# Dynamic Analysis 2020.12 Administration Tutorial

**Dynamic Analysis is part of Coverity Analysis.**

## Table of Contents

1. Overview of Dynamic Analysis .......... 1
2. Running Dynamic Analysis on Java programs .......... 5
3. Dynamic Analysis defects reports .......... 18
4. Troubleshooting .......... 24
A. Dynamic Analysis Agent command-line options and Ant task attributes .......... 31
B. Dynamic Analysis Broker information and reference .......... 35

# Chapter 1. Overview of Dynamic Analysis

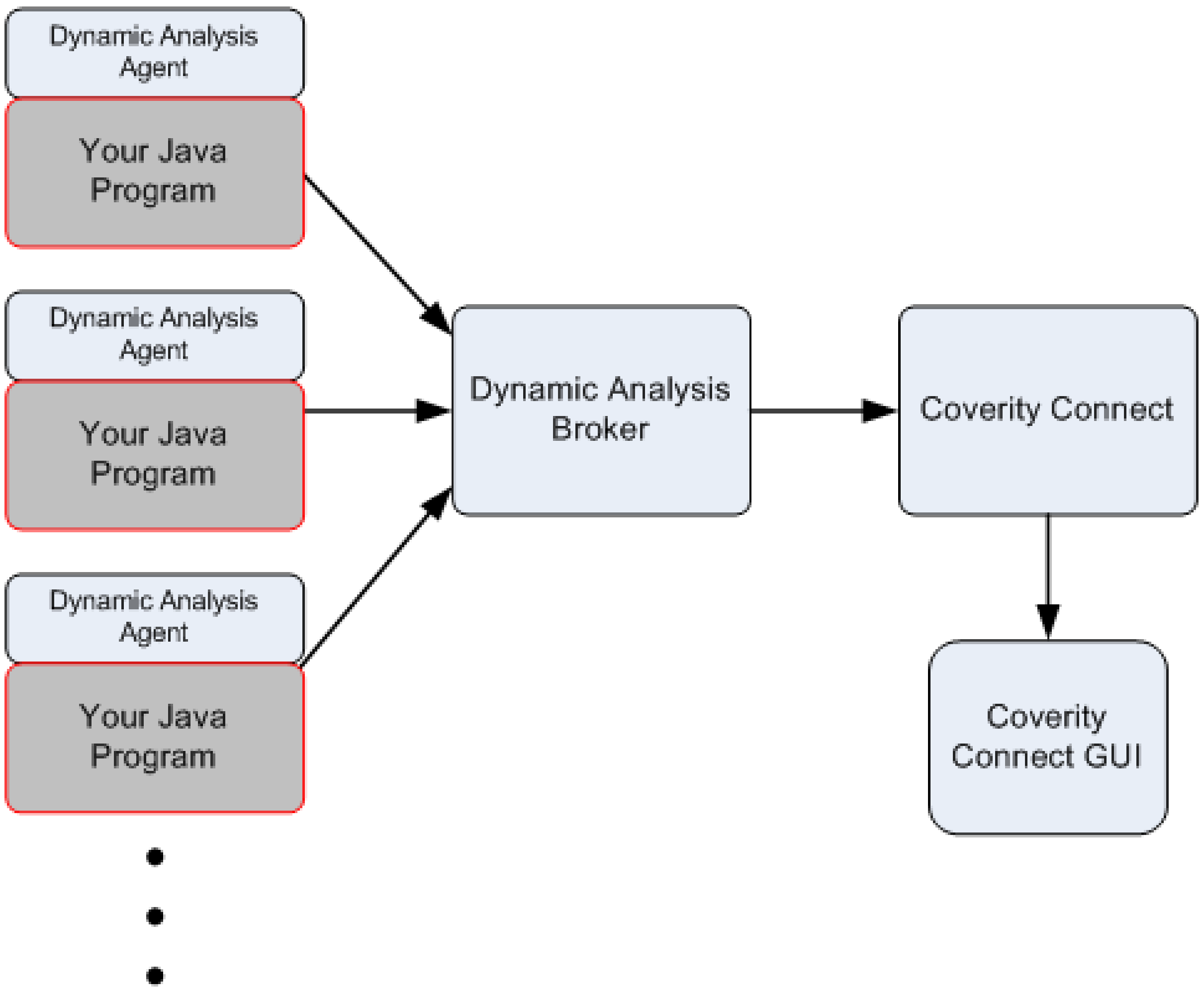
## Table of Contents

Dynamic Analysis finds actual and potential race conditions, deadlocks, and resource leaks by watching your Java program while it runs. Coverity Analysis also finds these types of defects, but Dynamic Analysis finds defects that are difficult or impossible for any static analysis software to find. While Static Analysis considers all possible execution paths, Dynamic Analysis focuses on those that it actually observes on test workloads. Neither is superior to the other, but their focus is different. Coverity Analysis and Dynamic Analysis, used together, provide the most thorough analysis of race conditions, deadlocks, and resource leaks. At this time, Dynamic Analysis only supports Java programs.

## 1.1. How Dynamic Analysis works

The following figure depicts the flow of data between a Java program, the Dynamic Analysis Agent, and the Dynamic Analysis Broker.

**Figure 1.1. Dynamic Analysis data flow**



A Dynamic Analysis Agent runs in a JVM (Java Virtual Machine) along with the Java program you want to dynamically analyze. The Dynamic Analysis Agent watches your program run and looks for actual resource leaks, potential race conditions, and potential deadlocks. If it finds any, it sends a defect report to the Dynamic Analysis Broker, which forwards the data to the Coverity Connect server.

Each Dynamic Analysis Agent can run on a different machine and watch for defects as different tests or instances of your application run. You can view the defect data and source code at any time during this process by bringing up Coverity Connect in a web browser.

## 1.2. Obtaining thorough analysis results

Dynamic Analysis observes a run of your Java program. Thus, the more thoroughly your test workload exercises code paths in your program, the more useful your Dynamic Analysis results. Dynamic Analysis cannot report defects on code that does not execute. Because race conditions and deadlocks require the interaction of multiple threads, Dynamic Analysis does not report these defect types if only one thread accesses a field or acquires locks during the run it observes.

To thoroughly test your application for race conditions and deadlocks, you must design tests that cause multi-threaded execution of your code base. This requires a technical understanding of concurrency in Java, how your program uses it, and how to simulate it in a test environment.

Dynamic Analysis amplifies your existing testing. It provides more comprehensive results if you test your program using varied workloads. Small single-threaded unit tests do not provide the type of workloads needed to yield good results. System tests, integration tests, load tests, or even ad hoc testing is likely to cover your application more thoroughly than most unit tests, and thus yield more useful results.

**Important**

When running Dynamic Analysis on a Java program, expect the program to run as much as 2-5 times slower than normal. Dynamic Analysis also requires additional memory. Without adequate heap memory, the Java program will fail with an `OutOfMemoryException`. In practice, the peak memory usage increases as much as 800%. To allocate additional heap memory, use the *`-Xmx`* parameter to the `java` command when you invoke your test. For more information about this error see the `OutofMemoryError` question and answer

## 1.3. Fitting Dynamic Analysis into your development and testing environments

Dynamic Analysis can be integrated into your development and testing environment in many ways, but most commonly you will want to integrate it into your manual and automatic tests.

**Ad hoc analysis.** In this model, you run your application with Dynamic Analysis, and exercise the program thoroughly, running it through different work flows and using multiple users or tasks simultaneously. Focus on concurrency or resource-related code, and exercise it so that multiple threads access shared data.

**Desktop analysis.** In this model, developers work on their code and run Dynamic Analysis on the functions they develop. They can run the entire program manually or use automated tests that simulate aspects of the application outside their control, rather than run the entire program. For example, they might run a test that runs their business logic code in an automated test framework that simulates user requests and database accesses. This scenario provides a focused analysis of the code being developed and the code with which it interacts. It also allows developers to fix defects before they check their code into the build.

The desktop and ad hoc analysis models are the easiest ways to start using Dynamic Analysis. To integrate Dynamic Analysis into your development environment, set up an automated process that sends you defect information as you develop, test, and build your software.

**Automated analysis.** In this model, testers run automated tests with Dynamic Analysis. You can set up Dynamic Analysis to run with your nightly builds or at some other regular test interval.

**Important**

While it might be useful to run Dynamic Analysis with your program in a production-like environment, we strongly recommend against running your programs with Dynamic Analysis in an actual production environment. The program runs much slower, requires more memory, and disables the security manager (see the `override-security-manager` Agent option in Section A.2). Dynamic Analysis also introduces more failure modes: if Coverity Connect goes down, the Broker and your program also go down (see the `failfast` Agent option, also in Section A.2).

## 1.4. Installing and licensing Dynamic Analysis

Dynamic Analysis is installed as part of Coverity Analysis and so requires a Coverity Analysis license. If you have a current `license.dat` file with Dynamic Analysis enabled, place it in `<install_dir>/bin/`. If you have an old Coverity Thread Analyzer or Dynamic Analysis license but no Coverity Analysis license, start the Dynamic Analysis Broker with the `--security-file <license_path>` option.

# Chapter 2. Running Dynamic Analysis on Java programs

## Table of Contents

This procedure in this chapter uses a simple program example to demonstrate how to perform a dynamic analysis. It explains how to perform this action from the command line. To run Dynamic Analysis using Apache Ant tasks, see Section 2.6.

**Note**

In most cases, examples in this chapter use UNIX command-line syntax. You can also run examples in Windows by using the appropriate Windows command-line syntax, typically `./some_command.exe` followed by any necessary command options.

**To run Dynamic Analysis on a Java program:**

1. Create Dynamic Analysis streams for your Java program in Coverity Connect.
2. Provide a Java program that compiles and runs to be dynamically analyzed.
3. Start the Dynamic Analysis Broker.
4. Run your Java program with the Dynamic Analysis Agent.
5. Stop the Dynamic Analysis Broker.

## 2.1. Step 1: Create Dynamic Analysis projects and streams in Coverity Connect

The first step in running Dynamic Analysis with your Java program is to create a Coverity Connect project that contains the streams to receive the source code and defects that Dynamic Analysis reports on your program. When you start the Dynamic Analysis Broker in Section 2.3, you will specify the Dynamic Analysis defect stream created in this step. You only need to create streams once for each code base.

**Requirement**

To complete this procedure, you must have Administrator or Configuration Manager privileges to Coverity Connect. If you do not, contact your Coverity Connect administrator.

