OIDC for HashiCorp Vault Access in GitHub Actions Workflows

Security White Paper

August 18, 2023

Prepared for:
Ari Kalfus and Tim Lisko
DigitalOcean

Prepared by:
Kelly Kaoudis and Will Brattain
Trail of Bits
kelly.kaoudis@trailofbits.com, will.brattain@trailofbits.com

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Analysis Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and mutually agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As such, this document should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Table of Contents

Introduction

Earlier in 2023, DigitalOcean published blog posts (Part 1, Part 2) discussing their use of OpenID Connect (OIDC) for sourcing secrets from HashiCorp Vault in GitHub Actions–based continuous integration and deployment (CI/CD) workflows. DigitalOcean then engaged Trail of Bits to assess the risks and benefits of the use of these technologies together for secrets management in CI workflows, using DigitalOcean's own OIDC deployment as a case study for other businesses considering implementing similar CI integrations.

Trail of Bits' threat modeling assessments are intended to provide a detailed analysis of the risks facing an application or system at a structural and operational level, assessing the security of its architecture as opposed to its implementation details. During these assessments, engineers rely heavily on frequent meetings with the client's developers, paired with extensive readings of any and all documentation the client can make available. Code review and dynamic testing are not an integral part of threat modeling assessments. However, engineers may occasionally consult the codebase or a live instance to verify specific assumptions about the system's design.

This white paper was produced following our April 27–May 12, 2023 assessment, during which we identified security findings related to DigitalOcean's OIDC deployment that we did not include here. In this white paper, we cover general security-adjacent guidance on sourcing secrets from HashiCorp Vault in GitHub Actions CI/CD workflows via OIDC.

We provide an overview of GitHub-to-Vault OIDC integrations, enumerate the components of DigitalOcean's own OIDC integration (in a more generalized fashion, tailored to this white paper, than we would in a typical threat model), group these components into trust zones, and enumerate potential connections between these zones. We detail threat actor profiles within the system and potential scenarios in which these actors could impact system confidentiality, integrity, or availability. Finally, we cover a non-exhaustive set of tradeoffs, considerations, and best practice recommendations.

We hope this work leads to further assessments and discussion of OIDC usage in CI/CD pipelines from the varied perspectives of system administrators, security researchers, and developers.

OIDC Overview

A security team removing hard-coded secrets from the company codebase should require developers to instead store their sensitive data in a secrets management system like HashiCorp Vault. Rather than including keys and passwords directly in configurations and code, developers reference those secrets' locations in Vault. At runtime, the infrastructure swaps in the secret values. This eliminates hard-coding and the potential for direct secrets interaction in many places, but a second-order problem arises: minimizing the risks of Vault access in CI.

CI servers like GitHub Actions runners often build, test, deploy, and sign artifacts across many different repositories. Granting a self-hosted runner access to an entire Vault would probably enable it to successfully complete all workflows that reference stored secrets, but would also present a tempting target for an internal adversary or a compromised employee looking to gain secrets access. Using OIDC to delegate short-term access to just a few secrets in Vault at a time eliminates any need to allow runners broad Vault access.

To bootstrap OIDC, some prerequisite trust relationships must be in place between system components. How exactly these relationships should be established depends on the risks that the security team in question is comfortable accepting or can mitigate. The Vault deployment must be configured to trust an OIDC identity provider like GitHub's. Setting up this relationship can have some potential pitfalls depending on how CI runners are hosted and establish trust with GitHub, and on which repositories or workflows each runner can access. The security team must also either create and support an extensive set of Vault access policies, or educate and enable developers to safely maintain their own policies.

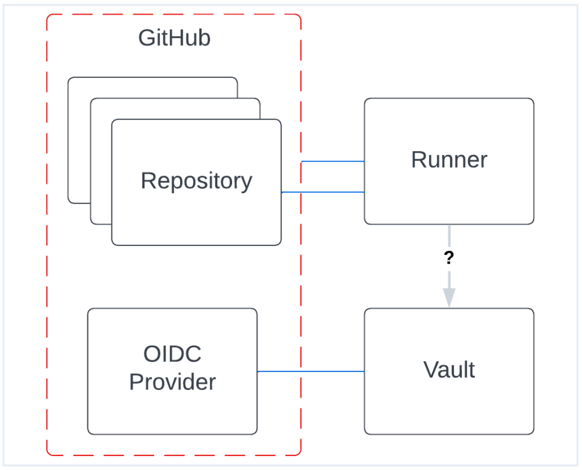

Figure 1: Prerequisite trust relationships needed for a self-hosted GitHub runner to retrieve secrets from Vault via OIDC on behalf of the workflow it is currently executing

The runner in figure 1 (above) is outside the dashed-line GitHub infrastructure trust boundary since it is self-hosted, though it has been registered following the process shown in figure 2 (below). A self-hosted runner must register with GitHub (or, if applicable, the local GitHub Enterprise deployment instead) using the GitHub API in order to run Actions workflows. We would consider *GitHub*-hosted runners to be within the GitHub infrastructure trust boundary.

