GitHub | Sep 2022

Integrating GitHub Advanced Security with third party reporting and analytics platforms

GitHub Advanced Security (GHAS) is an application security solution that enables companies to approach security with a developer-first mindset. Integrating GHAS with external reporting and security information and event management (SIEM) tools allow customers of the GitHub platform to improve their security posture by increasing the visibility of application security events.

Table of Contents

Abstract

This document is intended to capture strategies for integrating and ingesting alerts from the GitHub Advanced Security (GHAS) platform into external reporting, Security Information and Event Management (SIEM) services, and vulnerability analytics platforms.

Integrating the metrics provided by GitHub Advanced Security into an external reporting and analytics platform allows customers to gain deeper insight into their application security posture.

What we'll cover in this white paper

- Methods for extracting information from the GitHub platform for ingestion in a reporting tool
- Ideal integration methods
- Metrics to watch in your reporting/SIEM tool

Audience

The intended audience for this white paper is engineers looking to integrate the GHAS platform into existing internal SIEM tools, including:

- GitHub administrators
- Application Security teams
- Security Operations
- SOC Analyst teams
- SIEM and reporting tool vendors
- GitHub partners working with customers on GHAS implementations

Glossary of terms

- GHAS - GitHub Advanced Security
- SIEM - Security information and event management
- SARIF - Static Analysis Results Interchange Format, a standard JSON-based format for the output of static analysis tools.

Authors

Dan Shanahan - GitHub Advanced Security Field Specialist
Chad Bentz - GitHub Advanced Security Field Specialist
Alexander De Michieli - Senior Partner Engineer

Executive Summary

Enterprise data retention, audit requirements, and fluid design considerations are important in your security architecture. While GitHub does provide the Security Overview for viewing GHAS alerts, you may require a longer-term historical audit trail of the data, or even find the need to run more powerful queries, charting, and visualizations while joining additional data from your own environment logs.

There are four primary methods for extracting GitHub Advanced Security metrics from the GitHub platform. These methodologies apply to both GitHub Enterprise Cloud and GitHub Enterprise Server.

Webhooks
GitHub webhooks are HTTPS payloads that are immediately delivered to an endpoint when a specified event occurs. Webhooks are the preferred method for long-term metric collection and event alerting for the GitHub Advanced Security (GHAS) platform. Webhook events are sent for status changes on all events in the GHAS platform including code scanning, secret scanning, and Dependabot at the time the event occurs.

REST API
The GitHub REST API provides a snapshot view of the current status of your GHAS platform. The REST API provides information about the state of all alerts for code scanning, secret scanning, and repository vulnerabilities.

GraphQL API
Our GraphQL API is a powerful interface for querying information about your GitHub configuration. This API allows developers to extract information about repository vulnerabilities (Dependabot alerts)

Audit Log
To support debugging and internal and external compliance, GitHub Enterprise provides logs of audited user, organization, and repository events. The content of the audit log is not scoped specifically to GHAS events but provides insights into sensitive changes within your GitHub environment.

Integration Methods

Webhooks

Webhooks are a one-way communication method that sends a JSON payload via HTTPS to a receiver when an event happens on the GitHub platform.

Webhook messages are triggered each time a security finding is created, resolved, or marked as closed by a user. The webhook message contains information about the alert and where the finding was identified. The verbosity of the alert, along with the just-in-time nature of webhooks, creates an ideal integration target for SIEM tools.

Webhook Pros	Webhook Cons
● Easy configuration ● Protected with secret token ● Real-time event ● Minimal infrastructure required ● No rate limiting	● Tracking event lifecycle is difficult ● Potential for missed messages

Implementation

Creating a webhook is a two-step process. You'll first need to set up how you want your webhook to behave through GitHub: what events it should listen to. After that, you'll set up your server to receive and manage the payload.

Webhook configuration docs are available here.

Server-side Configuration

Your webhook receiver should be prepared to accept JSON payloads via a `POST` method. The ideal content-type header will be `application/json`.

Setting a webhook secret allows you to ensure that `POST` requests sent to the payload URL are from GitHub. When you set a secret, you'll receive the `X-Hub-Signature` and `X-Hub-Signature-256` headers in the webhook `POST` request. For more information on how to use a secret with a signature header to secure your webhook payloads, see "Securing your webhooks."

