|  |  |
| --- | --- |
| Engineering Note | TSE-Security Tools |
| 05 | How to Populate SARIF Logs for Effective Run-Over-Run Result Matching |
|  | [Subject] |

**Abstract**

*In which we describe how tools developers and results management systems should generate SARIF to most effectively drive baselining and run-over-run analysis comparisons.*

**Revisions**

|  |  |  |
| --- | --- | --- |
| **Date** | **Summary** | **Author/Editor** |
| 04/11/2018 | First draft | Everett Maus |
| 04/15/2018 | Second draft | Michael C. Fanning |
| 04/30/2018 | Edits based on internal review, add summary of guidance | Michael C. Fanning |
| 05/27/2018 | Edits based on Semmle review | Michael C. Fanning |
|  |  |  |

**Table of Contents**

[1. Goals 1](#_Toc515280522)

[2. Non-Goals 1](#_Toc515280523)

[3. Definitions 1](#_Toc515280524)

[4. Problem Statement 1](#_Toc515280525)

[4.1. Baseline/Fingerprint ‘Stressors’ 1](#_Toc515280526)

[5. Solution Space 1](#_Toc515280527)

[5.1.1. Elements of a Result Identity 1](#_Toc515280528)

[5.2. In-source Suppressions 1](#_Toc515280529)

[5.2.1. Advantages: 1](#_Toc515280530)

[5.2.2. Disadvantages: 1](#_Toc515280531)

[5.3. Side-car Suppressions 1](#_Toc515280532)

[5.3.1. Advantages: 1](#_Toc515280533)

[6. Guiding Principles 1](#_Toc515280534)

[7. Proposed Solution 1](#_Toc515280535)

[7.1. Guidance Summary 1](#_Toc515280536)

[7.2. SARIF Properties Population Summary 1](#_Toc515280537)

[7.3. run.Result factoring 1](#_Toc515280538)

[7.4. Observation target data availability 1](#_Toc515280539)

[7.5. run.logicalId 1](#_Toc515280540)

[7.6. run.versionControlProvenance 1](#_Toc515280541)

[7.7. tool.name and tool.version 1](#_Toc515280542)

[7.8. result.ruleId 1](#_Toc515280543)

[7.9. result.message.arguments 1](#_Toc515280544)

[7.10. result.fullyQualifiedLogicalName 1](#_Toc515280545)

[7.11. result.relatedLocations, result.codeFlows, result.stacks, result.graphTraversals 1](#_Toc515280546)

[7.12. result.fingerprints 1](#_Toc515280547)

[7.13. result.partialFingerprints 1](#_Toc515280548)

[7.14. result.suppressionStates 1](#_Toc515280549)

[7.15. physicalLocation (.region and .contextRegion) 1](#_Toc515280550)

[7.16. region.Snippet 1](#_Toc515280551)

[7.17. file.contents 1](#_Toc515280552)

[7.18. file.hashes 1](#_Toc515280553)

# Goals

* Provide detailed guidance and best practices for populating SARIF log files to best support future baselining solutions.
* Summarize high-level need for baselining and the challenges any effective solution will need to address.
* Summarize existing approaches for baseline fingerprinting and articulate their strengths and weaknesses.
* Propose and define common terminology for describing results fingerprinting and baselining.

# Non-Goals

* This note does not describe a complete results management solution, although we touch on some high-level requirements of such a system.
* This note does not specify a detailed algorithm for matching or persisting results fingerprints run-over-run.
* This note does not describe ‘baseline-free’ solutions that obviate a need for most baselining functionality.

# Definitions

**Scan Tool:** An automated process that runs against a scan scope or set of scan scope, and outputs a set of observations or quality results relevant to them.

**Scan Scope:** The target of a scan tool; may be source code or a source repository, a directory , binaries, a web endpoint, a process, network share, etc.

**Scan Result**: The output of an analysis of a scan scope by a scan tool, containing descriptions and relevant data about potential weaknesses within or other observations regarding the scan scope or a program element within the scan scope.

**Observation Target:** A program element with which a scan tool has associated a result.

**Baseline:** A set of results from a scan tool run at a specific point-in-time, to be compared to a separate run of an identically configured scan tool.

