# Code scanning with GitHub CodeQL

## How does CodeQL analyze code?

CodeQL analyzes code through a process that involves several steps:

1. Compilation: CodeQL starts by compiling the target codebase into an intermediate representation called the "CodeQL database." The compilation step varies depending on the programming language being analyzed. For compiled languages like C, C++, and Java, the code is typically compiled using the respective language compilers. For interpreted languages like Python and JavaScript, the source code is parsed and analyzed directly.
2. CodeQL Query Execution: Once the code is compiled into a CodeQL database, CodeQL queries are executed against the database. Queries are written in the CodeQL language and define the specific analyses to be performed. Queries can express data-flow analysis, control-flow analysis, taint tracking, security checks, and more. The queries examine the code's structure, relationships between elements, and data flows within the program.
3. Results Collection: As CodeQL executes the queries, it collects results based on the specified analysis criteria. These results typically include code patterns, vulnerabilities, data flow paths, control flow paths, and other relevant information. CodeQL assigns severity levels and categorizes the results based on the predefined query logic.
4. Result Presentation: After the analysis is complete, CodeQL presents the results in a human-readable format. This includes displaying the identified issues, vulnerabilities, or insights along with their locations in the codebase. The results may include code snippets, contextual information, and suggestions for remediation.
5. Continuous Integration/Static Analysis Integration: CodeQL can be integrated into continuous integration (CI) systems or used as a standalone static analysis tool. In CI environments, CodeQL can be set up to automatically analyze code changes and provide feedback on potential issues or vulnerabilities. It can also integrate with other static analysis tools to provide a comprehensive analysis of the codebase.

Overall, CodeQL combines the power of a sophisticated query language, data-flow analysis, and a rich database representation of the code to enable thorough and precise code analysis. It helps identify security vulnerabilities, code quality issues, and other potential risks in software projects.

Here are examples of how CodeQL analyzes code in a compiled language (C++) and an interpreted language (Python):

Example for Compiled Language (C++):

cpp

import cpp

// Define a simple query to find potential null pointer dereferences

from cpp import Method, Expr

where

exists(Method m |

exists(Expr e |

e.(cpp::NullPointerDereference)

and e.getAnAccess().getEnclosingCallable() = m

)

)

select m, "Potential null pointer dereference"

In this example, CodeQL is analyzing C++ code. The query searches for potential null pointer dereferences by examining methods (Method) and expressions (Expr). If it finds an expression that represents a null pointer dereference (cpp::NullPointerDereference) within a method, it selects the method and displays the message "Potential null pointer dereference".

Example for Interpreted Language (Python):

python

import python

# Define a simple query to find potential SQL injection vulnerabilities

from python import DataFlow::PathGraph, SinkNode

where

exists(DataFlow::PathGraph pathGraph |

exists(SinkNode sink |

pathGraph.getASink() = sink

and sink.hasTaintFlow()

and sink.getTaintFlow().toString().contains("SQL query")

)

)

select pathGraph, "Potential SQL injection"

In this example, CodeQL is analyzing Python code. The query searches for potential SQL injection vulnerabilities by examining the data flow within the code. It looks for a data flow path graph (DataFlow::PathGraph) that contains a sink node (SinkNode) representing a tainted value. If the tainted value corresponds to an SQL query, it selects the path graph and displays the message "Potential SQL injection".

These examples showcase the flexibility of CodeQL in analyzing code in different languages. The queries can be customized and extended to suit specific analysis requirements and uncover various types of issues in the codebase.

1. What is CodeQL?
   * CodeQL is an analysis engine used for automating security checks and variant analysis.
   * Code is treated as data, and vulnerabilities and errors are modeled as queries that can be executed against code databases.
   * You can use standard CodeQL queries or write custom queries for analyzing code.
2. Variant Analysis:
   * Variant analysis helps identify similar problems in code based on known security vulnerabilities.
   * CodeQL querying is an efficient way to perform variant analysis.
   * Standard CodeQL queries can identify seed vulnerabilities, and custom queries can find new vulnerabilities.
3. CodeQL Databases:
   * CodeQL databases contain data extracted from a codebase, including the abstract syntax tree, data-flow graph, and control-flow graph.
   * Each language has its own database schema, defining relations and tables for language constructs.
   * CodeQL libraries provide an object-oriented view of the data, making it easier to write queries.
4. Query Suites:
   * Query suites allow the selection of queries based on criteria like filename, location, or metadata properties.
   * They help group and organize queries for frequent use in CodeQL analyses.
   * Query suite definitions are stored in YAML files and can contain multiple queries.
5. Default Query Suites:
   * CodeQL has three default query suites: code-scanning, security-extended, and security-and-quality.
   * These suites include various security and quality queries, with different levels of precision and severity.
6. Query Language (QL) Packs:
   * QL packs organize files used in CodeQL analysis, including queries, libraries, query suites, and metadata.
   * CodeQL repositories provide QL packs for different languages like C/C++, C#, Java, JavaScript, Python, Ruby, and Go.
   * Custom QL packs can be created to contain your own queries and libraries.
7. QL Pack Structure:
   * A QL pack must have a file called qlpack.yml in its root directory.
   * The files and directories within the pack should be organized logically.
   * Queries are organized into directories based on specific categories.
   * Queries related to products, libraries, and frameworks have their own top-level directories.
   * There is a top-level directory named "<owner>/<language>" for query library (.qll) files.
   * Within the "<owner>/<language>" directory, .qll files should be further organized into subdirectories based on specific categories.

This summary provides an understanding of the structure and organization of QL packs, including the requirement for qlpack.yml, the categorization of queries, and the organization of query library files.

name: codeql/java-queries

version: 0.0.6-dev

groups: java

suites: codeql-suites

extractor: java

defaultSuiteFile: codeql-suites/java-code-scanning.qls

dependencies:

- codeql/java-all: "\*"

- codeql/suite-helpers: "\*"

This YAML represents a QL pack named "codeql/java-queries" with version "0.0.6-dev" targeting the "java" language. It specifies the "codeql-suites" as the suites directory and "java-code-scanning.qls" as the default suite file. Additionally, it lists the dependencies on "codeql/java-all" and "codeql/suite-helpers" with a wildcard version to include all versions.

# CodeQL analysis consists of three steps:

1. Database creation:
   * For compiled languages, the build process is monitored to extract a relational representation of each source file, including syntactic and semantic data.
   * For interpreted languages, the extractor runs directly on the source code to create an accurate representation of the codebase.
   * Each language supported by CodeQL has its own extractor to ensure accuracy.
   * Extracted data is imported into a CodeQL database.
2. Query execution:
   * CodeQL queries, written in the QL language, are executed against the CodeQL database.
   * Queries can be standard ones from the CodeQL repository or custom queries.
   * Queries can be executed using the CodeQL for VS Code extension or the CodeQL CLI.
3. Query results:
   * Results generated during query execution are interpreted to provide meaningful insights.
   * Queries contain metadata properties that indicate how results should be interpreted.
   * Interpretation highlights potential issues in the source code.
   * Results are output for code review and triaging.
   * CodeQL for Visual Studio Code automatically displays interpreted query results in the source code.
   * Results from the CodeQL CLI can be output in different formats for use with various tools.