Storing the GitHub API key in Vault itself could result in a kind of chicken-and-egg situation (see steps 1 and 2 below in figure 2), where the infrastructure retrieves the GitHub API key from Vault to register a newly created runner that eventually will *itself* be granted Vault access. If Vault configuration does not protect the API key from other accesses, it could be preferable to store such a "secret zero" that bootstraps further access in a separately secured location like a distinct secrets manager.

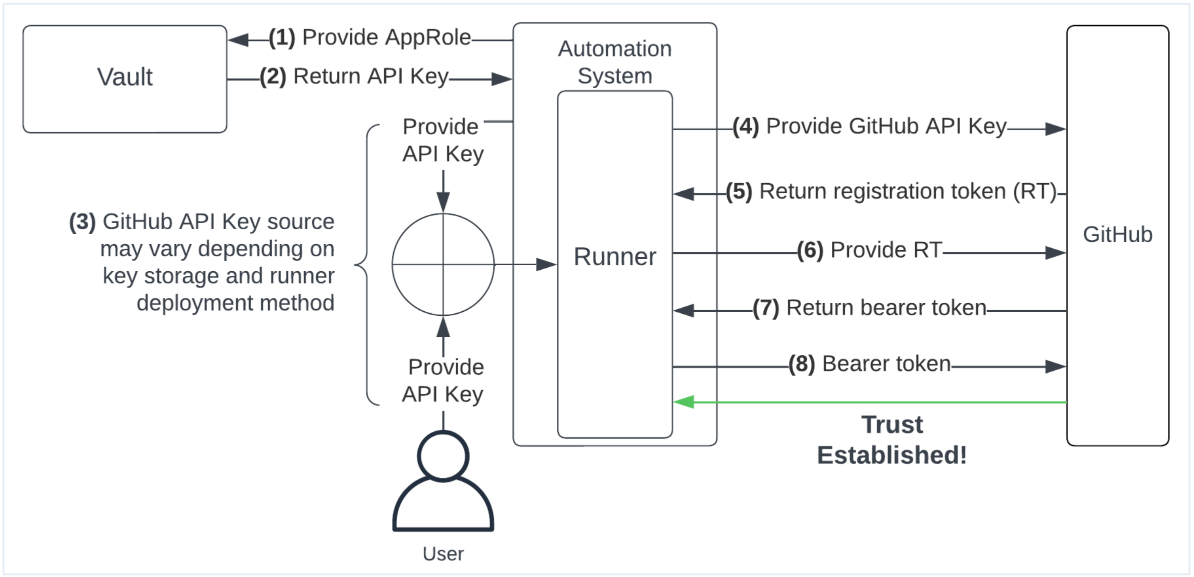

Figure 2: A simplified version of self-hosted runner configuration and registration

Secrets Access in an Actions Run

Once Vault trusts the OIDC provider, GitHub trusts the runner, and the repository admin or GitHub Enterprise admin has allowed the runner appropriately scoped repository access, the only remaining relationship we need to set up to let our runner access Vault-stored secrets referenced in GitHub Actions workflows is, naturally, between the runner itself and Vault.

OIDC handily lets us leverage the trust we already set up between the runner and GitHub, as well as between Vault and the OIDC provider (shown earlier in figure 1) to allow our runner to retrieve secrets from Vault. Note, however, that Vault must also be configured with policies defining which repositories should access which secrets; otherwise, all the

GitHub Actions workflows, runners, and repositories in our GitHub organization or enterprise would have access to the entire Vault deployment. DigitalOcean uses Terraform to apply this configuration to Vault. The OIDC protocol steps (also shown below in figure 3) are as follows:

1. The runner requests an OIDC JWT containing a signed set of identity claims from the OIDC identity provider.
2. If the runner is found to be legitimate, the OIDC provider gives it a time-limited JWT containing the requested claims.
3. The runner presents the JWT to Vault.
4. Vault verifies the JWT signature using information fetched from the identity provider.
5. If JWT signature verification succeeds, Vault then tries to match the JWT's claims against its access policies.
6. If the policy-matching procedure succeeds, Vault issues a time-limited access token to the runner.
7. If granted a Vault access token, the runner sends the Vault token back in the request to retrieve any secrets (that Vault policy configuration allows the repository to access) that were referenced in the GitHub Actions workflow.

