliation is required so fixes are merged back into the normal development flow.

7) Release tagging and traceability

- [ ] Each production release is tagged or otherwise uniquely identified.
- [ ] Tags or release identifiers map to the exact code that was deployed.
- [ ] The release record includes approvals, pipeline results, and artifact references.
- [ ] It is possible to reconstruct the approved release content later.
- [ ] The team can identify which changes were included in a given release.
- [ ] Rollback paths are based on known release points, not guesswork.

8) Access control and separation of duties

- [ ] Access to protected branches is limited to authorized roles.
- [ ] Permissions align with operational responsibilities.
- [ ] No single person can casually bypass the review and validation process.
- [ ] Privileged accounts are controlled and monitored.
- [ ] Access reviews are performed on a recurring basis.
- [ ] Temporary elevated access is time-bound and documented.

9) Release readiness and production verification

- [ ] The release candidate has been validated in an environment that matches production expectations.
- [ ] Release criteria are defined before deployment begins.
- [ ] The team verifies that the candidate matches what was approved.
- [ ] Release managers can confirm whether the branch content is safe to promote.
- [ ] Production deployment depends on passing evidence, not informal confirmation.
- [ ] Monitoring or rollback readiness is in place before promotion.

10) Operational fit and exception pressure testing

- [ ] The policy is strict enough to protect production but not so rigid that teams create workarounds.
- [ ] The release process can handle urgent fixes without becoming ad hoc.
- [ ] Developers and release managers know what to do when checks fail.
- [ ] The team has tested the process for a real or simulated hotfix scenario.
- [ ] The model supports both auditability and reasonable delivery speed.

- [ ] The team can explain why the current strategy fits its risk profile.

Quick decision check

Use a protected release branch model if you need:

- [ ] Strong auditability
- [ ] Reproducible release content
- [ ] Clear approval boundaries
- [ ] Controlled hotfix handling
- [ ] Reliable rollback and incident traceability

If two or more of these are missing, revisit the branching strategy before relying on it for production.

Final review questions

- [ ] Can you explain exactly how a commit reaches production?
- [ ] Can you prove what code was released?
- [ ] Can you handle an urgent patch without bypassing control?
- [ ] Can you reconstruct approvals and CI results later?
- [ ] Would an auditor or incident responder trust the branch history?

If the answer to any of these is no, the protected release model needs more work before production use.