Smoother Git CLI Workflows with 2FA Enabled

GitHub and GitLab for Annoyed Developers

1 Introduction

Once you enable two-factor authentication (2FA) on GitHub or GitLab, the old "type username + password in the terminal and forget about it" model stops working for HTTPS Git operations.

- Your normal account password is no longer accepted for `git push` over HTTPS.
- You may get repeated prompts for credentials, or generic "authentication failed" errors.
- The web flow for 2FA (codes, apps, SMS, etc.) does not map directly to the command line.

The good news: once you do a small one-time setup, you can get back to *smooth* `git push` / `git pull` with 2FA safely enabled.

This document focuses on HTTPS-based workflows with:

- Personal Access Tokens (PATs) as password replacements.
- SSH keys as a more "permanent" solution.

We cover both GitHub and GitLab, plus a short troubleshooting section.

2 GitHub

2.1 Personal Access Tokens

A GitHub Personal Access Token (PAT) is a long, random string that acts like a password for specific operations. When 2FA is enabled, GitHub will reject your normal password over HTTPS; you must use a PAT (or SSH key) instead.

Creating a PAT on GitHub

1. Log in to GitHub in your browser.
2. Click your avatar (top-right) → `Settings`.
3. In the left sidebar, go to:
   - `Developer settings` → `Personal access tokens`.
4. Choose the PAT type.
5. Click `Generate new token`.
6. Give it a meaningful name (e.g., "CLI Git operations on laptop").
7. Set an expiration (shorter is safer; e.g. 30 or 90 days).

8. Select scopes:
    - For basic repo push/pull on your own repos, you typically need:
        - Classic token: `repo` scope.
        - Fine-grained token: grant access only to the repositories and operations you actually need (e.g. read/write code).
9. Click `Generate token`.
10. **Copy the token now** and store it somewhere safe; you will not be able to see it again.

Using a PAT at the Command Line

1. Ensure your repository remote is HTTPS:
    - Run `git remote -v`.
    - You should see something like: `https://github.com/user/repo.git`.
2. Run an operation that needs auth, for example:
    - `git pull`
    - or `git push`
3. When Git prompts:
    - `Username for 'https://github.com'`:
      Enter your GitHub username.
    - `Password for 'https://github.com'`:
      Paste the PAT instead of your real password.

If you do nothing else, you will need to paste the PAT every time the credential cache expires. To really remove the annoyance, configure a credential helper.

Storing the PAT with a Credential Helper

Simple file-based helper This stores credentials in a plain-text file `~/.git-credentials` (chmod-protected, but not encrypted).

1. Configure globally:
    - `git config --global credential.helper store`
2. Run a Git command that prompts for credentials, enter username + PAT once.
3. Git writes the credentials to `~/.git-credentials`.
4. Next time, Git uses them automatically; no prompt.

OS-specific helpers (recommended on desktops) If available, use a more secure helper that integrates with the OS keychain:

- On macOS:
    - `git config --global credential.helper osxkeychain`
- On Windows (Git Credential Manager):
    - `git config --global credential.helper manager-core`

- On many Linux setups (Gnome keyring, libsecret, etc.), see Git docs for the appropriate helper.

Security Notes for PATs

- Use the minimum scopes needed.
- Use an expiration date and renew tokens instead of "no expiry".
- Revoke tokens immediately if your machine or token is compromised.
- Treat `~/.git-credentials` as sensitive; it must not be world-readable.
- Never commit PATs to a repository or share them.

2.2 SSH Keys

SSH keys give you a key-pair based login that GitHub accepts without prompting for 2FA on every operation. 2FA is enforced when you log into the web UI, but SSH key usage is tied to possession of the private key.

Generate an SSH Key Pair

Run this in your terminal:

1. `ssh-keygen -t ed25519 -C "your_email@example.com"`
2. When prompted for a file path, press Enter to use the default (typically `~/.ssh/id_ed25519`).
3. When asked for a passphrase:
    - Choose a non-empty passphrase for better security.

This creates:

- Private key: `~/.ssh/id_ed25519`
- Public key: `~/.ssh/id_ed25519.pub`

Add the Public Key to GitHub

1. Copy the public key:
    - `cat ~/.ssh/id_ed25519.pub`
    - Copy the full line starting with `ssh-ed25519`.
2. In GitHub's web UI:
    (a) Go to `Settings`.
    (b) In the left sidebar, click `SSH and GPG keys`.
    (c) Click `New SSH key`.
    (d) Give it a clear title (e.g. "Laptop SSH key").
    (e) Paste the contents of `id_ed25519.pub`.
    (f) Click `Add SSH key`.

Test the Connection

1. Run: `ssh -T git@github.com`
2. On first connect, accept the host key (type `yes`).
3. If successful, GitHub responds with a greeting message.

Switch an Existing Repo from HTTPS to SSH

In your local repo:

1. Check current remote:
   - `git remote -v`
2. Change the origin URL to SSH:
   - `git remote set-url origin git@github.com:user/repo.git`
3. Verify:
   - `git remote -v`

Now `git push` and `git pull` will use SSH, not HTTPS.

