Coverity Desktop Analysis 2020.12: User Guide

Table of Contents

Chapter 1. Desktop Analysis overview

Table of Contents

Coverity Desktop Analysis is a feature that allows source code to be checked for defects as it is written, on the developer's metaphorical "desktop".

This guide provides the following information for Coverity Desktop Analysis:

- Quick start Desktop Analysis guide.
- Key concepts and use cases that describe typical end-to-end workflows.
- Description of Desktop Analysis options for advanced use.
- Troubleshooting for common issues.

1.1. Choosing a user interface

Coverity Desktop Analysis tools can be used from the command line, an IDE, or from various text editors. The following sections provide instructions for getting started with Desktop Analysis using your preferred development environment.

Eclipse IDE

Desktop Analysis is offered as a plug-in to the Eclipse, WindRiver, and QNX Momentics IDEs. For complete use instructions, see *Coverity 2020.12 for Eclipse, Wind River Workbench, and QNX Momentics: User Guide*.

Visual Studio

Desktop Analysis is offered as a plug-in to the Visual Studio IDE. For complete use instructions, see *Coverity Desktop 2020.12 for Microsoft Visual Studio: User Guide*.

IntelliJ IDEA

Desktop Analysis is offered as a plug-in to IntelliJ IDEA and Android Studio. For complete use instructions, see *Coverity Desktop 2020.12 for IntelliJ IDEA and Android Studio: User Guide*.

Command line

Desktop Analysis can be used to analyze your code for issues directly on the command line. See Chapter 2, *Desktop Analysis quick start guide* to get started.

Editor or other IDE

Desktop Analysis can be configured to run from your editor or IDE without the Coverity Desktop plug-in. Complete the steps in Chapter 2, *Desktop Analysis quick start guide*, then see Section 2.4, “Running Desktop Analysis from an editor or IDE” for configuration and use instructions.

1.2. Requirements

Desktop Analysis runs on all of the platforms and compilers supported for use with Coverity Analysis, except AIX.

Desktop Analysis requires a minimum of 1.5 GB of RAM. We recommend at least 4 GB of RAM and 4 processor cores.

Chapter 2. Desktop Analysis quick start guide

Table of Contents

The following sections highlight the steps for configuring and running Coverity Desktop Analysis.

2.1. What you will need

Make sure that you have access to the following:

Coverity Connect stream name

Coverity Connect must be configured in advance to provide analysis summary data to Desktop Analysis users. Desktop Analysis relies on a "reference snapshot" to provide analysis summary data. This requires an initial analysis and commit to a stream enabled for Desktop Analysis. You will need the name of that stream.

See *Coverity Platform 2020.12 User and Administrator Guide* for information on configuring Coverity Connect for use with Desktop Analysis.

Coverity Connect access information

- Host name
- Port number and type (HTTP or HTTPS)
- User name
- Password

Source code to analyze

You can use Desktop Analysis with C, C++, C#, Java, JavaScript, PHP, Python, Ruby, Scala, or Swift code.

For C, C++, C#, Java, Scala, and Swift, you need to know the command to build the software, and which compilers your project uses.

Coverity Analysis installer and license file

The Coverity Analysis installer and license file are available from the Coverity customer portal, and may also be made available from the Coverity Connect downloads page by your Coverity Connect administrator. This is the recommended configuration.

Instructions for adding the installer and license file to the Coverity Connect downloads page are found in the *Coverity Platform 2020.12 User and Administrator Guide*.

2.2. Installing Coverity Analysis tools

Coverity Desktop Analysis is included in the Coverity Analysis tool set. To install it, run the installer and follow the on-screen prompts. When asked which installation mode to run, choose "Desktop Analysis" mode. For additional information, see the *Coverity 2020.12 Installation and Deployment Guide*.

2.3. Desktop Analysis on the command line

The following sections will walk you through initial configuration and use of Desktop Analysis from the command line. The process is broken down into the following general tasks:

- Create `coverity.conf`
- Add the "bin" directory to your PATH
- Run `cov-run-desktop --setup`
- Analyze a specific source file
- Review returned defects

2.3.1. Creating `coverity.conf`

The `coverity.conf` file contains settings required to run desktop analysis. It should be checked into your Source Code Management (SCM) repository, usually in the root directory. The `cov-run-desktop` command searches upward in the file tree from wherever it is invoked to find this file. If your code base already has a `coverity.conf` file, skip this section.

This file should contain at least:

- The host name and port number of the Coverity Connect server.
- The name of the stream on Coverity Connect that is associated with the particular code base.
- Shell command lines to perform a clean build (compile) of the code.

The configuration file uses JSON syntax, but unlike standard JSON, comments *are* allowed.

Create a `coverity.conf` file with the following contents:

```
{
  "type": "Coverity configuration",
  "format_version": 1,
  "settings": {
    "server": {
      "host": "coverity-server-name"                 // CC host name
    },
    "stream": "stream-name",                         // CC stream name
    "cov_run_desktop": {
      "build_cmd": ["make", "-j", "$(num_cores)"],   // build command
```

```
        "clean_cmd": ["make", "clean"]                 // clean command
      }
    }
  }
```

In place of "`coverity-server-name`", enter the host name of the machine running Coverity Connect. If it is using a non-default port number, for example 1234, add another attribute `"port" : "1234"` to the server section.

In place of "`stream-name`", enter the name of the stream that contains the reference snapshots that desktop analysis should use to get interprocedural summary information from.

In place of "`make`" and "`make clean`", enter shell commands to build and clean your code base. If you are not going to analyze any compiled code, use `"build_cmd": []` in your `coverity.conf` file to indicate that you do not have a build to capture. Even if you have a build command, the `clean` command can be set to `[]` or omitted, but you should include it if the build command omits calling the compiler when object files are newer than source files, as with `make`. Note that separate shell command words must be written as separate strings.

See Section 5.14, “coverity.conf file format” for full details on the structure and meaning of the configuration file. Note in particular the `compiler_configurations` element if you are using a compiler other than GNU C/C++ as "gcc" or "g++", Microsoft C/C++/C#, or Oracle Java.

2.3.2. Adding the "bin" to your PATH

In order to run `cov-run-desktop` from the command line, it is recommended (although not required) to add the "bin" directory of the `cov-analysis` installation directory to your PATH.

On Windows:

1. Go to Control Panel → System Properties → Advanced → Environment Variables.

2. Select *PATH* and click **Edit**.

3. Append a semi-colon (;) followed by the "bin" directory path.

4. Click **OK** twice to save.

5. Start a new command shell window.

On Unix, edit your shell startup file (for example, "`$HOME/.bashrc`") and add a line like:

```
PATH=$PATH:/path/to/cov-analysis/bin
```

where "`/path/to/cov-analysis`" is the directory where you chose to install the Coverity Analysis tools. Then save that file and start a new shell.

2.3.3. Run `cov-run-desktop --setup`

Run the following command:

```
> cov-run-desktop --setup
```

This will first attempt to create an "authentication key". It prompts for your Coverity Connect password. Once the key is created, you will not have to enter your password again to use desktop analysis.

It then configures your compilers and filesystem capture. The default configuration works for GNU, Microsoft, Oracle Java, and Clang compilers, as well as all filesystem capture languages that Coverity supports on your platform. See `compiler_configurations` if you are using another compiler or need customizations such as different file extensions for filesystem capture.

Finally, it runs the `clean_cmd` and `build_cmd` in `coverity.conf` (if non-empty) to capture a clean build of your compiled code so that Coverity tools know how to compile all of your compiled source files. You will need to run `cov-run-desktop --setup` again if new source files are added, or the command lines to compile them change. Interpreted code does not depend on compiler invocations, so will be captured automatically later, as needed.

To capture compiled code not captured during `--setup`, such as adding new files to your project, use `cov-run-desktop --build <build_cmd>`, where `<build_cmd>` only has to compile the uncaptured files. You can avoid this manual step by configuring a script to compile specific files on demand (see Section 5.6, “Compiling files on demand” for details). However, if you change how files are compiled, such as defined preprocessor directives, and need Coverity to capture those changes, you must re-capture a build using `cov-run-desktop --build`.

2.3.4. Analyzing a source file

To analyze a source file, run:

```
> cov-run-desktop <source_file_name>
```

If everything is configured correctly, this will parse the named source file and analyze it for defects, using a previously created reference snapshot with interprocedural summaries in order to understand how that file relates to the code around it.

`<source_file_name>` is the path and file name of the source file to be analyzed. You can pass as many files as you want to analyze.

For C and C++, `<source_file_name>` is typically the name of a `.c` or `.cpp` file. If you want to analyze a header file, see Section 5.5, “Analyzing non-primary source files (C/C++)”.

When the analysis completes, it prints the defects it has detected to the console, less any defects removed by the filtering options. The `cov-run-desktop` command reference explains how to adjust the output, including formatting and filtering.

Note

Depending on what file you choose, there may not be any defects detected. You can log into Coverity Connect, select a file that has defects detected by the central analysis, and pass it to `cov-run-desktop` to confirm those defects.

After your initial analysis, you can make additional changes to the source file and run `cov-run-desktop` again on the same file, or any other file in the project.

2.3.5. Reviewing defects

Once `cov-run-desktop` has completed, the console will display the standard output, which will consist of a completion message followed by a list of all defects found. For example, see the code below, which contains two defects:

```
int nullBug(int *p)
{
  if (p != NULL) {
    // ...
  }
  // ...
  return *p;              // oops, 'p' might be NULL here
}

int compareBug(char const *a, char const *b)
{
  if (strcmp == 0) {      // oops, forgot to actually call 'strcmp'
    return 1;
  }
  else {
    return 0;
  }
}
```

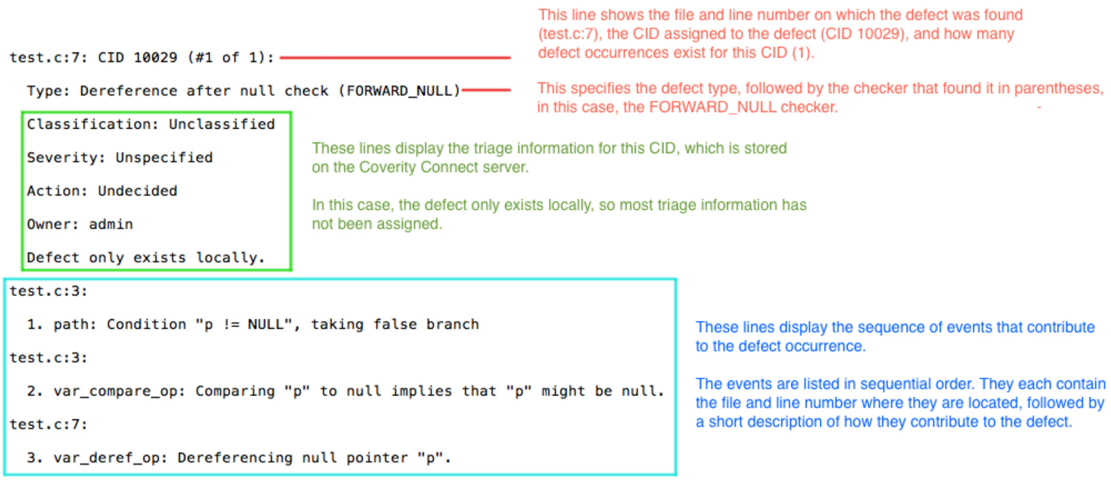
When run on this code sample, `cov-run-desktop` will return the following console output:

```
Detected 2 defect occurrences that pass the filter criteria.

test.c:7: CID 10029 (#1 of 1):
  Type: Dereference after null check (FORWARD_NULL)
  Classification: Unclassified
  Severity: Unspecified
  Action: Undecided
  Owner: admin
  Defect only exists locally.
test.c:3:
  1. path: Condition "p != NULL", taking false branch
test.c:3:
  2. var_compare_op: Comparing "p" to null implies that "p" might be null.
test.c:7:
  3. var_deref_op: Dereferencing null pointer "p".

test.c:12: CID 10028 (#1 of 1):
  Type: Function address comparison (BAD_COMPARE)
  Classification: Unclassified
  Severity: Unspecified
  Action: Undecided
  Owner: admin
  Defect only exists locally.
test.c:12:
  func_conv: This implicit conversion to a function pointer is suspicious: "strcmp ==
 NULL".
```

```
test.c:12:
  remediation: Did you intend to call "strcmp"?

cov-run-desktop took 12.4 seconds.
```

As the first line indicates, this output shows two defect occurrences (separated by a blank line), and a final message which says how long `cov-run-desktop` took to run - in this case, 12.4 seconds. Figure 2.1, “Defect output explained” provides detail on the output for an individual defect occurrence.

Figure 2.1. Defect output explained



```
test.c:7: CID 10029 (#1 of 1):

  Type: Dereference after null check (FORWARD_NULL)

  Classification: Unclassified

  Severity: Unspecified

  Action: Undecided

  Owner: admin

  Defect only exists locally.

test.c:3:

  1. path: Condition "p != NULL", taking false branch

test.c:3:

  2. var_compare_op: Comparing "p" to null implies that "p" might be null.

test.c:7:

  3. var_deref_op: Dereferencing null pointer "p".
```

This line shows the file and line number on which the defect was found (test.c:7), the CID assigned to the defect (CID 10029), and how many defect occurrences exist for this CID (1).

This specifies the defect type, followed by the checker that found it in parentheses, in this case, the FORWARD_NULL checker.

These lines display the triage information for this CID, which is stored on the Coverity Connect server.

In this case, the defect only exists locally, so most triage information has not been assigned.

These lines display the sequence of events that contribute to the defect occurrence.

The events are listed in sequential order. They each contain the file and line number where they are located, followed by a short description of how they contribute to the defect.

For additional information about checkers, and the defects they produce, see the *Coverity 2020.12 Checker Reference* ↗.

2.4. Running Desktop Analysis from an editor or IDE

It is possible to run Desktop Analysis from your editor or integrated development environment (IDE). Usage and configuration options are provided for some of the most common platforms in the following sections. Some basic configuration is needed prior to running local analysis from your editor, such as creating a `coverity.conf` configuration file, setting up your PATH, and running the `--setup` command. Be sure to complete these steps, as described in Section 2.4.3, “Desktop Analysis in the IDE”, prior to running Desktop Analysis from your editor or IDE.

Note

It is highly encouraged that one person (like an admin or team lead) create the `coverity.conf` configuration file and check it into your Source Code Management (SCM) repository, usually in the root directory. This will allow all users to benefit from preconfigured settings.