# What is QL?

QL (Query Language) is a declarative, object-oriented query language optimized for analyzing hierarchical data structures, particularly databases representing software artifacts.

Key points about QL:

1. Purpose of a query language:
   * Provide a platform to ask questions about information stored in a database.
   * Abstract away the details of the underlying database management system, simplifying query writing.
   * Express complex queries and ensure efficient execution.
2. QL syntax:
   * Similar to SQL, but based on Datalog, a declarative logic programming language.
   * Operations in QL are logical operations, making it a logic language.
   * Supports recursion and aggregates for concise and simple queries.
   * Object-oriented approach using classes modeled as predicates and inheritance as implication.
3. QL vs. general-purpose programming languages:
   * QL lacks imperative features like variable assignments and file system operations.
   * Operates on sets of tuples, and queries define results through set operations.
   * Set-based semantics facilitate processing collections of values without storage concerns.
4. Object orientation in QL:
   * QL offers the benefits of object orientation, including modularity, information hiding, and code reuse.
   * Defines a simple object model with classes represented as predicates and inheritance.

QL provides a powerful and expressive language for querying and analyzing data, particularly for software artifacts. It allows developers and researchers to write custom queries or utilize pre-existing open-source queries effectively.

# Code scanning and CodeQL

Code scanning and CodeQL can be set up for analysis and alert generation using different tools and workflows, depending on your requirements:

1. Analysis tool and alert generation options:
   * CodeQL: GitHub Actions or third-party continuous integration (CI) system.
   * Third-party: GitHub Actions or external generation with subsequent upload to GitHub.

Setting up code scanning with GitHub Actions and CodeQL:

1. Go to the Security tab of your repository.
2. Click "Set up code scanning" next to Code scanning alerts (enable GitHub Advanced Security if necessary).
3. Under "Get started with code scanning," choose to set up the CodeQL analysis workflow or a third-party workflow.
4. Customize the workflow if needed.
5. Commit the workflow changes directly to the default branch or create a new branch and start a pull request.
6. Code scanning will analyze your code based on the configured workflow triggers (e.g., push or pull request events).

Bulk setup of code scanning:

You can use a script to set up code scanning in multiple repositories simultaneously. Two examples are provided:

* jhutchings1/Create-ActionsPRs (PowerShell script)
* nickliffen/ghas-enablement (NodeJS script)

These scripts automate the process of adding a GitHub Actions workflow that enables code scanning to multiple repositories.

By following these steps and utilizing bulk setup scripts, you can efficiently configure code scanning with CodeQL on GitHub repositories.

# Customize your code scanning workflow with CodeQL - Part 1

To better configure the CodeQL code scanning workflows, you have options to specify additional queries that suit your organization's needs. When using CodeQL for scanning, the analysis engine generates a database from the code and executes queries on it. While CodeQL includes default queries, you can add more queries to run alongside the defaults.

There are two ways to specify the queries you want to run with CodeQL code scanning:

1. Using your code scanning workflow: You can edit the workflow file to reference additional queries. You have the flexibility to specify query packs or individual query files/directories/definitions.

* packs: Allows you to install CodeQL query packs (beta) and run the default queries or query suite included in those packs.
* queries: Enables you to specify single .ql files, directories with multiple .ql files, .qls query suite definition files, or a combination of these.

You can use both packs and queries together in the same workflow. However, it's not recommended to reference query suites directly from the github/codeql repository, as they may not be compatible with your other queries.

1. Using a custom configuration file: For workflows that generate CodeQL databases for multiple languages, you can specify the additional queries in a configuration file instead of the workflow file.

If you want to add CodeQL query packs (beta), you can use the with: packs: entry within the uses: github/codeql-action/init@v1 section of the workflow. In the packs field, you can specify one or more packages to use, along with the desired version. If no version is specified, the latest version will be downloaded. If the packages are not publicly available, you need to set the GITHUB\_TOKEN environment variable with a secret that has access to the packages.

Here's an example YAML configuration:

yml

- uses: github/codeql-action/init@v1

with:

packs: scope/pack1,scope/pack2@1.2.3,scope/pack3@~1.2.3

In the example, pack1, pack2, and pack3 are CodeQL query packs that will be downloaded from GitHub, and the default queries or query suites for each pack will be executed. The latest version of pack1 is downloaded, while version 1.2.3 of pack2 is specified, along with the latest version of pack3 compatible with version 1.2.3.

Remember, the CodeQL package-management functionality, including query packs, is currently in beta and subject to change.

To use queries in QL packs with CodeQL code scanning, you can follow these steps:

1. Add a with: queries: entry within the uses: github/codeql-action/init@v1 section of your workflow file.
2. Specify the queries by providing a comma-separated list of paths to the queries. If the queries are in a private repository, you can use the external-repository-token parameter to specify a token that has access to check out the private repository.

Here's an example YAML configuration:

yml

- uses: github/codeql-action/init@v1

with:

queries: COMMA-SEPARATED LIST OF PATHS

external-repository-token: ${{ secrets.ACCESS\_TOKEN }}

Additionally, you can specify query suites in the queries value. Query suites are collections of queries grouped by purpose or language. CodeQL code scanning provides several built-in query suites, including code-scanning, security-extended, and security-and-quality. When you specify a query suite, the CodeQL analysis engine runs the queries contained within the suite in addition to the default set of queries.

If you want to combine queries from both a workflow file and a custom configuration file, you need to ensure that the additional packs or queries specified in your workflow are used instead of those specified in the configuration file. To do this, prefix the value of packs or queries in the workflow file with the + symbol.

Here's an example illustrating the combination of queries:

yml

- uses: github/codeql-action/init@v1

with:

config-file: ./.github/codeql/codeql-config.yml

queries: +security-and-quality,octo-org/python-qlpack/show\_ifs.ql@main

packs: +scope/pack1,scope/pack2@v1.2.3

In the above example, the + symbol ensures that the specified additional packs and queries are used together with any specified in the referenced configuration file (codeql-config.yml).

By leveraging these configuration options, you can customize and combine queries to suit your specific needs in CodeQL code scanning.

# Customize your code scanning workflow with CodeQL - Part 2

To specify additional configuration options for CodeQL code-scanning workflows, you can use custom configuration files. Here are the key points to understand:

1. Custom Configuration File: You can create a custom configuration file to specify additional packs, queries, disable default queries, and define directories to scan during analysis. The file should be written in YAML format.
2. Referencing Configuration File: In your workflow file, use the config-file parameter of the github/codeql-action/init@v1 action to specify the path to your configuration file. For example: - uses: github/codeql-action/init@v1 with: config-file: ./.github/codeql/codeql-config.yml.
3. External Repository Configuration File: You can store the configuration file in an external repository and reference it by using the OWNER/REPOSITORY/FILENAME@BRANCH syntax. If the external repository is private, you'll need to provide an external-repository-token to grant access.
4. Specifying CodeQL Query Packs: In the custom configuration file, you can specify CodeQL query packs using an array format. Each pack can be defined with the pack name and an optional version. For example:

packs:

- scope/pack1

- scope/pack2@v1.2.3

- scope/pack3@~1.2.3

1. Specifying Additional Queries: In the custom configuration file, you can specify additional queries using the queries array. Each element can identify a single query file, a directory containing query files, or a query suite definition file. For example:

queries:

- uses: ./my-basic-queries/example-query.ql

- uses: ./my-advanced-queries

- uses: ./query-suites/my-security-queries.qls

1. Naming Queries: You can optionally provide names for individual queries in the queries array. This can help organize and identify the queries within the configuration file.
2. Other Configuration Options: The custom configuration file also allows you to disable default queries, specify directories to scan (paths), and exclude certain paths from analysis (paths-ignore).