**To set up projects and streams:**

1. Get the hostname for Coverity Connect, the Coverity Connect HTTP port to which you point your browser (default: 8080), and the commit port where your Broker sends defects (default: 9090).

   Your Coverity Connect administrator should have this information. These values were specified during the installation of Coverity Connect.

2. Open Coverity Connect in your web browser:

   ```
   http://<host>:<CIM_http_port>
   ```

For example, if Coverity Connect is installed on the `cim.example.com` with the default settings, the URL is `http://cim.example.com:8080.` If Coverity Connect does not start, see the *Coverity Platform 2020.12 User and Administrator Guide* for Coverity Connect start, stop and status instructions.

3. Create a new project that contains a *Java (dynamic analysis)* stream called `Example-dynamic`.

   In Coverity Connect, be sure to create a *Java (dynamic analysis)* stream (not a *Java* stream) so that Dynamic Analysis can commit defect data to it.

   For more information about creating projects and streams in Coverity Connect, see the *Coverity Platform 2020.12 User and Administrator Guide*.

4. Proceed to Section 2.2, “Step 2: Provide a program to be dynamically analyzed”.

## 2.2. Step 2: Provide a program to be dynamically analyzed

Because Dynamic Analysis watches a running program, you must have a program that compiles and runs without Dynamic Analysis.

**Note**

Though unnecessary for running Dynamic Analysis, you can also run Coverity Analysis on the source code to see the differences between the two analyses.

For recommendations that supplement the steps in this procedure, see Section 2.7, “Deploying Dynamic Analysis in a production environment”.

The following procedure uses a sample program, `Example.java`, to demonstrate the tasks that are required to perform a dynamic analysis. The sample forces race condition, deadlock, and resource leak defects to occur so that you can see how Dynamic Analysis and Coverity Connect work. `Example.java` is available at `<install_dir>dynamic_analysis/dynamic-analysis/demo/src/simple`.

**To build the sample program:**

1. Create a local, writeable copy of the Dynamic Analysis `/demo` directory.

   This directory is located under `<install_dir>dynamic_analysis/dynamic-analysis`.

2. Use one of the following build procedures.

   - Recommended way to build the program:

     a. Use `cov-configure` to set up your Java compiler.

     ```
     > <install_dir>dynamic_analysis/bin/cov-configure --java
     ```

     If successful, the console will output the following message:

     ```
     Generated coverity_config.xml at location <install_dir>dynamic_analysis/
     config/coverity_config.xml
     ```

```
Successfully generated configuration for the compilers: apt java javac
```

b. From your local copy of the Dynamic Analysis `/demo` directory, use the `cov-build` command to capture a build of the sample program to an intermediate directory.

   Execute the following commands:

```
> mkdir classes
> <install_dir>dynamic_analysis/bin/cov-build
    --dir intermediate_dir_name
    javac -d classes src/simple/Example.java
```

   You can specify any name for your intermediate directory.

   The `cov-build` process takes more time to complete than a build process that uses only the native compiler. Upon success, the console outputs the following message:

```
The cov-build utility completed successfully.
```

- Alternative build procedure to follow only if using `cov-build` is infeasible:

a. Compile your code base using `javac`.

b. Run `cov-emit-java --compiler-outputs` for each invocation of `javac`.

   For example:

```
> cov-emit-java --findsource src --findjars lib:build-lib/ --dir
    my/intermediate/dir --compiler-outputs build/classes/;build/junitclasses/
```

**Note**

Alternatively, you can run `cov-emit-java` alone, without `--compiler-outputs`, for each compiler invocation. Then, after running it for all invocations, you can run the `cov-emit-java` command a final time with `--compiler-outputs`.

   For more detailed information about `cov-emit-java`, see the Coverity Analysis.

3. Confirm that you can run the sample program without Dynamic Analysis:

```
> java -cp classes simple.Example
```

   The console displays the following sort of output when you run the program without Dynamic Analysis:

```
*** RESOURCE_LEAK example
*** RACE_CONDITION example
race=-260
race=0
race=-46
race=-23
race=370
race=53
```

```
race=-1324
race=248
race=-310
race=-205
*** DEADLOCK example (this example might actually deadlock and need to
be forcibly terminated)
```

> **Note**
>
> This test should finish in less than ten seconds. If it continues to run after thirty sections, use Ctrl+C to kill the program.
>
> The numbers after `race=` might vary per run.

4. Proceed to Section 2.3, “Step 3: Start the Dynamic Analysis Broker”.

## 2.3. Step 3: Start the Dynamic Analysis Broker

The Dynamic Analysis Broker performs two distinct tasks:

- It gathers source code and sends it to Coverity Connect.
- It accepts connections from Dynamic Analysis Agents and then sends their defect reports to Coverity Connect. You will run the Agent in Section 2.4.

You start the Dynamic Analysis Broker with the `cov-start-da-broker` command after creating a defect stream in Coverity Connect but before running your program with the Dynamic Analysis Agent.

> **Note**
>
> You can set several Broker options through a configuration file and environment variables. For more information about this topic, see Appendix B.

The following procedure demonstrates source transfer from the Dynamic Analysis Broker to Coverity Connect.

**To run the Dynamic Analysis Broker:**

1. Execute the `cov-start-da-broker` command from your local copy of the Dynamic Analysis `/demo` directory. For example:

   ```
   > <install_dir>dynamic_analysis/dynamic-analysis/bin/cov-start-da-broker
       --host cim.example.com --dataport 9090 --user jdoe
       --password secret --stream Example-dynamic
       --dir intermediate_dir_name
   ```

   The options perform the following actions:

   - The `--host` and `--dataport` options specify the Coverity Connect server.
   - The `--user` and `--password` options specify your Coverity Connect user name and password.

- The `--stream` option specifies the Coverity Connect stream that will receive source and defects from the Broker.

- The `--dir` option identifies the intermediate directory that you specified for the `cov-build` command. The Broker sends resources that are in the directory to Coverity Connect.

The console displays the following sort of output when you start the Broker:

```
Dynamic Analysis Broker 5.3.0 (cda5.3t-push-696.1) : ...
Using Java 1.6.0_21 ...

Saving logs and other information to run directory at
'/home/jdoe/CIC/dynamic-analysis/demo/cda_data/broker_1'.

Connecting to server at cim.example.com:9090.
Authenticating as 'jdoe'.
Preparing to commit to dynamic stream 'Example-dynamic'.
[SOURCE_COMMIT] starting xrefs.
[BROKER]: listening for agent connections on port 4422.
```

The console displays the following sort of output during the source commit to Coverity Connect:

```
[SOURCE_COMMIT] finished xrefs.
[SOURCE_COMMIT] starting source commit.
[SOURCE_COMMIT] finished source commit.
[SOURCE_COMMIT] output: 2010-10-26 18:05:58 UTC - Committing 8 files...
|0----------25-----------50----------75---------100|
****************************************************
2010-10-26 18:05:59 UTC - Committing cross-references for 8 files...
|0----------25-----------50----------75---------100|
****************************************************
2010-10-26 18:06:00 UTC - Committing 102 functions...
|0----------25-----------50----------75---------100|
****************************************************
2010-10-26 18:06:00 UTC - Committing 5 output files...
|0----------25-----------50----------75---------100|
****************************************************
New snapshot ID 10009 added.

Sending queued defects to server.  Subsequent defects will be streamed.
```

For additional information about this command and its options, see the `cov-da-start-broker` command documentation.

2. Proceed to Section 2.4, “Step 4: Run a program with the Dynamic Analysis Agent”.

## 2.4. Step 4: Run a program with the Dynamic Analysis Agent

The Dynamic Analysis Agent sends the defects that it finds in your running Java program to the Dynamic Analysis Broker, which forwards the data to a Dynamic Analysis stream in Coverity Connect, where you can view and triage the defects. After defining a Dynamic Analysis stream in Coverity Connect and starting the Dynamic Analysis Broker, you can run your program with the Dynamic Analysis Agent.

**To run the Dynamic Analysis Agent:**

1. While the Broker is running, open a new console to your local copy of the Dynamic Analysis `/demo` directory and execute the following command from it:

   ```
   > cov-build --test-capture --java-da --da-broker localhost \
     --build-id-dir intermediate_dir_name \
     --log-dir intermediate_dir_name \
     java -cp classes simple.Example
   ```

   With the `--java-da` and `--da-broker options`, the `cov-build` command injects the Java Dynamic Analysis agent into JVM invocations under an arbitrary command. In this case, the command is a single direct JVM invocation.

   Alternatively, you can manually configure the agent for a JVM invocation, as in this command:

   ```
   > java -javaagent:<install_dir>dynamic_analysis/dynamic-analysis/
       dynamic-analysis.jar -cp classes simple.Example
   ```

   On Windows, you need to surround the options in double quotes if you use a directory path that contains spaces. For example:

   ```
   > java "-javaagent:\Program Files\Coverity Integrity
           Center\dynamic-analysis\dynamic-analysis.jar"
       -cp classes simple.Example
   ```

   **Note**

   This step assumes that you usually start your Java applications with `java <arguments>`, so it adds the `-javaagent` argument to start the Dynamic Analysis Agent. The command takes the following form:

   ```
   > java -javaagent:<install_dir>dynamic_analysis/dynamic-analysis/
       dynamic-analysis.jar[=options] <arguments>
   ```

   Here, `options` specifies a list of `optionname=value` settings separated by commas without spaces. These options specify the behavior of the dynamic analysis. See Appendix A for examples and a description of the Dynamic Analysis Agent options.

   After you execute the command line, the Dynamic Analysis Agent watches the program run and looks for actual and potential race conditions, deadlocks, and resource leaks.