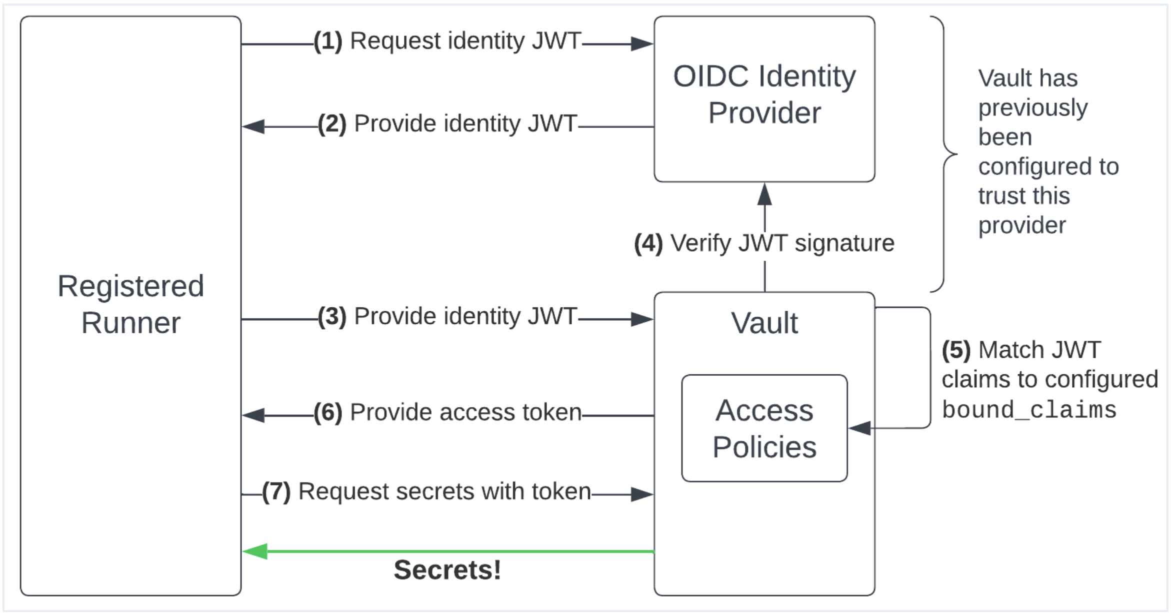

Figure 3: A GitHub Actions workflow uses OIDC to gain access to Vault-stored secrets.

Components and Trust Zones

The following table provides high-level, generalized descriptions of components and dependencies identified during our exploration of DigitalOcean's system. These system elements are classified into *trust zones*: logical clusters of shared functionality and criticality, between which the system enforces, or should enforce, interstitial controls and access policies. Each colored heading used below indicates a distinct trust zone.

Note that components marked by asterisks (*) were considered out of scope for the threat modeling assessment. We explored the implications of threats involving out-of-scope components that directly affect in-scope components, but we did not consider threats to out-of-scope components themselves. Any components or dependencies not directly described here are also out of scope.

Component	Description
Vault	These components are part of the DigitalOcean secrets-as-a-service system centered on a HashiCorp Vault deployment.
HashiCorp Vault	Vault mediates access to secrets sourced from a backing data store.
Load Balancer	The internal load balancer deployment manages connections to the internal Vault servers.
GitHub	This zone includes infrastructure and repositories that GitHub may host publicly. If an internal GitHub Enterprise deployment is in use, the components in this zone that are not noted as publicly available are instead part of the internal Enterprise deployment. Any GitHub component *not* described here is out of scope.
Actions	GitHub Actions workflows are repository- or branch-specific CI/CD files defined in YAML and stored within the repository in question. Actions workflows can consist of one or more tasks or steps, such as source linting, running tests, and application deployment. These tasks may require different permissions, secrets, and access rights.
HashiCorp `vault-action` (*)	This is a HashiCorp-maintained, public GitHub action that enables the use of Vault secrets in GitHub Actions builds. This component is out of scope.
OIDC Provider (*)	This refers to GitHub's OIDC provider. This component is out of scope.

Runners	A GitHub runner executes GitHub Actions workflows. An organization can choose to use GitHub-hosted or self-hosted runners. If GitHub hosts a runner, it exists within the GitHub trust zone.
Vault Policy Repository	This is an internal Git repository of access policies as well as scripts to manage and apply the access policies to Vault using Terraform.
Public GitHub OIDC-to-Vault Tutorial	This tutorial is configured as a GitHub template under the public, personal tutorial developer's account. It outlines the process by which a self-hosted runner executing a GitHub Actions workflow obtains a secret from Vault.
Terraform GitHub-to-Vault OIDC module	This publicly hosted Terraform module configures Vault to accept GitHub OIDC JWTs presented by runners.
Infrastructure	These components include other internal infrastructure on which the DigitalOcean Vault system depends. Any component not mentioned here should be considered out of scope.
Runners	A GitHub runner executes GitHub Actions workflows. An organization can choose to use GitHub-hosted or self-hosted runners. If a runner is self-hosted, it exists within the infrastructure trust zone.
Automation Systems (*)	Internally deployed automation systems such as Terraform handle updates to Vault components and dependencies. This dependency is out of scope.
Application Orchestration System (*)	The internal application orchestration system hosts and coordinates developer-maintained services. This dependency is out of scope.
Identity Server (*)	Internal employee and contractor accounts, as well as service principals (accounts), are granted access rights/capabilities for internal systems and data based on group or role membership. This dependency is out of scope.