GitHub Configuration

Webhooks can be configured at the repository and organization level. To enable a webhook, navigate to the settings page of either the repository or organization and choose Webhooks under Code, planning and automation. Select to create a new webhook, then enter the endpoint URL of your receiver, choose `application/json` as the content type, then enter the secret (if one was generated on your receiver).

You will be asked which events should trigger a webhook. The following events are GHAS specific, however, there are many other events that may be pertinent to your environment.

- Code scanning alerts
- Secret scanning alerts
- Secret scanning alert locations
- Security and analyses
- Repository vulnerability alerts (Dependabot alerts)

A complete example of a webhook receiver is available here.

REST API

The GitHub REST API provides a fully-featured platform for extracting and updating findings from GHAS. Information on findings from code scanning and secret scanning is made available via the REST API. Additionally, GitHub has provided developer libraries to make writing integrations easy.

The REST API is ideal in situations where you need to report on the current state of the GHAS platform. For example, an API request could be created to return all the closed code scanning alerts for a particular organization. It's important to note that the REST API does have rate limits in place. Personal access tokens maintain a rate limit of 5000 requests/hour. GitHub app tokens are limited to 15,000 requests/hr.

REST API Pros	REST API Cons
● CRUD interface ● Point-in-time view of the status of alerts ● Filterable queries	● Requires poller infrastructure ● Not event driven ● Rate limits

Implementation

Consuming GHAS events via the API will require a poller service that queries the GitHub REST API on a scheduled basis. OctoKit is a client library that makes writing these integrations easy by abstracting some of the more complex aspects of the API such as paging and authentication.

There will be two primary endpoints you will focus on, code scanning alerts and secret scanning alerts

Authentication

Production applications that interact with the GitHub API should be built and installed as a GitHub app. GitHub apps unlock many integration options and permission capabilities, as well as expand the API rate limits to 15,000 requests per hour.

In non-production environments, you can provide a personal access token to authenticate to the GitHub API. This token is scoped to the user and has a rate limit of 5,000 requests per hour.

NOTE: For documentation on the best integration methods with GitHub APIs, see Basics of GitHub Authentication

Code scanning API

The code scanning API provides visibility into all alerts generated by code scanning. This API returns results for an entire enterprise, organization, or repository. Our guidance is to query the enterprise API for all events on a scheduled basis to provide a snapshot of the current security posture. Querying the enterprise API simplifies the process by not requiring you to populate a list of all repositories or organizations.

Example query:

```
curl \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: token <TOKEN>" \
  https://api.github.com/enterprises/ENTERPRISE/code-scanning/alerts
```

Code Scanning Analysis API

The Code Scanning Analyses API is able to extract both scan summary information and detailed SARIF results. SARIF is a JSON results interchange format consumed by code scanning, and produced by static analysis tools like CodeQL. The SARIF specification allows for detailed information about runs of a code analysis tool and the results they produce. This notably includes data points not generally available from the API, such as tooling, rule, severity, locations, fingerprints, and code flows. For more information visit the SARIF support for code scanning article.

To retrieve SARIF results you must supply a repository `analysis id` as returned from Code Scanning List Analyses API and specify the `application/sarif+json` media type accept header. The API response includes a subset of the actual data that was uploaded for the specified analysis along with additional data such as the generated `github/alertNumber` and `github/alertUrl`. The additional alert properties can be used to correlate data from the Code Scanning API/Webhooks. Please note that the response size from this API may be exceptionally large.

Example query:

```
curl \
  -H "Accept: application/sarif+json" \
  -H "Authorization: token <TOKEN>" \

https://api.github.com/repos/OWNER/REPO/code-scanning/analyses/ANALYSIS_ID
```

Secret scanning API

The secret scanning API provides visibility into all alerts generated by the secret scanning service. This API returns results for an entire enterprise, organization, or repository. Again, like the code scanning API, the best practice is to query results from the secret scanning enterprise API to retrieve all events across the enterprise.

IMPORTANT: The secret scanning API returns the plain-text content of a secret scanning alert (containing the exposed secret). This information should not be logged and discarded if not needed.

Example query:

```
curl \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: token <TOKEN>" \
  https://api.github.com/enterprises/ENTERPRISE/secret-scanning/alerts
```

GraphQL API

GraphQL is a query language used to interact with APIs. The GitHub GraphQL API offers more precise and flexible queries than the GitHub REST API, as it allows you to accurately define the data you want. That being said, with accuracy comes complexity as all requests are validated and executed against the schema. GraphQL requires in general a more structured approach. For a list of docs, visit the Getting Started guide.

