SWAMP Output Files and Debugging Failures

James A. Kupsch

Version 1.2.1, 2020-05-20 09:30:24 -0500

Table of Contents

1. Introduction

This document describes the output of using the SWAMP frameworks to build a software package, assess it with a static analysis tool, and convert the raw results into parsed results in either SCARF or SARIF format. It also describes determining the success of this process and how to proceed to get the results or what files to inspect to aid with diagnosing the cause of failure.

The SWAMP uses Virtual Machines or Docker to perform a run: build, assessment and parsing of results. It is possible that only a subset of the steps occur. When the VM or Docker container exits, a set of files are left in the output directory. The contents of these files describe what occurred, or the output of commands and tools used to perform each step, or contain the artifacts of each step.

First this document gives an overview of the files typically produced. Then it describes how to inspect the files to determine a successful outcome, and how to proceed based on success or failure. Finally the output files are described and guidance is given on how to interpret them.

2. Overview of Files and How to Interpreting Results

After the SWAMP has run a VM or Docker container, a set of files is produced in the output directory. The common set of files produced is shown in Table 1. Not every run may produce every file, the files produced depend on the type of run performed and if it was successful.

Table 1. Common output files produced during an assessment. From indicates the configuration file containing the actual filename (b: build.conf, o: output_files.conf, p: parsed_results.conf, r: results.conf, -: hard coded).

Common Filename	From	Description
build.conf	o	SWAMP conf file describing build files
build.tar.gz	b	archive containing build output and artifacts
build_assess.out	o	output of assessment framework
capture.tar.gz	o	files captured at end of a run (for debugging)
env.sh	o	environment variable values (for debugging)
output_files.conf	-	paths of other output files
parsed_results.conf	o	SWAMP conf file describing parsed results files
parsed_results.tar.gz	p	archive containing result parser output, SCARF and SARIF results
results.conf	o	SWAMP conf file describing assessment result files
results.tar.gz	r	archive containing assessment output and raw results
run.out	o	output of framework's low-level provisioning and control commands
status.out	o	list of tasks performed with timing and status information

The file *output_files.conf* is the only file that has a fixed hard-code name, the other filenames and paths are determined by examining the contents of other files. **NOTE:** The rest of this document uses the common names of files and paths, but the actual names must be determined using the process described in Section 3, Section 5, Section 6, and Section 7.

The first step to working with the SWAMP output files is to determine the actual names of the top level files. Section 3 describes how to determine these using *output_files.conf*.

Once the filenames are determined, the next step is to inspect *status.out* to determine if the run was successful or not as explained in Section 4. If the run is successful, obtain the parsed result files in SCARF or SARIF format, as described in Section 7. If the run is unsuccessful, Section 4 explains how to diagnose the failure including other files to inspect.

3. output_files.conf

The file *output_files.conf* in the output directory is a SWAMP Conf formatted file (see Section 10) that contains key, value pairs with the paths to other files. These paths are relative to the directory containing this file. Table 2 lists the keys and their common values.

Table 2. output_files.conf attribute names and common values. Values are paths relative to the location of this file. If the attribute name is not present then unless otherwise noted, the content was not produced.

Attribute Name	Common Value	Attribute Missing Note
buildAssessOut	build_assess.out	content is in-line in run.out
buildConf	build.conf	use the Common Value
captureArchive	capture.tar.gz	
envSh	env.sh	
parsedResultsConf	parsed_results.conf	use the Common Value
resultsConf	results.conf	use the Common Value
runOut	run.out	
statusOut	status.out	

If *output_files.conf* exists, and an attribute does not exist in it, then that file was not produced unless otherwise noted. If *output_files.conf* does not exists, then proceed as if a synthesized *output_files.conf* was created in the output directory. The synthesized file contains an entry with each Attribute Name and Common Value pair shown in Table 2 if and only if a filename with the Common Value exists in the output directory, otherwise the entry is ommitted.

4. status.out

This section describes the *status.out* file produced when running a build, assessment, and/or parsing of results, and using it to debug failures. The *status.out* file is a summary of the tasks performed. The tasks appear in chronological order of completion. From the *status.out* file, you can quickly determine if the assessment succeeded, how long it took to run, where the time was spent, and in the case of failure, what task in the process caused the failure. Listing 1 shows a *status.out*

file.

Listing 1. Example status.out file of an assessment of a program written in C.

```
NOTE: begin
PASS: install-os-dependencies                                  18.683063s
PASS: install-strace (/opt/strace-4.10/bin/strace)               0.090283s
PASS: tool-unarchive                                            0.300803s
PASS: package-unarchive                                         0.243168s
PASS: configure                                                14.911602s
PASS: build                                                    64.587349s
PASS: build-trace-decode                                       21.838286s
PASS: assess (pass: 104, fail: 0)                              83.294354s
PASS: buildbug                                                171.294004s
PASS: build-archive                                             1.455068s
PASS: results-archive                                           0.029203s
PASS: resultparser-unarchive                                    0.016161s
PASS: parse-results (weaknesses: 214)                           4.561078s
PASS: parsed_results-archive                                    0.006893s
PASS: all                                                     211.602487s
NOTE: end
```

Three main types of tasks occur in *status.out* files produced in the SWAMP:

1. **informational** — provide information such as the *begin* and *end* tasks, and do not correspond to an activity (all of these have a status of NOTE),
2. **activity** — correspond to tasks or events that occurred as part of the build, assessment or result parsing, such as the *build* or *assess* tasks, and
3. **aggregate** — tasks that aggregate a sequence of preceding subtasks such as the *all* or *buildbug* tasks; the status (PASS or FAIL) depends on the status of the subtasks, and the duration is the total time from the first subtask starting to the end of the last subtask.

The rest of this document describes the status.out file format, the definitions of a valid status.out and successful run, the current tasks, and debugging task failures.

4.1. File Format

A *status.out* file consists of a chronologically ordered sequence of task records. Each task record conveys the status of a single task, and consists of two to six components. The task name and its status are required components. The duration and duration unit of the task, a short single line message, and a long multi-line message are optional. The remainder of this section shows the physical format, and then explains each component.

Listing 2 is a diagrammed status.out task without a long message.

Listing 2. Single status.out task with components labeled and no long message component.

```
PASS: task-name (short-message)                                    89.123456s
|     |          |                                                 |        |
|     task       shortMsg                                          dur      |
status                                                              durUnit
```

Each task may optionally have a multi-line message. Listing 3 adds a long message to the task shown in Listing 2.

Listing 3. Single status.out task with a two line long message.

```
PASS: task-name (short-message)                                    89.123456s
  ----------
  long message line 1
  long message line 2
  ----------
```

The individual components of a task record are described next.

4.1.1. Task

The task is a string that is a sequence of letters, numbers, the hyphen, and underscore characters. The task is separated from the status by a colon. Each task record requires a task, and each task in the file should be unique.

The currently defined tasks along with their description are found in Section 4.3.

4.1.2. Status

The status of a task is a string that is a sequence of letters, numbers, hyphen, and underscore characters. Each task record requires a status.

The currently defined status values are:

PASS indicates a successful task.

FAIL indicates an unsuccessful task.

SKIP indicates a task that was not run.

NOTE indicates a task that is informational. Two tasks with this status are the *begin* and *end* tasks that by convention are the first and last tasks.