**Fingerprint**. A persistent (stable) string value that attempts to uniquely identify a specific scan result run-over-run, even as the scan scope continues to change.

**Candidate Fingerprint**. A fingerprint that (along with others) will be considered when attempting to match a result run-over-run.

**Partial Fingerprint (or Fingerprint Contribution)**. A constituent element of a fingerprint that is merged with or weighed alongside other information to produce a candidate fingerprint.

**Result Matching:** The process of comparing scan tool results from one run of a scan tool to those of a second run of the same tool. The expected outcome of this process is to identify logically identical scan results that are shared by or unique to each run.

**Baseline Stressor**: A contributing factor that complicates result matching. E.g., code churn or a source file rename may make it difficult to correlate a result between two tool scans.

**Deduplication/Deduping:** A common term for result matching. This document avoids references to deduplication, as the phrase ‘result matching’ is more descriptive of the process.

**Suppress:** To add metadata to a finding to indicate that it can be ignored. A result may be suppressed for many reasons, e.g., if a result is a false positive, fires against code that does not ship or will be obsoleted soon, is deferred because it will be fixed in a future milestone, etc.

**In-Source (or In-lined) Suppression**: A suppression that is applied within the source code of an observation target. Where required language and file format features exist, in-source suppressions may be rendered as annotations applied directly to an observation target (such as a class declaration), as #pragma mechanisms that control compiler error/warning behaviors, or as source comments. Sometimes referred to as an ‘in-lined’ suppression.

**Side-car (or Global) Suppression**. A suppression that isn’t applied directly to an observation target but is persisted in a separate storage location (such as an XML, JSON or source code file but also a database or other store). Sometimes referred to as a ‘global’ suppression.

**Rebaseline**. The act of discarding a current baseline entirely in preference of a more recent result set. This is typically done for specific scenarios (such as updating a tool set to a new version) where it is preferable not to halt the production pipeline to resolve any change in the result set that was introduced.

**Results Management Solution:** An end to end system that consumes tool results, turns them into actionable tasks or business decisions, and tracks their progress towards final disposition (fixed, deferred, etc.).

# Problem Statement

Scan tools provide a mechanism for assessing quality at a specific point-in-time. It is important to apply this assessment continuously, at some appropriate cadence, to identify new problems that emerge as a result of ongoing changes in source code, deployment environments, etc. For continuous scanning to be efficient, it is critical to correlate, where possible, individual issues (which may already have been assessed or otherwise accounted for in a results management workflow) between multiple scan runs.

To provide run-over-run correlation of scan results, results management solutions may construct ‘fingerprints’ (unique identifiers) of active results and persist them to a ‘baseline’. Two baselines can be compared by a result matching algorithm to identify which results are unique to each run and which occur in both instances. Common scenarios that argue for effective results matching:

* **‘Stop the bleeding.’** Teams with emergent maturity often wish to, as a first step, ignore all technical debt that has accumulated in code and to begin blocking new results as they are introduced. This approach has a strong appeal to developer psychology, as developers are held accountable only for quality problems that they have introduced in their current work.
* **Efficient work item creation**. Once a work item is created for addressing a static analysis result, the problem is introduced into the engineering workflow process and other signal relating to the problem should be suppressed. If results cannot be effectively tracked run-over-run, there is a risk of opening duplicate tracking items, compromising efficiency.
* **Tracking ‘Won’t fix’ disposition**. Tool results are explicitly ignored for a wide variety of reasons, e.g., due to tool noise, identification of problems in non-shipping or soon-to-be-obsoleted code, etc. Requiring teams to revisit results that have already be determined to be out-of-scope is a significant source of team dissatisfaction.