7

If you are an Eclipse, Visual Studio®, IntelliJ, or Android Studio user, it is recommended that you use the Coverity Desktop plug-in for use with Desktop Analysis. More information on the plug-ins can be found in their respective usage guides.

For Eclipse, Wind River, or QNX, see the *Coverity 2020.12 for Eclipse, Wind River Workbench, and QNX Momentics: User Guide*.

For Visual Studio, see the *Coverity Desktop 2020.12 for Microsoft Visual Studio: User Guide*.

For IntelliJ or Android Studio, see the *Coverity Desktop 2020.12 for IntelliJ IDEA and Android Studio: User Guide*.

2.4.1. Using Desktop Analysis with Emacs

To run Desktop Analysis from within Emacs, arrange to run `cov-run-desktop` such that Emacs thinks it is running a compiler. It will then parse the defect output as compiler syntax errors and navigate to them accordingly. The Coverity Analysis tools include an example `elisp` function to do that:

`<install_dir_ca>/doc/examples/desktop-scripts/coverity.el`

To use this script, first load it. For example, copy it to the `.emacs.d` subdirectory of your home directory, then add to your `.emacs` or `.emacs.d/init.el` file the following lines:

```
(if (file-readable-p "~/.emacs.d/coverity.el")
  (load-file "~/.emacs.d/coverity.el"))
```

Then, restart emacs. Now, navigate to a source file in a directory underneath where `coverity.conf` is installed and type "`M-x coverity`" (Alt-X, "coverity", Enter) or hit `M-F9` (Alt F9).

This command will invoke `cov-run-desktop`, passing the current file's name as an argument, in the directory where the open file is located. `cov-run-desktop` will then search upward in the directory tree for a `coverity.conf` file, which must exist for this command to work, as it contains required information such as the server connection parameters. The console output of `cov-run-desktop` will then be parsed by Emacs the same way as compiler syntax errors. Use "`M-g n`" (Alt-G, "n") and "`M-g p`" to navigate forward and backward through the "errors", which are in reality defects and events detected by Coverity Desktop Analysis.

2.4.2. Using Desktop Analysis with Vim

As with Emacs, to use Desktop Analysis with Vim, invoke `cov-run-desktop` so Vim thinks it is running a compiler. One minor difference is that `cov-run-desktop` should be passed the "`--text-output-style oneline`" switch, as that produces an output format more suited to the way Vim displays and navigates syntax errors. An example script that does this is included with the Coverity Analysis tools:

`<install_dir_ca>/doc/examples/desktop-scripts/coverity.vimrc`

To load this into Vim, copy that file to someplace like the `.vim` subdirectory of your home directory, and add the following lines to your `.vimrc` file:

```
" load Coverity command
let coverity_vimrc = $HOME . "/.vim/coverity.vimrc"
if filereadable(coverity_vimrc)
  execute "source " . fnameescape(coverity_vimrc)
endif
```

Then start or restart Vim, and the "`:Coverity`" command will be defined. Invoke it when editing a source file that is underneath the directory where `coverity.conf` is installed to analyze that file. Use "`:copen`" to see the error list, "`:cnext`" and "`:cprev`" to navigate, and "`:cclose`" to close the error list, among other commands. Web search for "vim quickfix" for additional commands and documentation.

2.4.3. Desktop Analysis in the IDE

When Desktop Analysis is set up to work in an IDE, the `coverity.conf` configuration file can be used to deploy various settings to Desktop Analysis IDE users.

Note: Visual Studio currently supports all of the settings below, while the other IDEs only support setting up the following settings pages:

- Coverity Connect
- Stream
- SCM

The Visual Studio plugin takes advantage of the ability to specify settings on a per analysis configuration basis and also allows you to specify specific checker settings or extend_checker settings. This, along with user specified variables allow users to deploy very complex consistent settings to all users by creating the coverity.conf configuration file and check it into your Source Code Management (SCM) repository, usually in the root directory

Below are some `coverity.conf` examples that can be helpful for Desktop Analysis in the IDE plugin.

Example 1: Configure a default custom tools location, while allowing users to override it by specifying the `cov_install_dir` in their user-specific `coverity.conf` file if they need to:

```
{
    "type": "Coverity configuration",
    "format_version": 1,
    "format_minor_version": 5,
    "variables": {
        "cov_install_dir": "C:\\Program Files\\Coverity\\cov-analysis-8.7.0"
    },
    "settings": {
        "known_installations": [
            {
                version: "$(version)",
                platform: "$(platform)",
                kind: "cov-analysis",
                directory: "$(var:cov_install_dir)"
            }
        ]
    }
}
```

Example 2: To pre-configure some checkers for two named analysis configurations with specific settings (this can be expanded to customize settings for different analysis configuration names), use something like:

```
{
    "type": "Coverity configuration",
    "format_version": 1,
    "format_minor_version": 5,
    "settings": {
        "server": {
            "host": "<host name>",
            "port": <portnumber>,
            "ssl": true
        },
        "stream": "teststream",
        "scm": {
            "scm": "git"
        },
        "conditional_settings": [
            {
                "when": {
                    "configurations": [
                        "Default",
                        "Alternate AC 1"
                    ]
                },
                "settings": {
                    "cov_run_desktop": {
                        "reference_snapshot": "latest",
                        "checkers": {
                            "UNREACHABLE": {
                                "enabled": true
                            },
                            "IDENTICAL_BRANCHES": {
                                "enabled": false
                            }
                        }
                    }
                }
            }
        ]
    }
}
```

Chapter 3. Coverity Connect integration

Table of Contents

In order to keep the most current analysis data, Desktop Analysis frequently interacts with the Coverity Connect server. When you call `cov-run-desktop`, it receives the latest triage data and analysis settings from your reference stream and snapshot, respectively.

3.1. Using an authentication key

It is recommended that you create an authentication key to provide your Coverity Connect credentials to `cov-run-desktop`. The authentication key allows you to run Desktop Analysis without having to specify your password each time. To create an authentication key, run the following command:

```
> cov-run-desktop --create-auth-key
```

The command above gets the Coverity Connect host information from `coverity.conf`, prompts for your Coverity Connect password, and writes the authentication key to a default directory:

- Windows directory: `%APPDATA%/Coverity/authkeys`
- Unix directory: `$HOME/.coverity/authkeys`

The "`cov-run-desktop --setup`" command will create an authentication key if one has not already been created.

Once it has been created, every `cov-run-desktop` invocation that needs an authentication key will get it from that location.

Note

The ability to create an authentication key is not supported on the following platforms:

- AIX, FreeBSD, Linux-ia64, NetBSD

As a workaround, it is possible to create the key on another platform and then copy the authentication key file to a different machine.

3.2. Disconnected mode

It is possible to use Desktop Analysis when you are offline or otherwise unable to connect to the Coverity Connect server. This is done by passing the `--disconnected` option to `cov-run-desktop`. This will cause all options related to Coverity Connect (`--stream`, `--reference-snapshot`, etc.) to be ignored.

When in disconnected mode, `cov-run-desktop` will not be able to download summary or triage data from Coverity Connect, and instead relies on any cached data from previous local analyses. As a result, analysis summaries may be out of date or nonexistent, and the results of Desktop Analysis could be less accurate.

Chapter 4. Administering Desktop Analysis

Table of Contents

4.1. Creating `coverity.conf` for a code base

In order to set up a code base so that developers can easily run desktop analysis, it is recommended to create a file called `coverity.conf` and put it into the root directory of the source code management (SCM) repository. The file contains configuration information that will be shared by all developers working on that code base.

At minimum, it must contain:

- The host name and port number of the Coverity Connect server

  You can also use a url in place of a host name. For example, `https://example.com/coverity` or `https://cimpop:8080`.

- The name of the Coverity Connect stream that is associated with the code base

It is recommended that it also contain:

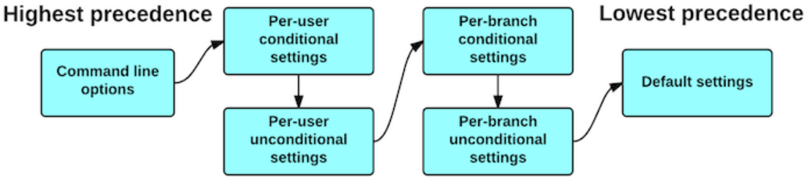
- The name of the SCM system in use
- Shell command lines to build (compile) and clean

The configuration file uses JSON syntax. An example file containing all the required and recommended elements is below:

```
  {
    "type": "Coverity configuration",
    "format_version": 1,
    "settings": {
      "server": {
        "host": "coverity-server.example.com"      // host name
      },
      "stream": "codebase-branch",                 // stream name
      "scm": {
        "scm": "git"                               // SCM name
      },
      "cov_run_desktop": {
        "build_cmd": ["make"],                     // build command
        "clean_cmd": ["make", "clean"],            // clean command
        "reference_snapshot": "scm"
      }
    }
```

```
}
```

Whenever `cov-run-desktop` is invoked, it will search upward from its invocation directory to find a `coverity.conf` file. If it finds one, the settings from that file are used unless the user overrides them with a command line option. The order of precedence for selecting which settings `cov-run-desktop` will use is illustrated below:


Highest precedence
Command line options
Per-user conditional settings
Per-user unconditional settings
Per-branch conditional settings
Per-branch unconditional settings
Lowest precedence
Default settings

As shown in the diagram, any settings specified on the command line take precedence over the settings in any `coverity.conf` file. From there, the per-user `coverity.conf` file (generally located in `$HOME/.coverity` or `%APPDATA%/Coverity`) is processed, with any *true* conditional settings taking precedence over top-level, "unconditional" settings. The conditional settings are processed in order, such that earlier conditional settings (whose condition is true) take precedence over later conditional settings. Next, the per-branch `coverity.conf` file (located in the code base directory) is processed. Finally, default values will be assigned to any settings that are not specified on the command line or configured in a `coverity.conf` file.

See Section 5.14, “coverity.conf file format” for full details on the structure and meaning of the configuration file. Note in particular Section 5.14.9, “CompilerConfiguration” if you are using a compiler *other than* GNU C/C++ as "gcc" or "g++", Microsoft C/C++/C#, or Oracle Java.

4.2. Editing the coverity.conf configuration file with a JSON code editor

You can edit the `coverity.conf` configuration file by setting up a JSON editor with a schema. The schema enables the editor to correctly display the configuration file, and allows validation and auto-completion of text. The Visual Studio Code editor from Microsoft is easy to use for this purpose. Visual Studio 2013 or newer can also be used.

Download the Visual Studio Code editor installer from Microsoft (code.visualstudio.com), and install the application. It is available for most platforms.

To configure Visual Studio Code:

1. Go to File → Preferences → User Settings.
2. Add a `files.associations` property to associate `coverity.conf` files with the JSON editor.
3. Add a `json.schemas` property to associate the `coverity.conf` schema with `coverity.conf` files.

A minimal settings file is shown below. The server host and port need to be modified to point to the Coverity Connect server the users are working with.

```
{
    "files.associations": {
        "coverity.conf": "json"
    },

    "json.schemas": [
        {
            "fileMatch": [
                "coverity.conf"
            ],
            "url": "http://coverity-server-name:8443/schemas/
coverity.conf.schema.json"
        }
    ]
}
```

If users are working with Visual Studio 2013 or newer, they may add a `$schema` property to the `coverity.conf` file itself, pointing to the schema location. Note that including this property in the `coverity.conf` file will prevent the file from working with releases prior to 2020.12, but this should not be a problem for users already working with 8.7.0 or newer. The `$schema` property enables JSON support in Visual Studio 2013 and Visual Studio Code, even if user settings are not modified as described above. A minimal `coverity.conf` file using this feature is shown here:

```
{
    "$schema": "http://coverity-server-name:8443/schemas/coverity.conf.schema.json",
    "type": "Coverity configuration",
    "format_version": 1,
    "format_minor_version": 5,
    "settings": {
        // ...
    }
}
```

4.3. Add `data-coverity` to the SCM exclusion list

By default, local analysis intermediate data is stored in a directory called "`data-coverity`", a sub-directory of whichever directory contains the `coverity.conf` file. Since these files should not be checked in to the Source Code Management (SCM) system, you should add the name "data-coverity" to the list of directories excluded by the SCM (often stored in an "ignore" file).

You can change where intermediate data is stored, either by editing `coverity.conf` or by overriding its settings on the `cov-run-desktop` command line. However, you should not put the intermediate data from different branches of a single code base into the same directory, because the analysis may be confused by the presence of different versions of the same artifacts. Furthermore, avoid putting this data someplace that will be removed when the build is cleaned, because then users would have to re-run build capture after cleaning the build.

Chapter 5. Desktop Analysis Reference

Table of Contents

5.1. Concepts

This section provides conceptual definitions for Coverity Desktop Analysis. These concepts are integral to more advanced usage of the Desktop Analysis tools.

Build Capture

Build capture is the process of running a build under monitoring software that records information about the build. To parse code accurately and consistently with your build settings, build capture is required to analyze compiled code with `cov-run-desktop`. Build capture is typically done in a separate step before analysis but can be done automatically, on demand, if configured for that.

Code Version Date

In contrast to the snapshot date, which is when a snapshot was committed (copied) into Coverity Connect, the code version date is the date when the code was checked in to the Source Code Management (SCM) system.

Desktop Analysis

Desktop Analysis is used by developers to locally analyze a small subset of the code base for which they are responsible. The developer can run Desktop Analysis on a specific file (or set of files) after making changes. By specifying a smaller number of files (or translation units) to analyze, the Desktop Analysis user gets analysis results back in a significantly shorter time.

In order to accurately locate the more complex static analysis issues, Desktop Analysis uses previously reported analysis summary information for any code that isn't included in the analysis run. Desktop Analysis connects to the Coverity Connect server to download analysis summary data.

Because the Coverity Connect server may contain several streams for organizing different versions of the code base, Desktop Analysis users must also specify a reference stream from which to retrieve summary information. This ensures that the analysis summaries are the most relevant to the code being analyzed. See *Coverity Platform 2020.12 User and Administrator Guide* for more information on stream configuration.

Primary Source File

For C and C++, the primary source file (PSF) of a translation unit is the file that is directly named on the compilation command line when compiling that translation unit. Header files, and other files that are read in the context of a PSF, are called "non-PSF" files. For Java, C#, and filesystem capture languages, every source file is considered a PSF of its own translation unit, even if it is not explicitly listed on a compiler command line.