By utilizing a custom configuration file, you have more flexibility in adjusting the behavior of CodeQL code-scanning workflows, including adding packs, specifying queries, and fine-tuning analysis settings.

To customize the behavior of CodeQL code scanning, you can adjust the configuration to disable default queries and specify directories to scan. Here's a breakdown of these options:

1. Disable Default Queries: If you only want to run custom queries and exclude the default security queries, set disable-default-queries: true in the configuration file. This is useful when constructing a custom query suite or to prevent duplicate query execution.
2. Specify Directories to Scan: For interpreted languages like Python, Ruby, and JavaScript/TypeScript, you can restrict code scanning to specific directories by adding a paths array to the configuration file. You can also exclude files in specific directories from analysis by adding a paths-ignore array.

Example:

yaml

paths:

- src

paths-ignore:

- src/node\_modules

- '\*\*/\*.test.js'

Note:

* paths determine the directories to include in the analysis.
* paths-ignore specify directories to exclude from analysis.
* Be careful not to confuse these keywords with the ones used for workflow triggers (on.<push|pull\_request>.paths), which determine when the actions run based on modified code.

1. Compiled Languages: For compiled languages, if you want to limit code scanning to specific directories, you need to configure appropriate build steps in your workflow. The commands to exclude a directory from the build will depend on your build system.
2. Analyzing Portions of a Monorepo: To analyze specific directories when modifying code in a monorepo, you should exclude the directories in your build steps and also use the paths-ignore and paths keywords in your workflow's trigger configuration (on.<push|pull\_request>).

By disabling default queries and specifying directories to scan, you can focus the code scanning process on your specific needs, improving efficiency and reducing noise in the analysis results.

# Use the CodeQL CLI

Here are the key points regarding using the CodeQL CLI to create databases, analyze them, and upload the results to GitHub:

1. CodeQL CLI Commands: The CodeQL CLI allows you to perform various tasks from the command line. The primary commands for generating and uploading results are:
   * database create: Creates a CodeQL database representing the hierarchical structure of supported programming languages in the repository.
   * database analyze: Runs queries to analyze each CodeQL database and produces a SARIF (Static Analysis Results Interchange Format) file summarizing the results.
   * github upload-results: Uploads the SARIF files to GitHub, where the results are associated with a branch or pull request and displayed as code scanning alerts.
2. Command-Line Help: You can use the --help option with any CodeQL CLI command to display its command-line help and learn about its usage and options.
3. Uploading SARIF Data: Uploading SARIF data to display as code scanning results on GitHub is supported for organization-owned repositories with GitHub Advanced Security enabled, as well as public repositories on GitHub.com.
4. Creating CodeQL Databases: To create CodeQL databases for analysis, follow these steps:
   * Check out the code you want to analyze, either the head of a branch or the head commit or merge commit of a pull request.
   * Set up the environment for the codebase, ensuring that dependencies are available.
   * Find the build command for the codebase, usually specified in a configuration file in your CI system.
   * Run the codeql database create command from the root of the repository, specifying the database name, build command, and language identifier(s) for single or multiple supported languages.

Example commands:

* For a single language:

bash

 codeql database create <database> --command <build> --language=<language-identifier>

 For multiple languages:

bash

* codeql database create <database> --command <build> --db-cluster --language=<language-identifier>,<language-identifier>

By using the CodeQL CLI, you can automate the process of creating CodeQL databases, analyzing code, and uploading the results to GitHub for effective code scanning and alert generation.

If you use a containerized build, you need to run the CodeQL CLI inside the container where your build task takes place.

The full list of parameters for the database create command is shown in the following table:

| **Option** | **Required Usage** |
| --- | --- |
| <database> | Specify the name and location of a directory to create for the CodeQL database. The command will fail if you try to overwrite an existing directory. If you also specify --db-cluster, this is the parent directory, and a subdirectory is created for each language analyzed. |
| --language | Specify the identifier for the language to create a database for, one of: cpp, csharp, go, java, javascript, python, and ruby (use Javascript to analyze TypeScript code). When used with --db-cluster, the option accepts a comma-separated list, or can be specified more than once. |
| --command | Recommended. Use to specify the build command or script that invokes the build process for the codebase. Commands are run from the current folder or, where it is defined, from --source-root. Not needed for Python and JavaScript/TypeScript analysis. |
| --db-cluster | Optional. Use in multi-language codebases to generate one database for each language specified by --language. |
| --no-run-unnecessary-builds | Recommended. Use to suppress the build command for languages where the CodeQL CLI does not need to monitor the build (for example, Python and JavaScript/TypeScript). |
| --source-root | Optional. Use if you run the CLI outside the checkout root of the repository. By default, the database create command assumes that the current directory is the root directory for the source files; use this option to specify a different location. |

Here are the key points regarding the examples provided for creating CodeQL databases:

1. Single Language Example:
   * This example creates a CodeQL database for a repository located at /checkouts/example-repo.
   * The JavaScript extractor is used to create a hierarchical representation of JavaScript and TypeScript code in the repository.
   * The resulting database is stored in /codeql-dbs/example-repo.
2. Multiple Languages Example:
   * This example creates two CodeQL databases for a repository located at /checkouts/example-repo-multi.
   * The --db-cluster flag is used to analyze more than one language.
   * The --language flag specifies the languages (Python and C++) for which databases should be created.
   * The --command flag specifies the build command for the codebase (make in this case).
   * The --no-run-unnecessary-builds flag tells the tool to skip the build command for languages where it is not needed (like Python).
   * The resulting databases are stored in the python and cpp subdirectories of /codeql-dbs/example-repo-multi.

Both examples show the initialization, build process, and finalization of the databases. The CLI command outputs the progress and status of the database creation process.

By following these examples, you can create CodeQL databases for your repositories, allowing you to analyze code, run queries, and generate meaningful results for code scanning purposes.

## Analyze a CodeQL database

After creating your CodeQL database, follow these steps to analyze it:

1. Optionally run codeql pack download <packs> to download any CodeQL packs (beta) that you want to run during analysis.
2. Run codeql database analyze on the database and specify which packs and/or queries to use.

Bash

codeql database analyze <database> --format=<format> \

--output=<output> <packs,queries>

Note

If you analyze more than one CodeQL database for a single commit, you must specify a SARIF category for each set of results this command generates. When you upload the results to GitHub, code scanning uses this category to store the results for each language separately. If you forget to do this, each upload overwrites the previous results.