Trust Zone Connections

At a design level, trust zones are delineated by the security controls that enforce the differing levels of trust within each zone. Therefore, it is necessary to ensure that data cannot move between components without first satisfying the intended trust requirements of its destination. We enumerate connections between components and trust zones below.

Originating Zone	Destination Zone	Data Description	Connection Type(s)	Authentication Type(s)
External Network	GitHub	In-browser user communications for repository access	HTTP	● TLS ● SSO and 2FA
	Infrastructure	In-browser dashboard viewing	HTTP	● TLS ● SSO and 2FA
	Vault	In-browser Vault UI interactions	HTTP	● TLS ● SSO and 2FA
Infrastructure	GitHub	Runner access to repositories (a connection that exists entirely within the GitHub trust zone if GitHub hosts the runner)	HTTP	● TLS ● GitHub API token ● Runner token
	GitHub	Runner access to an OIDC provider (a connection that exists entirely within the GitHub trust zone if GitHub hosts the runner)	HTTP	● TLS ● GitHub API token
	External Network	Single sign-on (SSO) for internal credentials checking	HTTP	● TLS ● SAML or OIDC

GitHub	External Network	Internal credentials verification against an SSO identity provider	HTTP	• TLS • SAML or OIDC
	Infrastructure	Runner deployment of or access to internal, coordinated services	HTTP	• Mutual TLS • Service principal/ account credential
	Vault	Vault policy updates via automation	HTTP	• TLS • Service principal/ account credential
	Vault	Vault tokens obtained and supplied by a runner	HTTP	• Mutual TLS • OIDC JWT • Vault token

Threat Actors

When conducting a threat model, we define the types of actors that could threaten the security of the system. We also define other “users” of the system who may be impacted by, or induced to undertake, an attack. For example, in a confused deputy attack such as cross-site request forgery (CSRF), a normal user who is induced by a third party to take a malicious action against the system would be both the victim and the direct attacker. Establishing the types of actors that could threaten the system is useful in determining which protections, if any, are necessary to mitigate or remediate vulnerabilities.

Actor	Description
OIDC Provider	The GitHub OIDC provider can issue JWTs to enable the retrieval of secrets from Vault.
External Attacker	An external attacker is an attacker on the public network who can eavesdrop on and potentially modify connections that route through the public network.
Internal Attacker	An internal attacker is an attacker on the internal network (or VPN) who can eavesdrop on and potentially modify connections that route through this network.
Employee	Employees can log in to access internal services and data.
Software Engineer	An engineer on a customer-facing feature team can create GitHub repositories in the internal enterprise and external organizations, can create commits and pull requests (including editing GitHub Actions workflow configurations), can trigger GitHub Actions workflow runs on GitHub runners that include Vault access, and can directly access Vault’s UI in a web browser through SSO.
Security Engineer	A security engineer approves Vault policy changes, can modify and deploy secrets management infrastructure, and has both general employee and organization-specific user account access rights to internal infrastructure.
Identity Administrator	An identity administrator can perform administrative actions on internal identity groups, roles, user accounts, and service accounts.

Threat Actor Paths

Additionally, we define the paths that threat actors can take through the various trust zones of the system. This is useful when analyzing the controls, remediations, and mitigations that may exist in the current architecture.

Originating Zone	Destination Zone	Actor	Description
GitHub	External Network	Internal Attacker, Employee	A compromised employee or an internal attacker may attempt to access external resources from a location such as a runner host machine, possibly using a malicious GitHub Actions workflow. Such an actor may also attempt to exfiltrate data from the host machine's or container's running environment.
	Infrastructure	Internal Attacker, Employee	With escalated internal access and privileges, an internal attacker could deploy over applications running on internal infrastructure with malicious or intentionally nonfunctional application versions.
	Vault	Internal Attacker, Employee	An insider may attempt to access secrets stored in Vault via the Vault UI or via a malicious GitHub Actions workflow or compromised runner. Such an attacker may be equipped with internal user or service accounts that have privileged groups and roles, or may have illicitly added privileged groups and roles to their own internal account(s).
	Vault	Internal Attacker	If able to compromise the Vault policy configuration repository, an internal attacker could create malicious policy updates and automatically apply them to Vault using automation run in the Vault policy repository's own GitHub Actions workflow. This could enable illicit access to keys and secrets in Vault.