Using an SSH Agent (Avoid Re-typing Passphrase)

To avoid typing your SSH key passphrase over and over:

1. Start an agent (if not already running), e.g. on many systems:
   - `eval "$(ssh-agent -s)"`
2. Add your key:
   - `ssh-add ~/.ssh/id_ed25519`
3. Enter the passphrase once per session; the agent caches it.

On macOS and some Linux desktops, your environment can automatically load keys into an agent on login.

3 GitLab

GitLab behaves similarly: 2FA breaks direct password usage for HTTPS Git operations. Use a Personal Access Token or SSH keys instead.

3.1 Personal Access Tokens on GitLab

Creating a PAT on GitLab

1. Log in to GitLab in your browser.
2. Click your avatar (top-right) → `Edit profile` (or `Preferences`, depending on version).
3. In the left sidebar, go to `Access Tokens` (sometimes under `User Settings` → `Access Tokens`).
4. Enter a name (e.g. "CLI HTTPS token").
5. Set an expiration date.
6. Select scopes you need for push/pull:

- Typically `api` or `read_repository` + `write_repository` depending on your GitLab instance version and scope options.

7. Click `Create personal access token`.
8. **Copy the generated token** and store it securely (you will not see it again).

Using the PAT over HTTPS

1. Ensure the repo remote is HTTPS:
   - `git remote -v` → e.g. `https://gitlab.com/user/repo.git`
2. Run `git pull` or `git push`.
3. When prompted:
   - For username: enter your GitLab username (or email if required).
   - For password: paste the PAT.

Avoid Repeated Prompts with Credential Helpers

Use the same Git credential helpers as for GitHub:

- Plain text store:
  - `git config --global credential.helper store`
- macOS keychain:
  - `git config --global credential.helper osxkeychain`
- Windows credential manager:
  - `git config --global credential.helper manager-core`

Enter the PAT once; subsequent HTTPS operations should be silent.

3.2 SSH Keys on GitLab

If you already generated an SSH key for GitHub, you can reuse the same public key on GitLab.

Add an Existing SSH Key

1. Copy your public key:
   - `cat ~/.ssh/id_ed25519.pub`
2. In GitLab:
   (a) Go to `Preferences` or `Edit profile`.
   (b) In the left sidebar, click `SSH Keys`.
   (c) Paste the public key.
   (d) Optionally set an expiration date for the key.
   (e) Click `Add key`.

Test the Connection

1. Run: `ssh -T git@gitlab.com`
2. Accept the host key on first connect.
3. You should see a GitLab welcome or identification message.

Switch a Repo Remote to SSH

1. Check the current remote:
   - `git remote -v`
2. Change origin URL to SSH:
   - `git remote set-url origin git@gitlab.com:user/repo.git`
3. Verify again with `git remote -v`.

From now on, `git push` / `git pull` will use SSH and will not prompt for 2FA codes.

4 Best Practices and Troubleshooting

PATs vs. SSH Keys

- **PATs**:
  - Easy drop-in replacement if you are already using HTTPS.
  - Good when you work on many machines occasionally.
  - Use with credential helpers for less friction.
- **SSH keys**:
  - Great for long-term, frequent use from the same machines.
  - Once configured, very low friction; works across hosts (GitHub, GitLab, etc.).
  - Use passphrases + SSH agent for a sane security/comfort balance.

In practice: for a primary development machine, SSH keys are usually the least annoying. For quick setups or shared machines, PATs via HTTPS might be simpler.

Typical Errors and Checks

"Authentication failed" with HTTPS

- Make sure you are using a PAT, *not* your account password.
- Verify that the PAT has not expired.
- Check that the PAT has the required scopes (e.g. `repo` or read/write repo scopes).
- Confirm the remote URL matches the correct host (GitHub vs GitLab).

Wrong Remote URL

- Run `git remote -v` and ensure:
  - HTTPS: `https://github.com/user/repo.git` or `https://gitlab.com/user/repo.git`
  - SSH: `git@github.com:user/repo.git` or `git@gitlab.com:user/repo.git`

- Fix with `git remote set-url origin <correct-url>`.

SSH Still Asks for Password or Fails

- Ensure the SSH key is added to the platform (GitHub/GitLab settings).
- Check file permissions:
  - Private key should be readable only by you (e.g. `chmod 600 ~/.ssh/id_ed25519`).
- Verify that the right key is used:
  - Use `ssh -vT git@github.com` or `ssh -vT git@gitlab.com` to see which key is tried.
- Ensure SSH agent is running and holds your key if you set a passphrase.

Security Recommendations (Short and Non-Negotiable)

- Never commit PATs, private keys, or `~/.git-credentials` to any repo.
- Use limited scopes and expirations for PATs; revoke them when no longer needed.
- Protect private keys with a passphrase and correct filesystem permissions.
- If a laptop or token is lost/stolen, immediately:
  - Revoke PATs from the web UI.
  - Remove or rotate SSH keys.

With PATs or SSH keys and a credential helper (or SSH agent) in place, you can keep 2FA on, stop fighting the prompts, and get back to `git push` and `git pull` behaving like they should.