   The console displays the following sort of output when you run the program:

```
Dynamic Analysis for Java 5.0.0 (cda5.0t-push-516.169) : . . .
  Using Java 1.6.0_14-ea (from Sun Microsystems Inc.) on . . .
  Connecting to DA broker at localhost:4422
  Java agent transformer installed
  *** RESOURCE_LEAK example
```

```
  *** RACE_CONDITION example
      race=-30
      race=74
      race=-3196
      race=-5359
      race=-1450
      race=264
      race=980
      race=-3240
      race=-12688
      race=-1619
  *** DEADLOCK example (this example might actually deadlock and need
        to be forcibly terminated)
```

Once the Dynamic Analysis Agent connects to the Broker, the Broker console displays the following sort of output:

```
[AGENT:localhost, workload:run_1, timestamp: . . .]: agent connected.
[AGENT:localhost, workload:run_1, timestamp: . . .]: finished
  normally.
Reported 3 defects before defect merging ( DEADLOCK:1
RACE_CONDITION:1 RESOURCE_LEAK:1 )
```

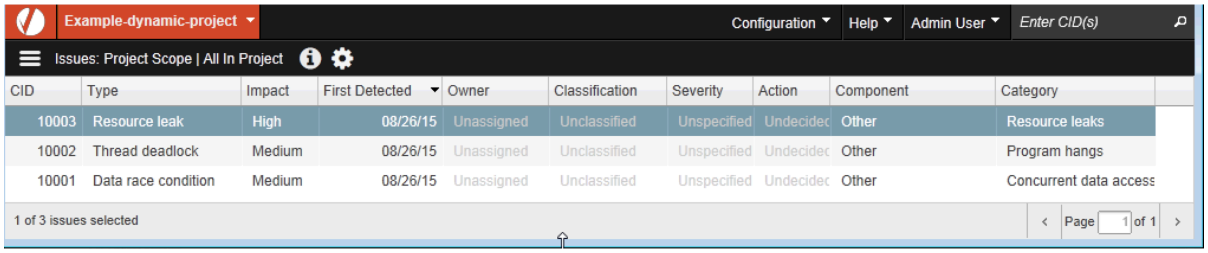
The second line, which includes `finished normally`, can happen only after you kill a hung process.

2. Log into Coverity Connect and view the four defects in the *Example* project.

   Note that you can see Dynamic Analysis defects in Coverity Connect as soon as the Broker finishes sending source code to Coverity Connect. You do not need to wait until your tests finish or the broker shuts down.

   To see the defects, click the link to the *Example-dynamic* project from the Projects menu.

**Figure 2.1. Defects in Coverity Connect Projects screen**

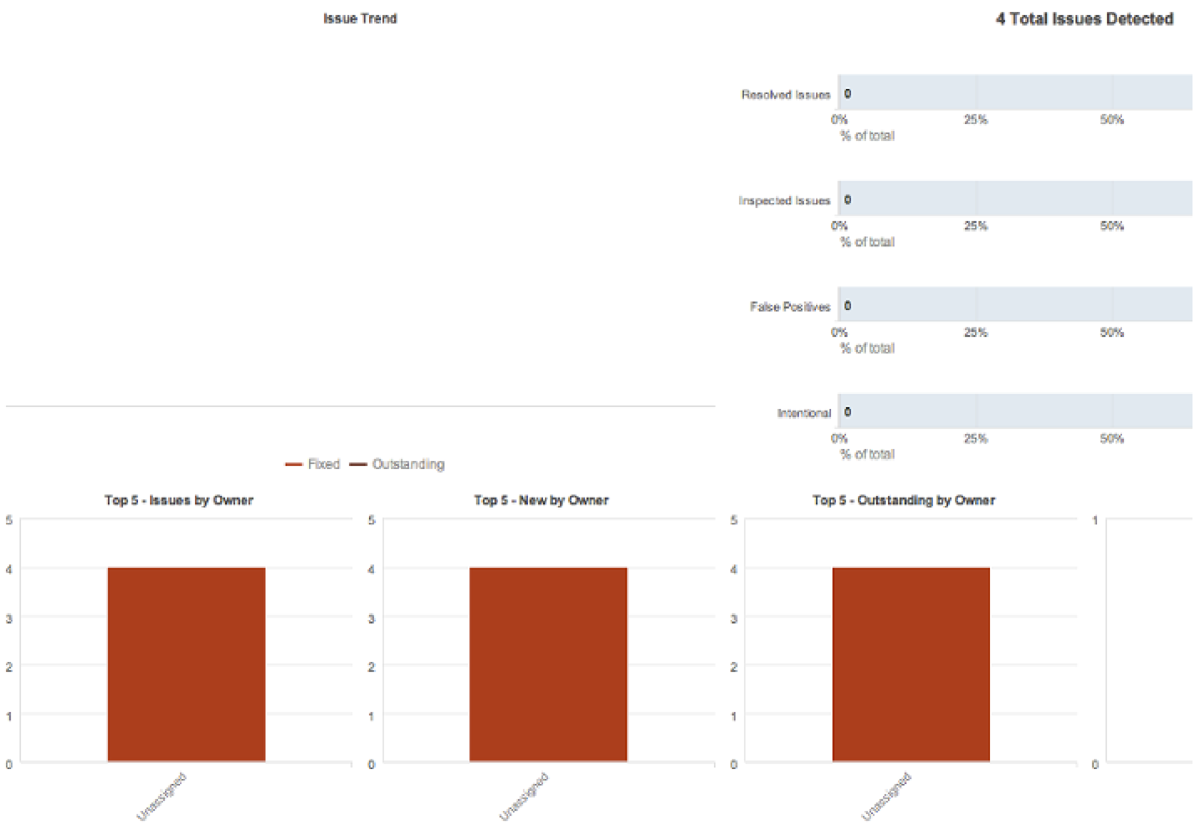


| CID | Type | Impact | First Detected | Owner | Classification | Severity | Action | Component | Category |
|---|---|---|---|---|---|---|---|---|---|
| 10003 | Resource leak | High | 08/26/15 | Unassigned | Unclassified | Unspecified | Undecidec | Other | Resource leaks |
| 10002 | Thread deadlock | Medium | 08/26/15 | Unassigned | Unclassified | Unspecified | Undecidec | Other | Program hangs |
| 10001 | Data race condition | Medium | 08/26/15 | Unassigned | Unclassified | Unspecified | Undecidec | Other | Concurrent data access |

1 of 3 issues selected    Page 1 of 1

You can also open the Quality dashboard in Coverity Connect to see a graphical representation of the defect total for the project.

**Figure 2.2. Defects in the Coverity Connect Dashboard screen**



For additional examples from this exercise, see Chapter 3, *Dynamic Analysis defects reports*.

3. Proceed to Section 2.5, “Step 5: Stop the Broker”.

## 2.5. Step 5: Stop the Broker

After the program finishes, it is a good practice to stop the Broker.

**To stop the Broker:**

- From another console, execute the `cov-stop-da-broker` command:

```
> <install_dir>dynamic_analysis/dynamic-analysis/bin/cov-stop-da-broker
```

The console displays the following output when you stop the Broker:

```
Sending request (BROKER_CLEAN_SHUTDOWN) to broker at localhost:4422.

[BROKER]: clean shutdown: shutting down when agents and source commit finish.
```

```
[BROKER]: done listening for agent connections.
[BROKER]: waiting for 0 agent(s) to complete.
[BROKER] closed connection with server.
[BROKER] sent 3 defects to the server.
```

**Note**

If you do not stop the Broker, it will stop by default 600 seconds after the last Dynamic Analysis Agent disconnects. You can change the number of seconds by using the `--shutdown-after` option to `cov-start-da-broker`.

See the `cov-stop-da-broker` reference page in the *Coverity 2020.12 Command Reference* for a list and description of the command options.

## 2.6. Using an Ant build file to run Dynamic Analysis on your program

You can also use an Ant build file to run Dynamic Analysis on a Java program. The following procedure provides general guidelines for adding Dynamic Analysis functionality to your Ant build file.

**Note**

To use a sample build file that follows these guidelines, see Section 2.6.1, “Using a sample Ant build file”.

For recommendations that supplement the steps in this procedure, see Section 2.7, “Deploying Dynamic Analysis in a production environment”.

**To add Dynamic Analysis Ant tasks to a build file:**

1. Include the following lines inside the `<project>` element in the build file:

   ```
   <typedef resource="com/coverity/anttask.xml"
     classpath=<install_dir>dynamic_analysis/library/coverity-anttask.jar">
   ```

2. Incorporate any of the following Dynamic Analysis Ant tasks into the build file:

   `cov-dynamic-analyze-java`
   : Run a Java program with Dynamic Analysis enabled using an Ant task.

     For recommended build techniques, see Section 2.7, “Deploying Dynamic Analysis in a production environment”.

   `cov-dynamic-analyze-junit`
   : Run tests from the JUnit testing framework with Dynamic Analysis enabled.

   `cov-start-da-broker`
   : Start the Dynamic Analysis Broker from an Ant task.

   `cov-stop-da-broker`
   : Stop the Dynamic Analysis Broker using an Ant task.

For more information about these tasks, see the *Coverity 2020.12 Command Reference*.

### 2.6.1. Using a sample Ant build file

Dynamic Analysis provides a sample Ant build file, `<install_dir>dynamic_analysis/dynamic-analysis/demo/build.xml`, that executes a Java class called `simple.Example`. This program is described in Section 2.2, “Step 2: Provide a program to be dynamically analyzed”.

**To run Dynamic Analysis on a sample program with `build.xml`:**

1. If you have not done so already, create a project in Coverity Connect called *Example* that preconfigures a set of streams for Dynamic Analysis.

   This step generates a Dynamic Analysis stream. If you performed this step in Section 2.1, you do not need to repeat it. You can proceed to the next step, instead.

2. If you have not done so already, create a local copy of the Dynamic Analysis `/demo` directory.

   This directory is located under `<install_dir>dynamic_analysis/dynamic-analysis/`. If you performed this step in Section 2.2, you do not need to repeat it. You can proceed to the next step, instead.

3. Update the attributes in your local copy of `demo/Example.broker-config` to match the Coverity Connect installation that supports your *Example-dynamic* stream.

   The default configuration file looks something like the following:

   ```
   # Update these attributes to match your Coverity Connect configuration.
   host=cim.example.com
   dataport=9090
   user=jdoe
   password=secret
   stream=Example-dynamic
   ```

   If you do not know the values to use, contact your Coverity Connect administrator.

