Creating CodeQL query suites

In this article

You can create query suites for queries you frequently use in your CodeQL analyses.

GitHub CodeQL is licensed on a per-user basis upon installation. You can use CodeQL only for certain tasks under the license restrictions. For more information, see "About the CodeQL CLI." If you have a GitHub Advanced Security license, you can use CodeQL for automated analysis, continuous integration, and continuous delivery. For more information, see "About GitHub Advanced Security."

About creating CodeQL query suites

CodeQL query suites provide a way of selecting queries, based on their filename, location on disk or in a CodeQL pack, or metadata properties. Create query suites for the queries that you want to frequently use in your CodeQL analyses.

Query suites allow you to pass multiple queries to CodeQL without having to specify the path to each query file individually. Query suite definitions are stored in YAML files with the extension `.qls`. A suite definition is a sequence of instructions, where each instruction is a YAML mapping with (usually) a single key. The instructions are executed in the order they appear in the query suite definition. After all the instructions in the suite definition have been executed, the result is a set of selected queries.

Note: Any custom queries that you want to add to a query suite must be in a CodeQL pack" and contain the correct query metadata. For more information, see "Using custom queries with the CodeQL CLI."

Locating queries to add to a query suite

When creating a query suite, you first need to specify the locations of the queries that you want to select. You can define the location of one or more queries using:

- A `query` instruction—tells CodeQL to look for one or more specified `.ql` files:

```
- query: <path-to-query>
```

The argument must be one or more file paths, relative to the CodeQL pack containing the suite definition.

- A `queries` instruction—tells CodeQL to recursively scan a directory for `.ql` files:

  ```
  - queries: <path-to-subdirectory>
  ```

  The path of the directory must be relative to the root of the CodeQL pack that contains the suite definition file. To find the queries relative to a different CodeQL pack, add a `from` field:

  ```
  - queries: <path-to-subdirectory>
    from: <ql-pack-name>
    version: ^x.y.z
  ```

  The `version` field is optional and specifies a range of compatible versions of this CodeQL pack. If you don't specify a version, then the most recent version of the pack is used.

- A `qlpack` instruction—tells CodeQL to resolve queries in the default suite of the named CodeQL pack:

  ```
  - qlpack: <qlpack-name>
    version: ^x.y.z
  ```

  The default suite of a query pack includes a recommended set of queries inside of that query pack. Not all query packs have a default suite. If the given query pack does not define a default suite, the qlpack instruction will resolve to all of the queries within the pack.

  The `version` field is optional and specifies a range of compatible versions of this CodeQL pack. If you don't specify a version, then the most recent version of the pack is used.

Note: When pathnames appear in query suite definitions, they must always be given with a forward slash, `/`, as a directory separator. This ensures that query suite definitions work on all operating systems.

You must add at least one `query`, `queries`, or `qlpack` instruction to your suite definition, otherwise no queries will be selected. If the suite contains no further instructions, all the queries found from the list of files, in the given directory, or in the named CodeQL pack are selected. If there are further filtering instructions, only queries that match the constraints imposed by those instructions will be selected.

Filtering the queries in a query suite

After you have defined the initial set of queries to add to your suite by specifying `query`, `queries`, or `qlpack` instructions, you can add `include` and `exclude` instructions. These instructions define selection criteria based on specific properties:

- When you execute an `include` instruction on a set of queries, any queries that match your conditions are retained in the selection, and queries that don't match are removed.
- When you execute an `exclude` instructions on a set of queries, any queries that match your conditions are removed from the selection, and queries that don't match

are retained.

The order of your filter instructions is important. The first filter instruction that appears after the locating instructions determines whether the queries are included or excluded by default. If the first filter is an `include`, the initially located queries will only be part of the suite if they match an explicit `include` filter. If the first filter is an `exclude`, the initially located queries are part of the suite unless they are explicitly excluded.

Subsequent instructions are executed in order and the instructions that appear later in the file take precedence over the earlier instructions. So, `include` instructions can be overridden by a later `exclude` instructions that match the same query. Similarly, `exclude`s can be overridden by a later `include`.

For both instructions, the argument is a constraint block—that is, a YAML map representing the constraints. Each constraint is a map entry, where the key is typically a query metadata property. The value can be:

- A single string.
- A `/`-enclosed regular expression.
- A list containing strings, regular expressions, or both.

To match a constraint, a metadata value must match one of the strings or regular expressions. When there is more than one metadata key, each key must be matched. The standard metadata keys available to match on are: `description`, `id`, `kind`, `name`, `tags`, `precision`, and `problem.severity`. For more information about query metadata properties, see "Metadata for CodeQL queries."

In addition to metadata tags, the keys in the constraint block can also be:

- `query filename`—matches on the last path component of the query file name.
- `query path`—matches on the path to the query file relative to its enclosing CodeQL pack.
- `tags contain`—one of the given match strings must match one of the space-separated components of the value of the `@tags` metadata property.
- `tags contain all`—each of the given match strings must match one of the components of the `@tags` metadata property.