## Baseline/Fingerprint ‘Stressors’

The following conditions can compromise issue fingerprints, with the result that a result cannot be tracked between two versions of a code base. A result matching system will be effective according to its ability to overcome these problems:

1. **File and directory renames/moves**. Baseline fingerprints that store fully-qualified paths to files that contain quality weaknesses can break when file name and directory paths change. This class of stressor tends to break side-car suppressions; because in-source suppressions are applied to observation targets within their containing file, they are very resilient to this kind of change.
2. **Intra-file code locations**. File line and column locations associated with results will change as developers continue to modify code. Baseline fingerprints that persist this data (typically in a side-car mechanism) are therefore fragile.
3. **Logical name changes**. It can be helpful to refer to an observation target by its logical name, i.e., a fully-qualified path that identifies an observation target within a container. A .NET method, for example, can be identified by its namespace, type name, and a method identifier with sufficient details (such as return value and parameters) to uniquely identify it within the type. A fingerprint that refers to an item’s logical name can be stable even when file/directory name and location details change. It is common for developers to keep directory and file names in sync with namespace and file details, however, so the value of logical name resilience is lessened in practice.
4. **Tool updates**. Tool updates present challenges to baselining systems and require consideration (warranting its own engineering note). A new version of a tool may produce new valid results as well as false positives or introduce new false negatives (where valid results disappear). Tools may also alter log file production or other built-in suppression behavior that impact baselines. Tool updates can also result in rule identifier changes that break suppressions.
5. **Non-deterministic analysis.** Tool results may be non-deterministic, due to bugs in the tool, as a by-design feature (such as a fuzzer that randomizes its mutation of input files), or due to non-determinism in the analysis target when compiled or, for dynamic checkers, as it behaves at runtime. Web endpoint scanning is a notorious generator of non-deterministic results.
6. **Build changes**. Changes to compiler settings (or mistakenly applying a baseline produced for a ‘debug’ build to a ‘release’ scan scope) can compromise baselines. UI scan scopes (such as web sites) may differ due to running as a specific language, with altered user agent string, etc.). To provide a clear example of this stressor, a debug build may contain code that provokes a scan result which is compiled away in a release (because the problematic code is located within a #if DEBUG conditional).
7. **Branch merges.** Baselines are typically kept in sync with a branch. When merging a branch back into another requires rationalizing the baseline associated with each branch against a current scan run against the merged code. The practical effect is that the current scan runs need to be compared against two divergent baselines,

# Solution Space

### Elements of a Result Identity

A result may be conceptually and uniquely identified by the following elements.

* **Tool/tool version**. Run-over-run scan results are organized by the tools that produced them. A tool’s behavior can change as its code is updated and so it is important to track versioning details for the program. A tool’s version implicitly includes the versioning of any plug-ins, scripts, etc., that drive scan tool behavior.
* **Rule and rule output**. An individual result is partly defined by a conceptual notion of a quality standard (e.g., ‘do not utilize broken cryptography algorithms’). A specific result may also be denoted by the user-facing text and/or additional relevant data provided by a rule.
* **Scan scope/scan scope version**. Each result is associated with an entity of a specific version at a specific point in time, e.g., SomeNamespace.SomeType::SomeFunction()) as it existed in the master branch of a repository synced to a specific commit id. Scan scope details can also include things like a container name, source code locations, etc. A scan scope version may not be well-defined in some cases (e.g., a running process).
* **Scan scope creation/composition data.** An identical set of scan scope files may result in substantially different outputs when compiled with differing build settings (macro definitions, optimized vs. not, etc.). It is quite common that these differences in compilation output provoke different scan tool outputs, with the result that baselines typically need to be strongly associated with a stable compiler toolchain configuration.
* **Scan data**. Every result is associated with some data that has been assessed to produce a result, e.g., a presumed code flow to a dangerous sink, or an observation that the broken MD2 hash function is used.
* **Result location**. Multiple instances of a rule must be differentiated by some logical or physical location details, e.g., several results of a rule that occur on the same line may be further qualified by column information to track them as distinct problems.

The information above is used to distill analysis results into a set of actionable results with no duplication.

## In-source Suppressions

In-source suppressions are indicators that are in-lined within source code to indicate that a tool result can be ignored. Scan observation target identifier and result locations are not rendered explicitly in an in-source suppression; instead, the in-source suppression is applied to the relevant observation target, e.g., as an attribute, annotation or code comment that is proximate to it.