A successful run is one where none of the tasks have a status of FAIL.

Current practice is for status values to be four uppercase characters.

4.1.3. Duration

The duration of a task is how long it took to complete. The duration is optional, and is not included for tasks that indicate a point in time. It consists of a decimal number (consisting of numbers and single optional period (decimal point) to separate the fraction part of the value), and a unit (consisting of alphabetic characters). Current units are listed in Table 3.

Table 3. Duration Units

Unit	Definition
s	seconds
none	seconds
ms	milliseconds
ns	microseconds

Current practice is to use a unit of *s* with a resolution of microseconds, have at least one digit to the left of the decimal point, all six digits after the decimal point, and the last character of unit appearing in column 80 if possible.

4.1.4. Short Message

The short message is used to provide additional information about the task, and is optional. If present, it is enclosed in parenthesis and appears immediately after the task. It consists of sequence of characters that does not include the new line character or a left parenthesis.

The main function of the short message is to provide counts for a few tasks, or to convey the reason a task failed or was skipped.

4.1.5. Long Message

The long message is used to provide additional multi-line information about the task, and is optional. The main function of the long message is to provide additional information about failed tasks.

The long message may contain multiple lines. A long message for a task starts is indicated by the line following the task being a delimiter consisting of a sequence of one or more spaces and then a sequence of one or more hyphen characters. The long message is placed in subsequent lines each prefixed with the number of spaces in the delimiter. Finally a new line is added and the delimiter is repeated to indicate the end of the long message. Processors must remove the prepended spaces of each line and final new line character to form the original data.

Current practice is to use two spaces and ten hyphens for the delimiter.

4.2. Valid *status.out* and Successful Run Definitions

A **valid *status.out*** file is syntactically correct, and in addition meets the following requirements:

1. *status.out* file exists in the output directory

2. no duplicate tasks
3. *begin* is the first task and has a status of NOTE
4. *all* task exists
5. *end* is the last task and has a status of NOTE

All *status.out* files produced during SWAMP build, assessment and/or parsing should be valid. If not, the assessment or host it is running failed in an unexpected way.

A **successful run** produces a valid status.out, and also meets the additional requirements:

6. *all* task has a status of PASS
7. no task has a status of FAIL

4.3. Task Definitions

This section documents the currently defined tasks. Each task includes a description and descriptions of the short message values defined for the task.

all

is the aggregate task that covers all other subtasks. Its status indicates the overall status of the entire run and its duration is the duration of the entire run.

android-update

is the task of updating an Android package to work with the Android SDK installed on the machine or VM.

assess

is the task of running the assessment tools. Additional information is provide in the short message indicating the number of invocations of the assessment tool and their success, failure or number of assessments skipped (due to missing files or other valid reasons not to run the assessment tool on a source file). Each assessment processes one or files. An example short message for *assess* is

```
    (pass: 93, fail: 0, skip: 3)
```

The status is FAIL if any of the assessments failed, SKIP if no files were found to assess (see Section 4.4.3.1), or PASS otherwise.

assess-acquire-license

is the task of acquiring the license to operate a tool that uses a license manager. If the status is FAIL, acquiring the license failed and the long message may contain more details about the reason for failure.

begin

is an informational task. This record is written upon start up.

build

is the task of building the software package with build monitoring. The build duration includes the added time to monitor the build process.

In the case of failure the long message has information on the non-zero exit status or signal that caused the failure. Additional information may be available in the long message. The *build_stdout.out* and *build_stderr.out* files found in the build archive for more information about the cause of the failure (see Section 5).

build-archive

is the task of archiving the build directory.

build-trace-decode

is the task of decoding the build monitoring for C and C++ builds.

build-unarchive

is the task of unarchiving the build directory.

buildbug

is the aggregate task covering all the build and assess tasks including *build*, *build-trace-decode*, and *assess*. If any of those fail, *buildbug* status is also failed.

chdir-build-dir

is the task of changing to the build directory after changing to the package directory. Only present on failure with the long message containing the name of the directory and the reason for the failure.

chdir-config-dir

is the task of changing to the config directory after changing to the package directory. Only present on failure with the long message containing the name of the directory and the reason for the failure.

chdir-package-dir

is the task of changing to the package directory. Only present on failure with the long message containing the name of the directory and the reason for the failure.

configure

is the task of configuring the software package.

In the case of failure the long message has information on the non-zero exit status or signal that caused the failure. Additional information may be available in the long message. The *config_stdout.out* and *config_stderr.out* files found in the build archive for more information about the cause of the failure (see Section 5).

end

is an informational task. This record is written as the framework is ending. If it is not present, the framework or VM likely crashed.

fetch-pkg-dependencies

is the task to fetch the dependencies using a non operating system package manager. This task may be a task that is always included in the task list such as with pip packages, or done as part of the build task and is only included if the build fails due to the package manager's failure to fetch dependencies. The short message contains the package manager (currently `gradle`, `maven`, `ant+ivy` or `pip`). The long message contains details about the failure.

flow-typed

is the task of running the flow-typed tool that fetches flow description files of JavaScript modules to improve the operation of the `flow` tool. This task may have a status of SKIP if the tools is not run; the short message contains the reason the tool was not run:

(internet inaccessible)	assessment run is configured to indicate that the VM is in an environment where the internet is inaccessible (*run.conf* file contains `internet-inaccessible = true`)
(disabled)	configuration file setting has disabled the use of flow-typed
(not configured)	flow-typed configuration options not found (old tool version)
(package.json missing)	package.json file is missing and required to use flow-typed
(node_modules missing)	node_modules directory is missing and required to use flow-typed

gem-install

is the task of installing the user's Ruby gem file (user's package) along with the gem dependencies downloaded from the network into the local Ruby user install directory

gem-unpack

is the task that unpacks the user's Ruby gem file to make the source files accessible from the file system (similar to the package-unarchive but with the command: `gem unpack <gem-file-path> –target <package-dir>`).

generator

is an informational task. Is included if the *status.out* file was created outside the assessment framework code. The short message contains the program that produced the *status.out* file. Typically there are subsequent tasks with details on why the framework did not run.

install-os-dependencies

is the task of installing OS dependencies for the run using the OS's package manager such as `yum` or `apt-get`.

A status of PASS and FAIL are based on the success or failure of the OS's package management system. A status of SKIP can occur from two sources indicated by the task's short message:

none	no OS packages to install
(internet inaccessible)	assessment run is configured to indicate that the VM is in an environment where the internet is inaccessible (*run.conf* file contains `internet-inaccessible = true`)

install-pip-dependencies

is the task to install PIP dependencies for Python packages.

install-strace

is the task to install the strace command packaged with c-assess. If an strace was packaged c-assess the short message is the location of strace binary. If no strace was packaged, the status is SKIP, the short message gives the reason, and the system strace binary found in the PATH is used.

network

is the task that that indicates a properly functioning network in the VM. It is only present on failure. The short message indicates the type of failure (the only currently defined value is *no-ipv4-address*).