Bash

codeql database analyze <database> --format=<format> \

--sarif-category=<language-specifier> --output=<output> \

<packs,queries>

The full list of parameters for the database analyze command is shown in the following table:

| **Option** | **Required Usage** |
| --- | --- |
| <database> | Specify the path for the directory that contains the CodeQL database to analyze. |
| <packs,queries> | Specify CodeQL packs or queries to run. To run the standard queries used for code scanning, omit this parameter. You can find the other query suites included in the CodeQL CLI bundle in /<extraction-root>/codeql/qlpacks/codeql-<language>/codeql-suites. For information about creating your own query suite, check out [Creating CodeQL query suites](https://codeql.github.com/docs/codeql-cli/creating-codeql-query-suites/) in the documentation for the CodeQL CLI. |
| --format | Specify the format for the results file generated by the command. For upload to GitHub, this should be: sarif-latest. |
| --output | Specify where to save the SARIF results file. |
| --sarif-category | Optional for single database analysis. Required to define the language when you analyze multiple databases for a single commit in a repository. Specify a category to include in the SARIF results file for this analysis. A category is used to distinguish multiple analyses for the same tool and commit, but performed on different languages or different parts of the code. |
| --sarif-add-query-help | Optional. Use if you want to include any available markdown-rendered query help for custom queries used in your analysis. Any query help for custom queries included in the SARIF output will be displayed in the code scanning UI if the relevant query generates an alert. |
| <packs> | Optional. Use if you have downloaded CodeQL query packs and want to run the default queries or query suites specified in the packs. |
| --threads | Optional. Use if you want to use more than one thread to run queries. The default value is 1. You can specify more threads to speed up query execution. To set the number of threads to the number of logical processors, specify 0. |
| --verbose | Optional. Use to get more detailed information about the analysis process and diagnostic data from the database creation process. |

### Basic example

This example analyzes a CodeQL database stored at /codeql-dbs/example-repo and saves the results as a SARIF file: /temp/example-repo-js.sarif. It uses --sarif-category to include extra information in the SARIF file that identifies the results as JavaScript. This is essential when you have more than one CodeQL database to analyze for a single commit in a repository.

Bash

$ codeql database analyze /codeql-dbs/example-repo \

javascript-code-scanning.qls --sarif-category=javascript

--format=sarif-latest --output=/temp/example-repo-js.sarif

> Running queries.

> Compiling query plan for /codeql-home/codeql/qlpacks/

codeql-javascript/AngularJS/DisablingSce.ql.

...

> Shutting down query evaluator.

> Interpreting results.

## Upload results to GitHub

SARIF upload supports a maximum of 5,000 results per upload. Any results over this limit are ignored. If a tool generates too many results, you should update the configuration to focus on results for the most important rules or queries.

For each upload, SARIF upload supports a maximum size of 10 MB for the gzip-compressed SARIF file. Any uploads over this limit will be rejected. If your SARIF file is too large because it contains too many results, you should update the configuration to focus on results for the most important rules or queries.

Before you can upload results to GitHub, you must determine the best way to pass the GitHub App or personal access token you created earlier to the CodeQL CLI. We recommend that you review your CI system's guidance on the secure use of a secret store. The CodeQL CLI supports:

* Passing the token to the CLI via standard input using the --github-auth-stdin option (recommended).
* Saving the secret in the environment variable GITHUB\_TOKEN and running the CLI without including the --github-auth-stdin option.

When you've decided on the most secure and reliable method for your CI server, run codeql github upload-results on each SARIF results file and include --github-auth-stdin unless the token is available in the environment variable GITHUB\_TOKEN.

Bash

echo "$UPLOAD\_TOKEN" | codeql github upload-results --repository=<repository-name> \

--ref=<ref> --commit=<commit> --sarif=<file> \

--github-auth-stdin

The full list of parameters for the github upload-results command is shown in the table below.

| **Option** | **Required Usage** |
| --- | --- |
| --repository | Specify the OWNER/NAME of the repository to which to upload data. The owner must be an organization within an enterprise that has a license for GitHub Advanced Security, and GitHub Advanced Security must be enabled for the repository unless the repository is public. |
| --ref | Specify the name of the ref you checked out and analyzed so that the results can be matched to the correct code. For a branch, use refs/heads/BRANCH-NAME; for the head commit of a pull request, use refs/pulls/NUMBER/head; or for the GitHub-generated merge commit of a pull request, use refs/pulls/NUMBER/merge. |
| --commit | Specify the full SHA of the commit you analyzed. |
| --sarif | Specify the SARIF file to load. |
| --github-auth-stdin | Optional. Use to pass the CLI the GitHub App or personal access token created for authentication with GitHub's REST API via standard input. This isn't needed if the command has access to a GITHUB\_TOKEN environment variable set with this token. |

# Customize languages and builds for code scanning

Here are the key points regarding changing the languages analyzed by CodeQL code scanning and adding custom build steps:

1. Default Language Analysis:
   * CodeQL code scanning supports multiple languages by default, including C/C++, C#, Go, Java, JavaScript/TypeScript, Python, and Ruby (beta).
   * The default CodeQL analysis workflow file includes a build matrix called "language" that lists the languages analyzed in your repository.
   * CodeQL automatically populates this matrix when you enable code scanning, and it optimizes the analysis by running each language analysis in parallel.
2. Modifying the Language Analysis:
   * If your repository contains code in multiple supported languages, you can choose which languages to analyze.
   * By editing the "language" matrix variable in the workflow file, you can add or remove languages from the analysis.
   * If your workflow includes the "language" matrix, CodeQL will only analyze the languages specified in the matrix.
3. Sequential Analysis without Matrix:
   * If your workflow does not contain a matrix called "language," CodeQL is configured to run analysis sequentially.
   * In this case, CodeQL automatically detects and attempts to analyze any supported languages in the repository.
   * To choose specific languages for analysis without using a matrix, you can use the "languages" parameter under the init action in the workflow file.

By customizing the language matrix or using the "languages" parameter, you can control which languages are analyzed by CodeQL code scanning. This allows you to focus the analysis on relevant code and exclude dependencies or code that you don't want to see alerts for. Additionally, you may need to modify the workflow to include custom build steps if your code uses a non-standard build process.

Here are the key points regarding adding custom build steps for code scanning:

1. Autobuild for Supported Compiled Languages:
   * CodeQL provides an autobuild action in the analysis workflow for C/C++, C#, and Java to automatically build the code.
   * This avoids the need to specify explicit build commands for these languages.
   * For Go projects, CodeQL runs a build to set up the project, but it extracts all Go files in the repository, not just the ones built.
2. Adding Custom Build Steps for Compiled Languages:
   * If your repository's C/C++, C#, or Java code has a non-standard build process, the autobuild step may fail.
   * In such cases, you need to remove the autobuild step from the workflow and manually add custom build steps.
   * Uncomment the run step in the workflow and add the appropriate build commands for your repository.
   * The run step executes command-line programs using the operating system's shell, allowing you to customize the build process by modifying or adding commands.
3. Language-Specific Build Commands:
   * If your repository contains multiple compiled languages, you can specify language-specific build commands.
   * For example, if your repository has C/C++, C#, and Java code, and the autobuild works for C/C++ and C# but fails for Java, you can configure the workflow to use autobuild for C/C++ and C# while adding custom build steps for Java.
   * Use conditionals (if statements) in the workflow to differentiate between languages and execute the corresponding build commands.