### Advantages:

* **In-source suppressions are resilient** to changes in both physical (file and directory path, file line and column data) and logical (namespace and type name, etc.) locations, a significant advantage for baselining scenarios.
* **In-source suppressions are efficient to consume.** Analysis tools are in the business of parsing target language and in-source suppressions are therefore easy to locate and consume without doing additional work (such as locating and parsing a side-car suppression file). Most tools look for and honor in-source suppressions during scan runs, without special configuration of the tool.
* **In-source suppressions provide useful documentation in some cases**. In cases where a scan tool finds a good result, that may be obvious to any developer reading the code, an in-source suppression can provide an explicit indication that a result is known but is ignored, by design. A source code comment or suppression justification may provide the rationale for this case.

### Disadvantages:

* **In-source suppression mechanisms vary widely in form and function by source language** and, in some cases, do not exist at all. JSON, for example, does not support code comments and injecting suppressions into the JSON itself may invalidate files according to their schema.
* **In-source suppressions do not allow easy rebaselining**. It is not possible to easily rebaseline a code base that depends on in source suppressions (for example, due to updating a tool) to accept the current results of a tool as valid. Instead, any tool update requires a developer to review/fix new results and author in-source suppressions for whatever isn’t fixed. In practice, the disadvantage introduces significant downward pressure on tool servicing (which prevent tools improvements and innovation).
* **In-source suppressions clutter code.** Some subset of suppressions can provide useful documentation of why a specific issue has not been fixed. In general, suppressions are used to eliminate false positives in tools, a poor excuse for compromising code readability.
* **In-source suppressions are scan tool-specific**. Introducing a new scan tool to an existing code base can be problematic, as the source code may need to be edited in many places to create a baseline. The addition of new in-source suppressions may further clutter code that is already contains other suppressions.
* **In-source suppressions provide a single, rigid instance of review state.** In-source suppressions are most effective when used to mark an issue as permanently ignored. In practice, many scan results are temporarily ignored (e.g., when work to resolve them is scheduled for a future milestone). No existing in-source suppressions provide nuanced, machine-parsable justifications that describe why a result is suppressed.
* **In-source suppressions are not easily documented/audited**. In-source suppressions allow developers an easy mechanism to avoid satisfying compliance policy, in a way that isn’t easily audited (due to the large # of suppression mechanisms that exist). It can be difficult to produce metrics for the # of in-source suppressions in a code base for the same reason. (NOTE: To overcome this problem, SARIF producers should emit suppressed results to log files and marked them as suppressed in-source. See the result.suppressionStates information below).
* **In-source suppressions are limited in and/or provide undesirable expressiveness.** The #pragma warning disable mechanism provides an identifier and no other details for suppressed results. It is common for other suppression mechanisms to provide an open ‘Justification’ field to document the reason for a suppression, which tends to be a vector for rendering developer opinions on tool quality that aren’t appropriate to ship in product code. No in-source suppression mechanism that we’re aware of provides an expiration mechanism.
* **In-source suppressions are rarely reconsidered by development teams**. In general, there is significant pressure to suppress results to unblock the ship process. It is extremely common, even in cases where a team intends to fix a result at a future time, for results suppressed in source code never to be reconsidered.

## Side-car Suppressions

A side-car suppression is one that is persisted to a store other than files that define the entity for which a quality weakness has been found. Side-car suppressions are often stored in checked in files, with well-known names and locations. Side-car suppressions can also be persisted to arbitrary other stores, such as a remote database.

### Advantages:

* **Side-car suppressions can be expressed in a generic format for all languages**.
* **Side-car suppressions can be more easily audited.**
* **Side-car suppressions provide clean separation of results management from production code.**
* **Multiple side-car suppression files can be used to maintain distinct views into quality.** One suppressions file could contain all current issues, for example and may be used to break builds on introducing any new problem. A second baseline might suppress all results except for a specific set that should be resolved in the current milestone. This latter baseline might be used to generate a burndown of progress towards getting clean.
* **Side-car suppressions provide open-ended expressiveness.**