As this is a temporary failure, a *retry* task is also included indicating the VM should be run again.

no-build-setup

is the task to find and validate source files to assess for C, C++ or Java packages with a build system of *no-build*. The number of source files found with valid extensions and the number of files that actually compile are specified in the short message in the format shown in the following example:

```
(source-files: 1, compilable: 1)
```

The status is FAIL if no source files are found or none are compilable. The short message can be used to distinguish the specific type of failure. If compilable is less than source-files, a warning should be issued (see Section 4.4.3.2).

package-checksum

is the task of validating the package's checksum against the value found in package.conf.

package-unarchive

is the task of unarchiving the package.

parse-results

is the task of running the result parser on the assessment results to produce parsed results in SCARF (SWAMP Common Assessment Results Format). The short message contains the number of weaknesses found as in the following example:

```
(weaknesses: 214)
```

parsed-results-archive

is the task of archiving the parsed results directory.

python2-install

is the task of installing the Python 2.x language.

python3-install

is the task of installing the Python 3.x language.

read-gem-spec

is the task of reading a Ruby package's gem file to determine what files to assess

resultparser-unarchive

is the task of unarchiving the result parser.

results-archive

is the task of archiving the results directory.

results-unarchive

is the task of unarchiving the results.

retry

is an informational task that indicates the VM should be recreated and restarted as a temporary condition was detected that caused the run to fail. The first failed task is the reason for the retry.

setup

is the task that is the low-level provisioning code that runs before or after the assessment framework.

The long message contains more details about the failure, and the short message The short message contains the more specific type of error. The long message contains more details. Currently defined short messages and their meanings are as follows:

(command-failed)	A command exited with a non-0 exit code.
(command-not-executable)	Command to be executed is missing or not executable.
(conf-file)	Error reading, processing or interpreting *run-params.conf*.
(conf-file-missing)	*run-params.conf* configuration file is missing.
(get-ip-addr)	Error getting IP address used to route to outside hosts.
(no-home-dir)	User's home directory does not exist.
(invalid-arg)	Invalid function argument or wrong number.
(invalid-env-var)	Environment variable is invalid or missing.
(invalid-run-params-var)	Configuration file variable is invalid or missing.
(log-open)	Error opening the output log file (*run.out*).
(redirect)	Error opening a file for stdout or stderr redirection.
(signal)	A signal caused an early exit.

swamp-maven-plugin-install

is the task to setup the Java framework's plug-in to monitor Maven builds.

tool-configure

is the task of configuring the tool. The long message may contain more details about the configuration.

tool-install

is the task of installing the tool.

tool-package-compatibility

is the task to check compatibility of the tool for use with the software package being assessed. For instance there are tools that require Java 1.6 or above and fail if they are used on Java 1.4 or 1.5 packages.

The short message contains the type of error. The long message contains more details about the incompatibility. Currently defined short messages and their meanings are as follows:

(android+maven)	The android-lint tool is not configured properly for a Maven project.
(gcc version)	The version of `gcc` used during the build is incompatible with the tool.
(Java language version)	The Java source language used during the build is incompatible with the tool.
(known tool bug)	The tool failed assessing the source code due to a known bug with the tool.
(ruby version)	The version of `ruby` used during the build is incompatible with the tool.

tool-runtime-compatibility

is the task to check compatibility of the tool for use in the runtime environment.

The short message contains the type of error. The long message contains more information about the incompatibility. Currently defined short messages and their meanings are as follows:

(JVM version)	The version of the JVM is incompatible with the tool.

(internet inaccessible) This tools as configured required access to the internet and can not be run in an environment where *internet-inaccessible* is set. Currently the tools *OWASP Dependency Check*, *Retire.js*, and *Sonatype AHC* by default require access to the internet. Either repackage the tools to contain a static version of the data (see the documentation inside the tool archive), or allow access to the internet.

tool-unarchive

is the task of unarchiving the results.

validate-package

is the task to validate that the Java Bytecode files specified to assess are valid in the package.

4.4. Debugging Task Failures

If the run is successful (see Section 4.2), there is nothing to debug, but there may be useful information to report to the user as described in Section 4.4.3. The rest of this Section describes how to debug *status.out* files that are invalid, valid but unsuccessful, and finally issues that should be brought to the attention for runs that are successful.

4.4.1. Missing or Invalid status.out File

A status of FAIL for the *all* task and at least one other task indicates a normal failure of some aspect of the run. Debugging these types of errors are explained in Section 4.4.2.

Other abnormal types of failures are indicated by a missing or invalid *status.out* file. Explanations for these failures are

status.out missing Internal problem starting the VM, such as /mnt/in or /mnt/out not being mounted.

duplicate tasks The assessment framework was run multiple times, or a bug in the framework. Possibly the VM was started multiple times.

first task not *begin* The VM or framework crashed or exited before the framework was run or early in the framework code.

***all* task missing** VM exited or crashed before the framework completed. A previous failure in *status.out* may indicate the cause, and inspecting *build_assess.out* (see Section 9) and *run.out* (see Section 8) may contain additional information.

last task not *end* VM exited or crashed before the framework completed. A previous failure in *status.out* may indicate the cause, and inspecting *build_assess.out* (see Section 9) and *run.out* (see Section 8) may contain additional information. Data from the run could be OK if the *all* task has a status of PASS, but should be treated with suspicion.

4.4.2. Debugging Specific Task Failures

If the *status.out* file is valid, but the *all* task has a status of FAIL, then the run has failed. The first step to debug a failed, but valid status.out is to create a list of all the tasks that have a status of FAIL, and to determine if there is a *retry* task.

If there is a *retry* task, the VM should be recreated and rerun as this indicates a likely temporary failure. Retrying should be attempted a small fixed number of times, such as three times total, until *retry* is no longer a task or the count is reached. If the retries were not successful, there is a persistent problem with the infrastructure and the diagnosis should proceed with the next step.

The next step is to locate the first task with a status of FAIL in the list below and use the advice in its section. If this task is not in the list below report it to the developers, and continue with subsequent elements until a failed task is in the list (there should be at least one since the *all* task must be in the list). Subsequent failed tasks should be ignored as they are likely caused by the first failure.

In the list below, if the task is marked as an *internal error*, this is is a problem with frameworks, VM hypervisors, or the environment. An internal error is not generally something that a user can fix by making changes to their package and how it is configured. Internal errors should be reported to the operator SWAMP or developers of the SWAMP software.

Unless otherwise noted in the advice, the files to inspect for additional information about the failure are files listed in the advice, then *build_assess.out*, and finally *run.out*. Generally *run.out* is only relevant for *setup* task failures. These files should be made available for inspection by the user.

Below are the list of tasks and advice on how to proceed if they fail.

all

Internal error. As an aggregate task some other task should have failed earlier in the list.

android-update

This failure indicates that the user's Android package does not conform to the convention Google uses to configure Android packages. This can be caused by copying an Android configuration file from an older Android SDK and modifying it instead of configure the package correctly to Google's convention.

A possible solution is to modify the package's SWAMP build options to enable the *Android Redo Build* option. This erases the existing Android build file and the Android SDK recreates the file from the metadata found in the package. This may not work if the user made changes that package depends on to the configuration file they copied as it is replaced.

The other option a developer has is to update their package to follow Google's build conventions using the latest Android SDK.

assess

Internal error. This is a failure of the assessment tool to run successfully. This may be caused by the tool not properly interpreting the input files due to deficiencies in the tool, a bug in the tool, or a misconfiguration of the tool.