Reference Snapshot

A snapshot is a set of source files and the defects that were detected in those source files. A reference snapshot additionally contains analysis summaries that are used during desktop analysis to supplement the information derived from locally compiled code, and analysis options that are used as the default starting point for local analysis options.

Static analysis

Static analysis is the process by which Coverity Analysis tools scan your source code for bugs without having to run the code. Static analysis makes use of various "checkers" to scan for hard-to-find software issues. Some of these checkers are relatively simple, finding issues based on patterns in local files. Other checkers combine information gleaned from across the code base to find more complex issues.

This process is generally executed on a periodic basis, analyzing the full code base. The results of the analysis are then committed to the central Coverity Connect server, where they can be viewed by the developer responsible.

Stream

Snapshots are organized into streams. All snapshots in a stream should be from the same code base and branch, be compiled for the same target platform, and analyzed with the same options.

Translation Unit

A translation unit is smallest unit of re-compilation for a compiler. In Java, for example, it is possible to recompile any single source file in a project, so a translation unit corresponds to a source file. In C/C++, a translation unit is a primary source file and all the other files it includes. Coverity tools capture artifacts for analysis as a collection of translation units, including those input files used to generate object code, as well as other files and information that form the context of the compilation. For example, in Java, this context includes bytecode files in the classpath. In C/C++, this context includes platform information about the compiler and defined preprocessor directives. Coverity Desktop build capture typically only records the primary source file and its corresponding command line, although there is an option (`--record-with-source`) to additionally record the supporting files.

Triage

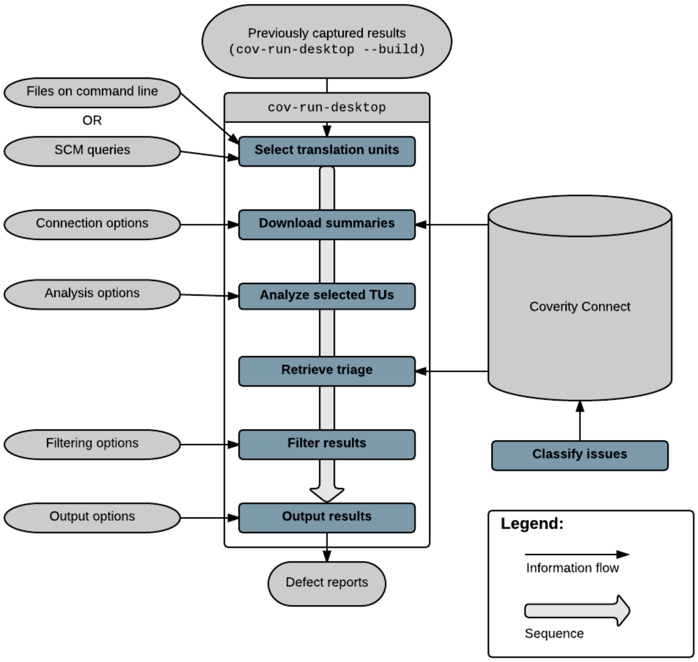
Triage is user-specified metadata associated with a defect, or more precisely, an equivalence class of defects that all share certain essential characteristics. The equivalence class is named by the numeric CID, the "Coverity ID". The most important triage attribute is *Classification*, which includes the settings of "False Positive" and "Intentional", both of which effectively mean the user wants to suppress the defect from that point forward.

All defects detected by desktop analysis have a triage record on the Coverity Connect server (unless using disconnected mode), even if they have not been committed as part of a snapshot.

5.2. The `cov-run-desktop` command

The `cov-run-desktop` command is the primary interface for using Desktop Analysis. The following diagram illustrates the main processes performed by `cov-run-desktop`, with explanations below:

Figure 5.1. `cov-run-desktop`



Select translation units to analyze

In general, `cov-run-desktop` analyzes the files passed explicitly at the end of the command line (`cov-run-desktop --dir <intermediate_directory> file1 file2`). However, it is also possible to use your Source Control Management system (SCM) to decide which files should be analyzed, using the `--analyze-scm-modified` option. See Section 5.4, “Source code management system integration” for more information. Another option is to analyze previously captured source, by using the `--analyze-captured-source` option.

Download analysis summaries from Coverity Connect

In order to generate results that are both fast and accurate, Desktop Analysis depends on analysis summaries from Coverity Connect to provide relevant information about your source code. When you call `cov-run-desktop`, use the `--stream` option to specify a reference stream from which to pull summary data.

Optionally, you can also use the `--reference-snapshot` command to specify a particular snapshot within your reference stream from which to pull analysis summaries. If unspecified, Desktop

Analysis uses the default snapshot (the snapshot created closest to the creation date of your intermediate directory). For more information, see `--reference-snapshot`.

Analyze selected translation units

With analysis summaries downloaded from Coverity Connect, Desktop Analysis can analyze individual files and understand their impact on the rest of your source code, without having to analyze all of the other files therein. Additionally, `cov-run-desktop` collects summaries of locally analyzed code. This keeps the analysis summary information as current as possible for subsequent local analyses on different source files.

Retrieve triage data for analysis results

Aside from analysis summaries, Coverity Connect also provides triage data for any previously known issues also found by Desktop Analysis. `cov-run-desktop` retrieves the triage information for these issues and displays it in the output.

Filter locally found issues

To return only the most relevant analysis results, Desktop Analysis automatically filters out any issues with Classification of "False Positive" or "Intentional," issues with Action set to "Ignore," those found in non-primary source files, as well as any issues found in third party code. Third party code is identified as any that belongs to a component with a file rule that contains "`[Tt]hird.*[Pp]arty`."

There are additional options that can be used to further specify the list of issues returned, or to remove the default filters from your results (--no-default-triage-filters).

See output and filtering options for more information.

Output returned analysis data

The `cov-run-desktop` command has options to specify the desired format and order of the defect output. By default, the list of defects is printed as text output that mimics compiler syntax errors. However, there are several options for customizing the defect output. These include changing the text output style, using JSON output format, and customizing the sort order of your defect list. The JSON output format is described in detail in Section 5.13, “Desktop Analysis JSON output syntax”.

See output and filtering options for information on other options related to Desktop Analysis output.

Classify locally found issues

Once you have received the results of a local analysis, some of the issues found may not be actual Bugs - they might be False Positives or Intentional. In order to mark them as such, you can run the `cov-run-desktop` command again, using the `--mark-fp` (False Positive) and/or `--mark-int` options accordingly. Each of these options specify a particular CID, and Classify it as either False Positive or Intentional, with a text comment explaining the classification. See `--mark-[fp | int]` for details.

5.3. Defining the analysis scope

The group of files specified for analysis by `cov-run-desktop` is known as the analysis scope. This can be specified explicitly (by passing the files to the `cov-run-desktop` command line) , determined by querying your SCM (using the `--analyze-scm-modified` option), or by using the source captured from previous builds (using the `--analyze-captured-source` option).

By default, Desktop Analysis does not return any defects found in files outside of the analysis scope. This means that, unless directly specified, defects in headers and other non-primary source files will not be found by `cov-run-desktop`. Additionally, previously detected defects in such files will be suppressed in the `cov-run-desktop` output. This behavior can be disabled by setting the `cov-run-desktop --confine-to-scope --confine-to-scope` option to "false".

5.4. Source code management system integration

Desktop Analysis can be integrated with your source code management (SCM) system to help determine which files have been recently modified, and thus require local analysis. Using the `cov-run-desktop` option, `--analyze-scm-modified`, will query your SCM to determine which of your source files have been modified locally, and then proceed with Desktop Analysis on those files.

You can also use your SCM to decide which reference snapshot to use with Desktop Analysis. By passing the `--reference-snapshot scm` option to `cov-run-desktop`, or setting "`settings.cov_run_desktop.reference_snapshot`" to "`scm`" in `coverity.conf`, Desktop Analysis will determine the creation date and time of your current code version, and use that as the date and time for which to determine the appropriate reference snapshot.

Note

When using `--analyze-scm-modified` or `--reference-snapshot scm`, you must also pass the `--scm` option, or set "`settings.scm.scm`" in `coverity.conf`, in order to specify which SCM you are using. For more information on these and other SCM-related options, see `cov-run-desktop`.

5.5. Analyzing non-primary source files (C/C++)

During the build capture step of Desktop Analysis, the `cov-run-desktop --build` command records only primary source files (PSF) and corresponding command lines, ignoring headers and other supporting files (non-primary source files).

If you would like to run Desktop Analysis on a header file, or other non-PSF, there are two ways to do so:

Analyze the non-PSF along with a PSF that includes it

Since the non-PSFs in your project are not recorded during the build, `cov-run-desktop` will not have the necessary context to analyze them when specified. To fix this, you can specify the non-PSF, along with a PSF which includes it, to the `cov-run-desktop` command.

For example, if you want to analyze a header file (`header.h`) which is included by the PSF `file1.c`, you would specify both files in the command:

```
> cov-run-desktop [OPTIONS] file1.c header.h
```

Recapture the build with `--record-with-source`

If you plan to analyze non-PSFs frequently, it may be beneficial to repeat the build capture step, using `cov-run-desktop --build --record-with-source`. This will record all of the source files in your project, including non-PSFs. Thus, each non-PSF in your project will be directly available to `cov-run-desktop`, as long as it is included by a primary source file.

When a header file (or other non-PSF) is specified, `cov-run-desktop` will search for a PSF that includes the header. If one is found, it will analyze the header as it was compiled in the context of the selected PSF (if more than one PSF includes the header, the one with the alphabetically first file name is selected). If no PSF is found to include the specified header, `cov-run-desktop` halts with an error.

Note that the `--record-with-source` option will slow down the build capture by 10-50%. This should be used only if you intend to regularly run analysis on non-PSFs.

5.6. Compiling files on demand

To analyze compiled code, `cov-run-desktop` must have seen a compiler invocation for that code so that it knows the proper context for parsing and analyzing the code. This is normally accomplished by capturing builds with `cov-run-desktop --setup` or `--build`, but an alternative is to configure a script for building specific files on demand, using the `specific_files_build_cmd` option in `coverity.conf`.

When `cov-run-desktop` is asked to analyze source files that are not already part of a captured compilation and are not captured with filesystem capture, `cov-run-desktop` will attempt to *auto-compile* them with the `specific_files_build_cmd`, if it is set. If successful, `cov-run-desktop` proceeds with analysis as usual, without the need for a specific build step by the user. Some with a configured `specific_files_build_cmd` might choose to skip build capture during `--setup`, with `--skip-build` or `build_cmd: []` in `coverity.conf`.

Consider a `coverity.conf` configuration example for auto-compilation:

```
cov_run_desktop: {
    "specific_files_build_cmd": [ "python", "scripts/compile-specific-files.py", "--
response-file=$(response_file_utf8)" ],
    "specific_files_regex": "[.](c|cpp)$"
}
```

First, like other commands specified in `coverity.conf`, arguments comprising the build command are specified as strings in a JSON array. `cov-run-desktop` will construct a temporary text file listing the files to compile, one per line, and the path to that file will be substituted for use of special variable `$(response_file_utf8)` or `$(response_file_platform_default)`. The script should read the list from that file, using either UTF-8 or platform default character encoding, depending on the variable used. Only one use of a response file special variable is allowed in the `specific_files_build_cmd`. If no response file special variable is used, files to compile will instead be appended as command line arguments. This is more convenient for writing scripts, but is not recommended because it risks exceeding OS limits on command line length when compiling many files on demand, especially on Windows.

The example runs the system's `python` command and gives it a relative path to a Python script, `scripts/compile-specific-files.py`. This relative path works because `cov-run-desktop` always executes the `specific_files_build_cmd` with the current directory set to `$(code_base_dir)`. On Unix-like systems, with proper permissions, and so on, it is possible to execute the `.py` script directly, but specifying the interpreter for the script is more portable to Windows platforms.

Some requirements for `specific_files_build_cmd` scripts:

- The script must be able to handle relative or absolute paths (in platform-native format, system default encoding), where relative paths will be relative to `$(code_base_dir)`, the working directory for the command.

- It must also handle more than one file specified for compilation, perhaps by grouping related files together for compilation or perhaps by naively building one at a time.

- It must not skip compilation if the object file is already newer than the source file. For builds driven by `make`, the script will likely need to `touch` source files before invoking `make`, or use the `-W` option of `make`, to force invocation of the compiler.

The `specific_files_regex` option is highly recommended with `specific_files_build_cmd` because it reduces or improves errors when attempting to analyze an uncaptured file that would fail when passed to the `specific_files_build_cmd`. In the example above, we know that our script only handles C and C++ source files, with `.c` and `.cpp` extensions in our code base. Thus, the regex ensures `specific_files_build_cmd` is only invoked with such files. This option works in concert with other regex filtering options, such as `restrict_modified_file_regex`.

Non-primary source files, mostly C and C++ header files, present a challenge for auto-compilation, but the `specific_files_regex` can make this work in some cases. For example, suppose `foo.c` includes `foo.h`, nothing has been captured, and `cov-run-desktop` is using the above configuration. Although `cov-run-desktop foo.h` will fail because we do not know how to capture a `.h` file automatically, `cov-run-desktop foo.c foo.h` will succeed because auto-compiling `foo.c` will enable `cov-run-desktop` to analyze `foo.h`, as a non-primary file included by `foo.c`.

Note

Advanced note: `specific_files_build_cmd` can directly invoke `cov-translate` on applicable compiler commands instead of actually running those compiler commands. This can be more efficient by avoiding compiler invocations, but it might be more difficult to set up and to debug.

5.7. Running security checkers with cov-run-desktop

By default, Desktop Analysis enables the same set of checkers as `cov-analyze` (minus any that do not support desktop analysis); in particular, it does not enable most security checkers by default. You can enable them with the `--webapp-security` and `--android-security` options to the `cov-run-desktop` command.

Not all security checkers support Desktop Analysis. Unsupported checkers will be disabled and have warnings issued to indicate that the checker requires running in `--whole-program` mode. See Section 5.8, “Analyzing your whole program with cov-run-desktop” for more information.

5.8. Analyzing your whole program with cov-run-desktop

If you emit your entire project and analyze it using `cov-run-desktop` and the `--whole-program` option, you can run some checkers that are otherwise not enabled for Desktop Analysis. Furthermore, other checkers, such as some Java/.NET security checkers, are more effective if they can analyze your whole project at once.