External Network	GitHub	External Attacker	An external attacker could compromise some inadequately protected externally available service hosted on internal infrastructure to obtain access to the internal network or to secrets and keys in the service's running environment. If runners are self-hosted, an external attacker who gains access to the internal network could pivot to the infrastructure that hosts runners. This could enable network traffic capture and Vault access.
	Infrastructure	External Attacker	If able to compromise the organization's SSO service, an external attacker could access internal services that authenticate users via SSO. Such an attack could be a link in an exploit chain leading to a broad Vault compromise or to the theft of individual secrets.
	Vault	External Attacker	If able to masquerade as a legitimate runner to GitHub or to otherwise compromise the GitHub OIDC provider, an external attacker could gain access to source code and configuration stored in private organization repositories and to internal keys and secrets stored in the internal Vault deployment via `vault-action`.
	Infrastructure	External Attacker	An external attacker could view unauthenticated internal dashboard service links and gain knowledge of internal systems. The attacker could leverage this information or their access to the dashboard service to pivot to other internal services.

Tradeoffs and Considerations

A secret can be a credential pair, a private key, or other data that should be shared only with an authenticated and authorized actor. Secret *zero* refers to an initial, privileged secret that allows the holder to access (or delegate access to) additional secrets. Using OIDC for Vault access in GitHub Actions requires prior use of a secret zero such as a Personal Access Token or a GitHub app API token to establish the necessary trust relationships. The use of OIDC for secrets access in GitHub Actions workflows may be appropriate if the benefits outweigh the risks that a potentially over-permissioned secret zero could introduce.

Risk Consolidation versus Usability and Convenience

For example, Alex, a security-conscious person, might memorize a single account password (a secret zero) to access a password manager on a personal phone. For most of Alex's use cases, this secret zero usage acceptably balances convenience, usability, and security.

However, a secret zero consolidates the individual risks of access to the secrets it unlocks. This might not always be an acceptable risk. If Alex now wants to bring the phone with the vault on it to a hostile country that wants access to the credentials in the password manager vault, the risk taken through the use of a password manager relying on the secret zero pattern could be higher.

If the vault is cloud-hosted instead of local, Alex's phone would have to make a network request to access it. Even if the remote vault server requires TLS, an attacker who can intercept and decrypt network traffic could obtain the password, replay it against the vault, and gain illicit access to the rest of the secrets in the vault.

Infrastructure Considerations

Any secret zero used to establish trust between GitHub itself and a *GitHub*-hosted runner does not leave the GitHub trust zone (i.e., the secret zero is GitHub's responsibility), which is handy for the rest of us. However, registering a *self*-hosted runner requires secrets interaction, as shown in figure 2. A developer who manually uses the GitHub API to create a new self-hosted runner must also provide the API key and runner registration token on the command line. While the developer could source these secrets from a different secrets manager from the main Vault deployment, if self-hosted runner registration is automated, the underlying infrastructure can handle the secrets instead of the developer. Still, if an attacker were to compromise infrastructure that has secret zero access, the secret zero–based trust scheme would collapse.

The following three strategies that we identified apply different risk considerations to the problem of who (or what) will handle the initially needed secrets zero to establish the foundation for OIDC.

Runner Management Strategy 1

Make each repository owner responsible for creating and managing their own runners. This means that a minimum of one secret zero per repository in the organization will exist, limiting the *maximum* impact of compromise of a single secret zero and increasing the clarity of audit logging in the event of a compromise. On the other hand, this could also result in manual secret zero management for some repositories, which is undesirable and could lead to a secret zero's accidental storage in places like a developer's local shell history. Some teams *could* also choose to rely on GitHub-hosted runners unless they have a need for self-hosted infrastructure, as in strategy 3.

Runner Management Strategy 2

Manage self-hosted runner registration at the organization's infrastructure layer. The infrastructure owns a single enterprise-level GitHub API access token used for registering all runners. This credential requires all of the capabilities necessary to manage runners for *all* repositories in the organization. The impact of a compromise of this highly privileged secret zero *minimally* equals the sum of the impacts of compromises of the many secrets zero described in strategy 1. If this secret zero is compromised, all of the organization's runners should also be considered compromised. However, a security engineer (or team) would have a somewhat easier time in helping with safe infrastructure setup and token storage under this strategy than under strategy 1.

Just-in-Time (JIT) Runners

If the organization uses JIT self-hosted runners, a given runner's access is valid only for the duration of the GitHub Actions workflow that the runner was spun up to execute. We find that the use of JIT runners could reduce the likelihood that an adversary could take over any particular self-hosted runner. However, we also find that it would likely not reduce the risk that an adversary who gains access to the enterprise's GitHub API access token could spin up adversarial runners. Additionally, limiting the use of self-hosted runners may help reduce this risk.

Runner Management Strategy 3

Relying only on GitHub-hosted runners may be a better option for a smaller team without the desire or staffing to host and manage its own CI/CD infrastructure, but it does introduce a third-party infrastructure dependency. No security engineer (except those that GitHub employs) should need to manage runner token–related risk if this strategy is followed.