Additional information about this failure may be found in the assessment results files (see Section 6) or in *build_assess.out*.

assess-acquire-license

Internal error. This may be due to networking issues between the VM running the tool, or an issue with the license server. The long message may contain more details on the failure.

(concurrency limit)	If the number of VMs running concurrently trying to use the licensed tool exceeds the maximum number of concurrent licenses allowed, recreating and rerunning the VM may solve the problem. This should not occur if the concurrency limits are properly enforced.
(network connection)	The host must be able to connect with the license server on the port it is listening for requests. It may not be able to due to the VM or network firewall blocking the connection, the license server host (or VM) is not up, or the license server is not running on the license server host.
(license expired)	Licenses have limited lifetimes and need to renewed periodically.

begin

Internal error. This task is informational and should always have a status of NOTE.

build

The build for the user's package failed. This can be caused by the user selecting the wrong build system, missing libraries, the package not being compatible with the environment of the selected platform (operating system), or the package containing code that is not syntactically valid.

Some packages download resources from the internet as part of the *build* task such as the use of a build system with an integrated package manager, or the tools like `wget` or `curl` during the build. For these packages, a *build* task status of FAIL can be caused by the remote resource being moved or removed, an incorrect specification for the remote resource, an off-line or malfunctioning remote server, the assessment host is in an isolated environment that prevents remote connections, or a network issue such as an incorrectly configured firewall or router. The user can remove the need for the remote resource, or correct the problem and try again at a later time. If the failure is due to the network the SWAMP administrator needs to correct the problem, or the user can try

again at a later time as the problem could be transient.

The long message of the task contains the exact command that failed to run successfully, and how it failed. Debugging this failure is accomplished by inspecting the output of the build. The output and error output for the build step is found in the *build_stdout.out* and *build_stderr.out* files found in the build archive (See Section 5).

The long message and the two output files should be provided for users to inspect if the *build* task fails, so they can determine what failed, and how to update the package archive or build parameters.

Users should be directed to try building the package on their own computer.

build-archive

Internal error. Error creating the build archive. A possible cause is that /mnt/out is full. The *build_assess.out* (see Section 9) may contain additional information.

build-trace-decode

Internal error. This indicates that the format of strace data produced by the build monitoring of C and C++ packages was not formatted as expected.

build-unarchive

Internal error. The build archive is corrupt or missing, or the file system is full.

buildbug

Internal error. As an aggregate task some other task should have failed earlier in the list.

chdir-build-dir

The *build directory* (relative to the *package directory*) specified by the user is invalid for the user's package; the directory may not exist or may have invalid permissions. The long message contains more details about the directory and type of problem. The user should be directed to update the build directory setting or to add the directory to package archive. This directory must exist in the user's package archive, if there is not a *configure command*. If there is a configure command, it may create the directory.

Users should be directed to build the package on their own computer using the paths specified.

chdir-config-dir

The *config directory* (relative to the *package directory*) specified by the user is invalid for the user's package; the directory may not exist or may have invalid permissions. The long message contains more details about the directory and type of problem. The user should be directed to update the *configure directory* setting or to add the directory to the package archive. This directory must exist in the user's package archive if there is a *configure command*.

Users should be directed to build the package on their own computer using the paths specified.

chdir-package-dir

The *package directory* specified by the user is invalid for the user's package; the directory may not exist or may have invalid permissions. The long message contains more details about the directory and type of problem. The user should be directed to update the package directory setting or to add the missing directory to the package archive. This directory must exist in the user's package archive.

Users should be directed to build the package on their own computer using the path specified.

configure

The configure task for the user's package failed (non-0 exit status). This can be caused by the user selecting the wrong build system, missing libraries, the package not being compatible with the environment of the selected platform (operating system), or configure command not existing.

Some packages download resources from the internet as part of the *configure* task such as the use of tools like `wget` or `curl` during configure. For these packages, a *configure* task status of FAIL can be caused by the remote resource being moved or removed, an incorrect specification for the resource, an off-line or malfunctioning remote server, the assessment host is in an isolated environment that prevents remote connections, or a network issue such as an incorrectly configured firewall or router. The user can remove the need for the remote resource, or correct the problem and try again at a later time. If the failure is due to the network the SWAMP administrator needs to correct the problem, or the user can try again at a later time as the problem could be transient.

The long message of the task contains the exact command that failed to run successfully, and how it failed. Debugging this task can be accomplished by inspecting the output of the configure task. The output and error output of the configure task is found in the *config_stdout.out* and *config_stderr.out* files found in the build archive (see Section 5).

The long message, and the two files should be provided for users to inspect if the *configure* task fails, so they can determine what failed, and how to update the package archive or build parameters.

Users should be directed to try building the package on their own computer.

end

Internal error. This task is informational and should always have a status of NOTE.

fetch-pkg-dependencies

Fetching the dependencies of the package using the build system's mechanism to fetch dependencies from the internet failed. This could be caused by the dependencies in the package's build settings file no longer existing (or being incorrect), a networking issue such as an incorrectly configured firewall, or the remote package manager server not being up or functioning incorrectly. The short message contains the name of the build system being used, and current is either `gradle`, `maven`, `ant+ivy` or `pip`. The long message contains more information about the dependencies that failed to download. Users should be shown the long message to communicate what failed.

If it is due to the dependencies in the package's settings no longer existing or being incorrect, the user needs to fix the package's dependencies, and upload an updated package archive. It is not atypical for old versions of dependencies to be removed from repositories.

If the failure is due to the network the SWAMP administrator needs to correct the problem, or the user can try again at a later time.

Occasionally a repository does not function correctly. The only recourse is to try again at a later time.

flow-typed

Internal error. The *flow-typed* command run as part of the flow tool failed. The stdout and stderr from flow-typed may have more details and can be found in the *results* directory in the results archive: *flow_typed_stdout1.out* and *flow_typed_stderr1.out*.

gem-install

Installing the user's gem package file failed. The output and error output of the gem install task is in gem_install.out and gem_install.err files found in the build archive. These files should be inspected for more information.

gem-unpack

Unarchiving the user's gem package file failed. The output and error output of the gem unpack task is in gem_unpack.out and gem_unpack.err files found in the build archive. These files should be inspected for more information.

generator

Internal error. This task is informational and should always have a status of NOTE.

install-os-dependencies

This is a failure to install OS dependencies using the OS package manager. This could be caused by the package name specified by the user being incorrect for the selected platform, a networking issue such as an incorrectly configured firewall, or the remote OS package manager server not being up or functioning incorrectly.

Users should be allowed to inspect *build_assess.out* (see Section 9) to determine the cause of the failure. If it is due to an incorrect OS package name, the user can correct this. If the failure is due to remote OS package manager server not functioning correctly, the run may succeed at a later time. If the failure is due to a networking issue the administrator of the SWAMP needs to correct the issue.

install-pip-dependencies

Installation of the PIP dependencies specified by the user failed. The output and error output of the PIP install task is in pip_install.out and pip_install.err files found in the build archive. These files should be inspected for more information.

install-strace

Internal error. The strace archive in c-assess is corrupt or missing, or the file system is full.

network

The VM's network is not functioning correctly. This is usually caused by an temporary issue on the hypervisor, and will likely work if tried again (a *retry* task is included to indicate this). If this issue persists, other parts of the infrastructure are having issues that the system administrator needs to correct.

no-build-setup

A build system type of *no build* was selected for C, C++ or Java source code. This fails for three reasons. The first is that no source files of the selected type (C/C++ or Java) were found in the *build directory* selected by the user. The second is that none of the source files found compiled without errors. The third is an internal error.