Before you run `cov-run-desktop` with the `--whole-program` option, be sure that you have emitted all the source files in your project. The `--whole-program` option tells `cov-run-desktop` that you'd like to do a deeper (but slower) analysis. It also tells `cov-run-desktop` that you've emitted your entire project, so that it is reasonable to enable checkers that need to examine the entire project. (A `--whole-program` analysis can yield poor results if it selects only a subset of files from the project.)

Note

If you are using the Coverity Desktop plug-in for Eclipse to perform Desktop Analysis, this behavior can be configured through the *Analysis Configurations* dialog. See the *Coverity 2020.12 for Eclipse, Wind River Workbench, and QNX Momentics: User Guide* for more information.

If you are using the Coverity Desktop plug-in for Visual Studio to perform Desktop Analysis, this behavior can be configured through the *Analysis Configurations* dialog. See the *Coverity Desktop 2020.12 for Microsoft Visual Studio: User Guide* for more information.

If you are using the Coverity Desktop plug-in for IntelliJ IDEA and Android Studio to perform Desktop Analysis, this behavior can be configured through the *Analysis Configurations* dialog. See the *Coverity Desktop 2020.12 for IntelliJ IDEA and Android Studio: User Guide* for more information.

5.9. Local defect owner assignment

Defects found by Desktop Analysis are automatically assigned to the user running `cov-run-desktop` (see the `--user` option). This applies to all newly detected defects (CIDs), which only exist locally and do not have a previously assigned owner.

This behavior is controlled by the `--set-new-defect-owner` option. When set to true, as it is by default, automatic owner assignment will take place for all new defects, as long as there are 100 or fewer local-only defects found. Because the owner assignment adds to the total runtime of the `cov-run-desktop` command, it is limited to 100 new defects by default. This means that if `cov-run-desktop` finds more than 100 local-only defects, no owner assignment will take place. However, if you require owner assignment on a larger number of local defects, the limit can be adjusted with the `--set-new-defect-owner-limit` option.

5.10. Desktop Analysis usage tracking

Defects that are first discovered by Desktop Analysis have their "First Detected By" attribute set to *Preview*. Defects found through other means will have different values for this attribute (*Snapshot* or *API*, for example). This distinction allows you to specifically track the number of defects discovered by Desktop Analysis. To do so, complete the following steps:

1. Log in to Coverity Connect.
2. Make sure you have the correct project open, and navigate to an Issues: Project Scope view ("All In Project" for example).

3. Click the "gear" icon to edit the view settings, and open the *Columns* tab.
4. Enable the *First Detected By* and *First Snapshot* columns.
5. Open the *Filters* tab, and click to open the *First Detected By* filter.
6. Check the box for *Preview*. This will filter all of the defects in the project to display only those *first* found by Desktop Analysis.

Once the view has been filtered, note in particular the *First Snapshot* column. This displays the snapshot when this CID was first committed to Coverity Connect. If this column is blank, that means that the defect was found by `cov-run-desktop` and not yet committed to Coverity Connect (this could be because the defect was fixed immediately after being discovered by Desktop Analysis, or simply because it was discovered very recently and has yet to be committed).

Note

When `cov-run-desktop` finds a new defect, it assigns it a CID and sets its owner to the current user. However, defect occurrence information is only communicated to Coverity Connect when the CID is committed with `cov-commit-defects`. This means that when you attempt to open a defect in Coverity Connect that has not yet been committed, you will see an error explaining that "No further information for this CID exists on Coverity Connect."

5.11. Reasons for results differences

When running Desktop Analysis, you may encounter analysis results that differ somewhat from those found by your central analysis server. This section is intended to highlight several of the most common causes for result disparities, and suggest ways to lessen or eliminate those differences. Note that this is not an exhaustive list of all scenarios in which results differences may be observed.

Missing summary data for adjacent functions

Desktop Analysis relies on summary data from the Coverity Connect server. This is used to provide information on functions and files within your project that are outside of the analysis scope. If `cov-run-desktop` does not have access to the most current summary data, your local analysis results may differ significantly from those found by central analysis.

The main causes for missing summary data are as follows:

- The central analysis did not capture and/or commit analysis summary data (controlled by the `cov-analyze --export-summaries` option).
- `cov-run-desktop` failed to properly download analysis summaries from the Coverity Connect server - possibly because the connection timed out.
- If `cov-run-desktop` is run in disconnected mode (specified with the `--disconnected` option), it won't attempt any connection with the server.

Target configuration differences

It is possible that the downloaded summaries were created while analyzing the code when it was configured for a different target (e.g. "debug" versus "release" builds, separate operating systems, etc). To prevent this issue, make sure that you associate your Desktop Analysis run with a reference stream that matches, or is sufficiently close to, your local target configuration.

Code version skew

Results differences may be present if the downloaded summary data is for a different version of the source code than what you have checked out. This can happen if you are not using the most recent reference snapshot for retrieving analysis summaries.

To prevent this issue, it is recommended that you set the `cov-run-desktop --reference-snapshot` option to `scm`, so that the selected reference snapshot is as close as possible to your current code version. See `--reference-snapshot` for more information.

Dependencies outside the analysis scope

In some configurations, changes made locally in one file can affect other code outside the analysis scope. When these dependencies flow out of, and then back into, the analysis scope, Desktop Analysis accuracy may suffer.

For example, suppose a project contains three functions, A, B, and C, where A calls B, and B calls C. If A and C are analyzed locally but B is not, the analysis will have to rely on summary data for function B. As such, any local changes to C will not be reflected in that summary data, and the analysis results may be inaccurate or incomplete.

Problem reporting on anonymous types

Certain checkers, including CHECKED_RETURN, occasionally fail to report defects on functions that involve anonymous types when called by `cov-run-desktop`. Consequently, if your project contains any defects in such functions, they may only be found during the full central analysis.

Insufficient information for dynamic language call graph

In dynamic languages supporting interprocedural analysis (currently JavaScript), in order to properly apply downloaded function summaries, the analysis call graph computation must process some "extra files" in addition to those selected in the analysis scope. This is handled automatically by `cov-run-desktop` based on information in the downloaded summary batch about the relationships between source files in the Central analysis. If a source file that you are analyzing is new, or is newly calling into another source file or library, that information might not be found by `cov-run-desktop`, and you can improve the accuracy of the analysis by manually including those files that are called into in the analysis scope.

Whole program checkers

For performance reasons, Desktop Analysis does not enable security checkers or other "whole program" checkers by default. This may cause discrepancies in your Central and Desktop Analysis results. See Section 5.7, “Running security checkers with cov-run-desktop” for more information.

5.12. Effect of component permissions

Desktop Analysis does not respect user permissions for components in Coverity Connect. This means that when a user does not have access to a particular component, Desktop Analysis will still be aware of any defects associated with that component. This could cause some confusion, as the "present in reference" attribute, along with all other triage information, is displayed correctly in the output. For example, if the issue was found centrally, "present in reference" will be set to "true" even if the issue is in an inaccessible component.

A common cause of this issue may be if an organization wants to suppress all issues from third party code. In this case, the system administrator could place all third party code into a private component, so that other Coverity Connect users would not see third party issues. However, in this scenario, those issues would still be present in the Desktop Analysis output. One way to ensure that this behavior does not occur is for the Coverity Connect administrator to set the "Action" attribute for all third party issues to "Ignore". This will cause Desktop Analysis to filter out those defects by default.

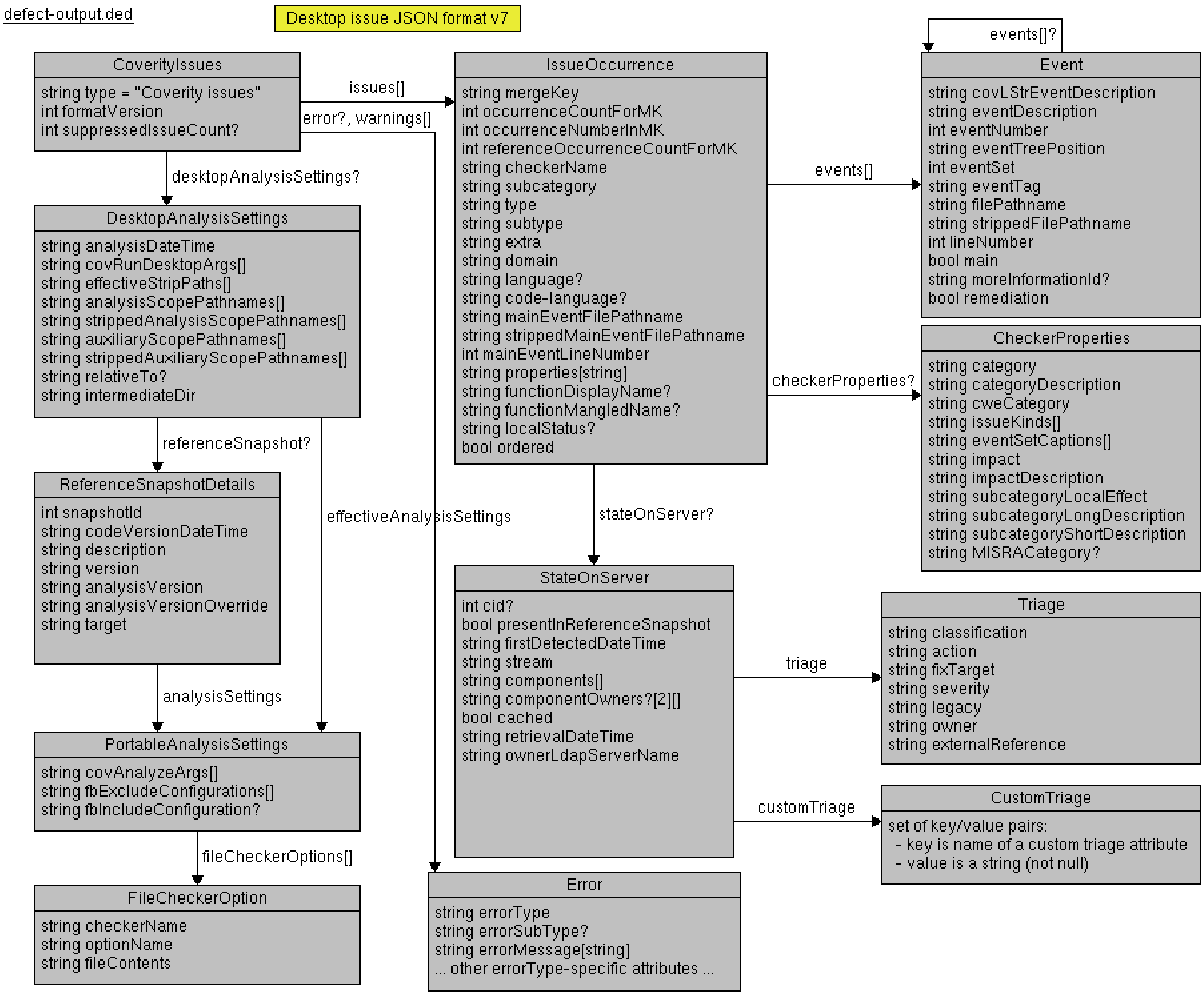
5.13. Desktop Analysis JSON output syntax

When specified with the `--json-output-v7` option, `cov-run-desktop` will write its output to a JSON file. This section describes the objects and attributes of the JSON output in detail.

Note

`--json-output-v1` through `--json-output-v6` are supported for backward compatibility. It is recommended that you use `--json-output-v7` in order to see the most complete set of information.

The structure of the JSON output (v7) is represented below.



5.13.1. CoverityIssues

This is the outermost object of the JSON output file, acting as an envelope with type and version information. It contains the following elements:

type: `<string>`

A string identifying the type of object contained in the file in order to assist tools in diagnosing cases where the wrong data or version is provided as input.

formatVersion: `<int>`
    An integer identifying the version of the file/object format.

suppressedIssueCount: `<int>`
    The number of issues that were detected but not output because of filters.

issues: `<[IssueOccurrence]>`
    A list of issue occurrences, each of which has several attributes described in the IssueOccurrence section. The order of this list is determined by the `--sort` option.

desktopAnalysisSettings: `<DesktopAnalysisSettings>`
    A list of the settings used to perform the desktop analysis. The values are described in the DesktopAnalysisSettings section.

error?: `<Error>`
    Indicates that we encountered an error; in this case, issues will be empty.

warnings[]: `<Error>`
    Indicates warnings, if any, that were produced by this run. Added in version 6. (The attribute `hadMissingSummaries` was subsumed by `warnings`, and was removed in version 6.)

5.13.2. IssueOccurrence

This object represents a single issue occurrence produced by the analysis. It contains the following elements for describing the occurrence:

mergeKey: `<string>`
    A 32 character hexadecimal string representing the hash of several issue details.

occurrenceCountForMK: `<int>`
    The number of occurrences in the parent CoverityIssues object that have the same `mergeKey` value.

occurrenceNumberInMK: `<int>`
    A number between 1 and `occurrenceCountForMK`, unique for this merge key.

referenceOccurrenceCountForMK: `<int>`
    The number of occurrences in the reference snapshot for containing CoverityIssues objects that have the same mergeKey value. The number of occurrences in the reference snapshot may be different than the number of occurrences in the local analysis reported in `occurrenceCountForMK`. Added in version 5.

checkerName: `<string>`
    The name of the checker that produced the issue.

subcategory: `<string>`
    The checker-specific subcategory which indicates the kind of issue found in this occurrence.

type: `<string>`
    The kind of issue, corresponding roughly to the checker. Added in version 7.

subtype: `<string>`
: A subdivision of `type`, denoting a sub-type of the issue type. Added in version 7.

code-language: `<string>`
: Represents the programming language of the source file. Added in version 7.

extra: `<string>`
: A checker-specific merging discriminator.

domain: `<string>`
: The analysis "domain," as carried in the defect XML.

language: `<string>`
: The name of the programming language containing the main event of the occurrence. Added in version 4.

mainEventFilePathname: `<string>`
: The absolute path name of the file containing the main event, reflecting its physical location on the machine where the build was performed.

strippedMainEventFilePathname: `<string>`
: The `mainEventFilePathname` after path stripping. Path stripping will not change the value of `mainEventFilePathname`.

mainEventLineNumber: `<int>`
: The 1-based line number of the occurrence's main event.

properties: `<[string]>`
: A set of key/value pairs for general properties of the issue.

functionDisplayName: `<string>`
: The display name of the function where the main event occurs. The display name usually follows the syntax of the programming language to uniquely denote the function. This may be absent if the main event is not in a function or if the analysis does not know or report the function.

functionMangledName: `<string>`
: A unique string generated to identify the function where the main event occurs. This will be absent only when `functionDisplayName` is also absent.

ordered: `<boolean>`
: When true, the sequence of events in `events[]` and their children is significant. Specifically, the events are in chronological order along a particular code path that the analysis believes would misbehave. When false, the event tree is not in a meaningful order.

events: `<[Event]>`
: A list of top-level events in this issue report, each of which has several attributes described in the Event section. The events are ordered chronologically, relative to their specific event set.

stateOnServer: `<StateOnServer>`
: A list of attribute values for the issue as they exist on the Coverity Connect server. These values are described in the StateOnServer section. This object is `null` in disconnected mode.

checkerProperties: <`[CheckerProperties]`>

A list of property values for the checker that discovered the issue. These values are described in the CheckerProperties section. This information is retrieved from Coverity Connect, so may be absent in disconnected mode if not previously stored.

localStatus: <`string?`>

The string localStatus represents the local status with respect to reference status. Used when --include-missing-locally true is provided. The values are currently:

- "local": the defect is reported by local analysis but is not present in reference snapshot.
- "missing": the defect is found the reference snapshot, but not in local analysis.
- "present": the defect is both in the reference snapshot and local analysis.