OIDC Provider Hosting

If an external third party hosts the OIDC identity provider, access to internal secrets then depends on that external third party's availability. If an external attacker were able to masquerade as a legitimate runner to the OIDC provider, or otherwise compromise the third party, the trust scheme would collapse.

Indicators For and Against OIDC Use

The use of OIDC may be a great next step for a security team in an organization or company that already has some level of maturity with regard to secrets handling and data breach response. In this section, we list signals that we think could be useful in deciding whether to introduce OIDC for secrets access in GitHub Actions workflows.

Positive Indicators

- The organization or company currently uses a secrets store such as HashiCorp Vault or AWS Secrets Manager.
- The organization or company desires to significantly limit developers' handling of secrets.
- The organization or company desires to limit the number of long-lived secrets in direct use in their Actions workflows.
- The organization or company can accept the operational risk of relying on the availability of GitHub/GitHub Enterprise.
- The organization or company can feasibly rely only on GitHub-hosted runners, *or* the organization or company uses self-hosted runners and has adequately secured its runner deployment process(es).
- The organization or company strictly regulates *all* traffic and access to and from self-hosted runners and underlying infrastructure, following the principle of least privilege.
- The security department or team has sufficiently knowledgeable resources that can implement and then support an OIDC integration indefinitely.

Contra-Indicators

Do not implement OIDC for secrets access in CI/CD only on the basis that another company, organization, or security department uses it. OIDC is a great option after a certain level of security maturity has already been achieved within an organization. Indicators that introducing OIDC to allow secrets access in GitHub Actions workflows may *not* be suitable for a company's use without implementing other security measures beforehand include the following:

- No secrets management system is currently in use. Secrets and keys may be checked into version control or stored locally on developer machines. (This could even result in the compromise of a secret zero.)
- The secrets store or management system in use does not integrate with any OIDC provider.

- The organization or company has not considered how to restrict privileges required by any secret(s) zero used to establish trust relationship(s) (e.g., between the infrastructure and the OIDC provider) before OIDC-based secrets access can occur.
- The organization or company has not considered how to limit access to each secret zero.
- The organization or company does not have safeguards in place to limit the blast radius of illicit secrets store access if developers create a weak or overbroad secrets store access policy.
- The organization or company does not have personnel with sufficient security and infrastructure knowledge to implement OIDC between GitHub and the secrets store in use and to support the integration for an indefinite amount of time.
- The employees at the organization or company have not undergone general security education, meaning it is unlikely they would effectively leverage OIDC secrets access capabilities.
- The organization or company has not implemented widely known and accepted data breach and incident response handling processes.

Alternatives to OIDC Use

If OIDC does not make sense for a particular organizational use case, an alternative, particularly for smaller-scale cases, could be sourcing needed sensitive information/keys from GitHub secrets coupled with the use of GitHub environments to restrict possible access to a given secret within a GitHub Actions workflow.

Recommendations

In this section, we briefly cover some good patterns that DigitalOcean currently follows to secure its OIDC deployment, along with some of the recommendations we made to DigitalOcean during the threat modeling exercises we conducted. This set of recommendations should not be considered an exhaustive list of best practices for OIDC usage enabling runner access to Vault. Please be sure to refer to GitHub and HashiCorp documentation as well.

Tokens, Permissions, and Access Scope

- If possible, do not use a classic Personal Access Token to authenticate to the GitHub API. Prefer an appropriately scoped fine-grained token or an app API token.

- Following the principle of least privilege, grant only the minimum possible permissions for GitHub API tokens, runner tokens, and `GITHUB_TOKEN` at the organization or enterprise level. Grant additional token permissions, especially write permissions, on an as-needed, per-workflow basis.

- Untrusted repositories, GitHub Actions workflows, and runners, as well as GitHub Actions workflows that could run untrusted code, must not be able to retrieve JWTs that enable access to the Vault deployment.

Vault System Availability

- If there is a fallback Vault deployment, keep it up to date with the primary deployment, and define a consistent failover process from the active Vault cluster to the fallback so that failover works smoothly and does not cause follow-on availability issues when performed in an emergency.

    - Automate the failover process and run it weekly, or even daily, at a scheduled time.

    - Widely publicize this scheduled time within the company or organization so that any outages and related issues can be properly attributed and fixed.

    - Implement any infrastructural changes necessary to ensure that the availability of the entire secrets system remains acceptable during failovers.

- Aggressively limit direct access of other systems within the internal network to any components in either the primary or fallback Vault cluster deployment (nodes, backing storage nodes, etc.), and ensure that there is no possibility of direct external access to Vault.