4. Run the following Ant command from your local copy of the `demo` directory:

   ```
   > ant da.simple.example
   ```

   The following console output is edited and abbreviated:

   ```
   > ant da.simple.example
   Buildfile: build.xml
   . . .
   simple.agent:
   [cov-dynamic-analyze-java] Loading Dynamic Analysis . . .
   [cov-dynamic-analyze-java] Dynamic Analysis for Java 5.0.0 . . .
   [cov-dynamic-analyze-java] Using Java 1.6.0_10 . . .
   [cov-dynamic-analyze-java] Connecting to DA broker at localhost:4422
   [cov-dynamic-analyze-java] Java agent transformer installed
   [cov-dynamic-analyze-java] *** RESOURCE_LEAK example
   ```

```
[cov-dynamic-analyze-java] *** RACE_CONDITION example
[cov-dynamic-analyze-java]     race=4378
[cov-dynamic-analyze-java]     race=28059
[cov-dynamic-analyze-java]     race=736
[cov-dynamic-analyze-java]     race=633
[cov-dynamic-analyze-java]     race=4574
[cov-dynamic-analyze-java]     race=-720
[cov-dynamic-analyze-java]     race=-1256
[cov-dynamic-analyze-java]     race=-3443
[cov-dynamic-analyze-java]     race=897
[cov-dynamic-analyze-java]     race=-1709
[cov-dynamic-analyze-java] *** DEADLOCK example (example might need to be
terminated)
```

### 2.6.2. Description of the sample Ant build file

The comments in `<install_dir>dynamic_analysis/dynamic-analysis/demo/build.xml` describe its targets and how they work together. The following lines define the property name and a path ID that is called by the `da.simple.example` target.

```
<property name="config.file" location="Example.broker-config"/>
<!-- classpath to run the examples and demos -->
<path id="demos.classpath">
  <pathelement location="classes"/>
  <pathelement location="lib/plot.jar"/>
</path>
```

The `da.simple.example` target calls the following targets:

- `test.cim.connection`

  Validates the configuration in `Example.broker-config`. This target uses `cov-start-da-broker` with `onlytestconnection="true"`. If this target fails, you need to fix the configuration file.

  ```
  <target name="test.cim.connection"
          description="Test that the settings in 'Example.broker-config'
                       suffice to commit defects &CIM;.">
    <cov-start-da-broker
      failonerror="true"
      configfile="${config.file}"
      onlytestconnection="true"
    />
  </target>
  ```

- `source.commit`

  Commits the source code to Coverity Connect. This target uses `cov-start-da-broker` with `onlysourcecommit="true"`. Splitting the source commit from the dynamic defect commit – especially in Ant build files– is a good practice because it ensures that the whole target will fail if the source commit fails.

  ```
  <property name="demos.idir" location="example-idir"/>
  ```

```
<target name="source.commit" depends="build-project">
  <cov-start-da-broker
    failonerror="true"
    configFile="${config.file}"
    dir="${demos.idir}"
    onlySourceCommit="true"
  />
</target>
```

- `broker.listen`

  Starts the Broker in the background. This target uses `cov-start-da-broker` with `onlylisten="true"` to listen for Dynamic Analysis Agent connections and to stream their defects to Coverity Connect.

  ```
  <target name="broker.listen">
    <cov-start-da-broker
      failonerror="true"
      configfile="${config.file}"
      onlylisten="true"
     />
  </target>
  ```

- `simple.agent`

  Runs the example with the Dynamic Analysis Agent while connected to the Broker. This target uses the `cov-dynamic-analyze-java` Ant task, which sets the `repeat-connect` option of the Agent to 20 by default. Note the similarity of this task to the `simple.example` task, which is how you would run it without Dynamic Analysis.

  ```
  <target name="simple.agent" depends="build-project">
    <cov-dynamic-analyze-java
      classname="simple.Example" failonerror="true">
      <classpath refid="demos.classpath"/>
    </cov-dynamic-analyze-java>
  </target>
  ```

- `broker.shutdown`

  Shuts down the Broker. This target uses the `cov-stop-da-broker` Ant task.

  ```
  <target name="broker.shutdown">
    <cov-stop-da-broker shutdowntype="clean"/>
  </target>
  ```

## 2.7. Deploying Dynamic Analysis in a production environment

The following techniques are recommended.

- Build your program for use with Dynamic Analysis:
  - To help ensure that the Dynamic Analysis Broker sends all of your source and class files to Coverity Connect, use `cov-build` (instead of the alternative build process) to capture these resources

  before you run the Broker. For examples of these build processes, see Section 2.2, “Step 2: Provide a program to be dynamically analyzed”.

  - Make sure the source code and bytecode that you capture through the build process is the same as the bytecode that you run with Dynamic Analysis. Mismatches are likely to make Coverity Connect report incorrect line numbers for the location of defects, events in defects, and stack traces in events. As a consequence, it might be difficult for developers to find the issues in the source code.
  - Make sure that your bytecode contains debug symbols and that no code coverage tools have obfuscated or mangled it. Deployments that use aspect-oriented programming tools such as AspectJ or other bytecode re-writing tools such as the EMMA code coverage tool can remove debugging information or make it inaccurate. Running Dynamic Analysis on bytecode that has been altered by such tools can cause an improper alignment between the Dynamic Analysis defect reports in Coverity Connect and the source code.

- Send the source code and defect reports to Coverity Connect separately. This practice allows you to manage the two steps independently on different machines or at different times. If a problem occurs, the two steps can fail separately.

  On a build machine, you can run `cov-build` and then use the `--only-source-commit` option to `cov-start-da-broker` to send source code to Coverity Connect without any defect reports. At another time, and perhaps on a different machine, you can run `cov-start-da-broker` with the `--only-listen` option and then run your tests with the Dynamic Analysis Agent.

  If the broker and agent are running on different machines, you need to set the hostname or IP address on which the Broker is running when you start the Dynamic Analysis Agent. You use the `broker-host` option for this purpose. For examples, see Section A.1, “Sample Dynamic Analysis start commands”. For a list of options that you might find useful, see Section A.2, “Dynamic Analysis Java Agent options and Ant attributes”. You need to make sure that no firewall blocks the connection.

- Run tests that exercise as much of your program as possible and that perform multi-threaded operations: Simple unit tests are unlikely to result in good defect reports. For details on this topic, see Section 1.2, “Obtaining thorough analysis results”.

# Chapter 3. Dynamic Analysis defects reports

## Table of Contents

Dynamic Analysis generates reports for any RESOURCE_LEAK, RACE_CONDITION, and DEADLOCK defects that it finds in a Java program. For more information about the specific types of defects, see the *Coverity 2020.12 Checker Reference*. We use the program in `<install_dir>/dynamic-analysis/demo/src/simple/Example.java` from Chapter 2 as a case study to describe Dynamic Analysis defect reports. To follow along with the discussion using `Example.java`, bring up Coverity Connect and select the project named *Example-dynamic*.

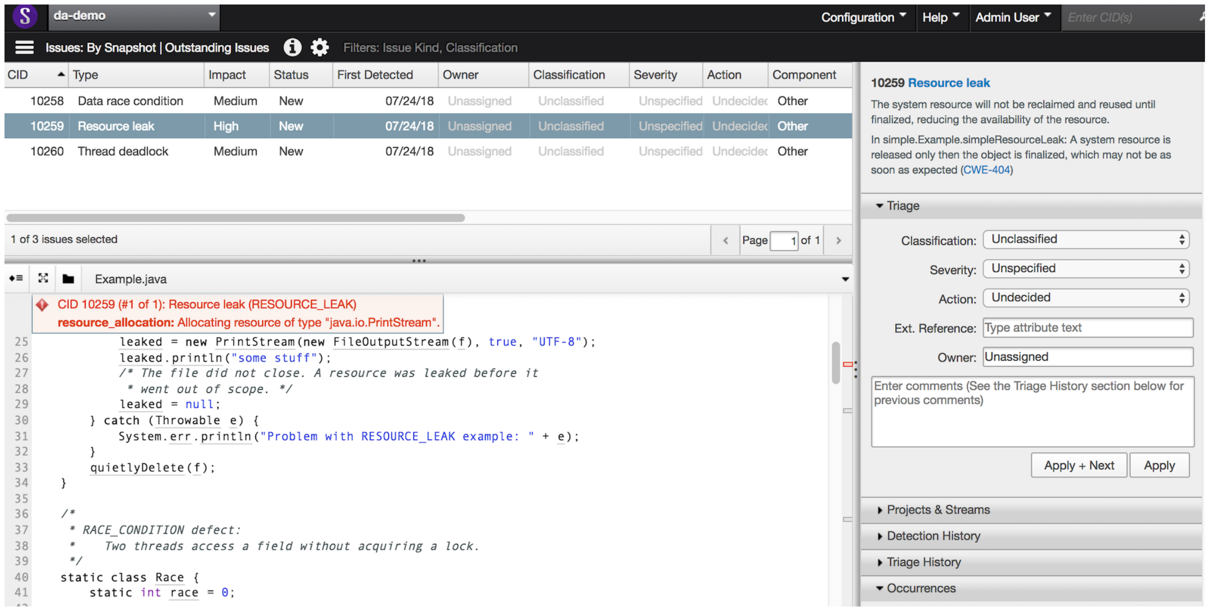
**Note**

If you are following the case study, you have installed and started Coverity Connect, and stepped through the procedures in Chapter 2. At this point you have compiled, run, dynamically analyzed, and committed defect data for `Example.java` to Coverity Connect. After selecting your project, you'll see a list of returned defects, similar to those in Figure 2.1, “Defects in Coverity Connect Projects screen”.

## 3.1. A RESOURCE_LEAK defect

From your defects list, click the *Resource leak* defect. Coverity Connect displays a screen that looks like Figure 3.1, “A RESOURCE_LEAK defect reported by Dynamic Analysis and displayed in Coverity Connect”.

**Figure 3.1. A RESOURCE_LEAK defect reported by Dynamic Analysis and displayed in Coverity Connect**


da-demo
Configuration
Help
Admin User
Enter CID(s)
Issues: By Snapshot | Outstanding Issues
Filters: Issue Kind, Classification

| CID | Type | Impact | Status | First Detected | Owner | Classification | Severity | Action | Component |
|---|---|---|---|---|---|---|---|---|---|
| 10258 | Data race condition | Medium | New | 07/24/18 | Unassigned | Unclassified | Unspecified | Undecided | Other |
| 10259 | Resource leak | High | New | 07/24/18 | Unassigned | Unclassified | Unspecified | Undecided | Other |
| 10260 | Thread deadlock | Medium | New | 07/24/18 | Unassigned | Unclassified | Unspecified | Undecided | Other |

1 of 3 issues selected

Page 1 of 1

Example.java

CID 10259 (#1 of 1): Resource leak (RESOURCE_LEAK)
**resource_allocation:** Allocating resource of type "java.io.PrintStream".

```
            leaked = new PrintStream(new FileOutputStream(f), true, "UTF-8");
            leaked.println("some stuff");
            /* The file did not close. A resource was leaked before it
             * went out of scope. */
            leaked = null;
        } catch (Throwable e) {
            System.err.println("Problem with RESOURCE_LEAK example: " + e);
        }
        quietlyDelete(f);
    }

    /*
     * RACE_CONDITION defect:
     *    Two threads access a field without acquiring a lock.
     */
    static class Race {
        static int race = 0;
```

10259 Resource leak

The system resource will not be reclaimed and reused until finalized, reducing the availability of the resource.