When `--include-missing-locally` is false (default), this field is always null. Added in version 5.

5.13.3. Event

This object represents an atomic piece of evidence in a defect report. The events in a defect occurrence form a tree with the defect itself as the root.

covLStrEventDescription: <`string`>

The event description, represented in a format used internally by `cov-run-desktop`.

eventDescription: <`string`>

The localized event description. Localization may not be available in disconnected mode, in which case English is the default language.

eventNumber: <`int`>

The ordinal number of the event in its parent `event` object.

eventTreePosition: <`string`>

A dotted hierarchical number (`1.2.3` for instance) that reflects the event's position in the defect event tree. The last number is the same as `eventNumber`, the next-to-last number is the same as the `eventNumber` for this event's parent, and so on. The first number is the `eventNumber` of a top-level event.

eventSet: <`int`>

A non-negative integer that identifies which set of events this particular event is part of. In most cases, all events will be in event set `0`, the main defect path. Some defects, however, have events in multiple event sets.

eventTag: <`string`>

The event tag. For a path event, the tag is `path`.

filePathname: <`string`>

The absolute path name of the file containing the event, as it was known on the build machine.

strippedFilePathname: <`string`>

The `filePathname` after path stripping.

lineNumber: `<int>`
The 1-based line number where the event occurs in its file.

main: `<boolean>`
True if this is the main event - the event where the ostensible misbehavior happens. Only one top-level event will be main, and no lower-level events can be main.

moreInformationId: `<string>`
An identifier that can be used to find checker documentation. It is absent if the checker does not have known documentation.

remediation: `<boolean>`
Indicates that this event represents remediation advice.

This attribute was introduced in `json-output-v2`.

events: `<[Event]>`
If present, this is a non-empty sequence of child events that explain why this event was concluded.

5.13.4. StateOnServer

This object represents information about the issue that is only known to the Coverity Connect server. This object is `null` in disconnected mode.

cid: `<int>`
The numeric Coverity ID (CID) of the merged issue. The CID is found in the triage store for the stream specified in the `cov-run-desktop` command.

triage: `<Triage>`
An object containing the current values of the *built-in* triage attributes for this defect. These values are described in the Triage section.

customTriage: `<CustomTriage>`
An object containing the current values of the *user-defined* triage attributes for this defect.

presentInReferenceSnapshot: `<boolean>`
True if the issue is present in the reference snapshot specified in the `cov-run-desktop` command, false if not.

firstDetectedDateTime: `<string>`
The date and time when the issue was first detected in the stream. It expresses the date and time with the granularity of seconds in the time zone where the producing program is invoked, and includes that time zone expressed as a positive or negative offset from GMT.

For example: `2013-05-04T19:47+07:00`

stream: `<string>`
The name of the stream specified in the `cov-run-desktop` command.

components: <[string]>
A list of components in which the issue occurrences with the same merge key appear.

componentOwners: <[string]>
A pair of strings for the component's `componentDefaultOwner` and `componentDefaultOwnerLdapServer`. The values of the two fields will be null if the component does not have a default owner.

cached: <boolean>
True if `cov-run-desktop` was run in disconnected mode and the state was cached from a previous connected run. If false, `cov-run-desktop` was able to obtain up-to-date triage data by connecting to Coverity Connect.

retrievalDateTime: <string>
The date/time when the state was last retrieved. If `cached` is false, this is the time that `cov-run-desktop` was run. Otherwise, it is the invocation time of the last `cov-run-desktop` process that successfully retrieved data from Coverity Connect.

ownerLdapServerName: <string>
The LDAP server of the defect owner. An empty string indicates that the `ownerLdapServerName` is null. Added in version 3.

5.13.5. CheckerProperties

This object contains information that puts this defect into broad classes of related defects according to various classification schemes, some that are industry standard, and some that are created by Coverity. (The term "checker properties" is legacy nomenclature.)

category: <string>
The English name of the Coverity-defined broad category into which this defect falls. It is refined by `subcategoryShortDescription`.

categoryDescription: <string>
This legacy attribute is the same as "category".

cweCategory: <string>
CWE classification. It is either a CWE ID as a decimal integer, or the string "none".

issueKinds: <[string]>
Alphabetically sorted list of strings indicating the "kind" of issue. The valid strings are:

- QUALITY

  The defect is likely to affect the perceived quality of the product that contains it.

- SECURITY

  The defect may be a security vulnerability.

- TEST

  This is a test policy violation, meaning the tests do not adequately exercise the associated code. Added in version 3.

eventSetCaptions: `<[string]>`
A list of descriptions for each of the event sets appearing in the top-level issue occurrence.

impact: `<string>`
Level of the potential impact of the issue. Values are `High`, `Medium`, `Low`, or `Audit`.

impactDescription: `<string>`
This legacy attribute is the same as "impact".

subcategoryLocalEffect: `<string>`
The local effect of the given subcategory.

subcategoryLongDescription: `<string>`
Long description of the nature of the subcategory.

This attribute is a subset of HTML. Specifically, it may contain the elements "`a`", "`br`", "`code`", "`em`", and "`i`". The "`a`" element may contain the "`target`" and "`href`" attributes. All attributes are delimited by double-quote characters. There may be numeric entity references, both decimal and hexadecimal, as well as "`lt`", "`gt`", and "`amp`" named entities.

subcategoryShortDescription: `<string>`
This is a refinement of the "category" attribute. It corresponds to the "Type" column in Coverity Connect.

MISRACategory: `<string>`
Category for MISRA defects, one of `Advisory`, `Required`, or `Mandatory`. This field is optional. When not present, the whole field is absent, `null` is not used. (Always absent before version 7.)

5.13.6. Triage

This object carries the current values of *built-in* triage attributes for the defect.

classification: `<string>`
The current issue Classification.

action: `<string>`
The current Action to be taken on the issue.

fixTarget: `<string>`
The current Fix Target for the issue.

severity: `<string>`
The current Severity of the issue.

legacy: `<string>`
: The current Legacy value of the issue.

owner: `<string>`
: The username of the currently assigned owner of the issue, or an empty string if there is no assigned owner.

externalReference: `<string>`
: The "external reference" string for the issue.

5.13.7. CustomTriage

The `CustomTriage` object holds a set of key/value pairs for any *user-defined* triage attributes.

5.13.8. DesktopAnalysisSettings

This object captures the main inputs to `cov-run-desktop`.

analysisDateTime: `<string>`
: The date/time of the `cov-run-desktop` invocation that produced the present JSON output file.

covRunDesktopArgs: `<[string]>`
: The command line arguments to `cov-run-desktop` that produced the present output file, with any `--password` argument removed.

effectiveStripPaths: `<[string]>`
: The set of effective strip paths.

analysisScopePathnames: `<[string]>`
: The set of source file pathnames defining the analysis scope. These paths are not stripped.

strippedAnalysisScopePathnames: `<[string]>`
: The stripped version of `analysisScopePathnames`.

auxiliaryScopePathnames: `<[string]>`
: The set of primary source file pathnames of translation units that were analyzed in addition to `analysisScopePathnames` for the purpose of enabling analysis of something in that set. Defects in the auxiliary scope are not reported.

strippedAuxiliaryScopePathnames: `<[string]>`
: The stripped version of `auxiliaryScopePathnames`.

referenceSnapshot: `<ReferenceSnapshotDetails>`
: If a reference snapshot is specified or inferred, this object contains details about the snapshot. This object is described in the ReferenceSnapshotDetails section.

effectiveAnalysisSettings: `<PortableAnalysisSettings>`
: The core analysis settings used for this invocation of `cov-run-desktop`. These are the effective combination of the settings retrieved from Coverity Connect (if any) and the settings specified on the `cov-run-desktop` command line (if any). See PortableAnalysisSettings.

relativeTo: <`string`>
: If present, this corresponds to `--relative-paths --relative-to <relativeTo>` . If absent, this corresponds to `--relative-to false`.

intermediateDir: <`string`>
: The intermediate directory used by `cov-run-desktop`. Added in version 3.

5.13.9. ReferenceSnapshotDetails

This object contains information about the reference snapshot stored on the Coverity Connect server.

snapshotId: <`int`>
: The numeric snapshot ID.

codeVersionDateTime: <`string`>
: The SCM date/time associated with the version of the code that was analyzed.

description: <`string`>
: The snapshot `description` attribute.

version: <`string`>
: The snapshot `version` attribute.

analysisVersion: <`string`>
: The snapshot's `analysisVersion`. Added in version 3.

analysisVersionOverride: <`string`>
: The `analysisVersionOverride` associated with the reference stream. Added in version 3.

target: <`string`>
: The user-provided `target` attribute.

5.13.10. PortableAnalysisSettings

This object contains the "portable" analysis settings for the current invocation of `cov-run-desktop`.

coding_standard_configs: <`string`>
: The contents of the file passed to `--coding_standard_configs`, if any.

covAnalyzeArgs: <`[string]`>
: A sequence of `cov-analyze` command line arguments.

fileCheckerOptions: <`FileCheckerOption`>
: The contents of the argument file to checker options that specify files.

fbExcludeConfigurations: <`string`>
: A sequence of the contents of files passed to `--fb-exclude`, if any.

fbIncludeConfigurations: <`string`>
: The contents of the file passed to `--fb-include`, if any.

5.13.11. FileCheckerOption

This object contains the contents of the file that was passed as an argument to a checker option.

checkerName: <`string`>
Name of the checker to which the option was passed (e.g. `WRAPPER_ESCAPE`).

optionName: <`string`>
Name of the option (e.g. `config_file`).

fileContents: <`string`>
The contents of the file passed as an option.

5.13.12. Error

If `cov-run-desktop` is unable to perform the requested action, the output will include the `Error` object.

errorType: <`string`>
The type of the error that caused `cov-run-desktop` to fail.

errorSubType: <`string`>
The subtype of the error.

errorMessage: <`string`>
A message describing the nature of the error.

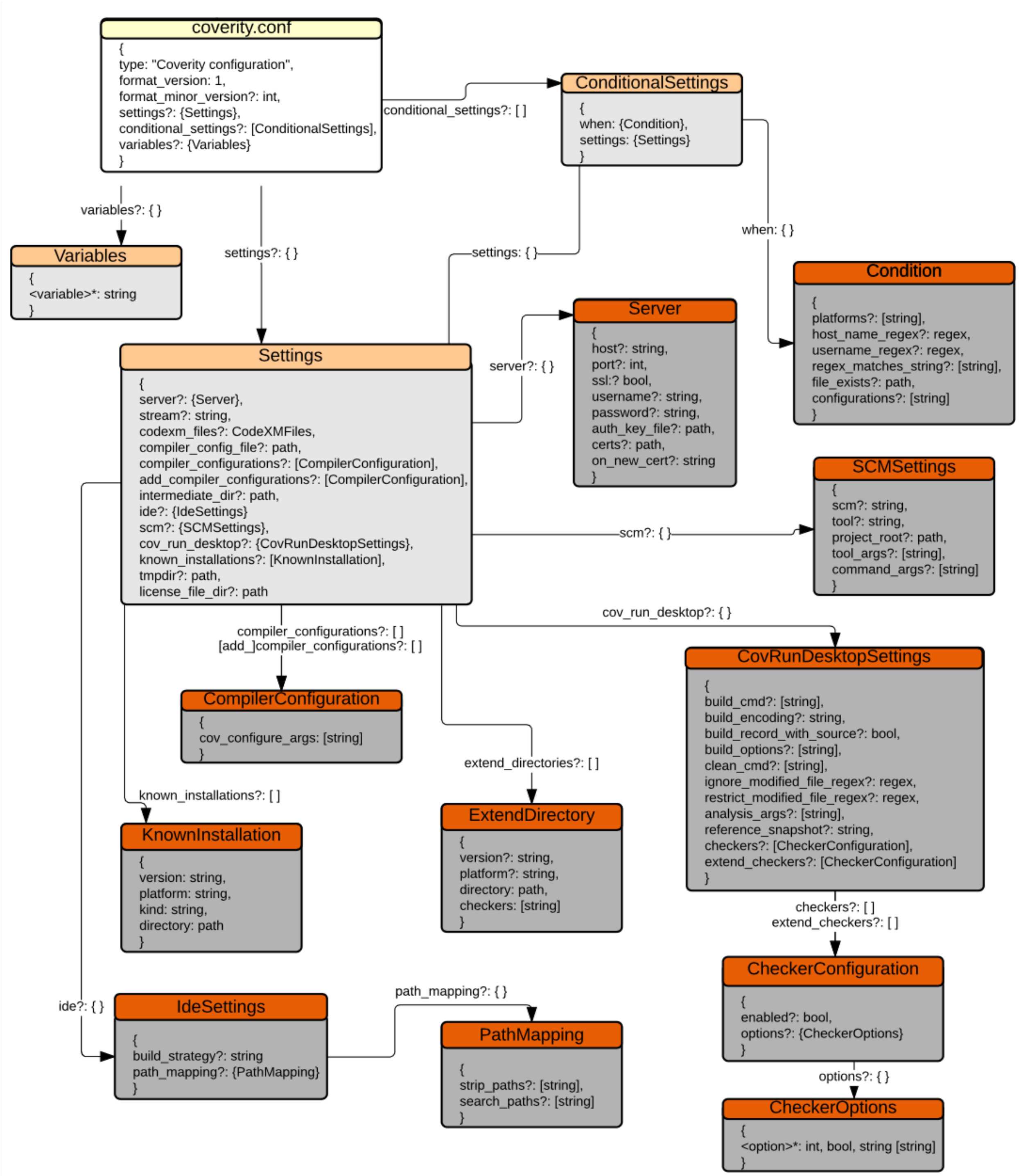
Note

The `Error` object may contain additional attributes, specific to its `errorType`. Any program that reads this format must ignore any unrecognized error attributes.