Limiting the (Re)use of Potentially Exposed Secrets Stored in Vault

- Perform regular automated scanning of source code at rest for secrets, keys, PII, and any other plaintext or encoded sensitive information.
    - Some enterprise static analysis tools, such as Checkmarx, can be configured to run such checks with every scan. Alternatively, the use of dedicated tooling such as Trufflehog can be scripted.
- Any secrets that have ever been exposed in source code or configuration, even internally, should be considered compromised and immediately rotated.
- When triaging exposed secrets, ensure that Git commit logs and any other relevant logs such as developers' local shell histories are also amended to remove all direct references to such secrets or credentials.
- Automate the core steps of secret rotation. (Consider incorporating static source and configuration analysis tooling such as Semgrep, Trufflehog, and Checkmarx to ensure that all secrets references are gone.)

Identity Management

- Ensure that each role and group granting access to sensitive information (such as credentials and secrets stored in Vault and PII stored elsewhere) has several possible membership approvers who are employees with knowledge of the capabilities and secrets that the group or role secures at all times.
    - Only the dedicated approver(s) should be able to admit internal accounts to the group or role in question, and only with valid and appropriately logged justification. For any group or role, there should be at least two defined *possible* membership approvers, with at least one signoff required from the pool, in case of an emergency.
    - A greater number of required signoffs from a pool of N possible approvers (M-of-N) may be desirable for more sensitive groups or roles.
    - The approver(s) should follow a short, written checklist (stored in Confluence, Sharepoint, or another widely accessible internal wiki) of definite criteria to grant or deny group/role membership, depending on the severity of potential compromise if the group/role membership were granted to a malicious insider, exposing sensitive data as a result.
    - Periodically audit the criteria for membership in all groups and roles, and ensure that representatives from both the security team and the group/role

    approver(s) pool think that these criteria are reasonable with regard to the privileges gained with group membership.

- An appropriate case for creating any new account/principal in production should require logged justification, such as the onboarding of a documented new employee following the approval of that new employee's future manager.
    - Examples of such justification include a signoff from the requestor's manager and a signoff from any approvers for groups or roles that the service account will join.
    - Creating new service accounts should also require logged justification.
- Logging should be immutable, actively monitored, and periodically reviewed for abnormalities such as inappropriate assignment of privileges and access rights. This will not only help with incident response but will also enable quick correction if an account is assigned inappropriate permissions.
    - Log justifications for denying or accepting group or role admittance.
    - Implement and follow a clear, simple process for revoking and correcting abnormalities and inappropriate role/group assignment.
    - Implement an incident response process for quickly addressing abnormalities that appear to be the result of malicious activity and for following up with further investigation if needed.
- When employees or contractors leave the organization or service accounts are no longer required, the account(s) should be promptly decommissioned so that access is not retained.
    - Privilege/group/role removal from user accounts should be automated when organizational structure changes or when an individual moves from one internal group to another.
    - Periodically audit all active accounts on production identity server(s) and remove any that do not pertain to active employees or in-use, deployed services.
- Sudoers and administrator access lists on individual machines like runner hosts should not include individual human accounts, but these rights may be granted on an as-needed basis via group or role membership.

- Debugging and observability tooling should replace the sudoer or administrator access of any human accounts on production machines outside of a very small, select set of administrators.
- The set of people with sudoer or administrator access on production machines should be limited to only those whose job requires direct production machine administration.
- *All* production machine access and sensitive VM or container access should always be audit-logged and time-stamped.

- All internal non-service (employee and contractor) accounts must authenticate with a second factor sourced from a different piece of hardware.
  - Ideally, the second factor should be an air-gapped hardware token like a Yubikey, or a one-time password (OTP) code from an application like Duo or 1Password running on a separate device.
  - If possible, avoid using push notifications, SMS, and voice-based authentication as a second factor for internal human accounts, for the following reasons:
    - Users may get in the habit of simply approving every two-factor push notification.
    - SMS can be compromised through SIM swapping.
    - Voice-based authentication can be spoofed through the use of recordings or machine learning.
  - Educate internal account holders on how their second-factor method is intended to work.
  - Ensure that no employee will ever ask for a 2FA code or hardware token output via any communications method, including email or SMS, for any reason.
    - Document in a widely accessible place such as the internal wiki (Confluence, Sharepoint) that *no* employee will ever ask for an OTP code or hardware token output.
    - Every account holder must always sign in using their own password as the first factor and their own OTP code or hardware token output as the second factor.