**Disadvantages:**

* **Side-car suppressions are fragile when file/directory and logical name details change.** A side-car suppression must maintain a reference to an observation target location, which entails either ‘physical’ (e.g., file name + line) or ‘logical’ (namespace + type + method) details. It is very typical for expected code churn to change data that breaks these references. It is not easy to provide mechanisms that automatically update baseline references of this kind. Logical identifiers entail additional costs as these cannot be rendered in a language-agnostic way (i.e., java syntax for rendering logical names differs from C++, etc.). Some logical constructs (such as explicitly referring to a parameter of a method) can’t be referred to within a syntax that is built-in to the relevant language.
* **Side-car suppressions provoke merge breaks.** It is a useful practice to depend on a checked in file as a project baseline. Developers that simultaneous add new entries to the end of a list of suppressions in separate branches often break automatic merging algorithms. [One solution here is to sort suppression entries, which requires developer cooperation or additional automation]. Structured suppression files (such as JSON) exacerbate the merging problem, as these algorithms, by design, do not account for internal file formats.
* **Side-car suppressions must be explicitly referenced in order to assess result management state.**

# Guiding Principles

* Result matching is hard. **SARIF producers should provide all possible SARIF data relevant to result matching**, even when that data is optional according to the SARIF schema. A standard SARIF SDK component will provide the heavy lifting for matching results run-over-run.
* Tool authoring is hard. **SARIF producers participation in a result management system should be limited to a single phase and consist solely of producing a focused SARIF file**, the contents of which is limited to information that only the scan tool can provide. The SARIF SDK post-processing tool will normalize this content and further enrich it. The goal is for the broad results management system to accept this content and function end-to-end without requiring tools authors to provide additional functionality (such as plugging into the result matching or false positive reporting phases).
* Fingerprints can be fragile. **SARIF producers should not recapitulate information that is duplicated elsewhere in the SARIF format within partial fingerprints.** For example, no line location details should be used when producing a partial fingerprint. Instead, a SARIF post-processor, supported by the SDK should use the line locations that are expressed elsewhere in the format to assist in produce complete fingerprints that enable result matching. NOTE: in some rare circumstances, a tool owner may be able to provide a final fingerprint that matches other data emitted in SARIF. For example, a checked in secrets detection may use a password and account combination as a fingerprint (as this data is ok to express or should not be checked into any observation target at all). This data might be expressed as a code snippet and also recapitulated as a SARIF result.fingerprints member.
* To make result matching effective, we assume that **SARIF result matching post-processors will have access to contents of any file that is referenced within the log file**. This may be accomplished by having access to version control systems as well as embedding any content within SARIF that is not managed by a version control system.
* Suppressions should not mask new quality results when introduced into a code base. **SARIF producers must therefore give special attention to result and rule id factoring** (see below).

# Proposed Solution

SARIF producers should adhere to the following guidance when generating log files.

## Guidance Summary

SARIF producers should do the following:

* **Do** factor results so that a single suppression of a result suppresses a single actionable problem.
* **Do** emit results that uniquely identify results by including details specific to them. **Do not** emit static user-facing strings for checks.
* **Do** emit file locations according to section 3.3.4 of the SARIF spec.
* **Do** emit tool.name and tool.version properties.
* **Do** emit run.logicalId.
* **Do** emit result.ruleId.
* **Do** format messages by populating the message.arguments property.
* **Do** emit result.fullyQualifiedLogicalName if one exists for a result observation target.
* **Do** emit code flows, graphs, related locations, and/or stacks if this data is relevant to your tool’s scan results.
* **Do** emit all scan results, even for those that are suppressed in-source. Mark suppressed results by setting result.suppressionStates to ‘suppressedInSource’.
* **Do** emit physicalLocation.region properties if possible.
* **Do** populate physicalLocation.contextRegion but only in cases where your tool can provide a better contextRegion than a post-processing tool.
* **Do** populate result.partialFingerprints but only in cases where your tool can provide a partial fingerprint that does not consist of data that is expressed elsewhere in the SARIF format.
* **Do not** populate result.fingerprints.
* **Do** run the SARIF SDK post-processing tool (in an environment where it can access all files referenced within SARIF) after generating your tool’s context to populate the following SARIF data:
  + result.contextRegion
  + region.snippet
  + the ‘files’ table (particularly for files referenced in SARIF that are not under version control)
  + file.hashes

## SARIF Properties Population Summary

The following SARIF properties are relevant to fingerprinting. Several optional SARIF properties are required (if the data is available) for baselining scenarios.