By customizing the build steps, you can handle non-standard build processes and ensure that the code is properly built before the CodeQL analysis. This allows you to adapt the workflow to the specific requirements of your repository's compiled languages.

# In summary, here are the key points regarding code scanning with CodeQL:

1. Workflow File Configuration:
   * Code scanning with CodeQL utilizes a workflow file to configure the scanning process.
   * The workflow file specifies the location of queries, the languages to analyze, and whether to use autobuild or manual build steps.
   * This configuration allows you to tailor the scanning process to your specific needs.
2. Integration with Third-Party Tools:
   * GitHub supports the integration of third-party scanning and alerting tools in the code scanning process.
   * This flexibility enables you to leverage additional scanning and security tools that complement CodeQL.
3. CodeQL CLI:
   * CodeQL provides a command-line interface (CLI) that allows you to create and analyze databases offline.
   * Using the CLI, you can generate analysis results and upload them to GitHub using the SARIF file format.
   * This offline analysis capability enhances flexibility and enables integration into existing workflows.
4. Automation and Security Benefits:
   * Implementing automated code vulnerability scanning with CodeQL provides significant benefits.
   * It enables automated scanning of code and facilitates generating pull requests to fix vulnerable code.
   * CodeQL's extensive library of queries in multiple languages helps create more secure code with minimal engineering effort.
   * The successful rollout of automated code vulnerability scanning improves developer productivity and enhances the security of the company's product.

By understanding and utilizing CodeQL's capabilities, you can effectively implement and configure automated code vulnerability scanning across your organization, leading to increased productivity and improved security.

# What is GitHub Advanced Security?

GitHub has many features that help you improve and maintain the quality of your code. Some of these are included in all plans, such as dependency graph and Dependabot alerts. Others only run in a limited functionality mode on public repositories. There are also others that require a GitHub Advanced Security license to run on private repositories.

In this unit, you'll learn more about GitHub Advanced Security. You'll also discover what a project with GitHub Advanced Security looks like.

## The GitHub Advanced Security features

The table below summarizes the availability of GitHub Advanced Security features on public and private repositories.

| **Feature** | **Public repository** | **Private repository without Advanced Security** | **Private repository with Advanced Security** |
| --- | --- | --- | --- |
| Code scanning | Yes | No | Yes |
| Secret scanning | Yes (limited functionality only) | No | Yes |
| Dependency review | Yes | No | Yes |
| Security Overview | No | No | Yes |

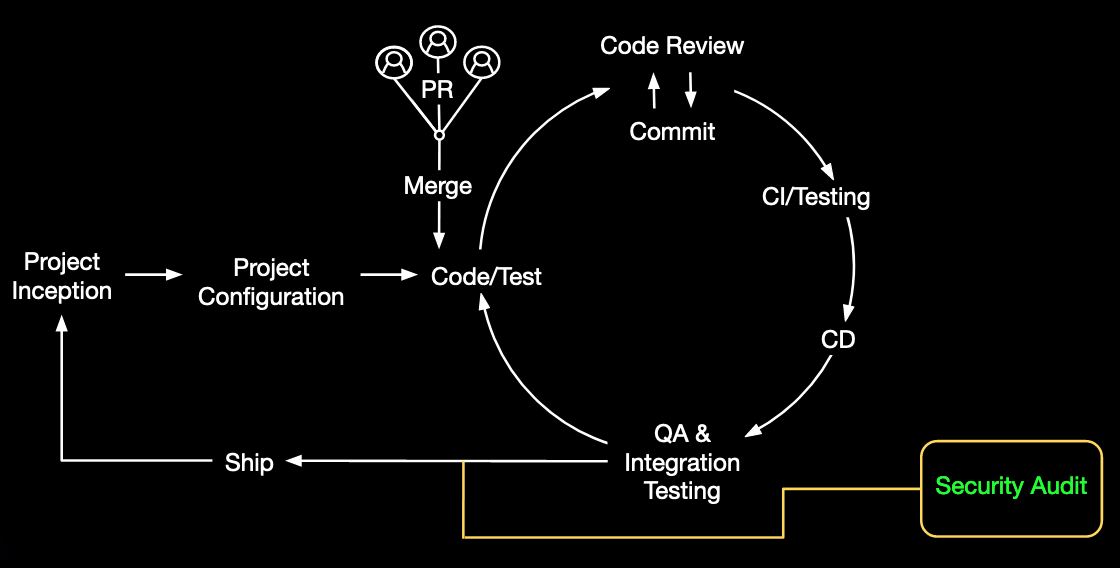
As outlined in the preceding table, all GitHub Advanced Security features except for the Security Overview are enabled by default for all public repositories on GitHub.com. To access these features on private and internal repositories, you need a GitHub enterprise account with a GitHub Advanced Security license.

A GitHub Advanced Security license provides the following additional features for private and internal repositories:

* **Code scanning**: Automatically detect common vulnerabilities and coding errors
* **Secret scanning**: Receive alerts when secrets or keys are checked in, exclude files from scanning, and define up to 100 custom patterns
* **Dependency review**: Show the full impact of changes to dependencies and see details of any vulnerable versions before you merge a pull request
* **Security Overview**: Review the security configuration and alerts for an organization and identify the repositories at greatest risk

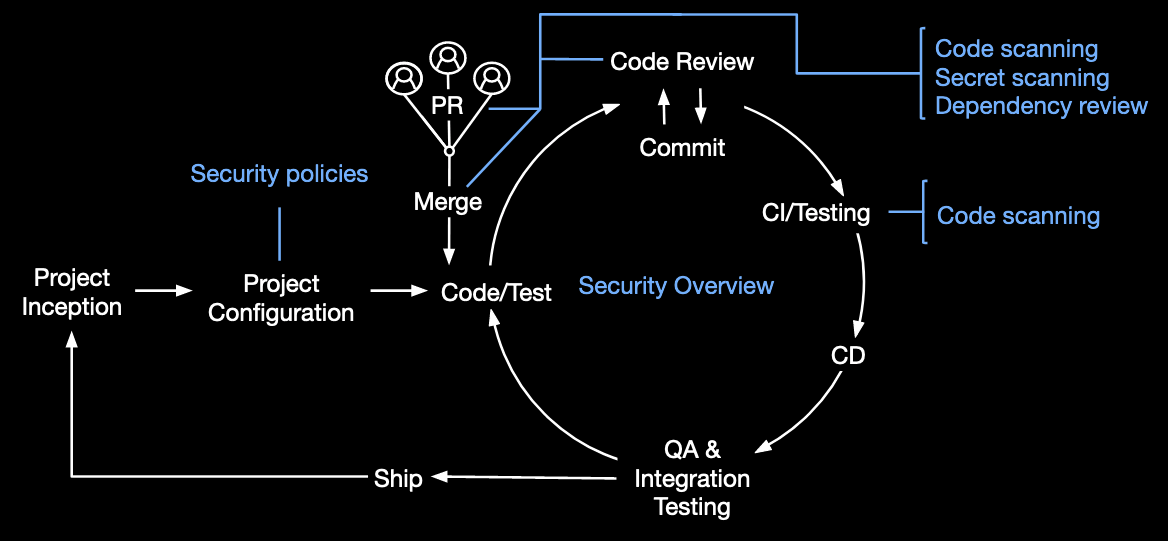
## GitHub Advanced Security in the software development lifecycle

So what difference do the GitHub Advanced Security features make in your software-development lifecycle? Let's have a look at a basic security scenario first.