The short message of the task contains data to determine the type of error. If missing, an internal error occurred. If no files of the appropriate type were discovered in the package directory, the compilable and source-files are both 0. If compilable is 0, source files were found, but none compiled without error.

The *source-compiles.xml* file in the build archive (see Section 5) contains the complete list of source files discovered, compilation success, how compiled, and the compiler output. The contents of this file (see Section 5.3) should be parsed and presented to the user to show the user the actual files discovered and if they were compilable.

package-checksum

The checksum of the package archive in the package.conf does not match the checksum of the package archive file. The package archive or package.conf is corrupt, or the checksum is invalid.

package-unarchive

The user's package archive is corrupt, the archive is missing, the archive is of an unknown format, or the file system is full. The long message of the task contains more information about command that was run, and why the failure occurred.

The user should be shown the long message of task, and *build_assess.out* (see Section 9) to determine the problem with the archive. The user should then replace the archive with a valid version.

Users should be directed to try unarchiving the package archive on their own computer.

parse-results

Internal error. Error creating the parsed results archive. The result parser failed to convert output to SCARF. This can be caused by an unexpected format of result file, the result file being corrupted, or a bug in the result parser. The stdout and stderr of the result parser (see Section 7) may help diagnose the problem.

parsed-results-archive

Internal error. A possible cause is that /mnt/out is full.

python2-install

Internal error. This task is indicates that the Python 2.x installation failed. Either the framework is configured incorrectly, is corrupt, or the disk is full.

python3-install

Internal error. This task is indicates that the Python 3.x installation failed. Either the framework is configured incorrectly, is corrupt, or the disk is full.

read-gem-spec

Failed to read the gem specification file from the user's gem package file. The output and error output of this task is in *gem-name*-err.spec file files found in the build archive. This file should be inspected for more information.

resultparser-unarchive

Internal error. The result parser archive is corrupt or missing, or the file system is full.

results-archive

Internal error. Error creating the results archive. A possible cause is that /mnt/out is full.

results-unarchive

Internal error. The results archive is corrupt or missing, or the file system is full.

retry

Internal error. This task is informational and should always have a status of NOTE. If the status is NOTE, the VM should be recreated and rerun.

setup

Internal error. A failure occurred in the low-level provisioning code that runs before or after the assessment framework. The short and long messages provide more details on the error.

swamp-maven-plugin-install

Installation of SWAMP maven plugin failed. The main reason this task fails is due to networking issues or Maven Central unavailability. The long message of the task contains more information about the failure. Rerunning this in the future may succeed if the issue is a temporary problem with the network or Maven Central.

tool-configure

Internal error. The tool configuration failed.

tool-install

Internal error. This task is indicates that the tool installation failed. Either the tool is configured incorrectly, the tool is corrupt, or the disk is full.

tool-package-compatibility

The tool is incompatible with the version of the language of package's source code. The short message is the type of error, and the long message contains additional details about the incompatibility. If possible, the user interface should not allow such an assessment to run, but in practice it may be difficult to determine the package's language version. The user can select a different tool or version of the tool that is compatible. The short and long messages should be displayed to the user. Currently defined short messages and how to resolve them are as follows:

(android+maven)	The android-lint tool is usually not configured properly for a Maven project. The user should update their project to use Maven or Gradle as a build system.
(gcc version)	The version of `gcc` used during the build is incompatible with the tool. This occurs when using the *Parasoft C/C++test* tool (versions 9.5.4, 9.5.6 and 9.6.1) and a platform with a version of `gcc` of 5.0 or newer. Platforms with a `gcc` version of 5.0 or newer include *Fedora 22* and newer, and *Ubuntu 16.04* and newer. The user can use another platform with an older version of `gcc` or try to install, and use an older version of `gcc`.
(Java language version)	The Java source language used during the build is incompatible with the tool *error-prone* version 2.0 and later does not support Java 1.5 sources. The user can try to modify the package to build using a newer version of Java, or they can use a different version of the tool (*error-prone* version 1.x supports Java 1.5 sources).
(known tool bug)	A known bug with the tool was triggered by the package. Until the tool is updated to fix the bug, users will not be able to get results using this tool with their package.
(ruby version)	The version of ruby used during the build is incompatible with the tool. The `reek` tool requires a minimum version of the ruby interpreter. The user can try an older version of the tool, or update the package to use a newer version of ruby.

tool-runtime-compatibility

The tool is incompatible with the platform's runtime environment such as the JVM or operating

system. The short message is the type of error, and the long message contains more information about the incompatibility. The user interface should not allow such an assessment to run. The user can select a different tool, version of tool, or platform that is compatible. The short and long messages should be displayed to the user.

tool-unarchive

Internal error. The tool archive is corrupt or missing, or the file system is full.

validate-package

The *package classpath* specified for the Java Bytecode package is not valid. Jar files or directories listed in the classpath are not present in the package. The long message contains more information about the failure. Correct the *package classpath* option, or use a new archive that contains the missing files or directories.

4.4.3. Potential Issues In Successful Runs

This section describes conditions where the run did not fail, but where user's should be alerted to potential issues with the package's configuration.

4.4.3.1. Status of *assess* task is SKIP

An *assess* task with a status of SKIP indicates that no files were found to assess.

This may be intentional as the user submitted the packages as a web package and ran all web tools, but the package did not contain all files types such as CSS. In this case, the use of tool such as csslint would have a status of SKIP for the *assess* task.

If the package is a C/C++, Java, Ruby, Android, or Python package, the user's package archive or package settings are almost certainly incorrect. Possible causes include:

- The user built the software on their computer, and then archived the already built package. When the software is built as part of an assessment, the build system detects that it is already built and does compile any of the source files. Since the SWAMP assesses only those files compiled during the build task, no files are assessed. The user should upload a freshly checked out version of their software, (or run their build system's *clean* target), archive the unbuilt package, and upload it again.
- The package contains no source files.
- The package's *build* or *configure* task settings are incorrect.

4.4.3.2. *no-build-setup* task exists and not all source files were compilable

For C/C++ and Java builds with a build system of *no-build*, the system first determines candidate source files in the build directory based on file extensions. Then it assesses only those files that build successfully using default compiler options. If at least one source file compiles then this is task is successful. If the number of compilable source files is less than the number of candidate source files, then there were files with an appropriate file extension that were not assessed; user's should be warned of this condition. See no-build-setup in Section 4.3 for details on how to

determine these counts. The details of which source files were discovered and which compiled successfully can be found in the file *source-compiles.xml* described in Section 5.3.

4.4.4. Known Issues with Tools

This section documents known issues with tools. The issues are mainly do to tools being incompatible with the package's source language or the runtime environment used to run the tool.

Android Lint

Most Android Java packages that use the Maven build system do not properly setup the environment for Android Lint to run correctly, causing the tool to fail. When an Android package is not configured correctly to use the android-lint tool, the *tool-package-compatibility* task FAILs with a short message of *android+maven*.

dawnscanner

There is a known bug in dawnscanner that causes it to fail on some source code. In this case, the *tool-package-compatibility* task FAILs with a short message of *known tool bug*.

error-prone

Version 2.0.0 and later of error-prone only requires the use of a Java source language 1.6 (Java 6) or later. When a package that uses an incompatible version of the Java language is used, the *tool-package-compatibility* task FAILs with a short message of *Java language version*.