5.14. `coverity.conf` file format

The `coverity.conf` file format is based on JSON, but with comments allowed. Comments may begin with "//" and go to the end of the line.

The overall structure of `coverity.conf` is depicted in the following diagram (note that optional fields are marked with a "?"). For a working `coverity.conf` example, see Section 5.14.19, “Example coverity.conf file”.



The following sub-sections describe each of the constituent elements. Several conventions and notations are used:

- Attributes (also known as fields or properties) are named beginning with a lowercase letter and consist of lowercase letters and underscores.

- Classes are named beginning with an uppercase letter and consist of mixed case names. Class names do not actually appear anywhere in the configuration file because JSON does not have a notion of a class.

- Required attribute declarations are written in the following format:

  ```
  "<attribute_name>": <type>
  ```

  This format means that there must be an attribute called "`<attribute_name>`" and its value must be consistent with `<type>`.

- A type is one of the following:

  - The scalars `bool`, `int`, or `string`

  - `regex` or `path` (these are like `string` but have special meaning)

  - A class name (meaning the value must be an object that conforms to the structure defined for that class)

  - One of the above followed by "`[]`" (meaning the value must be an array of the indicated type)

- Optional attribute declarations are written in the following format:

  ```
  "<attribute_name>"?: <type>
  ```

  Notice the "?" before the "`:`".

  This format means that the attribute can be omitted. The majority of attributes in this format are optional because the configuration can be created by combining multiple configuration sources.

5.14.1. coverity.conf

The root of the JSON object tree is called `coverity.conf` because that is the name of the file, and the file consists of exactly one such object. It has the following attributes:

type: "Coverity configuration"
  The `type` must be exactly "Coverity configuration" to label the file's contents.

format_version: 1
  The version must be the number 1. Future releases of the Coverity tools may introduce additional permitted version numbers.

format_minor_version: 7
  The minor version is currently at 7. Future releases of the Coverity tools may introduce additional permitted minor version numbers.

variables?: UserDefinedVariables

Defines a variable that can be used within the coverity.conf file itself using a `variables` property. Each variable is defined by a name and a corresponding value.

settings?: Settings

Defines the unconditional settings stored in this file.

conditional_settings?: ConditionalSettings[]

Defines the conditional settings in this file. Each object has some conditions that must be true for the object to take effect. Conditional settings override unconditional settings, and earlier conditional settings take precedence over later conditional settings (if both are active).

5.14.2. UserDefinedVariables

`UserDefinedVariables` contains user defined variable names that can be defined to specific values. These defined variable names are expanded in strings (See Table 5.1, “Simple variables” for details).

The `variables` property is a top-level property in the configuration file, and is unconditionally evaluated. Variables may be referenced in the expression associated with a `regex_matches_string` condition for a conditional settings block.

If a variable is defined in both a project-specific `coverity.conf` file as well as the `coverity.conf` file in the user's home directory, the value in the user-specific file takes precedence. In addition, this process is applied on a per-variable basis, so a user only needs to define values in the user-specific file which they intend to override.

5.14.3. ConditionalSettings

`ConditionalSettings` combines `Settings` with conditions under which they apply. If the conditions are true in the environment where the file is being read, then the corresponding settings are active (and may override settings from elsewhere).

The attributes of this class are:

when: Condition

The condition(s) that must be true for the `settings` to be active.

settings: Settings

When the conditions specified in the other attributes are satisfied, these settings become active, overriding same-named attributes from unconditional settings.

5.14.4. Condition

A `Condition` contains one or more simple predicates (attributes) that must be true for the condition to be met. The attributes are:

platforms?: string[]

If present, then the current host platform must be among those in the sequence for the condition to be satisfied. The possible platforms are:

- aix
- freebsd
- freebsd64
- linux
- linux-ia64
- linux64
- macosx
- netbsd
- netbsd64
- solaris-sparc
- solaris-x86
- win32
- win64

hostname_regex?: regex

If present, the fully qualified hostname of the host machine must match (as a substring, unless anchors are specified) the specified perl-syntax regular expression. Note that regular expression matching for `hostname_regex` is case-insensitive.

username_regex?: regex

If present, the OS username of the user invoking the tool reading the configuration file must match the specified regular expression.

regex_matches_string?: string[]

If present, this array must contain an *even* number of strings, such that they form a sequence of (regex,string) pairs. Each regex is matched against each corresponding string; all must match for the condition to be met.

For example:

```
    "regex_matches_string": [
        "^user1$", "$(env:USER)",         // user = user1, AND
        "^win", "$(platform)"             // OS is Windows
    ]
```

file_exists?: path

If present, this condition is true if there is any named file system entity (file, directory, named pipe, etc.) at the given location.

configurations?:string[]

If present, this condition is true if the current analysis configuration name exactly matches any of the listed strings.

Configuration names are specified using the –`configuration-name <name>` on the command line or matching an Analysis Configuration name in the Coverity Desktop Analysis plugins. If no configuration name is specified, the current configuration is *Default*.

5.14.5. Settings

The `Settings` class contains the configuration parameters that directly influence the operation of tools. It contains the following attributes:

add_compiler_configurations?: CompilerConfiguration[]

A supplementary set of compiler configurations. These do not override anything; instead, all active configuration sources' `add_compiler_configurations` attributes contribute additional configurations to the total set that will be active.

codexm_files?: CodeXMFiles

This property contains `CodeXMFiles` objects that define individual CodeXM checkers which are provided for analysis. CodeXM checkers are enabled once they are specified in the `codexm_files` property.

compiler_config_file?: path

The name of the `coverity_config.xml` file where the compiler configuration information shall be stored. It corresponds to the `--config` command line option.

The default value is "`$(code_base_dir)/data-coverity/v$(version)/config/coverity_config.xml`".

compiler_configurations?: CompilerConfiguration[]

This array contains specifications for how to configure compilers. Each element corresponds to one invocation of `cov-configure` to run when setting up desktop analysis on the developer's workstation.

The default value is:

```
    "compiler_configurations": [
      {
        "cov_configure_args": [
          "--javascript",
          "--if-supported-platform"
        ]
      },
      {
        "cov_configure_args": [
          "--php",
          "--if-supported-platform"
        ]
      },
      {
        "cov_configure_args": [
          "--python",
          "--if-supported-platform"
        ]
      },
      {
        "cov_configure_args": [
          "--ruby",
          "--if-supported-platform"
        ]
      },
      {
        "cov_configure_args": [
          "--gcc"
        ]
      },
      {
        "cov_configure_args": [
          "--java"
        ]
      },

      {
        "cov_configure_args": [
          "--scala"
        ]
      },
      {
        "cov_configure_args": [
          "--swift"
        ]
      },
      {
        "cov_configure_args": [
          "--msvc"
        ]
      },
      {
        "cov_configure_args": [
          "--cs"
        ]
      },
      {
        "cov_configure_args": [
          "--clang"
        ]
      }
    ]
```

That means that you only need to set this value if using a compiler other than GNU C/C++ under the name gcc or g++, Microsoft C/C++/C#, Oracle Java, or Clang. All interpreted languages (filesystem capture) supported for Coverity analysis on the current platform are part of this default configuration.

Otherwise, you need to create one `CompilerConfiguration` element for each compiler used during the build. For example, if you are using ccache with GCC, then you should set `compiler_configurations` to:

```
[
  {
    // cov-configure --gcc
    "cov_configure_args": ["--gcc"]
  },
  {
    // cov-configure --compiler ccache --comptype prefix
    "cov_configure_args": ["--compiler", "ccache", "--comptype", "prefix"]
  }
]
```

See Configuring compilers for Coverity Analysis for more information on how to use `cov-configure`. Once you have a set of `cov-configure` command lines that configure your compilers, put them in `compiler_configurations` or `add_compiler_configurations`.

If an overriding configuration specification (from another file, or from a conditional configuration) specifies this attribute, then the *entire* set of configurations is replaced with the overriding set.

cov_run_desktop?: CovRunDesktopSettings

Settings specific to the operation of `cov-run-desktop`.

extend_directories?: ExtendDirectories

`ExtendDirectory` objects which define the individual custom checkers provided for analysis. The extend checker names defined here are used in the `extend_checkers` property of `CovRunDesktopSettings`.

intermediate_dir?: path

Specifies the location of the intermediate directory, which is one piece of the local state maintained by desktop analysis. It corresponds to the `--dir` command line option.

The default value is "`$(code_base_dir)/data-coverity/v$(version)/idir`".

Note

Note that Desktop Analysis can only be run on an intermediate directory created on the same machine, and in the same source code directory, as the analysis will take place (i.e. the build and analysis processes must take place on the same machine and directory, and the intermediate directory must not be moved).

known_installations?: KnownInstallation[]

Sequence of known installations of Coverity tools.

license_file_dir?: path

Directory where automatically downloaded license files are stored.

The default value is "`$(code_base_dir)/data-coverity/v$(version)/lic`".

server?: Server
: Settings that specify how to access the Coverity Connect server.

stream?: string
: The name of the Coverity Connect stream from which to get analysis summaries. The stream should contain snapshots obtained by analyzing the same code base and branch as will be analyzed on the desktop.

  There is no default for this setting, so it must be set in a `coverity.conf` file or on the command line.

scm?: SCMSettings
: Settings for interacting with the SCM.

tmpdir?: path
: Directory in which to store temporary data (data that, under normal circumstances, is removed by the same process that created it). The default value is determined based on operating system.

5.14.6. Server

The `Server` class contains Coverity Connect server configuration attributes:

host?: string
: The host name or numeric (IPv4 or IPv6) address of the Coverity Connect server. No dependent variables may appear in this string.

  There is no default for this setting, so it must be set in a `coverity.conf` file or on the command line.

port?: int
: The port number of the HTTP or HTTPS service of the Coverity Connect server.

  The default is 8080 if `ssl` is false and 8443 if it is true.

ssl?: bool
: if true, then the communication with Coverity Connect will be protected with the SSL/TLS protocol. If false, communication will be conducted in cleartext.

  The default is false.

username?: string
: The Coverity Connect username to use when authenticating.

  The default value is "$(env:COV_USER:USER:USERNAME)".

  That means that if the COV_USER environment variable is set, its value is used. Otherwise, if USER is set then it is used; finally, USERNAME is tried.

password?: string
: The password corresponding to the user. If it is empty, no password is specified.

  The default value is "".

For security reasons, it is not recommended to put a password into `coverity.conf`, but there may be cases where it is expedient to do so anyway.

auth_key_file?: path

Name of a file to use as an authentication key.

The default value is "$(cov_user_dir)/authkeys/ak-$(server_host_as_fname)-$(server_port)".

That means that, by default, authentication keys are stored in a directory associated with the invoking user. Consequently, a single key can be used with many code bases and branches.

certs?: path

Use the CA certificates in the given file path. Requires ssl.

on_new_cert?: string

Indicates whether to trust (with trust-first-time) self-signed certificates, presented by the server, that haven't been seen before. The accepted string values are `trust` and `distrust`. Requires ssl.

5.14.7. CovRunDesktopSettings

The `CovRunDesktopSettings` class carries settings specific to the `cov-run-desktop` program. It has the following attributes:

allow_suffix_match?: bool

When true, `cov-run-desktop` behaves as if `--allow-suffix-match` were passed on the command line. This option is only recommended for enhanced backward compatibility.

analysis_args?: string[]

Arguments that `cov-run-desktop` should treat as if they appeared on its command line with the purpose of altering the behavior of the underlying analysis. This can be used to cause desktop analysis to use different options from the full, central analysis, although that is not the recommended usage.

If relative file names appear in these arguments, they will be interpreted as relative to the code_base_dir.

build_cmd?: string[]

A command line, as a sequence of shell words, that will invoke a build. This is run by "`cov-run-desktop --build`".

build_encoding?: string

A string to use as the argument to `cov-build --encoding` during build capture. The `source_encoding` option is preferred if all source code uses the same character encoding, but this option takes precedence for build capture when both are specified.

build_record_with_source?: bool

When true, `--record-with-source` is passed to `cov-build`. The default is false.

build_options?: string[]

A sequence of additional option words to pass to `cov-build`. The sequence of words must form a valid option and argument sequence for `cov-build`.

For example:

```
    "build_options": [
        "--encoding", "UTF-8",
        "--append-log",
        "--capture-ignore", "NTVDM.EXE"
      ]
```

checkers?: <Checkers>

`Checkers` then defines `CheckerSettings` specific to the operation of cov-run-desktop.

An example of the checkers and extend_checkers class is:

```
{
  // other cov-run-desktop settings...
  "checkers": {
    "ARRAY_VS_SINGLETON": {
      // inherit enabled value from tools or stream by not specifying the "enabled"
 property
      "options": {
        "stat_cutoff": 7
      }
    },
    "BAD_FREE": {
      "enabled": false
    }
  },
  "extend_checkers": {
    "MY_CUSTOM_CHECKER": {
      "enabled": true,
      "options": {
        "O": 3
      }
    }
  }
}
```

clean_cmd?: string[]

A command line (word sequence) that will clean the build. This is used by "`cov-run-desktop --clean`".

coding_standard_configs?: string[]

This accepts a list of strings, since this option can be specified multiple times on the command line. Provides the path(s) to configuration file(s) for a coding standard to run as part of the analysis. This option is required to enable C/C++ MISRA analysis. It can also be used for enabling other standards, such as CERT-C and AUTOSAR, see the description of the `--coding-standard-config` option to the `cov-analyze` command for more information.

extend_checkers?: <Checkers>

`Checkers` then defines `CheckerSettings` specific to the operation of `cov-run-desktop`.

fs_capture_build_options?: string[]

Like `build_options` but used when `cov-run-desktop` invokes `cov-build` to automatically capture files with filesystem capture.

ignore_modified_file_regex?: regex

A regex to be treated like the argument to `cov-run-desktop --ignore-modified-file-regex`. This is meant for use with the `--analyze-scm-modified` switch in order to avoid trying to analyze things that are not actually source code files.

ignore_uncapturable_inputs?: bool

Corresponds to the `--ignore-uncapturable-inputs` option to `cov-run-desktop`. This option is not recommended for general use because it can hide errors that would be revealed in the process of more thoughtful configuration.

restrict_modified_file_regex?: regex

A regex to be treated like the argument to `cov-run-desktop --restrict-modified-file-regex`. This is meant for use with the `--analyze-scm-modified` switch in order to avoid trying to analyze things that are not actually source code files.

reference_snapshot?: string

Corresponds to the `cov-run-desktop --reference-snapshot` option.