|  |  |  |  |
| --- | --- | --- | --- |
| **Property or object name** | **Required per SARIF schema** | **Required for fingerprinting**  **(‘required’ designation only applies if data is available)** | **Can be populated by SARIF SDK post-processing library** |
| run.logicalId | Optional | Suggested | No |
| run.versionControlProvenance | Optional | Suggested | Maybe |
| tool.name | Required | Required | No |
| tool.version | Optional | Required | Maybe |
| result.ruleId | Optional | Required | No |
| result.message.arguments | Optional | Required | No |
| result.fullyQualifiedLogicalName | Optional | Required | No |
| result.locations,  result.relatedLocations,  result.stacks, result.codeflows, result.graphTraversals | Optional | Required | No |
|  |  |  |  |
| result.fingerprints | Optional | Avoid except when converting legacy non-SARIF log files to SARIF | Yes |
| result.partialFingerprints | Optional | Use only when tool provides fingerprint data not expressed elsewhere in the SARIF format | Yes |
| result.suppressionStates | Optional | Required | Maybe |
| file.contents | Options | Required (for non-version controlled files | Yes |
| file.hashes | Optional | Required | Yes |
| physicalLocation.region | Optional | Required | No |
| physicalLocation.contextRegion | Optional | Required | Yes |
| region.snippet | Optional | Required | Yes |

## run.Result factoring

A SARIF result instance is one or more (but typically one) observation or quality weakness associated with a single observation target. This set of one or more weaknesses should comprise an issue that is sensible to manage as a single problem (in other words, the body of findings should either be fixed or ignored as a set). From the perspective of a results management system, the first responsibility of a SARIF producer is to emit results that are factored properly. If a user suppresses one result but sees other results are still active (because they refer to a common underlying quality weakness), the results are not factored properly. A user in this case will reasonably object to having to repeat the act of suppressing the other duplicate results. Similarly, a single SARIF result should not refer to multiple distinct problems, as it is not clear from a single suppression that each unique result suppressed by it has been accurately diagnosed. Suppressing multiple results with a single suppression also makes it more difficult to assess costs associated with addressing technical debt (or fixing false positives) that have accumulated in a code base. If one result can refer to 100 distinct issues but a second only refers to one, the metric of ‘two results’ is meaningless for developer costing exercises.

IMPORTANT:

* Scan tools should be careful to eliminate, where possible, results that ‘waterfall’ because of a root cause that is flagged by another check. In this situation, a tool provider should limit reporting to the rule associated with the salient, root case quality problem. Limiting reporting in this way lower costs for result investigation and management. To provide a concrete example, a class may be flagged as a candidate to be ‘sealed’, preventing other classes from inheriting from it. A second rule might look for problems in unsealed classes and fire more results. In this case, if the class should incontrovertibly be sealed (and therefore not inheritable) a single result should fire for that problem only (and any other results based on the fact that the class is improperly inheritable should not be emitted).

## Observation target data availability

The SARIF SDK post-processer and many results viewers cannot work properly without sufficient access to the source files and other scan data that are required to understand (and fingerprint) a result. Viewers and post-processing scenarios can operate in a different environment that was used to produce a SARIF log file. For this reason, SARIF producers should follow best practices for rendering URLs in a way that both provides immediate access to scan data content or provides information to make locating that data more efficient.

IMPORTANT:

* **Do** read section 3.3.4 (‘Guidance on the use of fileLocation objects’) carefully and adhere to its guidance
* **Do** ensure that links are rendered in a way that maximizes the ability of post-processing tools and other consumers to locate file contents. In general, tools should focus on providing detailed code locations in SARIF. Post-processing tools, depending on the scenario, can be used to inject snippets, or even entire file contents, as necessary.
* An exception to the above can occur in cases where observation targets are not under version control, are auto-generated and/or overwritten during analysis. For these cases, **do** emit relevant code snippets and/or file contents directly to the SARIF log file (see [file.contents](#_file.contents) for more details).

## run.logicalId

The run.logicalId property establishes the conceptual identity or purpose of a run and consists of a primary component (the portion of the text string that precedes any forward slash in the identifier) and an optional secondary component, the beginning of which is denoted by a forward slash. The primary component of this identifier should not contain any version information associated with the scan scope. It may be used as a user-facing string when reporting and so a human-readable description of the analysis (e.g., ‘Nightly comprehensive SDL tools run’) may be preferable.