Parasoft C/C++test

Versions 9.5.x and 9.6.x of Parasoft C/C++test do not recognize `gcc` compiler versions 5.0 and newer, and fails on platforms where the build uses such a version. Platforms with a `gcc` version of 5.0 or newer include *Fedora 22* and newer, and *Ubuntu 16.04* and newer. When a package that uses an incompatible version of the `gcc` is used, the *tool-package-compatibility* task FAILs with a short message of *gcc version*.

PHPMD

PHPMD has a bug that causes it to crash when analyzing certain types of PHP code. When this known failure occurs, the *tool-package-compatibility* task FAILs with a short message of *known tool bug*.

reek

The `reek` tool requires a minimum version of ruby. If the version of ruby used by the package is too old for `reek`, the *tool-package-compatibility* task FAILs with a short message of *ruby version*. There is also a known bug that causes `reek` to fail on some source code. In this case, the *tool-package-compatibility* task FAILs with a short message of *known tool bug*.

5. Build Related Files

The build related files are the files of the software package, the output of the configure and build tasks, and the build artifacts such as source code, libraries and executables created. If the configure or build tasks fail, these output files may be useful in diagnosing the cause of the failure.

The path of the file *build.conf* is determined by Section 3. All other names/paths are determined by

the contents of the *build.conf*. The rest of this section will refer to files using their Common Value shown in Table 4. All files besides *build.conf* and *build.tar.gz* are members with the given path of the *build.tar.gz* archive.

The rest of this section details the structure of the files *build.conf*, *build_summary.xml*, and *source-compiles.xml*. The *build* directory contains the source code of the package and all artifacts generated during the *configure* and *build* tasks. The other files are unstructured and contain the stdout and stderr of the *build* and *configure* tasks.

5.1. build.conf

The file *build.conf* in the output directory is a SWAMP Conf formatted file (see Section 10) that contains key, value pairs with the paths to other files. The key, value pairs are documented in Table 4. The *build-archive* attribute is the path (relative to the directory containing *build.conf*) to the archive containing the build related files, and the remaining attributes are paths within the archive.

Table 4. build.conf attribute names and common values. Unless otherwise noted, the value is the path relative to the build-dir path inside the build-archive archive.

Attribute Name	Common Value	Note
build-archive	build.tar.gz	path of build archive relative to this file
build-dir	build	path of directory within the archive containing build data
build-stderr-file	build_stderr.out	stderr of the build task (if run)
build-stdout-file	build_stdout.out	stdout of the build task (if run)
build-summary-file	build_summary.xml	build information file
config-stderr-file	config_stderr.out	stderr of the config task (if run)
config-stdout-file	config_stdout.out	stdout of the config task (if run)
source-compiles	source-compiles.xml	data about source files discovered for no-build C/C++ and Java builds

If an archive member exists in the *build.tar.gz* archive with a directory of *build* and a filename of the Common Value, then contents of *build.conf* should be treated as if the file contained the Attribute Name with the Common Value. This provides backwards compatibility for *build.conf* that did not include stdout and stderr values.

5.2. build_summary.xml

build_summary.xml is an XML file that contains information about the build. The format varies between frameworks and will be documented in the future.

5.3. source-compiles.xml

The file *source-compiles.xml* is an XML file that describes the files assessed when the *no-build* build type is used with C/C++ and Java. When *no-build* is selected the specified build directory is scanned for potential source files based on file extensions and language: C (.c), C++ (.C, .cc, .cp, .cpp, .CPP, .cxx, or .c++), and Java (.java). These potential source files are then compiled individually using default compiler flags based on the langage: C (`gcc -c source`), C++ (`g++ -c source`), and Java (`javac -g -implicit:class source`). If the potential source file successfully compiles it becomes a build source file and is assessed, otherwise it is not assessed. For each potential source file (based on file extension) found in directly in the build directory, the following information is represented: filename, file type, command used to compile, exit code, exit signal, and compiler output.

The structure and description of *source_compiles.xml* is show in Listing 4. As an XML file, the order of element is within another element is not significant and should not be relied on, except for the `<arg>` elements which are ordered. Also elements are allowed to be missing and indicate that there is no value.

Listing 4. Structure and description of source-compiles.xml XML file and its elements. ⋯ represents a text value.

```
XML Elements                                          Description

<source-compiles>
  <package-short-name>...</package-short-name>        package name
  <package-version>...</package-version>              package version
  <platform>...</platform>                            platform name
  <source-compiles-uuid>...</source-compiles-uuid>    UUID of this file
  <package-root-dir>...</package-root-dir>            package root directory
  <build-root-dir>...</build-root-dir>                build root directory
  <no-build-version>...</no-build-version>            no-build version
  <build-sys>...</build-sys>                          build-sys type
  <source-compile>                                    per source candidate info
    <source-file>...</source-file>                      filename
    <source-type>...</source-type>                      type C, C++, or Java
    <command>                                           compilation command info
      <executable>...</executable>                        name of executable
      <args>                                              arguments
        <arg>...</arg>                                      arguments of command
      </args>
      <environment>                                       environment variables
        <env>...</env>                                      format: name=value
      </environment>
      <cwd>...</cwd>                                      working directory
    </command>
    <execution-successful>...</execution-successful>  success: 'true' or 'false'
    <exit-code>...</exit-code>                        exit code of command
    <exit-signal>...</exit-signal>                    exit signal of command
    <output>...</output>                              stdout/err of command
  </source-compile>
</source-compiles>
```

This file will include one `<source-compile>` for each source file candidate. To determine if a source file was compilable first check if the element `<execution-successful>` exists, if it does then the contained text `true` indicates a compilable file and `false` indicates a non-compilable file. If the `<execution-successful>` element does not exist, then check if the compilation command exited normally (`<exit-code>` element exists) and without error (`<exit-code>` containing text is `0`). A non-compilable source file will either have exited by a signal (`<exit-signal>` element exists) or with a non-0 exit code (`<exit-code>` containing text is not `0`).

The text of the `<output>` element contains the combined compilation stdout and stderr. For non-compilable source, it will contain the error message as to why it failed. For compilable source it will generally by empty, but may contain compiler warnings.

The `<arg>` element in *no-build-version* of 2.0 or earlier does not include `argv[0]`; `argv[0]` should be assigned the value of the `<executable>` element's text.

6. Assessment Related Files

The assessment related files are the raw (native) tool result files from running the assessment tool on the software package, and the output from running the tools. If the assess task fails, these output files may be useful in diagnosing the cause of the failure.

The name/path of the file results.conf_ is determined by Section 3. All other names/paths are determined by the contents of the *results.conf*. The rest of this section will refer to files using their Common Value shown in Table 5. All files besides results.conf_ and *results.tar.gz* are members with the given path of the *results.tar.gz* archive.

The rest of this section details the structure of the files *results.conf*, and *assessment_summary.xml*. The other files are dependent on the tool run and contain the native tool results, the tool's stdout or stderr, or other tool generated files.