The GraphQL API has its own limits which are different from the REST API's rate limits. To accurately represent the server cost of a query, the GraphQL API calculates a call's rate limit score based on a normalized scale of points. Visit the Rate Limits guide to learn more about the rate limits with the GraphQL API

GraphQL Pros	GraphQL Cons
● More precise calls ● Reduced count of calls to the API endpoint	● More complex configuration ● Limited integration options (only Dependabot events)

Implementation

To communicate with the GraphQL server, you'll need an OAuth token with the right scopes (refer to REST API documentation). While The REST API has numerous endpoints, the GraphQL API has a single endpoint, which remains constant no matter what operation you perform: `https://api.github.com/graphql`

Here's an example of a GraphQL call to retrieve Dependabot events. If you want to learn more about the **RepositoryVulnerabilityAlert** object, consult this resource.

Audit Log

Audit logs contain events that are generated by activities within your enterprise, from the current month and up to the previous six months. The audit log also retains git events such as cloning, fetching, and pushing for seven days. There are a multitude of recorded events that cover nearly every important aspect of your enterprise; visit the Audit Log Events guide for a comprehensive list. We recommend visiting the Accessing Audit Logs guide to learn how to view them.

Although many activities are recorded, it's important to note that Audit Logs aren't supposed to substitute for alerts from the GitHub Advanced Security (GHAS) platform. The Audit Logs contain a wealth of information that can help your SOC analysts discover threats against your platform. Using correlation between audit log and GHAS data may assist you during an investigation. SIEM platform integrations such as the Microsoft Sentinel - Continuous Threat Monitoring for GitHub can provide tailored threat hunting queries for your environment.

Audit Log Streaming

Audit log streaming automatically writes a copy of all audit logs in an external location such as S3, Azure Blob Storage, Google Cloud Storage, or Splunk. Log streaming is the de facto mechanism for ingesting most platform events at the enterprise level (across all organizations).

Audit Log Streaming Pros	Audit Log Streaming Cons
● Easy configuration ● Well-tested integrations ● Near real-time ● Resilience (7 days of playback) ● Pause for maintenance	● Limited set of integration options ● Only a subset of GHAS alert data

Implementation

Review the list of currently supported Audit Log Streaming integrations for GitHub Enterprise Cloud to review a provider that best fits your needs. An Enterprise Owner is required to configure the stream for any providers. Follow the detailed instructions provided for the specific integration of your choosing. Further, consider the ability to pause the audit log stream in the event of an outage or known maintenance at your provider.

Audit Log Polling

If your organization is using GitHub Enterprise Cloud you can interact with the Audit Log via the GraphQL API and REST API. With both methods, you can access:

- Your organization or repository settings

- Changes in permissions
- Added or removed users in an organization, repository, or team
- Users being promoted to admin
- Changes to permissions of a GitHub App

Please note that you can't retrieve Git events using the GraphQL API. If you wish to do that, you should use the REST API instead. Likewise, keep in mind that the Audit Log retains Git events for seven days, which is different from other audit log events that can be retained for up to seven months.

For more information, visit the GitHub Enterprise Cloud documentation.

Implementation

You can programmatically access the GitHub Audit Log events with the REST or GraphQL API.

This endpoint is only available for organizations on GitHub Enterprise Cloud. You must be an organization owner, and you must use an access token with the `admin:org` scope to use this endpoint. GitHub Apps must have the `organization_administration` read permission to use this endpoint. Visit the Using the audit log API for your enterprise documentation for a great starting point.

By default, the response includes up to 30 events from the past three months. If you wish to retrieve more (or less) events, you should use pagination. For more information about the audit log REST API, see "Organizations."

The GraphQL response can include data for up to 90 to 120 days. For example, you can make a GraphQL request to see all the new organization members added to your organization. For more information, see the "GraphQL API Audit Log."

Metrics

In this section, we will discuss the core metrics from each of the capabilities of GHAS which should be monitored and reported on. These metrics are not an exhaustive list, but only a starting point.

Code Scanning

The role of code scanning is to review source code for security vulnerabilities found in the Static Analysis process. The following metrics provide insight into the status of your security scanning posture. All of these metrics are available (or can be extracted) in both the REST API and webhooks.