The default value is "idir-date".

source_encoding?: string

A string to use as the argument to `cov-build --encoding`, both during build capture and filesystem capture.

specific_files_build_cmd?: string[]

Specifies a command, typically a custom script, that can compile uncaptured files on demand. The command will be executed with `$(code_base_dir)` as the current directory and source files added to the end of the command, as paths relative to that directory or as absolute paths. See "Compiling files on demand" for details.

specific_files_build_options?: string[]

Like `build_options` but used when `cov-run-desktop` invokes `cov-build` with the `specific_files_build_cmd`. When unspecified, any settings in `build_options` are used.

specific_files_regex?: regex

When specified, specifies a pattern that files must match in order to attempt auto-compilation with `specific_files_build_cmd`. For example, if the command can only compile C and C++, the regex might be `\\.(c|cpp)$` to avoid attempting to compile header files or source code from other languages. When this option is unspecified, all specified files for analysis not previously captured with build capture and not captured with filesystem capture will be passed to `specific_files_build_cmd`. See "Compiling files on demand" for details.

5.14.8. SCMSettings

The `SCMSettings` class provides values that correspond to the `--scm*` arguments to `cov-run-desktop`. It has the following attributes:

scm?: string
Corresponds to `cov-run-desktop --scm`.

tool?: string
Corresponds to `cov-run-desktop --scm-tool`.

project_root?: path
Corresponds to `cov-run-desktop --scm-project-root`.

tool_args?: string[]
Corresponds to `cov-run-desktop --scm-tool-arg`. This accepts a list of strings, since this option can be specified multiple times on the command line.

command_args?: string[]
Corresponds to `cov-run-desktop --scm-command-arg`. This accepts a list of strings, since this option can be specified multiple times on the command line.

5.14.9. CompilerConfiguration

A `CompilerConfiguration` describes how to invoke `cov-configure` one time to configure one compiler, or a family of compilers that can all be configured by a single invocation. It has the following attribute:

cov_configure_args: string[]
A command line word sequence to pass to `cov-configure`.

5.14.10. KnownInstallation

A single `KnownInstallation` object records the existence of an installation of the Coverity Analysis tools. It has the following attributes:

version: string
The version number of the Coverity Analysis tools.

platform: string
The platform that this installation is intended to run on, as a Coverity platform identifier like "`linux64`". See Condition.platform for a complete list.

kind: string
The kind of tool installed at this location. Currently, the only possible value is "`cov-analysis`".

directory: path
An installation directory for the tools identified by `version` and `kind`.

5.14.11. ExtendDirectories

The `ExtendDirectories` class allows users to run Extend SDK checkers during the `cov-run-desktop` analysis. It has the following attributes:

version?: string
A version number which the custom checker works with; this value matches the version specified in the `KnownInstallation` object. During evaluation, custom checkers defined for a version other

than the effective value of the $(version) variable are ignored. If this value is omitted, the checker is assumed to be compatible with the current version of the tools.

platform?: string

The platform which the custom checker works with; this value matches the platform specified in a `KnownInstallation` object. During evaluation, custom checkers defined for a platform other than the effective value of the `$(platform)` variable are ignored. If this value is omitted, the checker is assumed to be compatible with the current platform.

directory?: path

A directory containing the custom checker executables.

checkers?: string[]

The names of custom checkers. These names will be equivalent to the executable names, but without the executable file extension.

An example of using extend_directories is:

```
{
  // other settings...
  "extend_directories": [
    {
      "version": "$(version)",
      "platform": "$(platform)",
      "directory": "$(install_dir)/bin/sdk",
      "checkers": [
        "CUSTOM_CHECKER_A",
        "CUSTOM_CHECKER_B"
      ]
    }
  ]
}
```

5.14.12. Checkers

The `Checkers` class declares a relationship between a specific checker and a `CheckerSettings` object that holds settings to configure that checker.

<name_of_checker>?: CheckerSettings

The name of the checker that is configured by the `CheckerSettings` object.

5.14.13. CheckerSettings

The `CheckerSettings` class allows users to configure specific checker settings during the `cov-run-desktop` analysis.

When merging two Settings objects, the checkers and extend_checkers objects are merged.

- The enabled property of the higher-priority `CheckerConfiguration` is applied

- The `CheckerOptions` objects are merged on a per-property basis, with values assigned to individual properties of the higher-priority Settings object replacing the corresponding values of the lower-priority Settings object (including string[] properties)

The `CheckerSettings` class has the following attributes:

enabled?: bool
If `true`, then enable the checker. If `false`, then disable the checker. If unspecified, the checker is enabled according to the settings in the reference snapshot.

options?: CheckerOptions
`CheckerOptions` objects which define the specific options to configure the checker.

5.14.14. CheckerOptions

The optional `CheckerOptions` class contains option names and the option values in the following format:

<optionName>?: <optionValue>
Each option value may be one of the following:

  - A number
  - The Boolean value `true` or `false`
  - A string
  - An array of strings. Whereas the previous cases all correspond to passing a single `--checker-option` option on the `cov-analyze` or `cov-run-desktop` command line, an array value here means the same thing as passing multiple `--checker-option` options.

5.14.15. string

A string is an ordinary JSON string, except that it may contain variable substitution placeholders that are substituted during evaluation to yield the effective configuration value.

Variable substitutions are denoted by "`$(`", then a name, then "`)`". For example, "`$(server_port)`" is substituted for the value of `settings.server.port`.

The defined variables are listed in the tables below:

Table 5.1. Simple variables

Simple Variable	Substitution
`env:VAR1:VAR2:...:VARn [=<default_value>]`	Environment variable lookup. First, VAR1 is looked up, and if that is defined and not empty then it is the substituted value. Otherwise, VAR2 is looked up, and so on. If none of the variables is defined as non-empty, then an empty string is substituted. Optionally, you can append =, which is a string that will be substituted if none of the environment variables are set.

Simple Variable	Substitution
`var:VAR1:VAR2:...:VARn [=<default_value>]`	Variable lookup. First, VAR1 is looked up, and if that is defined and not empty then it is the substituted value. Otherwise, VAR2 is looked up, and so on. If none of the variables is defined as non-empty, then an empty string is substituted. Optionally, you can append =, which is a string that will be substituted if none of the variables are set.
`dollar`	The character "$".
`lparen`	The character "(".
`rparen`	The character ")".
`platform`	The Coverity platform identifier string. This is one of the allowable values for `Condition.platforms`.
`version`	The Coverity tools version, for example "7.5.0" or "7.5.0.3".
`install_dir`	Directory where the invoked tool is installed. It has a `bin` subdirectory, among others.
`num_cores`	The number of detected CPU cores on the local host machine, or "1" if that can not be determined.

Table 5.2. Dependent variables

Dependent Variable	Substitution
`cov_user_dir`	A directory where user-specific and application-specific settings are stored. On operating systems other than Windows, this is "`$(env:HOME)/.coverity`". On Windows it is "`$(env:APPDATA)/Coverity`".
`code_base_dir`	The directory containing `coverity.conf` for the code base, if one is found; otherwise, it is the working directory where the tool was invoked.
`server_host_as_fname`	The effective value of `settings.server.host`, except mapped to a string that is safe to use as a file name.
`server_port`	The effective value of `settings.server.port`.

Table 5.3. Special variables

Special Variable	Substitution
`response_file_utf8`	In `specific_files_build_cmd`, this expands to the path of a temporary text file listing files to compile, one per line with UTF-8 character encoding.
`response_file_platform_default`	In `specific_files_build_cmd`, this expands to the path of a temporary text file listing files to compile, one per line with platform default character encoding.

Note

Special variables are only allowed in certain contexts.

5.14.16. regex

A regular expression ("regex") denotes a potentially infinite set of strings. Another string is said to "match" the regex if it is a member of its denoted set. In this file format, regexes use the Perl syntax, except they must be written as JSON strings, which means doubling backslashes and putting a backslash before certain other characters like double-quote.

5.14.17. path

A path is the name of either a file or directory on the file system. This must be an absolute path (that is, it must begin with "/" or "\" or a drive letter and colon and then a slash character).

If the path names a file or directory to be created, any necessary parent directories will be automatically created.

5.14.18. CodeXMFiles

The CodeXMFiles class allows users to run CodeXM checkers during the `cov-run-desktop` analysis. It has the following attributes:

directory?: path
A directory containing the custom CodeXM checker definitions.

files?: string
The names of files that define CodeXM checkers.

Here is an example that shows how to use the `codexm_files` property:

```
{
    // other settings...
    "codexm_files": [
        {
            "directory": "$(install_dir)/codexm",
            "files": [
                "CODEXM_CHECKER_A.cxm",
                "CODEXM_CHECKER_B.cxm"
            ]
          }
       ]
}
```

5.14.19. Example `coverity.conf` file

```
{
  "type": "Coverity configuration",
  "format_version": 1,
  "format_minor_version": 5,
  "settings": {
    "server": {
      "host": "d-linux64-03.sf.coverity.com",
               // REQUIRED
      "port": 5122,
      "ssl": false,
               // default
      "username": "$(env:COV_USER:USER:USERNAME)",
               // default
      "auth_key_file": "$(cov_user_dir)/authkeys/ak-$(server_host_as_fname)-
$(server_port)"          // default
    },
    "stream": "prevent-harmony",
               // REQUIRED
    "compiler_config_file": "$(code_base_dir)/data-coverity/v$(version)/config/
coverity_config.xml", // default
    "compiler_configurations": [
      {
        "cov_configure_args": ["--gcc"]
      },
      {
        "cov_configure_args": ["--java"]
      }
    ],
    "intermediate_dir": "$(code_base_dir)/data-coverity/v$(version)/idir",
               // default
    "license_file_dir": "$(code_base_dir)/data-coverity/v$(version)/lic",
               // default
    "scm": {
      "scm": "git"
    },
    "cov_run_desktop": {
      "build_cmd": ["make", "-j$(num_cores)"],
      "build_options": [
            "--encoding", "UTF-8",
            "--cygwin",
            "--delete-stale-tus"
      ],
      "clean_cmd": ["make", "clean"],
      "restrict_modified_file_regex": "^(?!.*/(cmd-)?(j|cs)?test.*).*\\.(java|c|cpp|
cc)$",
      "analysis_args": [
        "--enable-fb"
      ],
      "reference_snapshot": "scm"
    },
    "known_installations": [
      {
        "version": "7.5.0",
        "platform": "linux64",
        "kind": "cov-analysis",
        "directory": "/home/user1/opt/cov-analysis-linux64-7.5.0"
      },
      {
        "version": "7.5.1",
        "platform": "linux64",
        "kind": "cov-analysis",
        "directory": "/home/user1/opt/cov-analysis-linux64-7.5.1"
      }
    ],
          “ide” {
               “path_mapping”: {
                   “strip_paths”: [
                       “path1”,
                       “path2”
                   ],
                   “search_paths”: [
                       “path3”,
                       “path4”
                   ]
               },
               “build_strategy”: “CUSTOM”
  },
  "conditional_settings": [
    {
      "when": {
        "platforms": ["win64", "win32"]
      },
      "settings": {
        "compiler_configurations": [                                    // These
 settings are the default,
          {                                                             // except
 that by default they apply
            "cov_configure_args": ["--gcc"]                             // to all
 platforms.
          },
          {
            "cov_configure_args": ["--java"]
          },
          {
            "cov_configure_args": ["--msvc"]
          },
          {
            "cov_configure_args": ["--cs"]
          }
        ],
        "cov_run_desktop": {
          // adds "cs" extension on Windows
          "restrict_modified_file_regex": "^(?!.*/(cmd-)?(j|cs)?test.*).*\\.(java|c|
cpp|cc|cs)$"
        }
      }
    },
    {
      // On linux64, configure ccache checked in to platform-packages.
      "when": {
        "platforms": ["linux64"]
      },
      "settings": {
        "add_compiler_configurations": [
          {
```

```
      "cov_configure_args": [
       "--compiler",
       "$(code_base_dir)/linux64-packages/bin/ccache",
       "--comptype",
       "prefix"]
     }
    ]
   }
  }
 ]
}
```

5.14.20. `IDESettings`

`IDESettings` are used by the Coverity Desktop Analysis plugins and contain a few optional field settings that the user can configure. The following Desktop Analysis IDE settings exist:

build_strategy?: string

When set to `CUSTOM`, the plugin will use the default custom build settings specified in `build_cmd`. If a build setting isn't specified while using the Desktop Analysis IDE, the plugin will use the IDE specified build commands.

There is no default for this setting, so it must be set in a `coverity.conf` file or should be declared via command line.

path_mapping?: PathMapping

This setting allows you to specify strip and search paths for remote issues so that you can map them to local files.

For more information about where these settings exist in the IDE, see *Coverity 2020.12 for Eclipse, Wind River Workbench, and QNX Momentics: User Guide*

5.14.21. PathMapping

The `PathMapping` settings are used by the Coverity Desktop Analysis plugins to map any code defects (retrieved from Coverity Connect) to local files. The following `PathMapping` settings can be configured:

strip_paths?: string

Paths that are listed under this setting are stripped from the front of a defect's file path. This setting tries to resolve the path to a local file location.

search_paths?: string

Paths that are listed under this setting are added to the search locations of local files that contain remote issues.

5.14.22. Backwards compatibility

For newly added features in coverity.conf that need to be used with older versions of cov-run-desktop, the ext4 form may be used for compatibility with older releases:

- The string `or` is equivalent to `:` (colon).

- The string `else` is equivalent to = (equals).
- The syntax `$(ext4_var_or_NAME)` is equivalent to `$(var:NAME)`.
- The `variables` property may be placed within an `ext4` property.
- Any other new attribute, property or condition, may be placed in an `ext4` element or property. This will include things like:
    - The configurations condition
    - The checkers and extend_checkers properties in the `CovRunDesktopSettings` object.