6.1. results.conf

The file *results.conf* in the output directory is a SWAMP Conf formatted file (see Section 10) containing key, value pairs with the paths to other files. The key, value pairs are documented in Table 5. The *results-archive* attribute is a path to the archive of the assessment related files, and the remaining attributes are paths within the archive.

Table 5. results.conf attribute names and common values. Unless otherwise noted, the value is the path relative to the results-dir path inside the results-archive archive.

Attribute Name	Common Value	Note
results-archive	results.tar.gz	path of results archive relative to this file
results-dir	results	path of directory within archive containing results data
assessment-summary-file	assessment_summary.xml	path of directory within archive relative to *results-dir* of build summary data

6.2. assessment_summary.xml

The file *assessment_summary.xml* is an XML file the contains information about the *assessment* task. Its structure and description is show in Listing 5. As an XML file, the order of element is within another element is not significant and should not be relied on, except for the `<arg>` elements which are ordered. Also elements are allowed to be missing and indicate that there is no value.

Listing 5. Structure and description of assessment_summary.xml XML file and its elements. ⋯ represents a text value.

```
XML Elements                                                Description

<assessment-summary>
  <assessment-summary-uuid>...</assessment-summary-uuid>  assessment UUID
  <build-root-dir>...</build-root-dir>                    build root directory
  <package-root-dir>...</package-root-dir>                package root within build
  <package-name>...</package-name>                        package name
  <package-version>...</package-version>                  package version
  <assess-fw>...</assess-fw>                              assessment framework
  <assess-fw-version>...</assess-fw-version>              framework version
  <tool-type>...</tool-type>                              tool name
  <tool-version>...</tool-version>                        tool version
  <platform-name>...</platform-name>                      platform name/version
  <start-ts>...</start-ts>                                start time since Epoch
  <stop-ts>...</stop-ts>                                  end time since Epoch
  <assessment-artifacts>                                  assessment(s) info
    <assessment>                                            assessment info
      <build-artifact-id>...</build-artifact-id>              build id
      <report>...</report>                                    file/dir of tool output
      <stdout>...</stdout>                                    assessment stdout
      <stderr>...</stderr>                                    assessment stderr
      <execution-successful>...</execution-successful>
      <exit-code>...</exit-code>                              assessment exit code
      <exit-signal>...</exit-signal>                          assessment exit signal
      <start-ts>...</start-ts>                                assessment start time
      <stop-ts>...</stop-ts>                                  assessment stop time
      <command>                                               assessment command
        <cwd>...</cwd>                                          command cwd
        <environment>                                           command environment
          <env>...</env>                                          format:  name=value
        </environment>
        <executable>...</executable>                            command executable
        <args>                                                  command args
          <arg>...</arg>                                          arg value
        </args>
      </command>
    </assessment>
  </assessment-artifacts>
</assessment-summary>
```

An assessment consists of one or more invocations as an assessment tool along with setup step (these are not accounted for in *assessment_summary.xml*). The *assessment_summary.xml* contains the sequence of commands executed and their individual output files with one `<asseessment>` element for each instance of the tool execution. To locate assessments that failed, inspect each `<assessment>` element in *assessment_summary.xml* for failure. To determine if an `<assessment>` was a failure, first check if the element `<execution-successful>` exists, if it does then the contained text `false` indicates a failure while the text `true` indicates a successful assessment. If the `<execution-`

successful> element does not exist, then check if the assessment command exited normally (<exit-code> element exists) and without error (<exit-code> containing text is 0). A failed assessment will either have exited by a signal (<exit-signal> element exists) or with a non-0 exit code (<exit-code> containing text is not 0).

In the event of a failure, details of the error may be found in the stdout, stderr, and/or report files. The report path may be a directory in which case the information may be contained in file within this directory. Additional information may be found in *build_assess.out* (or *run.out*).

For assessment tools that have a setup phase and a failure occurs in the setup phase, the *assessment* task will have a status of FAIL, and additional information about the failure may be found in *build_assess.out* (or *run.out*).

For some older frameworks, the <arg> element may not include argv[0]. In this case <executable> should be used for argv[0]. Note that some versions of some frameworks use <end-ts> instead of <stop-ts> so if <stop-ts> does not exist use the value of <end-ts>.

7. Parsed Results Related Files

The parsed results related files are assessment results in a common format and the output from running results parser. The SWAMP result parser translate the raw (native) tool result files into a common format: SCARF (SWAMP Common Assessment Result Format, https://github.com/mirswamp/swamp-scarf-io) or SARIF (Static Assessment Result Interchange Format, https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=sarif and https://github.com/oasis-tcs/sarif-spec/).

The file *parsed_results.conf* in the output directory is a SWAMP Conf formatted file (see Section 10) containing key, value pairs with the paths to other files. The key, value pairs are documented in Table 6. The *results-archive* attribute is a path to the archive of the assessment related files relative to this conf file. The *parsed-results-dir* is a path within the archive, and the remaining attributes are paths relative to the *parsed-results-dir* within the archive or informational values. If the *parsed-results* task fails, these output files may be useful in diagnosing the cause of the failure.

Table 6. parsed_results.conf attribute names and common values. Unless otherwise noted, the value is the path relative to the results-dir path inside the results-archive archive.

Attribute Name	Common Value	Note
metrics		number of metrics in the SCARF file (SARIF does not support metrics at this time
parsed-results-archive	parsed_results.tar.gz	path of parsed results archive relative to this file
parsed-results-dir	parsed_results	path of parsed results directory within the parsed results archive

Attribute Name	Common Value	Note
parsed-results-file	parsed_results.xml	SCARF file (**DEPRECATED:** use scarf-file)
resultparser-stderr-file	resultparser_stderr.out	stderr of the result parser
resultparser-stdout-file	resultparser_stdout.out	stdout of the result parser
sarif-file	parsed_results.sarif.json	SARIF file (if missing no SARIF file was produced)
sarif-`TYPE`-file	parsed_results-`TYPE`.sarif-external-properties.json	Where `TYPE` is one of the following: • addresses • artifacts • graphs • invocations • logicalLocations • policies • webRequests • webResponses • results • taxonomies • threadFlowLocations • translations
sarif-`TYPE`-`N`-file	parsed_results-`TYPE`-`N`.sarif-external-properties.json	Where `TYPE` is as above and `N` is a non-negative integer
scarf-file	parsed_results.xml	SCARF file (if missing use deprecated parsed-results-file attribute, if also missing, no SCARF file was generated)
status		*PASS* or *FAIL* indicating success or failure of the result parser respectively (status of the *parse-results* task)
weaknesses		number of weaknesses in the output files

The *metrics*, *status*, and *weaknesses* attributes are informational. In the event the *result-parser* task

failed the *resultparser-stdout-file* and *resultparser-stderr-file* files may provide additional information on why the *result-parser* failed. All other attributes are files for one common formats.

Based on SWAMP settings, SCARF, SARIF or both output formats may be produced; and SARIF output may consist of a single or multiple files (SCARF is always a single XML formatted file). For efficiency, SARIF files may be split into multiple JSON formatted files. The main SARIF file is denoted by the attribute *sarif-file* and sub-files are denoted by the *sarif-<SUBTYPE>-file* (where <*SUBTYPE*> is an arbitrary string) attributes as shown in Table 6. If SARIF is generated, then the files denoted by the attribute keys *sarif-file* and *sarif-<SUBTYPE>-file* are the complete collection of files required to process the SARIF output.