Webhook Payload Example
REST Payload Example

Commonly Reported Metrics for Code Scanning:

Metric	Description
Count of current open findings	Count across the entire environment of all open findings
Count of new findings over time	New events plotted on a timeline
Count of closed findings over time	Closed events plotted on a timeline
Mean time to resolution	The resolution duration (closed at - opened at) averaged over a time period
Count by severity	Chart depicting counts by severity
Count by organization	Chart depicting counts by organization
Count by repository	Chart depicting counts by repository
Count by tool name	Chart depicting counts by tool name
Count by rule	Chart depicting counts by rule
Count by dismissed reason	Chart depicting counts by the reason it was dismissed
Count by language	Chart depicting counts by programming language. Webhook events include this data by default. API responses do not.

Secret Scanning

Secret scanning identifies plain-text credentials from our secret scanning partners located in a repository and prevents these credentials from being written to GitHub using push protection. The following metrics provide insight into the counts of secrets available in your repositories. These metrics are available by polling the REST API or by receiving webhook notifications.

Commonly Reported Metrics for Secret Scanning:

Metric	Description
Count of current open secrets	Total count across the entire environment of secrets in an open state
Count of opened secrets over time	Open events plotted on a timeline
Count of closed secrets over time	Closed events plotted on a timeline
Mean time to resolution	The resolution duration (closed at - opened at) averaged over a time period
Count by resolution type	Chart depicting counts by resolution type
Count by secret type	Chart depicting counts by secret type
Count by repository	Chart depicting counts by repository
Count by organization	Chart depicting counts by organization
Count by repository visibility level	Counts per repository visibility type (private, public, internal)
Count of push protection bypass	Chart depicting counts and reason for push protection bypass

Repository Vulnerabilities (Dependabot alerts)

Dependabot is the GitHub service that secures the software development supply chain. These metrics give insight into the status of dependencies with vulnerabilities within your environment and are accessible by querying the GraphQL API or by receiving webhooks.

Commonly Reported Metrics for Repository Vulnerabilities:

Metric	Description
Count of current open vulnerabilities	Total count across the entire environment of vulnerabilities in an open state
Count of opened vulnerabilities over time	Open events plotted on a timeline
Count of closed vulnerabilities over time	Closed events plotted on a timeline
Mean time to resolution	The resolution duration (closed at minus opened at) averaged over a time period
Count by severity	Chart depicting counts by severity
Count by repository	Chart depicting counts by repository
Count by organization	Chart depicting counts by organization

GitHub Audit Log

The following table describes audit log entries that may prove useful for security teams. These events are not necessarily tied to GHAS capabilities but should be monitored to ensure the integrity of the GitHub platform. This information is collected by polling the audit log REST or GraphQL APIs or by configuring audit log streaming. The entire list of GitHub audit log events is available here.

Commonly Reported Audit Log Metrics:

Action	Example events
dependabot_alerts	Users disabling Dependabot for an org
dependency_graph	User disables dependency graph for an org
enterprise	A new actions runner group is created
integration_installation	A new GitHub marketplace app is installed
ip_allow_list	The IP allow list is disabled for a repository
ip_allow_list_entry	A new IP address is added to the ip allow list
org	A new user is added to the org, MFA is disabled on an org, a new user is added to an Advanced security policy, a new self-hosted action runner is added
org_credential _authorization	Member authorizes credentials for use with SAML single sign-on
org_secret_scanning_custom_pattern	A custom secret scanning pattern is deleted
oauth_application	A new oauth application is created
repo	Self hosted runner is created, member is added,
repository_advisory	A repository security advisory is created
repository_dependency_graph	The dependency graph is disabled for a repository
repository_secret_scanning	Secret scanning is disabled for a repository
repository_secret_scanning_custom_pattern	Secret scanning custom pattern is disabled for a repository
repository_secret_scanning_push_protection	Push protection is disabled for a repository

repository_vulnerability_alerts	Dependabot alerts are disabled on a repository
secret_scanning	Secret scanning is disabled at the org level by an admin
secret_scanning_new_repos	An admin disables secret scanning for all new repositories
secret_scanning_push_protection	A user bypasses a secret scanning push protection alert
team	A new team is added to a repository, a new member is added to a team

References

Getting Starting with Ingesting GHAS alerts

- Presentation

Authentication to GitHub APIs

- Documentation

REST API

- Secret Scanning API
- Code Scanning API

GraphQL

- Dependabot (repository vulnerability alert)

WebHooks

- Secret Scanning Alert
- Code Scanning Alert
- Dependabot Alert

Log Streaming

- Blog
- Docs

Log Forwarding (Enterprise Server only)

- Docs