In simple.Example.simpleResourceLeak: A system resource is released only then the object is finalized, which may not be as soon as expected (CWE-404)

Triage

| | |
|---|---|
| Classification: | Unclassified |
| Severity: | Unspecified |
| Action: | Undecided |
| Ext. Reference: | Type attribute text |
| Owner: | Unassigned |

Enter comments (See the Triage History section below for previous comments)

In the image above, you can see the defect description and impact statement that is reported by Dynamic Analysis when a RESOURCE_LEAK is observed. It shows that some resource (like a socket or a file handle) has been allocated, but not properly closed. The resource has leaked. The bottom section of the report lists the streams where the defects were reported and the events that provide clues for finding the sources of the defect.

Here's an example of some code where Dynamic Analysis, reported in the `simpleResourceLeak` method, shows the RESOURCE_LEAK defect events (in italicized comments):

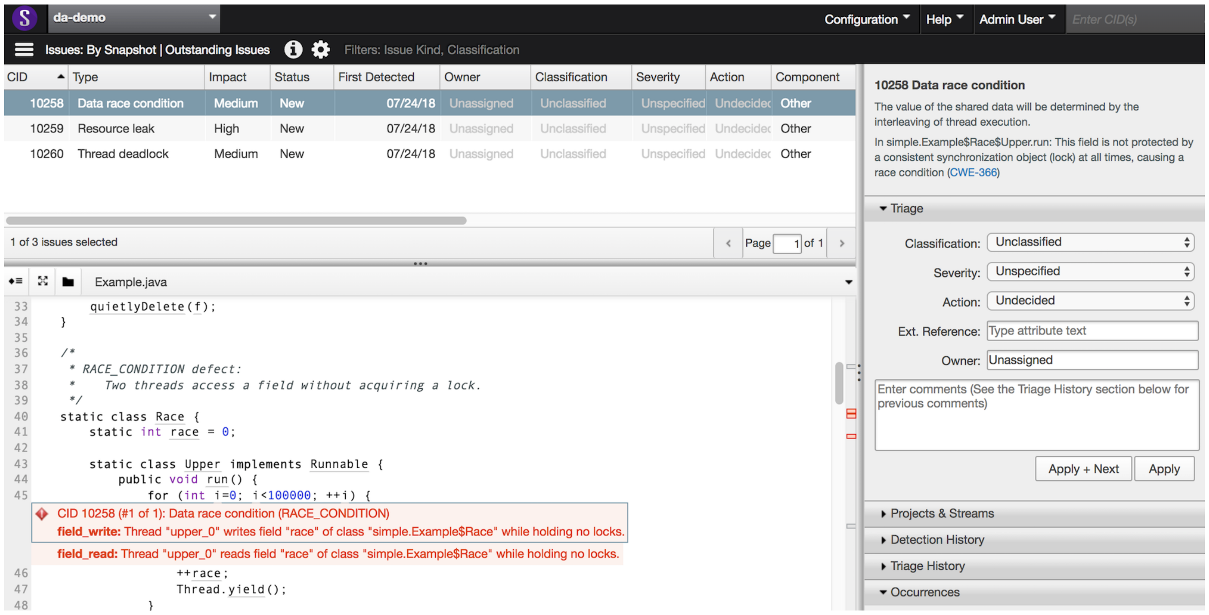
```
 /*
 * RESOURCE_LEAK defect:
 *     File is opened for output and later not closed.
 */
static PrintStream leaked;
public static void simpleResourceLeak() {
    System.out.println("*** RESOURCE_LEAK example");
    File f = null;
    try {
        f = File.createTempFile("da-example", null);
/* Allocating resource of type "java.io.PrintStream". */
        leaked = new PrintStream(new FileOutputStream(f), true, "UTF-8");
        leaked.println("some stuff");
        /* The file did not close. A resource was leaked before it
         * went out of scope. */
        leaked = null;
    } catch (Throwable e) {
        System.err.println("Problem with RESOURCE_LEAK example: " + e);
    }
    quietlyDelete(f);
}
```

Dynamic Analysis shows a `resource_allocation` event at the location in the code where the resource is opened. The left pane of Coverity Connect shows the stack trace of method calls that lead up to this event. In the preceding example, Dynamic Analysis observes that the `FileOutputStream` is opened and not closed, and reports a defect. The file handle (or file descriptor) associated with the `FileOutputStream` remains open until `leaked` goes out of scope and the garbage collector gets around to reclaiming its storage. For details about the RESOURCE_LEAK checker, see the *Coverity 2020.12 Checker Reference*.

## 3.2. A RACE_CONDITION defect

If you click the RACE_CONDITION defect, Coverity Connect displays a screen similar to Figure 3.2.

**Figure 3.2. A Dynamic Analysis RACE_CONDITION defect displays in Coverity Connect**



da-demo
Configuration
Help
Admin User
Enter CID(s)
Issues: By Snapshot | Outstanding Issues
Filters: Issue Kind, Classification

| CID | Type | Impact | Status | First Detected | Owner | Classification | Severity | Action | Component |
|---|---|---|---|---|---|---|---|---|---|
| 10258 | Data race condition | Medium | New | 07/24/18 | Unassigned | Unclassified | Unspecified | Undecided | Other |
| 10259 | Resource leak | High | New | 07/24/18 | Unassigned | Unclassified | Unspecified | Undecided | Other |
| 10260 | Thread deadlock | Medium | New | 07/24/18 | Unassigned | Unclassified | Unspecified | Undecided | Other |

1 of 3 issues selected — Page 1 of 1

Example.java

```
        quietlyDelete(f);
    }

    /*
     * RACE_CONDITION defect:
     *    Two threads access a field without acquiring a lock.
     */
    static class Race {
        static int race = 0;

        static class Upper implements Runnable {
            public void run() {
                for (int i=0; i<100000; ++i) {
CID 10258 (#1 of 1): Data race condition (RACE_CONDITION)
field_write: Thread "upper_0" writes field "race" of class "simple.Example$Race" while holding no locks.
field_read: Thread "upper_0" reads field "race" of class "simple.Example$Race" while holding no locks.
                    ++race;
                    Thread.yield();
                }
```

**10258 Data race condition**

The value of the shared data will be determined by the interleaving of thread execution.

In simple.Example$Race$Upper.run: This field is not protected by a consistent synchronization object (lock) at all times, causing a race condition (CWE-366)

Triage

Classification: Unclassified

Severity: Unspecified

Action: Undecided

Ext. Reference: Type attribute text

Owner: Unassigned

Enter comments (See the Triage History section below for previous comments)

In the image above, we see the defect description and impact statement of a RACE_CONDITION that's been reported by Dynamic Analysis. A RACE_CONDITION defect is found when multi-threaded accesses to a field are not protected by synchronizing on the same lock. The bottom section of the report lists the streams where the defects are reported and also shows the events that help you identify the source of the defect

Here's an example of some code that shows the RACE_CONDITION defect events in italicized comments:

```
/*
 * RACE_CONDITION defect:
 *    Two threads access a field without acquiring a lock.
 */
static class Race {
    static int race = 0;

    static class Upper implements Runnable {
        public void run() {
            for (int i=0; i<100000; ++i) {
/* Thread "upper_0" writes field "race" of class "simple.Example$Race" while holding
 no locks. */
/* Thread "upper_0" reads field "race" of class "simple.Example$Race" while holding no
 locks. */
                ++race;
                Thread.yield();
            }
        }
    }

```

```
             static class Downer implements Runnable {
                 public void run() {
                     for (int i=0; i<100000; ++i) {
/* Thread "downer_0" reads field "race" of class "simple.Example$Race" while holding
 no locks. */
                         --race;
                         Thread.yield();
                     }
                 }
             }

             public static void simpleRaceCondition() {
                 System.out.println("*** RACE_CONDITION example");
                 for (int i=0; i<10; ++i) {
                     race = 0;
                     runThreadsToCompletion(
                            new Thread(new Upper(), "upper_" + i)
                          , new Thread(new Downer(), "downer_" + i)
                     );
                     System.out.println("    race=" + race);
                 }
             }
         }
```

You might expect the preceding code to print `race=0` after each iteration of the `for` loop in `simpleRaceCondition()`. Each iteration of the `for` loop in `simpleRaceCondition()` starts two concurrent threads. One thread runs `Upper.run()` and the other runs `Downer.run()`. The `Upper` thread increments `race` 100,000 times and the `Downer` thread decrements `race` 100,000 times, which one might expect to make `race==0`. However, if you run the preceding code, you see many different outputs for `race`. The output depends on the details of how the `Upper` and `Downer` threads are scheduled. For example, the following thread schedule shows what can go wrong:

1. `Upper` reads `race==0`.

2. `Downer` reads `race==0`.

3. `Upper` computes `race+1==1` and stores `1` back into `race`.

4. `Downer` computes `race-1==-1` and stores it back into `race`.

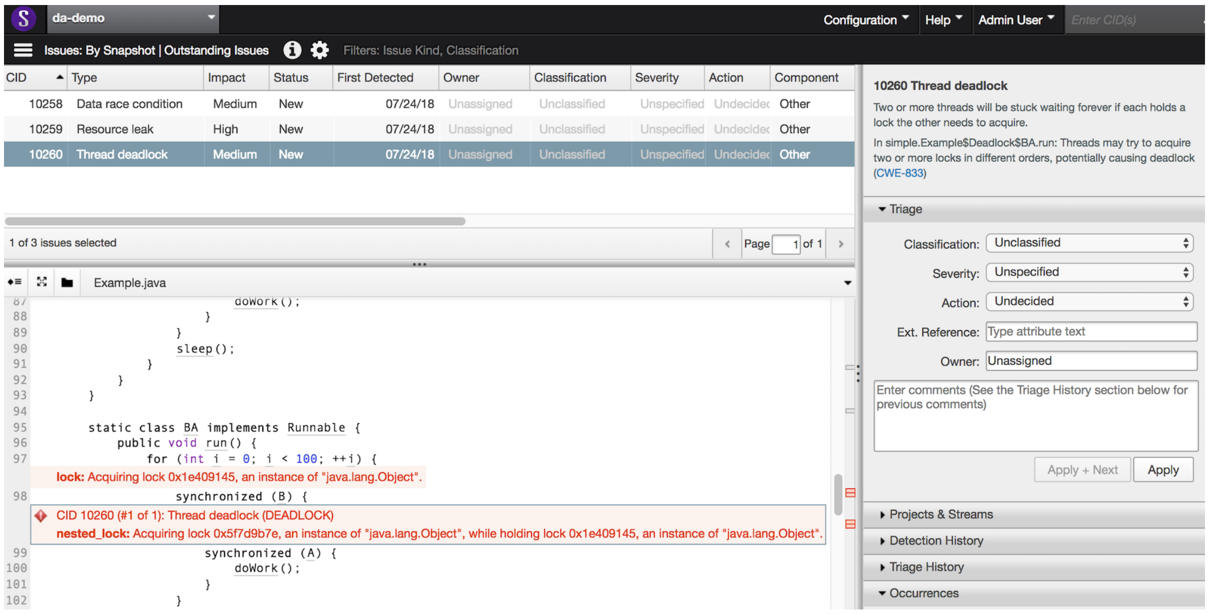
5. Now `race==-1`.