Examples of filtering which queries are run

A common use case is to create a query suite that runs all queries in a CodeQL pack, except for a few specific queries that the user does not want to run. In general, we recommend filtering on the query `id`, which is a unique and stable identifier for each query. The following three query suite definitions are semantically identical and filter by the query `id`:

This filter matches all the queries in the default suite of `codeql/cpp-queries`, except for the two queries with the excluded identifiers:

```
- qlpack: codeql/cpp-queries
- exclude:
    id:
      - cpp/cleartext-transmission
      - cpp/cleartext-storage-file
```

In this example, a separate `exclude` instruction is used for each query:

```
- qlpack: codeql/cpp-queries
- exclude:
    id: cpp/cleartext-transmission
- exclude:
    id: cpp/cleartext-storage-file
```

In this example, a regular expression excludes the same two queries. It would also exclude any future queries added to the suite with identifiers that begin: `cpp/cleartext-`:

```
- qlpack: codeql/cpp-queries
- exclude:
    id:
      - /^cpp\/cleartext-.*/
```

To define a suite that selects all queries in the default suite of the `codeql/cpp-queries` CodeQL pack, and then refines them to only include security queries, use:

```
- qlpack: codeql/cpp-queries
- include:
    tags contain: security
```

To define a suite that selects all queries with `@kind problem` and `@precision high` from the `my-custom-queries` directory, use:

```
- queries: my-custom-queries
- include:
    kind: problem
    precision: very-high
```

Note that the following query suite definition behaves differently from the definition above. This definition selects queries that are `@kind problem` *or* are `@precision very-high`:

```
- queries: my-custom-queries
- include:
    kind: problem
- include:
    precision: very-high
```

To create a suite that selects all queries with `@kind problem` from the `my-custom-queries` directory except those with `@problem.severity recommendation`, use:

```
- queries: my-custom-queries
- include:
    kind: problem
- exclude:
    problem.severity: recommendation
```

To create a suite that selects all queries with `@tag security` and `@precision high` or `very-high` from the `codeql/cpp-queries` CodeQL pack, use:

```
- queries: .
  from: codeql/cpp-queries
- include:
    tags contain: security
    precision:
    - high
    - very-high
```

Note: You can use the `codeql resolve queries /path/to/suite.qls` command to see which queries are selected by a query suite definition. For more information, see "resolve queries."

Reusing existing query suite definitions

Existing query suite definitions can be reused by specifying:

- An `import` instruction—adds the queries selected by a previously defined `.qls` file to the current suite:

  ```
  - import: <path-to-query-suite>
  ```

  The path to the imported suite must be relative to the CodeQL pack containing the current suite definition. If the imported query suite is in a different QL pack you can use:

  ```
  - import: <path-to-query-suite>
    from: <ql-pack>
    version: ^x.y.z
  ```

  The `version` field is optional and specifies a range of compatible versions of this CodeQL pack. If you don't specify a version, then the most recent version of the pack is used.

  Queries added using an `import` instruction can be filtered using subsequent `exclude` instructions.

- An `apply` instruction—adds all of the instructions from a previously defined `.qls` file to the current suite. The instructions in the applied `.qls` file are executed as if they appear in place of `apply`. Any `include` and `exclude` instructions from the applied suite also act on queries added by any earlier instructions:

  ```
  - apply: <path-to-query-suite>
  ```

  The `apply` instruction can also be used to apply a set of reusable conditions, saved in a `.yml` file, to multiple query definitions. For more information, see the examples below.

Reusability Examples

To use the same conditions in multiple query suite definitions, create a separate `.yml` file containing your instructions. For example, save the following in a file called `reusable-instructions.yml`:

```
- include:
    kind:
    - problem
    - path-problem
    tags contain: security
    precision:
    - high
    - very-high
```

Add `reusable-instructions.yml` to the same CodeQL pack as your current query suite. Then, in one or more query suites, use the `apply` instruction to apply the reusable instructions to the current suite. For example:

```
- queries: queries/cpp/custom
- apply: reusable-instructions.yml
```

This will filter the queries in `queries/cpp/custom` to only include those that match the reusable conditions.

You can also create a suite definition using `reusable-instructions.yml` on queries in a different CodeQL pack. If the `.qls` file is in the same CodeQL pack as the queries, you can add a `from` field immediately after the `apply` instruction:

```
# load queries from the default suite of my-org/my-other-custom-queries
- qlpack: my-org/my-other-custom-queries

# apply the reusable instructions from the my-org/my-custom-instructions CodeQL
pack
- apply: reusable-instructions.yml
  from: my-org/my-custom-instructions
  version: ^1.2.3 # optional
```

A common use case for an `import` instruction is to apply a further filter to queries from another query suite. For example, this suite will further filter the `cpp-security-and-quality` suite and exclude `low` and `medium` precision queries:

```
- import: codeql-suites/cpp-security-and-quality.qls
  from: codeql/cpp-queries
- exclude:
    precision:
      - low
      - medium
```

If you want to `include` queries imported from another suite, the syntax is a little different:

```
- import: codeql-suites/cpp-security-and-quality.qls
  from: codeql/cpp-queries
- exclude: {}
- include:
    precision:
      - very-high
      - high
```

Notice the empty `exclude` instruction. This is required to ensure that the subsequent `include` instruction is able to filter queries from the imported suite.

Naming a query suite

You can provide a name for your query suite by specifying a `description` instruction:

```
- description: <name-of-query-suite>
```

This value is displayed when you run resolve queries, if the suite is added to a "well-known" directory. For more information, see "Specifying well-known query suites."

Saving a query suite

Save your query suite in a file with a `.qls` extension and add it to a CodeQL pack. For more information, see "Customizing analysis with CodeQL packs."

Specifying well-known query suites

You can use CodeQL packs to declare directories that contain "well-known" query suites. You can use "well-known" query suites on the command line by referring to their file name, without providing their full path. This gives you a simple way of specifying a set of queries, without needing to search inside CodeQL packs and distributions. To declare a directory that contains "well-known" query suites, add the directory to the `suites` property in the `qlpack.yml` file at the root of your CodeQL pack. For more information, see "Customizing analysis with CodeQL packs."

Using query suites with CodeQL

You can specify query suites on the command line for any command that accepts `.qls` files. For example, you can compile the queries selected by a suite definition using `query compile`, or use the queries in an analysis using `database analyze`. For more information about analyzing CodeQL databases, see "Analyzing your code with CodeQL queries."

Further reading

- "CodeQL queries"