8. run.out

The *run.out* file contains the output of provisioning the VM or Docker container and properties of the environment before and after the assessment framework is run. The output of the assessment framework is also contained in this file in older versions of the framework.

This file may provide additional information if the *setup* task fails.

In older versions of the SWAMP frameworks this file is unstructured. In current versions of the frameworks, the file is structured. Lines starting with `==========` are control lines while other lines are data related to the prior control line. Common control files and their format are:

- Information message of <MSG>.

```
========== === <MSG>
```

- Shows variable <VAR> has the value <VALUE>.

```
========== === $<VAR>: <VALUE>
```

- For subsequent commands that are run, redirect stdout, stderr or both (<TYPE> being *stdout*, *stderr* or *stdout and stderr* respectively) to the file <FILE>. <APPEND> may be empty (file overwritten) or *append* (file appended).

```
========== === Redirect <APPEND> <TYPE> to <FILE>
```

- End redirection of stdout or stderr for commands (<TYPE> being *stdout* or *stderr* respectively.

```
========== === End <TYPE> redirect
```

- Execute <CMD> and wait for it to complete. <CMD> is escaped using Bash escaping. The stdout and stderr of the command are located in <OUTPUT> and span as many lines as necessary (unless the stdout and stderr were redirected to a file. After the command completes a new-line

character is written to the output. If the command exits with a non-zero exit code, an <ERROR_MSG> control line is produced with the exit code.

```
========================== + <CMD>
<OUTPUT>
```

- Execute <BACKGROUND_CMD> in the background. <BACKGROUND_CMD> is escaped using Bash escaping.

```
========================== +& <BACKGROUND_CMD>
```

- Informational error message of <ERROR_MSG>. Typically this relates to the previous command being run.

```
========================== ### ##### <ERROR_MSG>
```

- Timing data for a command with <TIMING_DATA> being the command, the start time, stop time, or duration.

```
==========================     | <TIMING_DATA>
```

9. build_assess.out

The *build_assess.out* file contains the output of the framework that is not placed in other files. This file is unstructured and is dependent on the framework run. This file may be useful to diagnose errors in tasks where the output of the task is not in other files.

10. SWAMP Conf File Format

Several SWAMP files use the SWAMP Conf file format described in this section. These are text files containing a set of key, value pairs. Keys are a sequence of characters that do not include whitespace characters, ':' or '=' characters. The value can be any arbitrary sequence of character.

The SWAMP Conf file has six types of constructs. Two of them are lines that are ignored and the other four are different mechanism to express the key that trade-off readability for expressibility.

For non-ignored lines, the key is determined by first skipping any initial whitespace. Then all characters up to the next whitespace character, ':' or '=' are the key (it is an error is the key string is empty). Next all non-newline whitespace characters are skipped to determine the type of line as described below. It is an error if the next character is not a ':' or '='.

The six type of constructs are:

- **Blank Lines** consisting of only whitespace characters are ignored.

```

```

- **Comments** have a '#' as the first non-whitespace character and are ignored.

```
# comment
      # another comment
```

- **Trimmed Value Assignment** has an '=' as the next non-whitespace character after the key. The value is the characters after the '=' to the end of the line with leading and trailing whitespace removed. In the example below k1 has the value 'v1' and k2 has the value 'v2 = 7'.

```
k1=v1
   k2  =   v2 = 7
```

- **Literal Value Assignment** has ':=' as the next non-whitespace characters after the key. The value is the characters after the ':=' to the end of the line retaining all whitespace except the terminating newline character. In the example below k3 has the value 'v1' and k4 has the value ' v2 = 7'.

```
k3:=v1
   k4  :=   v2 = 7
```

- **Multiple Line Value Assignment** has ':<N>L=' (where <N> is a positive integer) as the next non-whitespace characters after the key. The value is characters after the ':<N>L=' and the next <N> - 1 lines retaining all whitespace except the final terminating newline character. In the example below k5 has the value ' line 1\n line 2' and k6 has the value ' line 1\n line 2\n x=7'.

```
k5:2L=  line 1
 line 2
   k6 :3L=  line 1
 line 2
 x=7
```

- **Multiple Character Value Assignment** has ':<N>C=' (where <N> is a positive integer) as the next non-whitespace characters after the key. The value is the next <N> characters after the ':<N>C=' retaining all characters include newlines as-is. Processing of the next key, value pair are started as if the character after the value is the first character of a line. In the example below k7 has the value '12\n4' and k8 has the value '12345'.

```
k7:4C=12
4     k8 :5C=12345
```

For greatest readability, producers should use start the key in the first column and put a space after the key. They should also prefer Trimmed Value Assignments with space before the value. If the key has leading or trailing whitespace Literal Value Assignment should be used and Multiple Line Value Assignment should be used only if the value contains multiple lines. Multiple Character Value Assignment should be avoided, but if used a new-line should follow the value.

Listing 6 shows pseudocode to parse a SWAMP Conf File into an associative array.

Listing 6. Pseudo code to parse SWAMP Conf files. $<N> represents the N^{th} group matched `$<N> matched' is a boolean value of the N^{th} group matching at all.

```
KeyValuePairs ParseSwampConfFile(filename)
{
    file = open(filename)
    keyValuePairs = ()
    buffer = ''
    while (true) {
        if (buffer == '') {
            if (eof(file))  {
                return keyValuePairs
            }
            buffer = ReadLine(file)
        }
        if (buffer =~ /^\s*$/)  {
            # Blank Line, ignore
            buffer = ''
        }  elsif (buffer =~ /^\s*#/)  {
            # Comment, ignore
            buffer = ''
        }  elsif (buffer =~ /(^[^:=]*)(:[^=]*)?=(.*)$/)  {
            k = TrimWhiteSpaceBothEnds($1)
            if (k == '')  {
                return ERROR('key name contains no characters')
            }
            afterEquals = $3
            if ($2 matched)  {
                lengthModifier = $2
                if ($lengthModifier = ':')  {
                    # Literal Value Assignment
                    count = '1'
                    type = 'L'
                }
                if ($lengthModifier =~ /:([0-9]+)([LC])/)  {
                    count = StringToInt($1)
                    type = $2
                    v = afterEquals
                    if (type == 'L')  {
                        # Multiple Line Value Assignment
                        for (i = 2; i <= count; ++i)  {
                            if (eof(file))  {
                                return ERROR('unexpected end of file');
```

```
                            }
                            v += ReadLine(file)
                        }
                        v = RemoveFinalNewLine(v)
                        buffer = ''
                    }  elsif (type == 'C')  {
                        # Multiple Character Value Assignment
                        v = afterEquals
                        while (length(v) < count)  {
                            if (eof(file))  {
                                return ERROR('unexpected end of file');
                            }
                            v += ReadLine(file)
                        }
                        buffer = CharsAfterPos(v, count)
                        v = CharsUptoAndIncluding(v, count)
                    }
                }  else  {
                    return ERROR('invalid length modifier')
                }
            }  else  {
                # Trimmed Value Assignment
                v = TrimWhiteSpaceBothEnds(afterEquals)
                buffer = ''
            }
            keyValuePairs[k] = v
        }  else  {
            return ERROR("bad line, no = found")
        }
    }
}
```