Many other variations are possible, hence the many numbers after "`race=`" in the output. Dynamic Analysis notices this race condition in the output. When Dynamic Analysis watches this preceding program run, it notices that the field `race` is accessed by two different threads that do not hold a lock that could guard `race` and prohibit problematic thread schedules. Thus, Dynamic Analysis reports a potential RACE_CONDITION defect in this code. Notice that Dynamic Analysis reports `field_read` and `field_write` events where threads `upper_0` and `downer_0` access `race`.

## 3.3. A DEADLOCK defect

If you click the DEADLOCK defect, Coverity Connect displays a screen similar to Figure 3.3:

**Figure 3.3. A Dynamic Analysis DEADLOCK defect displays in Coverity Connect**


da-demo
Configuration
Help
Admin User
Enter CID(s)
Issues: By Snapshot | Outstanding Issues
Filters: Issue Kind, Classification

| CID | Type | Impact | Status | First Detected | Owner | Classification | Severity | Action | Component |
|---|---|---|---|---|---|---|---|---|---|
| 10258 | Data race condition | Medium | New | 07/24/18 | Unassigned | Unclassified | Unspecified | Undecidec | Other |
| 10259 | Resource leak | High | New | 07/24/18 | Unassigned | Unclassified | Unspecified | Undecidec | Other |
| 10260 | Thread deadlock | Medium | New | 07/24/18 | Unassigned | Unclassified | Unspecified | Undecidec | Other |

1 of 3 issues selected

Page 1 of 1

Example.java

```
                    doWork();
                }
            }
            sleep();
        }
    }
}

static class BA implements Runnable {
    public void run() {
        for (int i = 0; i < 100; ++i) {
lock: Acquiring lock 0x1e409145, an instance of "java.lang.Object".
            synchronized (B) {
CID 10260 (#1 of 1): Thread deadlock (DEADLOCK)
nested_lock: Acquiring lock 0x5f7d9b7e, an instance of "java.lang.Object", while holding lock 0x1e409145, an instance of "java.lang.Object".
                synchronized (A) {
                    doWork();
                }
            }
```

**10260 Thread deadlock**

Two or more threads will be stuck waiting forever if each holds a lock the other needs to acquire.

In simple.Example$Deadlock$BA.run: Threads may try to acquire two or more locks in different orders, potentially causing deadlock (CWE-833)

Triage

| | |
|---|---|
| Classification: | Unclassified |
| Severity: | Unspecified |
| Action: | Undecided |
| Ext. Reference: | Type attribute text |
| Owner: | Unassigned |

Enter comments (See the Triage History section below for previous comments)

In this screenshot, Dynamic Analysis reports a DEADLOCK defect that explains the defect description and impact statement (on the right side of the screen). A DEADLOCK occurs when threads try to acquire two or more locks in different orders. The bottom section of the report lists the streams where the defects were reported and the event that provides a clue for finding the source of the defect.

Here's an example of some code that shows the DEADLOCK defect events in italicized comments:

```
    /*
     * DEADLOCK defect:
     *     Two threads acquire two locks in different orders.
     */
    static class Deadlock {
        static Object A = new Object();
        static Object B = new Object();

        static class AB implements Runnable {
            public void run() {
                for (int i = 0; i < 100; ++i) {
/* Acquiring lock 0x1d03a4e, an instance of "java.lang.Object". */
                    synchronized (A) {
/* Acquiring lock 0xd5cabc, an instance of "java.lang.Object", while holding lock
 0x1d03a4e, an instance of "java.lang.Object". */
                        synchronized (B) {
                            doWork();
                        }
                    }
                    sleep();
                }
            }
        }

        static class BA implements Runnable {
            public void run() {
                for (int i = 0; i < 100; ++i) {
/* Acquiring lock 0xd5cabc, an instance of "java.lang.Object". */
                    synchronized (B) {
/* Acquiring lock 0x1d03a4e, an instance of "java.lang.Object", while holding lock
 0xd5cabc, an instance of "java.lang.Object". */
                        synchronized (A) {
                            doWork();
                        }
                    }
                    sleep();
                }
            }
        }

        public static void simpleDeadlock() {
            System.out.println("*** DEADLOCK example (this example may"
                + " actually deadlock and need to be forcibly terminated)");
            runThreadsToCompletion(
                    new Thread(new AB(), "AB")
                  , new Thread(new BA(), "BA")
            );
        }
     }

```

You might expect the preceding example to call `doWork()` 200 times while holding locks A and B (100 times from `AB.run()` and 100 times from `BA.run()`). Often, those 200 calls are all that happens. However, the AB thread and the BA thread can deadlock: they can enter a state in which both are stuck and neither can escape.

Consider what happens if AB acquires lock A, but before it can acquire lock B, BA acquires it. Now thread AB is holding A and is waiting on lock B, but thread BA is holding lock B and is waiting on lock A. Neither thread will release the lock it holds nor acquire the lock it waits for, and so neither thread can progress. As Dynamic Analysis watches this program run, it notices the potential for these threads to hold locks while waiting for other locks in such a way that deadlock is possible. Thus, Dynamic Analysis reports a DEADLOCK on this code.

# Chapter 4. Troubleshooting

## Table of Contents

This chapter describes various Dynamic Analysis problems and solutions.

## 4.1. I'm having problems starting the Dynamic Analysis Broker or connecting to Coverity Connect.

Solution:

Use the following steps to ensure that the Broker and Coverity Connect operate and connect properly.

1. Check the Broker start command options.

   Verify Coverity Connect server hostname (specified with the `--host` command line option) the user name (`--user`), password (`--password`), and dynamic analysis stream name (`--stream`). The host name for Coverity Connect server is the same as the one you connect to when accessing the Coverity Connect web interface. The user name and password are the same as those used to log on to the web interface of Coverity Connect. All of these options, excluding stream, are similar to options for the `cov-commit-defects` command (the command used to commit Coverity Analysis results to Coverity Connect). If you are unsure about these settings, check with your Coverity Connect administrator.

   To test the Broker connection to Coverity Connect, run a valid `cov-da-start-broker` command with the `--only-test-connection` option. If you get a message similar to the following, then your options are correct and the Broker successfully listens for defects for Dynamic Analysis instances:

   ```
   SUCCESS: everything seems to work (see messages above). Exiting now
   since this is only a test.
   ```

2. Check the Broker source commit.

   You can test this by running the Broker source commit function separately from the Broker listen/send function (that is, the function that listens for Dynamic Analysis Agent connections and sends defects to Coverity Connect). Run the Broker with all of the options from step 1, plus `--only-source-commit` and any source commit options necessary to do a source commit. Wait for the source commit process to finish, then run the Broker with `--only-listen` and start the Dynamic Analysis Agents. Once the source commit function and the listen/send functions are properly debugged and configured, you can run both Broker functions together (the default behavior).

3. Make sure there is no firewall or other security software preventing the Broker from listening on its port, the Broker contacting Coverity Connect, or Agents contacting the Broker.

## 4.2. Where are the `cov-da-start-broker` log files and other information files?

By default, `cov-da-start-broker` log files and other information files are in the directory `<broker_dir>/cda_data/broker_<number>/`, where `<Broker_dir>` is the directory where you ran `cov-start-da-Broker`. The file `log_broker.txt` records Broker activity such as the status of a source commit, Agent connections and disconnections and warnings. The console on the machine where the Broker runs displays most of the same data, although `log_broker.txt` might contain additional data. The other log files are used for support. You can change the location of the run directory with the `--rundir` option.

## 4.3. In starting the Broker and Agent from a script, the Broker starts, but the Agent fails with `Could not connect to Broker`.

Example:

```
Fatal error: Could not connect to Broker at localhost:4422
(java.net.ConnectException: Connection refused)
```

Solution:

When starting the Broker and Agent from a script, be sure to set the `repeat-connect` option for the Agent to something greater than 0 (20 is a good starting point, but a slow network or overloaded machines might require a longer wait). The Broker needs time to start up and listen for Agent connections. Having the Agent start immediately and trying to connect might fail.

## 4.4. I think Dynamic Analysis Agent might be causing a problem with my program.

To determine if Dynamic Analysis Agent is causing a problem with your program, verify that your program runs correctly by itself in the same environment:

Does your Java program run by itself without errors?

- On the same machine?
- On the same JVM?
- With the same environment and configuration?
- With the same command line (if running your program from the command line, batch file, script and so on)?
- Running the `cov-dynamic-analyze-junit` task with `enabled="false"` or turning a `cov-dynamic-analyze-junit` task into a junit task or a `cov-dynamic-analyze-java` task into a Java task?
- Are you running on a supported JVM?

## 4.5. The Agent is running with a program when the VM runs out of memory and throws an `OutOfMemoryError` error that is related to heap memory.

**Note**

If this error occurs with the Broker, see Section 4.7, “The Broker runs out of memory and throws an OutOfMemoryError error that is related to heap memory.”.

Dynamic Analysis causes your application to use as much as eight times the normal memory. Adjust the maximum heap size by using the JVM `-Xmx` argument (for details, run the `java -X` command-line documentation).

For example:

```
java -Xmx2000
```

Switch to a new machine with more RAM, if possible.