This example illustrates a traditional "security as a gate" approach in which a single security test or a series of security tests take place during the quality-assurance phase. In this scenario, security usually ends up being a bottleneck to shipping the software. This is what your company wants to fix by shifting security left.

Now let's look at the same software development lifecycle with GitHub Advanced Security.



In this scenario, security is set up right from the beginning via security policies at the project configuration stage. Developers then get alerted of potential security issues at every step of the development process:

* Code scanning scans for potential vulnerabilities and coding errors at every commit and merge.
* Secret scanning also scans for tokens and private keys that might have been accidentally committed at every commit and merge.
* Dependency review keeps track of the project dependency changes and their impact on the project security by comparing the repository manifest files to the databases of known vulnerabilities at every pull request.

In addition, the Security Overview offers administrators a high-level view of the project's security status. This lets administrators identify problematic repositories that require intervention.

When it's time for quality assurance, your code's security has already been reviewed multiple times. There's less chance for a bottleneck right before shipping, and less technical debt.

# Enable GitHub Advanced Security

You learned in the previous unit how GitHub Advanced Security fits into your software-development lifecycle and which features are included for different GitHub plans. It's now time to take the first step towards implementing GitHub Advanced Security in your organization.

In this unit, you'll learn how to enable GitHub Advanced Security at the organization level according to your enterprise plan.

## Enable GitHub Advanced Security for Enterprise Cloud

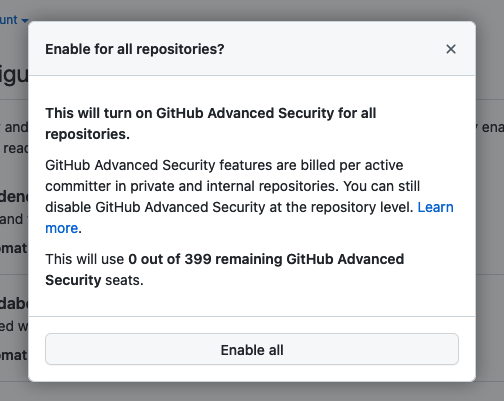
Enabling GitHub Advanced Security at the organization level automatically turns on GitHub Advanced Security for all private and internal repositories in your organization.

Note

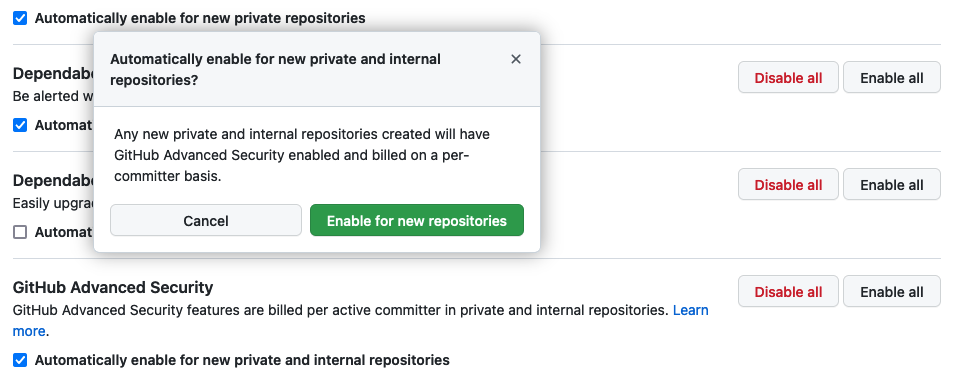
If you enable GitHub Advanced Security in your organization, committers to the organization repositories will use seats on your GitHub Advanced Security license.

Follow the steps below to enable GitHub Advanced Security for all the repositories in your organization:

1. In your organization, navigate to **Settings > Code security and analysis**.
2. Under **Configure security and analysis features**, click the **Enable all** button next to **GitHub Advanced Security**. The control for **GitHub Advanced Security** is disabled if you have no available seats in your GitHub Advanced Security license.
3. Review the impact of enabling Advanced Security on all repositories and click **Enable all**.



If you'd like to automatically enable GitHub Advanced Security on new private and internal repositories added to your organization, select the **Automatically enable for new private and internal repositories** checkbox under **GitHub Advanced Security**, review the impact of enabling Advanced Security on all new private and internal repositories, and click **Enable for new repositories**.



## Enable GitHub Advanced security for Enterprise Server

Before you can enable GitHub Advanced Security for all the repositories in your organization on GitHub Enterprise Server, you must first enable Advanced Security for your GitHub Enterprise Server instance. You can do this in two ways: via the GitHub user interface or via the administrative shell (SSH).

Whether you plan on using the user interface or the administrative shell, make sure you meet the following prerequisites before completing one of the procedures described in the next sections:

* Your license for GitHub Enterprise Server has been upgraded to include GitHub Advanced Security and you've uploaded it to your GitHub Enterprise Server instance.
* You've reviewed the prerequisites for the features you plan to enable:
  + [Prerequisites for code scanning](https://docs.github.com/en/enterprise-server@3.3/admin/advanced-security/configuring-code-scanning-for-your-appliance#prerequisites-for-code-scanning)
  + [Prerequisites for secret scanning](https://docs.github.com/en/enterprise-server@3.3/admin/advanced-security/configuring-secret-scanning-for-your-appliance#prerequisites-for-secret-scanning)
  + [Prerequisites for Dependabot](https://docs.github.com/en/enterprise-server@3.3/admin/configuration/configuring-github-connect/enabling-the-dependency-graph-and-dependabot-alerts-for-your-enterprise)

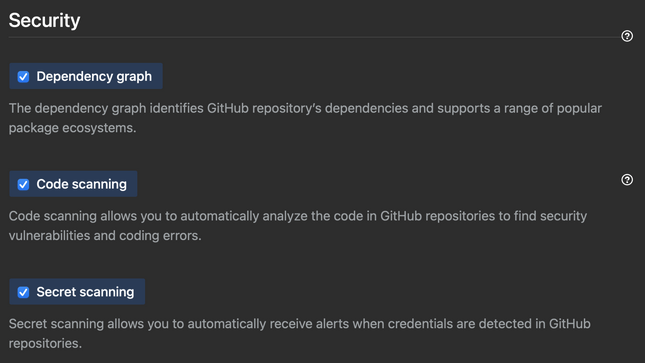
Note

Enabling GitHub Advanced Security features on your Enterprise Server instance will cause user-facing services on GitHub Enterprise Server to restart. You should time this change carefully to minimize downtime for users.

### Via the GitHub user interface

Follow the procedure below to enable the GitHub Advanced Security features on your GitHub Enterprise Server instance:

1. In the **Site admin** page of your GitHub Enterprise Server account, navigate to **Management Console**.
2. In the left sidebar, click **Security**.
3. Under **Security**, select the features that you want to enable.