The secondary component is used to provide details that are relevant to the construction of the scan scope (which may be required to differentiate baselines between them), e.g., ‘Nightly comprehensive SDL tools run/DEBUG x86’.

IMPORTANT:

* It is typical that baselines will be created on a one-to-one basis with the full stable identifier. Two baselines that differ by the full stable identifier will not be regarded as valid to compare to each other.

## run.versionControlProvenance

The run.versionControlProvenance property documents the specific version of a set of scan scopes by listing the version control details of that content (server URL, commit number, etc.). This information is important for managing baselines and should be provided where possible. A result management system may maintain a baseline for each branch of a repository, for example. The commit number associated with a baseline can be used to determine which version of it supersedes another.

IMPORTANT:

* The SARIF SDK post-processing tool will provide a mechanism for injecting version control details into any SARIF file.

## tool.name and tool.version

Due to the lack of a centralized mapping of shared conceptual rule identifiers (which would allow a suppression for a single rule id to be valid for implementations of that conceptual problem provided by multiple rules), the tool name is a required for baselining purposes. Tool version should also be provided, to assist automation in driving the special scenario of refreshing baselines on tool update.

IMPORTANT:

* SARIF producers **MUST** update their version for any implementation changes that can result in analysis differences when applied to identical scan scopes. The broader SARIF result management system will use the change in tool version to trigger rebaselining and other updates that prevent unexpected breaks in engineering workflow.
* The ‘version’ of a tool implicitly includes the version of any plug-ins, scripts, etc., that drive scan tool behavior. It is not clear that SARIF allows for version details related to a tool’s extensibility model. This problem is current tracked in an [OASIS github issue](https://github.com/oasis-tcs/sarif-spec/issues/179).

## result.ruleId

The result.ruleId property identifies a specific observation or quality weakness that is detected and should always be provided if available.

IMPORTANT:

* Rule identifiers that are overly broad will result in overly broad suppressions.

## result.message.arguments

Every SARIF result contains a ‘message’ property that provides user-facing text describing the result. A message instance can be parameterized by separating the dynamic values produced for the result and storing these in the message.arguments property. The other message properties (such as message.text) can provide a format string into which message.arguments will be copied to produce a finished string.

Explicitly populating a message with arguments clearly separates the dynamic data from the static user-facing content associated with a rule. Changes in the dynamically generated content may indicate a unique issue.

IMPORTANT:

* **Do** populate message.arguments the dynamic information that helps uniquely identify a result.
* **Do not** emit static user-facing strings for a rule that contain no details specific to the result. This compromises result matching. It also compromises user experience, as it is difficult to manage multiple messages in a user experience such as an error list that repeat the error message. Consider the difference in utility between two instances of the user-facing message (“Method variable was not initialized.”) and (“Method variable ‘pBuffer’ was not initialized.” and “Method variable ‘nCount’ was not initialized.)

See section 3.19.2 of the SARIF v2 specification for an example of populating message.arguments.

## result.fullyQualifiedLogicalName

A logical location is a conceptual or structural path to an observation target that is not based on details such as file paths for the version-controlled files that define it. A managed type, for example, consists of a logical namespace and type name. A <div> element is contained within a structured HTML/XHTML document. An observation target may also be persisted to a compressed archive or embedded within a container document such as a PDF. A logical location may remain stable when the physical location details (such as file path and line locations) change. A tool developer should therefore populate the logical location details if possible. NOTE: in practice, particularly in managed code development, it is common for directory and file paths to remain in sync with namespace and type names, and so an effective baselining system should be prepared for both candidate computed fingerprints to break.

IMPORTANT:

* A logical name is an internal reference that typically should not contain a reference to the name of its container (such as a Java archive or managed assembly file name).

## result.relatedLocations, result.codeFlows, result.stacks, result.graphTraversals

Results can be annotated with a complex set of locations that comprise a quality concern (for example, a code path from receipt of untrusted data from a web source that flows to a dangerous sink). These details contribute to the conceptual identity of a result and an effective baselining system will account for them.