If you are running on a 64-bit machine and a 64-bit JVM that supports the `-XX:+UseCompressedOops` option, use it. If you are using an older 64-bit VM without `-XX:+UseCompressedOops` support, upgrade. If you are using a 32-bit VM on a 64-bit machine with a lot of RAM, use a 64-bit JVM and the `-XX:+UseCompressedOops` option. If you can run the program on a smaller workload that demands less memory, then do so.

You can also exclude instrumentation of non-essential classes using the `exclude-instrumentation` or `instrument-only` options to the Agent (see Section A.2, “Dynamic Analysis Java Agent options and Ant attributes”). These options allow you to specify non-essential classes that Dynamic Analysis does not need to watch. In general such excluding causes Dynamic Analysis to run faster and use less memory, but it might also miss defects. In particular, it misses all `RESOURCE_LEAK` defects in excluded code, as well as any lock acquisitions or field accesses in excluded code that could contribute toward a `RACE_CONDITION` or `DEADLOCK` report. For example, one might exclude classes related to the application server or other third party code, or include only `com.mycompany`.

Another possibility is to live with the `OutofMemory` exceptions if the program is able to continue to run and Dynamic Analysis is able to report useful defects. Although unlikely, some programs can and do continue despite bursts where they go out of memory and skip some processing. If all else fails, run Dynamic Analysis once with the race detector disabled and again with only the race detector enabled.

## 4.6. The Agent is running with a program that dies with an `OutofMemoryError` error that is related to permanent generation (or `permgen`).

**Note**

If this error occurs with the Broker, see Section 4.8, “The Broker is running with a program that dies with an OutofMemoryError message about permanent generation (or permgen).”.

Increase the size of the permanent generation with the `-XX:MaxPermSize` command line option to the JVM.

For example:

```
-XX:MaxPermSize=128M
```

If you use a JVM other than the Sun Hotspot VM, the error message and command line option might be different.

**Note**

The Permanent Generation space is removed in Java 1.8. The JVM will ignore the options `-XX:PermSize` and `-XX:MaxPermSize`.

## 4.7. The Broker runs out of memory and throws an `OutOfMemoryError` error that is related to heap memory.

**Note**

If this error occurs with the Agent, see Section 4.5, “The Agent is running with a program when the VM runs out of memory and throws an OutOfMemoryError error that is related to heap memory.”.

Use one of the following solutions to increase the heap size:

- Create or edit a file called `CIC/dynamic-analysis/bin/cov-start-da-broker.vmoptions` that contains the following single line:

  ```
  -Xmx2000M
  ```

- Use the following option with the `cov-start-da-broker` command:

  ```
  -J-Xmx2000M
  ```

  You can test that the command is passing arguments to the JVM by running the following command and making sure that it prints the help message for the JVM:

  ```
  > cov-start-da-broker -J-help
  ```

- Specify the `INSTALL4J_ADD_VM_PARAMS` environment variable through your shell, in your shell script, or on the command line. The following example sets it on the command line:

  ```
  > INSTALL4J_ADD_VM_PARAMS="-Xmx2000M" cov-start-da-broker
  ```

  **Note**

  This setting does not work on Windows platforms.

  You can test that the setting is passing arguments to the JVM by running the following command and making sure that it prints the help message for the JVM:

```
> INSTALL4J_ADD_VM_PARAMS="-help" cov-start-da-broker
```

## 4.8. The Broker is running with a program that dies with an `OutofMemoryError` message about permanent generation (or `permgen`).

**Note**

If this error occurs with the Agent, see Section 4.6, “The Agent is running with a program that dies with an OutofMemoryError error that is related to permanent generation (or permgen).”.

**Note**

The Permanent Generation space is removed in Java 1.8. The JVM will ignore the options `-XX:PermSize` and `-XX:MaxPermSize`.

Use one of the following solutions to increase the `permgen` size:

- Create or edit a file called `CIC/dynamic-analysis/bin/cov-start-da-broker.vmoptions` that contains the following single line:

  `-XX:MaxPermSize=128M`

- Use the following option with the `cov-start-da-broker` command:

  `-J-XX:MaxPermSize=128M`

  You can test that the command is passing arguments to the JVM by running the following command and making sure that it prints the help message for the JVM:

  ```
  > cov-start-da-broker -J-help
  ```

- Specify the `INSTALL4J_ADD_VM_PARAMS` environment variable through your shell, in your shell script, or on the command line. The following example sets it on the command line:

  ```
  > INSTALL4J_ADD_VM_PARAMS="-XX:MaxPermSize=128M" cov-start-da-broker
  ```

  **Note**

  This setting does not work on Windows platforms.

  You can test that the setting is passing arguments to the JVM by running the following command and making sure that it prints the help message for the JVM:

  ```
  INSTALL4J_ADD_VM_PARAMS="-help" cov-start-da-broker
  ```

Allocate more memory if the 128M allocation is too small.

If you are having trouble committing source or do not want to wait for the source commit to complete, and you do not need to see the source, you can run the Dynamic Analysis Broker with the `--only-listen` option and commit defects without source.

## 4.9. How can I speed up the Dynamic Analysis process?

Use one or more of the following processes to speed up Dynamic Analysis:

- Give your program as much heap space as the process can allocate physical memory using `-Xmx`. This is described in Section 4.5, “The Agent is running with a program when the VM runs out of memory and throws an OutOfMemoryError error that is related to heap memory.”). More is better even if the application is not throwing an `OutOfMemoryError`.
- Exclude instrumentation of non-essential classes as described in Section 4.5, “The Agent is running with a program when the VM runs out of memory and throws an OutOfMemoryError error that is related to heap memory.”
- Run Dynamic Analysis twice. The first time, run it with only the `RACE_CONDITION` detector enabled. The second time, run it with only `DEADLOCK` and `RESOURCE_LEAK` enabled.
- If you have multiple tests to run, start concurrent Dynamic Analysis Agents on different machines, each of which runs on a different test or instance of your application. The Broker acts as a multiplexor that collects and compiles defect data from various Agents to send to Coverity Connect.

## 4.10. I get defects inside `javax.swing.*`. Are there defects in Swing?

Most likely your application is accessing Swing objects outside the event-dispatch thread. This is likely a bug. Look further up in the stack trace to locate where your code first accesses Swing. To use Swing properly within the presence of multiple threads, see this article: http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html

## 4.11. I get an `InvalidClassException` when I run Dynamic Analysis.

Other instrumentation tools, such as EMMA, produce class files with invalid debug information. The JVM verifier accepts these files, but Dynamic Analysis might not. If you see this problem try the following:

- Exclude instrumentation of the problematic class. That is, specify that Dynamic Analysis not watch its execution using the `exclude-instrumentation` or `instrument-only` options to the Agent (see Section A.2, “Dynamic Analysis Java Agent options and Ant attributes”).
- Send the class or a reproducer to Coverity if possible.

## 4.12. I've committed source code to the wrong stream, now all the source code in that stream is gone.

Commit the right source code to that stream (see cov-start-da-broker --only-source-commit). Any defects committed between committing the wrong source code and committing the right source code might not properly associate with their source code.

## 4.13. How can I run the Dynamic Analysis Agent on tests that were started with a Maven build?

Use the following steps as a guide. Both steps vary by environment.

1. Make sure that Maven forks a new JVM to run your tests.

2. Modify the arguments that start this JVM.

   In place of `<jvmargs>` in the following examples, you need to specify `-javaagent:<install_dir_cic>/dynamic-analysis/dynamic-analysis.jar` where `<install_dir_cic>` is your Coverity Analysis installation directory.

   - If you are using the Maven Surefire Plugin, try the following command-line argument to launch your tests with different VM arguments:

     ```
     mvn -DargLine="<jvmargs>" test
     ```

   - If you are using the Maven 1.x default style of running JUnit tests, try the following command-line argument:

     ```
     maven -Dmaven.junit.fork=true -Dmaven.junit.jvmargs="<jvmargs>
     ```

For additional guidance, see the Maven documentation and your testing plugin.

**Note**

Do not rely on unit tests for comprehensive Dynamic Analysis results. For more information, see Section 1.2, “Obtaining thorough analysis results”.

# Appendix A. Dynamic Analysis Agent command-line options and Ant task attributes

## Table of Contents

This appendix describes the Dynamic Analysis Agent command-line options and attributes for the `cov-dynamic-analyze-java` and `cov-dynamic-analyze-junit` Apache Ant tasks. It also provides three Dynamic Analysis Agent start examples.

## A.1. Sample Dynamic Analysis start commands

Three sample Dynamic Analysis start commands are shown below. Note that you cannot have spaces between the options and comma.

**Example 1:** This example starts the Java program and Dynamic Analysis Agent. It sets the options `broker-host`, `broker-port`, `failfast` (exit the program if Dynamic Analysis can't connect to the Broker), and `repeat-connect` (try to reconnect to the Broker 60 times before giving up).

```
> java -javaagent:<install_dir>/dynamic-analysis/dynamic-analysis.jar=
    broker-host=sf.host.com,broker-port=4423,failfast=False,repeat-connect=60
    -cp classes/ simple.Example
```

**Example 2:** This example starts the Java program and Dynamic Analysis Agent and disables the `RACE_CONDITION` detector.

```
> java -javaagent:<install_dir>/dynamic-analysis/dynamic-analysis.jar
    =broker-host=sf.host.com,broker-port=4423,detect-races=False -cp classes/
    simple.Example
```

**Example 3:** This example uses the `exclude-instrumentation` option to tell Dynamic Analysis not to watch any of the classes in the example program. This reports no defects. Instead the console displays the message *Excluding instrumentation for classes whose fully qualified names start with `simple.`* The commands also set the `failfast` and `repeat-connect` options.

```
> java -javaagent:<install_dir>/dynamic-analysis/dynamic-analysis.jar=
    exclude-instrumentation=simple.,failfast=true,repeat-connect=60
```

## A.2. Dynamic Analysis Java Agent options and Ant attributes

Below are the Dynamic Analysis Java Agent options. Default values are in parenthesis. (For the Ant attributes for `cov-dynamic-analyze-java` and `cov-dynamic-analyze-junit`, see `cov-dynamic-analyze-java` reference page in the *Coverity 2020.12 Command Reference* for details and options.)

`detect-deadlocks=<boolean>`
    Detect deadlocks. (True)

`detect-races=<boolean>`
    Detect race conditions. (True.)

`detect-resource-leaks=<boolean>`
Detect resource leaks. (True.)