1. Under the left sidebar, click **Save settings** and wait for the configuration run to complete.

When GitHub Enterprise Server has finished restarting, follow the procedure described in the previous "Enable GitHub Advanced Security for Enterprise Cloud" section to enable GitHub Advanced Security for all the repositories in your organization.

### Via the administrative shell

You can enable or disable features programmatically on your GitHub Enterprise Server instance via the administrative shell and command-line utilities for GitHub Enterprise Server. For example, you can enable any GitHub Advanced Security feature with your infrastructure-as-code tooling when you deploy an instance for staging or disaster recovery.

Follow the procedure below to enable the GitHub Advanced Security features on your Enterprise Server instance:

1. SSH into your GitHub Enterprise Server instance.
2. Enable the GitHub Advanced Security features you want to implement in your instance:
   * To enable code scanning, enter ghe-config app.minio.enabled true followed by ghe-config app.code-scanning.enabled true.
   * To enable secret scanning, enter ghe-config app.secret-scanning.enabled true.
   * To enable the dependency graph, enter ghe-config app.dependency-graph.enabled true.
3. Apply the configuration by entering ghe-config-apply.

When GitHub Enterprise Server has finished restarting, follow the procedure described in the previous "Enable GitHub Advanced Security for Enterprise Cloud" section to enable GitHub Advanced Security for all the repositories in your organization.

# Manage access to GitHub Advanced Security

In the previous unit, you learned how to enable GitHub Advanced Security according to your enterprise plan.

This unit walks you through how to configure GitHub Advanced Security for a project. It explains how to manage access to security alerts and set up security policies at the organization and repository level.

## Manage access to security alerts

When setting up GitHub Advanced Security for a project, you want to make sure the right people in your organization can view and resolve any alerts. The roles and permissions required to view these alerts depend on the type of alert.

This table shows the minimum roles and permissions needed to view each type of alert in the repository's **Security** tab:

| **Type of security alert** | **Roles and permissions needed** |
| --- | --- |
| Code scanning alerts | Write permission on repository |
| Secret scanning alerts | Repository administrators and organization owners |
| Dependabot alerts | Repository administrators and organization owners |

Additionally, repository administrators and organization owners can give secret scanning and Dependabot alert access to users and teams with write permission on their repositories from the repository **Security and analysis** settings.

With the right set of roles and permissions, the developers involved in your security workflow can take the following actions:

* For code scanning alerts: commit corrections to the code, dismiss alerts that do not require any action, or delete alerts to clean up code scanning results.
* For secret scanning alerts: delete detected secrets, create new tokens, and update code that uses the detected secrets, or dismiss alerts that do not require any action.
* For Dependabot alerts: update vulnerable dependencies or dismiss alerts that do not require any action.

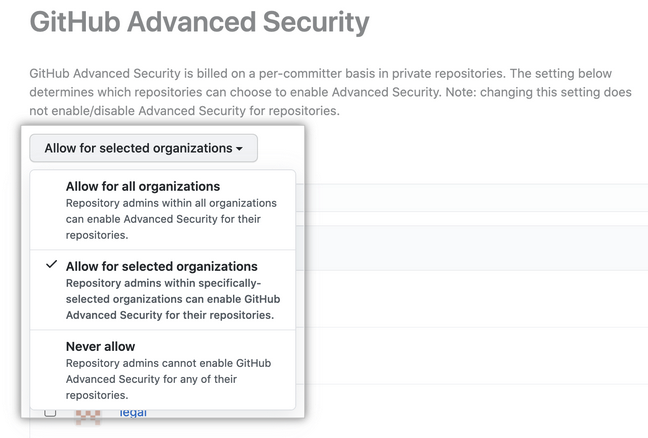
## Set a security policy at the organization level

A good way to make sure that everyone in your organization is using GitHub Advanced Security is to set up a security policy at the organization level. For example, you can set a policy that allows all repository administrators in your organization to enable features for Advanced Security for their repositories.

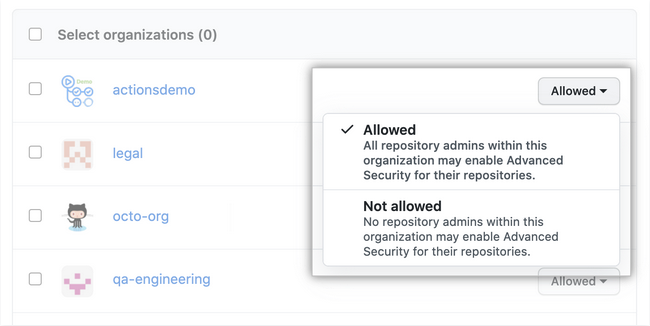
Policies can be configured for all organizations owned by your enterprise account, or for individual organizations that you choose.

Follow these steps to set up a security policy at the organization level:

1. In your enterprise sidebar, navigate to **Policies > Advanced Security**.
2. Under **GitHub Advanced Security**, select the drop-down menu and select a policy for the organizations owned by your enterprise.



1. Optionally, if you chose **Allow for selected organizations** to the right of an organization, select the drop-down menu to allow or disallow Advanced Security for the organization. Note that disallowing Advanced Security for an organization prevents repository administrators from enabling Advanced Security features for additional repositories, but does not disable the features for repositories where the features are already enabled.



Note

Keep in mind that GitHub bills for Advanced Security on a per-committer basis when setting up a policy at the organization level.

## Set a security policy at the repository level

Equally important when setting up a GitHub project is to document how to report security vulnerabilities for the project. To do so, you can add a SECURITY.md file to the project repositories' root, docs, or .github folders. When someone creates an issue in a repository, they will see a link to your project's security policy.

After someone reports a security vulnerability in your project, you can use GitHub Security Advisories to disclose, fix, and publish information about the vulnerability.

Follow these steps to set up a security policy at the repository level:

1. In your repository, navigate to **Security > Security policy**.
2. Click **Start setup**.
3. In the new SECURITY.md file, add information about supported versions of your project and how to report a vulnerability.
4. Commit the change to the repository.

# Manage the GitHub Advanced Security features and alerts

Now that the security for your project is set up, all you have to do is to monitor and manage the GitHub Advanced Security features and alerts for your project.

In this unit, you'll learn how to use the Security Overview to monitor the security risks in your project. This unit also covers how to use the GitHub Advanced Security endpoints to manage the GitHub Advanced Security features and alerts.

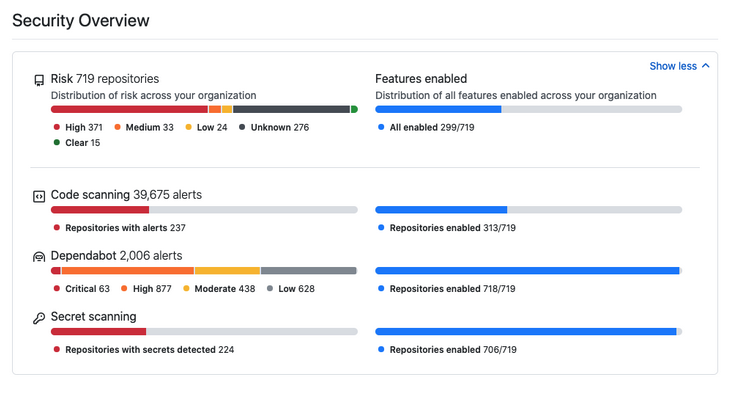
## Use the Security Overview

Note

The Security Overview is currently in beta and subject to change.