Network Access for Self-Hosted Runner Infrastructure

- Require mutual TLS between as many internal services as feasible. This will ensure that two internal systems can establish a communication channel only if *both* systems successfully authenticate themselves and verify the other's certificate chain.
- Ensure that the load balancers internally fronting the Vault deployment are assigned static IP addresses. Then, allowlist only those addresses to directly connect to Vault.
- Enable log retention for GitHub Actions workflow runs in GitHub (or GitHub Enterprise if applicable) for a period complying with any security and data retention restrictions that the organization must obey. This period should at least be long enough to detect anomalous behavior, which may be one-off or repeated.
- Ensure that an internal party such as the security operations team (SOC)/incident response team receives alerts when anything anomalous happens, like an outbound network connection from a Vault server. Ensure that there is a thoroughly documented process to guide the team in interpreting and responding to alerts.
- Use granular runner groups to dedicate particular self-hosted runner(s) that may otherwise be sourced from within more generic company-wide "production" or "staging" pools to teams and organizations. This will provide clearer audit records for incident response and a better sense of the account(s) that the infrastructure acted on behalf of, especially in a larger organization or company.
- Ensure that self-hosted GitHub runners are assigned IP addresses only from a particular range. Restrict load balancer–mediated Vault access to only this address range.
- Restrict the network access possible to/from self-hosted runners. Allowlist business-necessary network accesses and deny all other access attempts.

Additional References

- GitHub Docs: Hardening for self-hosted runners
- GitHub Docs: How the permissions are calculated for a workflow job
- GitHub Docs: Enforcing a policy for artifact and log retention in your enterprise
- GitHub Docs: Controlling access to larger runners
- GitHub Docs: Managing access to self-hosted runners using groups
- GitHub Docs: Accessing GitHub resources

Host System Access from Self-Hosted Runners

- Organizations that are self-hosting runners should prefer using JIT runners. A JIT self-hosted runner can execute at most one GitHub Actions workflow run during its lifetime. This can help reduce the likelihood of compromise.
- Self-hosted runners should be used only with private GitHub repositories without public forks, because a repository fork could run malicious code on the self-hosted runner to, for example, exfiltrate keys and secrets from the running environment.
- Enable as much system-level logging and monitoring for anomalous traffic, processes, and so on, on runner hosts as possible, and ensure that the incident response team or another responsible party has access to these logs and has processes in place to automatically detect and handle anomalies.
- Ensure that workflow- and job-specific container state and sensitive data are not persisted, saved, or shared between GitHub Actions containers.
- Use GitHub environments and limit job permissions to restrict environment sharing and potential secret exposure between jobs in the same GitHub Actions workflow or container.
- Runner workflow containers should never be able to interact directly with the runner host environment. Enforce this via system-level protections on the hardware, VMs, or containers where runners can be scheduled. Such protections include (but are not limited to) the following:
  - gVisor or SELinux policies for the purposes of managing the Docker daemon and reducing the likelihood of runner container breakouts
  - Seccomp, AppArmor, and similar security controls for the purposes of restricting process and program capabilities, such as system call (e.g., `execve`) usage on the runner host

Additional References

- GitHub: Potential impact of a compromised runner
- O'Reilly: *Container Security* (Liz Rice)
- Ryotak: Stealing GitHub staff's access token via GitHub Actions
- Red Hat: Docker SELinux Security Policy

- gVisor: Documentation
- Seccomp: SECure COMPuting with filters
- Ubuntu Wiki: AppArmor
- Red Hat: OpenShift Sandboxed Containers 101
- Trail of Bits: Understanding Docker container escapes

Vault Policy Management

- Define the process that the security team and the feature team who owns the secret(s) stored in Vault and the related service(s) or codebase in GitHub must follow, working together, to do the following:
    - Remediate any Vault policy problems
    - Check for and clean up any possible artifacts of incorrect policy application such as debug logs
    - Rotate any potentially exposed secret values
- Implement a checklist of requirements that every Vault policy pull request must pass, such as a Semgrep scan.
    - All Vault policy pull requests must have two approvers: a different member of the pull request creator's team and a security team member. This can be automatically enforced in GitHub or GitHub Enterprise.
    - Flag severe anomalies discovered by the required scan passes and known conflicts for secondary human review by security engineers and the policy change creator. If a pull request is flagged, do not allow it to merge until human review passes.

Reducing the Potential Risk and Impact of OIDC JWT Replay

- Write automation, possibly incorporating static analysis tooling such as Semgrep or Checkmarx, to examine all secret and policy additions to Vault as these additions are made. Ensure that any access to secrets stored in Vault is appropriately limited following the principle of least privilege.
- To limit what an attacker could gain by replaying a particular OIDC JWT within its validity period, consider mapping each Vault token requested via OIDC to only a single credential or credential pair within Vault:

    - Each value accessible via a particular Vault key or credential should consist of, for example, a single API token or a single username and password pair.
        - *No* secret should consist of multiple credential sets or multiple sensitive items stored under the same key.
        - Each access to a secret in Vault should require its own time-limited OIDC token exchange.
- There should be few to *no* policies that allow access to all secrets in a particular area of Vault, or that allow all or many users of Vault to access a particular secret.

Hands-On Evaluation (Penetration Testing)

- Schedule an internal or external penetration test/hands-on evaluation of any self-hosted GitHub runners' deployment and configuration to determine the potential extent of compromise of the internal network and infrastructure from runners.