Chapter 6. Troubleshooting Desktop Analysis

This troubleshooting section provides instructions for fixing the following common issues with Desktop Analysis:

1. There is no captured compilation that contains the file
2. `PARSE_ERROR` reported in file to analyze
3. `WARNING:` compiler output does not exist
4. `[ERROR] No snapshot in stream "<X>" has analysis summaries...`
5. `Issues with HTTP client proxies`
6. Differences between Central and Desktop Analysis results
7. `cov-run-desktop --clean or --build`: "The system cannot find the file specified."

Files not yet captured and not captured automatically

There are essentially three ways `cov-run-desktop` can capture a file for analysis, and this error indicates that none of them currently apply to the listed files. `cov-run-desktop` cannot be sure which was intended to apply, so the user has to determine that. The three capture methods are these:

- (a) Compiled under `cov-run-desktop --setup` or `--build`. This is the typical way of preparing `cov-run-desktop` to handle compiled source files.
- (b) Automatically captured using the filesystem capture configuration. Files for analysis that have not already been captured will be checked against filename patterns established in the filesystem capture configuration (part of the *compiler configuration*), such as glob `*.js`. When there is a match, such as `foo.js`, those files will be captured according the the relevant configuration.
- (c) Automatically compiled and captured with a *specific files* build script. (See Section 5.6, “Compiling files on demand” for details.) If `specific_files_build_cmd` is specified in `coverity.conf`, files for analysis that have not already been captured, do not match any filesystem capture configuration, and match the `specific_files_regex` if specified, will be passed to that build script. If the script returns a failure code (non-zero), that will be reported to the `cov-run-desktop` user as a specific error.

Therefore, the resolution to this error (`not yet captured and not captured automatically`) depends on which of these methods you expected to apply. Method (b) below is the only choice for interpreted code (filesystem capture code), including JavaScript, PHP, Python, and Ruby. Methods (a) and (c) are for compiled code, including C, C++, Objective-C, C#, Java, and Scala. Method (a) is the simpler and more typical of the two. If none of these apply, because a file is not source code or not supported for Coverity desktop analysis, consider one of the (n) solutions listed toward the end, below.

Troubleshooting method (a), compiling code under cov-run-desktop --setup or --build:

If you are expecting a file to have been captured already, the `cov-manage-emit` command can be used to help diagnose the cause. As a first step, run the following command to see a list of all the *primary source files* (PSFs) that were captured:

```
cov-manage-emit --dir <idir> list
```

A PSF is the "main" file of a translation unit (TU, also called a compilation unit), the file whose name is specified on the compiler command line. Other files that are implicitly read in order to compile the PSF, such as header files in C/C++ and other source and bytecode files in Java and C#, are "non-primary" source files.

If you are trying to analyze a PSF, but it is not in the output of the `list` command, it means a compilation of that file was not captured.

Solution (a1): Re-capture the build

If the file to analyze was added since the last time you ran `cov-run-desktop --build`, it is necessary to re-run that command to capture a compilation of the new file. It is simplest to re-capture a full build, although it is also possible to just capture a compilation of that one file.

Solution (a2): Configure another compiler

Another reason a PSF might be missing is that its compilation was seen by `cov-build` but not recognized as a compilation at the time. The `cov-configure` command tells `cov-build` what commands to consider as compilers. Every compiler used in your build should be configured with `cov-configure`.

Compiler configuration can be a complex process. The `<idir>/build-log.txt` file contains information about what commands were seen and which were treated as a compilation. If examining that file is not sufficient to discover the cause of an uncaptured compilation command, contact Coverity Support for assistance.

Solution (a3): Non-primary source files and record with source

If the file you want to analyze is a non-PSF, such as a header file, then it is typically necessary to add `--record-with-source` to the `cov-run-desktop --build` command line. This causes the build to preprocess every translation unit and record all of the source files that are read, rather than just record the command line. Consequently, `cov-run-desktop` will be able to find a translation unit that includes a given header file, if one exists.

The list of both primary and non-primary source files can be printed with the command:

```
cov-manage-emit --dir <idir> --tu-pattern 'file(".")' print-source-files
```

Here, the `--tu-pattern` is just a dummy pattern that matches all TUs, provided because `print-source-files` requires a pattern.

Solution (a4): Specify a primary source file as well as a header

Another tactic to analyze header files is to specify to `cov-run-desktop` *both* the header file to analyze and some PSF that includes that header file. `cov-run-desktop` will recognize the PSF, compile it, then notice that it includes the header. This is different from just analyzing the PSF, because `cov-run-desktop` only reports defects in files that are specified on the command line.

Finally, detailed debugging of build capture with `cov-run-desktop` is possible by consulting the `build-log.txt` file and the `output/cov-run-desktop-log.txt` file in the intermediate directory, usually `data-coverity/vN.N.N/idir`.

Troubleshooting method (b), automatic filesystem capture:

Solution (b1): Configure filesystem capture

Although the default compiler configuration used by cov-run-desktop includes standard configurations for all supported interpreted languages, if you have specified compiler_configurations in coverity.conf or a compiler configuration file with --config, these might not include filesystem capture. For example, to configure filesystem capture for Python *.py and PHP *.php, use this in coverity.conf (see Section 5.14.5, “Settings”):

```
"compiler_configurations": [
  {
    "cov_configure_args": ["--python"]
  },
  {
    "cov_configure_args": ["--php"]
  }
],
```

Note: Although the naming can be misleading, for historical reasons, the configuration for filesystem capture is part of the "compiler configuration".

Solution (b2): Expand file patterns for filesystem capture

Suppose your project uses the convention of a .j extension for JavaScript files. The default configuration for JavaScript capture provided by cov-run-desktop looks for files matching *.js (and *.html and others). The following adds configuration for capturing *.j files as JavaScript:

```
"add_compiler_configurations": [
  {
    "cov_configure_args": ["--comptype", "javascript", "--file-glob", "*.j"]
  }
],
```

Note: add_compiler_configurations extends the default configuration for cov-run-desktop. Use compiler_configurations to replace it.

Also refer to the cov-configure documentation in the *Coverity 2020.12 Command Reference* for more details.

Solution (b3): Use a supported platform for the source language

Coverity analysis of interpreted languages is not supported on some platforms. If you are using the default compiler configuration, this could explain why source files for some languages are not being captured. Providing an explicit configuration as in solution (b1) would cause cov-configure to report an error if a given filesystem capture language is not supported. You can also consult the "Supported platforms" section in the *Coverity 2020.12 Installation and Deployment Guide* for more information.

Also refer to the cov-configure documentation in the *Coverity 2020.12 Command Reference* for more details.

Finally, detailed debugging of filesystem capture with `cov-run-desktop` is possible by consulting the `autocapture-log.txt` file and the `output/cov-run-desktop-log.txt` file in the intermediate directory, usually `data-coverity/vN.N.N/idir`.

Troubleshooting method (c), automatic compilation of specific files

If you do not see this line on the console before receiving the `not yet captured and not captured automatically` error:

```
[STATUS] Attempting to compile files not known to the emit...
```

then `cov-run-desktop` did not invoke the build script, either because none is configured or because no files were considered applicable.

Solution (c1): Configure a `specific_files_build_cmd`

See Section 5.6, "Compiling files on demand" for details on configuring a custom build script.

Solution (c2): Adjust `specific_files_regex` to match the file name

If a file should be compiled by the build script and using `specific_files_regex` (recommended), make sure the regex matches that file name.

If you do see this line on the console before receiving the `not yet captured and not captured automatically` error:

```
[STATUS] Attempting to compile files not known to the emit...
```

then `cov-run-desktop` did invoke the build script, and it reported success.

Solution (c3): Specify a primary source file as well as a header

Auto-compilation cannot typically handle an uncaptured header file when no primary file that includes it is part of the files for analysis. Consider including one, as in solution (a4) above.

Solution (c4): Adjust `specific_files_regex` to match the file name (again)

It is possible that some files were properly passed to the build script and some were not due to this.

Solution (c5): Ensure that the build script always invokes the compiler

If the build script is base on `make`, measures must be taken to ensure the compiler is always invoked, even if object files are newer than source files. One way to fix this is to have the script `touch` the source files before running `make`.

Solution (c6): Configure the compiler

Compilers used by the build script must be part of the compiler configuration. See solution (a2) above.

Finally, detailed debugging of automatic compilation with `cov-run-desktop` is possible by consulting the `autocompile-log.txt` file and the `output/cov-run-desktop-log.txt` file in the intermediate directory, usually `data-coverity/vN.N.N/idir`.

Possible solutions when none of method (a) through method (c) apply:

- Solution (n1): Remove the file(s) from consideration, command line

  If a list of files was specified on the command line, remove the ones not to be analyzed.

- Solution (n2): Remove the file(s) from consideration, SCM modified

  If using `--analyze-scm-modified`, use options `--ignore-modified-file-regex` and/or `--restrict-modified-file-regex`, or their `coverity.conf` equivalents, to exclude unanalyzable files by name or extension. Example `coverity.conf` addition:

  ```
  {
    // ... other settings ...
    "settings": {
      "cov_run_desktop": {
        // ADD the following:
        "restrict_modified_file_regex": "\\.(c|cpp|cc|java|cs)$"
      }
    }
  }
  ```

- Solution (n3): Ignore uncapturable inputs (not recommended)

  If other solutions are impractical for eliminating uncapturable, unanalyzable files, or if you want a quick, temporary solution, you can specify `--ignore-uncapturable-inputs true` on the command line, or use the `ignore_uncapturable_inputs` `coverity.conf` setting.

Files not found

This generally indicates the user specified a path on the command line for a file to analyze, but the file was not found, either as an absolute path or relative to the current working directory.

- Solution 1: Name a file that actually exists. Did you mistype the path?

- Solution 2: Change to or specify the right directory

  If you want to analyze `foo.c`, but that is not in the current working directory, please change to that directory or include it in the specified path to `foo.c`.

- Solution 3: Enable suffix matching (not recommended).

  Earlier versions of `cov-run-desktop` permitted specifying any path suffix of a file captured in the intermediate directory, regardless of the current working directory. To restore this functionality for backward compatibility, use `--allow-suffix-match` or `allow_suffix_match` in `coverity.conf`.

Unable to find or capture specified files

This message is only generated when using --allow-suffix-match, and indicates you either have a "file(s) not found" problem or a "file(s) not yet captured and not captured automatically" problem, but the nature of --allow-suffix-match doesn't allow the tool to distinguish the two. Refer to the troubleshooting guidance for those errors.

`PARSE_ERROR` reported in file to analyze

A `PARSE_ERROR` pseudo-defect is reported when the Coverity compiler is unable to compile a source file. It may be that the file contains an ordinary syntax error; generally, you should compile the file with your usual compiler to check for syntax errors before running Desktop Analysis.

If there are no syntax errors detected by the usual compiler, a possible reason for a problem is that the compiler options for your build have changed — for instance, changing `-I` or `-D` flags for C/C++, or adding a `classpath` entry for java. In these cases, the solution is to re-capture a full build using `cov-run-desktop --build`.

Another possibility is that you have encountered an incompatibility between that compiler and the Coverity compiler. In that case, the typical solution is to adjust the compiler configuration as defined with `cov-configure` to work around the problem. For more information, see "Configuring compilers for Coverity Analysis" in the *Coverity Analysis 2020.12 User and Administrator Guide*. In some cases, it may be necessary to contact Coverity Support for assistance.

`WARNING:` compiler output does not exist

This warning is caused, when analyzing Java code, by the absence of a `.class` file corresponding to some `.java` file that was selected for analysis. These `.class` files are used as input to the SpotBugs component (which is responsible for reporting the `FB.*` class of defects). To fix this, ensure that your usual compiler has run on the code first so it will generate the `.class` files, then run `cov-run-desktop`.

`[ERROR] No snapshot in stream "<X>" has analysis summaries...`

This error is caused by one of the following scenarios:

- The reference stream (specified by the `--stream` option) does not contain any snapshots with analysis summaries.

- The reference stream does contain one or more snapshots with analysis summaries, but their Code Version Date is more recent than the date specified to, or inferred by, the `cov-run-desktop` command (see `--reference-snapshot` for information on how this is determined).

To fix this, log in to Coverity Connect to identify a candidate reference snapshot, if one exists. It may be necessary to enable Desktop Analysis in stream configuration, and then commit a new snapshot.

To find a candidate reference snapshot:

1. In Coverity Connect, open a *Snapshots* view for your project (*All In Project* for example).

2. Click the "gear" icon to edit the view settings, and open the *Columns* tab.

3. Enable the *Has Analysis Summaries* and *Code Version Date* columns.

4. Return to the *Snapshots* view. Any snapshot that has "True" in the *Has Analysis Summaries* column contains analysis summaries. Verify that your reference stream contains a snapshot with analysis summaries, and ensure that its Code Version Date is not more recent than the date specified by the `--reference-snapshot` option.

   For more information about Code Version Date, see the `cov-analyze` option, `--code-version-date`.

If no candidate reference snapshot exists:

1. Navigate to Configuration → Projects & Streams.

2. Select the relevant stream, and click on the *Desktop Analysis* tab.
3. Ensure that the *Enable Desktop Analysis* option is selected.
4. Commit a new analysis to this stream. This will contain analysis summaries as long as the `cov-analyze --export-summaries` option is not explicitly set to `false`.

If --reference-snapshot scm option is used:

This issue may be caused if your codebase's last update date is before the reference snapshot was committed to the stream. To fix this:

1. Update your codebase and push the changes (this will cause your SCM repository to be updated more recently than the reference snapshot).
2. Re-run Desktop Analysis with `--reference-snapshot scm`.

Issues with HTTP client proxies

Desktop Analysis may fail or return inaccurate results when run on networks using HTTP client proxies. Specifically, issues are known to arise when the `http_proxy` environment variable is a machine name rather than an IP address, or when there are wildcards in the `no_proxy` environment variable.

Differences between Central and Desktop Analysis results

You may notice analysis results that differ slightly from your Central Analysis results. There are several reasons that this may occur; see Reasons for results differences.

`cov-run-desktop --clean or --build`: "The system cannot find the file specified."

On Windows platforms, you may find that a command that works in the `cmd.exe` shell does not work in `settings.cov_run_desktop.build_cmd` or `clean_cmd` in `coverity.conf` (or on the `cov-run-desktop --build` command line), as it fails with the error message, "The system cannot find the file specified."

One possible reason is, unlike `cmd.exe`, `cov-run-desktop` does not automatically try file extensions other than "`.exe`". In particular, programs with extensions "`.com`", "`.bat`", and "`.cmd`" must be specified explicitly for `cov-run-desktop` to invoke them.