`use-resource-models=<File>`
(No Ant attribute.) Provide the RESOURCE_LEAK detector a file containing a list of additional resource management methods. Specifying `OPEN` and `CLOSE` methods for a class tells the RESOURCE_LEAK detector to report a RESOURCE_LEAK when an instance of that class has an `OPEN` method called without a subsequent `CLOSE` method being called.

The format of this file is as follows:

```
<type>,<resource class name>,<resource method name>, <method
signature>,<num params>,<index of resource>
```

Where:

- `<type>` is one of:
  - `OPEN` if this method creates the resource.
  - `CLOSE` if this method releases the resource.
  - `OPEN_WRAPPER` if this method wraps another resource.
- `<resource class name>`: The class name for the resource. This name adheres to the syntax used by the JVM specification.
- `<resource method name>`: The method name. This name adheres to the syntax used by the JVM specification.
- `<method signature>`: The signature for the method. The signature adheres to the syntax used by the JVM specification.
- `<num params>`: The number of parameters the method takes.
- `<index of resource>`: The index of the wrapped resource in the parameter list. This should be greater than `0` and less than `<num params>`. It is only valid for `OPEN_WRAPPER`, and should be `-1` for all other cases.

Here are some simple examples:

```
# Specify the create/release methods for UserResource
OPEN, com/coverity/tests/resourceLeak/UserResource, <init>,()V,0,-1
CLOSE, com/coverity/tests/resourceLeak/UserResource, dispose,()V,0,-1

# Specify a wrapper for UserResource
CLOSE, com/coverity/tests/resourceLeak/UserWrappingResource, dispose,()V,0,-1
OPEN, com/coverity/tests/resourceLeak/UserWrappingResource,
  <init>,(Ljava/lang/String;)V,1,-1
OPEN_WRAPPER, com/coverity/tests/resourceLeak/UserWrappingResource,
  <init>,(Lcom/coverity/tests/resourceLeak/UserResource;)V,1,0
```

The `jdkResourceList.txt` file in `<install_dir>/dynamic-analysis/dynamic-analysis.jar` contains many more examples.

`broker-host=<host_or_IP>`
Specify the hostname or IP address on which you ran the Broker. (localhost)

`broker-port=<port_number>`
Specify the port on which the Broker is listening. This is set with the `broker-port` option to `cov-start-da-broker`. If you intend to run more than one Broker instance simultaneously on the same machine, it is a good practice to use non-default ports to avoid collisions. (4422)

`collections-file=<filename>`
(No Ant attribute.) Load a list of collection operations that the RACE_CONDITION detector uses to detect races in collections. This option implies setting `instrument-collections` to `true`. (None.)

`exclude-instrumentation=<colon_separated_list_of_prefixes>`
Exclude classes from being watched by Dynamic Analysis to speed up Dynamic Analysis. However, Dynamic Analysis does not detect defects in excluded code nor as a result of actions performed in excluded code (such as field access or lock acquisitions).

This option consists of a colon-separated list of prefixes of the fully qualified names to exclude. For example:

"`exclude-instrumentation=A.B`" excludes any class whose name starts with "`A.B`", such as "`A.B`", "`A.B.c`", or "`A.Bc`".

"`exclude-instrumentation=A.B.`" (with a period) excludes "`A.B.c`", but not "`A.B`" nor "`A.Bc`".

(The default is to exclude nothing.)

`failfast=<boolean>`
If True, the Dynamic Analysis Agent exits the program it is watching if the Dynamic Analysis Agent loses its connection to the Broker or has other problems. If False, the Dynamic Analysis Agent quietly allows the program to continue running, even if Dynamic Analysis Agent cannot run properly. (False.)

`instrument-only=<colon_separated_list_of_prefixes>`
Specify a colon-separated list of classes watched by Dynamic Analysis. Dynamic Analysis excludes all other classes (same as if they were all specified as options to `exclude-instrumentation`). As with `exclude-instrumentation` above, using this option might speed up Dynamic Analysis, but at the cost of missing defects. This option consists of a colon-separated list of prefixes of fully qualified names to include. For example:

"`instrument-only=A.B`" includes any class whose name starts with "`A.B`", such as "`A.B`", "`A.B.c`", or "`A.Bc`".

"`instrument-only=A.B.`" includes "`A.B.c`", but not "`A.B`" nor "`A.Bc`".

(The default is to include everything.)

`instrument-arrays=<boolean>`
Watch reads and writes into arrays to report race conditions. (False.)

`instrument-collections=<boolean>`
: Detect race conditions on collections. For example suppose `map` is a `java.util.Map` and one thread executes `map.put("key", "value")` without holding any locks. If Dynamic Analysis sees another thread access this map, it reports a `RACE_CONDITION`. (True.)

`override-security-manager=<boolean>`
: Install a permissive security manager (`java.lang.SecurityManager`) that allows all operations except the installation of other security managers. This option exists because a restrictive security manager causes Dynamic Analysis to fail. Setting this option to `true` might allow Dynamic Analysis of your program to proceed. If this option is `false`, running Dynamic Analysis on your application might require adjusting your security policy or excluding classes that run within the restrictive security manager (see the `exclude-instrumentation` and `instrument-only` options). (False.)

`repeat-connect=<non-negative_number>`
: If this option is set to a number greater than zero, and the initial attempt to connect to the Broker fails, then the Agent tries to reconnect to the Broker that number of times, with a one second pause between attempts, before giving up. For best results, set this option to something greater than zero when starting both the Broker and Agents from a script or build file. (0 when running the Agent from the command line and 20 when running it through the `cov-dynamic-analyze-java` or `cov-dynamic-analyze-junit` Ant tasks.)

# Appendix B. Dynamic Analysis Broker information and reference

## Table of Contents

This section describes some useful Dynamic Analysis Broker technology. For a list and description of the Broker Ant attributes and command line options for `cov-start-da-broker` and `cov-stop-da-broker`, see the following reference pages in the *Coverity 2020.12 Command Reference*.

- `cov-start-da-broker` Ant task reference page
- `cov-stop-da-broker` Ant task reference page
- `cov-start-da-broker` command- line reference page
- `cov-stop-da-broker` command-line reference page

## B.1. Configuring the Broker using environment variables and the configuration file

This appendix describes how you can set certain `cov-start-da-broker` options with environment variables and in a configuration file as well as on the command line. This can help integrate Dynamic Analysis into your development and testing environment as well as saving you from extra typing.

For a complete description of these variables and options, see the `cov-da-start-broker` Ant task reference page.

### B.1.1. `cov-start-da-broker` environment variables

The following `cov-start-da-broker` options can be set as environment variables.

- `COVERITY_HOST`. Specifies the Coverity Connect server. Corresponding command line option: `--host`.
- `COVERITY_DATAPORT`. Specifies the port on the Coverity Connect server to which commits go. Corresponding command line option: `--dataport`.
- `COVERITY_USER`. Specifies a user name for logging into the Coverity Connect server. Corresponding command line option: `--user`.
- `COVERITY_PASSWORD`. Specifies the password for logging into the Coverity Connect server. Corresponding command line option: `--password`.

If we set the environment variables as follows:

```
COVERITY_HOST=cim.example.com
COVERITY_DATAPORT=9090
```

```
COVERITY_USER=jdoe
COVERITY_PASSWORD=secret
```

Then our sample command line from Section 2.3, “Step 3: Start the Dynamic Analysis Broker” becomes:

```
> <install_dir_cic>/dynamic-analysis/bin/cov-start-da-broker --stream Example-dynamic
 --dir intermediate_dir_name
```

### B.1.2. `cov-start-da-broker` configuration file

The `-cf <configuration_file>` `cov-start-da-broker` option allows you to specify a Java properties file that contains Broker options. The format for specifying properties is as follows:

```
<config_property>=<value>
```

`<config_property>` is usually the same name as the command line argument.

For example, suppose we have the following stored in a file called `Example.cda-broker-config`:

```
host=cim.example.com
dataport=9090
user=jdoe
password=secret
stream=Example-dynamic
```

The command line for running the example becomes:

```
> cov-start-da-broker -cf example.broker-config
```

The following Dynamic Analysis properties can be set in a configuration file:

```
broker-port=<port>
config-file=<file>
dataport=<port_number>
host=<hostname>
output-dir=<directory>
password=<password>
rundir=<run_directory>
run-prefix=<prefix>
security-file=<license_path>
shutdown-after<seconds>
stream=<dynamic_stream_name>
user=<user_name>
```

### B.1.3. Configuration file guidelines

Use the following guidelines when creating a configuration file:

- The character encoding for Java properties files is ISO 8859-1. Characters that cannot be directly represented in this encoding can be written using Unicode escapes.

- The colon (`:`) is a special character for Java properties files. Specifying paths may involve adding escape colons.

The following web sites document the format of the Java properties files more thoroughly.

http://java.sun.com/javase/6/docs/api/java/util/Properties.html

http://en.wikipedia.org/wiki/.properties

### B.1.4. Configuration option processing order

The Dynamic Analysis `cov-start-da-broker` processes configuration options in the following order, with later options overriding earlier ones:

1. Sets the `user` to the operating system user that runs the Dynamic Analysis Broker process.
2. Environment variables.
3. Command line arguments in order. Later command line options can override earlier ones.
4. Any configuration files specified with `-cf` are processed in their turn; later command line options can override what is set. Because of the nature of Java properties files, duplicate entries (including the `config-file` property) in a configuration file are resolved arbitrarily. Don't set a property (including `config-file`) more than once inside a configuration file.
5. The `--only-test-connection` option overrides any conflicting options.
6. `--help` and `--version` override everything.

## B.2. Running the Broker from a script

Running Dynamic Analysis from a script involves starting the Broker in the background, launching Agents, and shutting down the Broker. Several features of the Broker support running it from a script:

- The `--rundir` option allows you to specify a run directory where the Broker writes its logs and several other information files (mentioned below). Remove or move any old run directories in any script that starts the Broker. User errors, like a faulty command line, cause the Broker to leave the old run directory in place, which can potentially confuse your script.
- When the Broker opens its socket to listen for Agents, it writes a file called `broker.started` to its run directory.
- When the Broker finishes successfully (including explicit shutdowns with `cov-stop-da-broker` and the `cov-start-da-broker shutdown-after` option), it writes a file called `broker.ok` to its run directory.
- When the Broker fails due to some kind of error, and if it has gotten far enough along that it has created a run directory, it writes a file called `broker.fail` to its run directory. If the Broker fails due to an incorrect command line or other configuration error, it does not create a new run directory and does not touch an existing run directory.