The Security Overview is available in the **Security** tab of organizations and repositories. You can use it to get a high-level view of the security status of your organization or to identify problematic repositories that require intervention.

* At the organization level, the Security Overview displays aggregate and repository-specific security information for repositories owned by your organization. You can also filter information per security feature.
* At the team level, the Security Overview displays repository-specific security information for repositories that the team has admin privileges for.
* At the repository level, the Security Overview shows which security features are enabled for the repository and offers the option to configure any available security features not currently in use.



Thanks to its high level of interactivity and numerous filters, the Security Overview is useful for both broad and specific analyses of your organization's security status. For example, you can use it to monitor adoption of features by your organization or by a specific team as you roll out GitHub Advanced Security to your enterprise, or to review all alerts of a specific type and severity level across all repositories in your organization.

## Use the GitHub Advanced Security endpoints

The following table explains what endpoints are available for each Advanced Security feature with links to their documentation.

| **Feature** | **Endpoints** | **Documentation** |
| --- | --- | --- |
| Code scanning | Retrieve and update code scanning alerts from a repository  Create automated reports for code scanning alerts in an organization  Upload analysis results generated using offline code scanning tools | [Code Scanning API](https://docs.github.com/en/rest/reference/code-scanning) |
| Secret scanning | Enable or disable secret scanning for a repository  Retrieve and update secret scanning alerts from a private repository | [Repos API](https://docs.github.com/en/rest/reference/repos#update-a-repository)  [Secret Scanning API](https://docs.github.com/en/rest/reference/secret-scanning) |
| Dependency review | Enable and disable dependency alerts and the dependency graph for a repository  Enable and disable security fixes for a repository  View dependency information | [Repos API](https://docs.github.com/en/rest/reference/repos)  [GraphQL API](https://docs.github.com/en/graphql) |

Note that if you decide to use GitHub Actions to automate your security workflows, it's important to correctly set up the permissions for the GITHUB\_TOKEN used to make authenticated API calls. The GITHUB\_TOKEN has default permissions depending on scope:

| **Scope** | **Default access (permissive)** | **Default access (restricted)** | **Maximum access by forked repos** |
| --- | --- | --- | --- |
| actions | read/write | none | read |
| checks | read/write | none | read |
| contents | read/write | read | read |
| deployments | read/write | none | read |
| id-token | read/write | none | read |
| issues | read/write | none | read |
| metadata | read | read | read |
| packages | read/write | none | read |
| pull-requests | read/write | none | read |
| repository-projects | read/write | none | read |
| security-events | read/write | none | read |
| statuses | read/write | none | read |

You can modify the permissions for the GITHUB\_TOKEN in individual workflow files. If the default permissions for the GITHUB\_TOKEN are restrictive, you may have to increase the permissions to allow some actions and commands to run successfully. If the default permissions are permissive, you can edit the workflow file to remove some permissions from the GITHUB\_TOKEN. As a good security practice, you should grant the GITHUB\_TOKEN the least required access.

You can also use the permissions key in your workflow files to modify permissions for the GITHUB\_TOKEN for an entire workflow or for individual jobs. This allows you to configure the minimum required permissions for a workflow or job. When the permissions key is used, all unspecified permissions are set to no access, with the exception of the metadata scope, which always gets read access.

YAML

name: Create issue on commit

on: [ push ]

jobs:

create\_commit:

runs-on: ubuntu-latest

permissions:

issues: write

steps:

- name: Create issue using REST API

run: |

curl --request POST \

--url http(s)://[hostname]/api/v3/repos/${{ github.repository }}/issues \

--header 'authorization: Bearer ${{ secrets.GITHUB\_TOKEN }}' \

--header 'content-type: application/json' \

--data '{

"title": "Automated issue for commit: ${{ github.sha }}",

"body": "This issue was automatically created by the GitHub Action workflow \*\*${{ github.workflow }}\*\*. \n\n The commit hash was: \_${{ github.sha }}\_."

}' \

--fail

In the preceding example, write access is granted for one scope for a single job.

Additionally, you can use the permissions key to add and remove read permissions for forked repositories, but typically you can't grant write access. The exception to this behavior is if you selected the **Send write tokens to workflows from pull requests** option in the GitHub Actions settings.

# Summary

The goal of this module was to introduce the different GitHub Advanced Security features, show where they fit in your software development lifecycle, and how to leverage them.

Throughout the module, you learned which security features are available with a GitHub Advanced Security license, and at which step these can be integrated in your software development lifecycle. You've also learned how to enable GitHub Advanced Security according to your enterprise plan, and how to set up access to the Advanced Security features across your organizations and repositories.

Finally, you learned how to make the most of GitHub Advanced Security by using the Security Overview and the Advanced Security endpoints. You're now ready to roll out GitHub Advanced Security in your organization!

## Learn more

Here are some links to more information on the topics we discussed in this module:

**White papers:**

* [The complete guide to developer-first application security](https://assets.ctfassets.net/wfutmusr1t3h/397ElOPOMY8H6wSwfFvf4z/06ed44457b6fb3a9bd77134c098749ea/GitHubAdvanced_SecurityEbook.pdf)
* [Adopting and scaling GitHub Advanced Security in your company](https://resources.github.com/whitepapers/Scaling-GitHub-Advanced-Security/)

**Documentation:**

* [GitHub security features](https://docs.github.com/en/code-security/getting-started/github-security-features)
* [About GitHub Advanced Security](https://docs.github.com/en/get-started/learning-about-github/about-github-advanced-security#about-advanced-security-features)
* [About code scanning](https://docs.github.com/en/code-security/code-scanning/automatically-scanning-your-code-for-vulnerabilities-and-errors/about-code-scanning)
* [About secret scanning](https://docs.github.com/en/code-security/secret-scanning/about-secret-scanning)
* [About dependency review](https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-dependency-review)
* [About the security overview](https://docs.github.com/en/code-security/security-overview/about-the-security-overview)
* [Enabling or disabling a feature for all existing repositories](https://docs.github.com/en/organizations/keeping-your-organization-secure/managing-security-and-analysis-settings-for-your-organization#enabling-or-disabling-a-feature-for-all-existing-repositories)
* [Enabling GitHub Advanced Security for your enterprise](https://docs.github.com/en/enterprise-server@3.3/admin/advanced-security/enabling-github-advanced-security-for-your-enterprise)
* [Enforcing a policy for the use of GitHub Advanced Security in your enterprise](https://docs.github.com/en/enterprise-cloud@latest/admin/policies/enforcing-policies-for-your-enterprise/enforcing-policies-for-advanced-security-in-your-enterprise#enforcing-a-policy-for-the-use-of-github-advanced-security-in-your-enterprise)
* [Adding a security policy to your repository](https://docs.github.com/en/code-security/getting-started/adding-a-security-policy-to-your-repository)
* [REST API](https://docs.github.com/en/rest)
* [GraphQL API](https://docs.github.com/en/graphql)
* [Automatic token authentication](https://docs.github.com/en/actions/security-guides/automatic-token-authentication)