IMPORTANT:

* The longer a code flow, stack, etc., the more likely it is to be affected by an arbitrary code change without necessarily changing the validity of a finding. The observation that one of these elements has not changed between two runs may be more helpful when result matching, therefore, and less helpful (or not at all) when they have.

## result.fingerprints

A fingerprint is a standalone, final computed identifier that uniquely identifies a conceptual issue associated with a specific observation target. It is rare for tools to have sufficient information to produce an effective computed fingerprint. This slot is provided for existing tools with complex results production capabilities that provide one (Fortify is an example) and for post-processing tools that construct fingerprints based on additional context (such as version control information for things like file name changes) that are available to them.

IMPORTANT:

* Avoid populating this property except to store values produced by legacy pre-SARIF tools (where legacy formats are converted into SARIF).

## result.partialFingerprints

A partial fingerprint is a value that can be combined with other SARIF data (such as tool, rule, and logical location details) to produce a candidate, final computed fingerprint. Each partialFingerprint value should be specified by a human-readable key (e.g., ‘MD5 hash of code path to call site’) that is stable over time, given the specifics of the algorithm that produced the fingerprint. If the code that produces a fingerprint changes sufficiently that two versions of the algorithm may produce a different value for the same SARIF content, the name of the fingerprint should change. It is fine to introduce a versioning convention in the fingerprint name, e.g., ‘MD5 hash of code path to call site - v1’.

IMPORTANT:

* **Do not** populate partial fingerprints with data that can be persisted elsewhere in the SARIF format. The logical name of an observation target, for example, is useful as a partial fingerprint, but should be persisted to the result.logicalLocation.fullyQualifiedName property.
* **Do not** persist partial fingerprints that need to be combined with other partial fingerprints (and other SARIF data) to be effective. Instead, think of each partial fingerprint as an identifier that, in combination with logical name, physical location, and other information in the SARIF format comprises a candidate fingerprint.

## result.suppressionStates

The result.suppressionStates properties provides an array of values that indicates whether a result has been suppressed, either in source or an external (side-car) suppressions store. If an analysis tool itself is aware of in-source suppressions, it should always emit suppressed results to the log file and mark each as ‘suppressedInSource’. A result management processing tool may also decorate results as ‘suppressedExternally’.

IMPORTANT:

* If results that are suppressed in source code are not persisted to the SARIF log file, they are not viewable for auditing or other compliance purposes. This omission allows engineers to easily workaround compliance policy without being held accountable for whatever business risk accrues from suppressing, rather than fixing, a scan result.

## physicalLocation (.region and .contextRegion)

Physical locations consist of a reference to a container file (typically managed under version control) and a specific region or range of bytes within it. This data can be combined with tool and rule data to provide a candidate fingerprint. SARIF producers should provide the most detailed physical location details possible (including populating the physical.region property if at all possible). The primary purpose of a result physical location region is to highlight the specific scan data that demonstrates a quality weakness. (NOTE: This region is sometimes updated in new tool versions (to improve the diagnosability of a problem with the result that a different region of file text is highlighted for an identical problem. This fact needs to be accounted for when rebaselining for tool updates.)

The result.locations property is the primary source of location data associated with a scan result. Physical locations are associated with several other SARIF properties (related locations, code flows, graph traversals, etc.) that are populated by more sophisticated scan tools.

IMPORTANT:

* **Do** populate physicalLocation.region in any circumstance where it can be computed. The region object supports several approaches for populating its members (specifying an offset + length, for example, or specifying start and end line and column locations). Tools need only populate the region using one of these mechanisms. The SARIF SDK can exhaustively populate the remainder of the region object as a post-processing step.
* **Do** populate physicalLocation.region correctly. The **region** object should be limited to the most concise area that demonstrates a quality result. It is the region that will be highlighted and navigated to when a developer double-clicks a result in an error list. It is common for viewers to apply squigglies and pop-up messages for these regions (and so tools producers should avoid populating the region property with large areas of source code unless that range is critical to diagnosing a result).
* **Avoid** populating the physicalLocation.contextRegion object unless your tool can provide a region that improves on the general-case region that the SARIF SDK can provide. The physicalLocation.contextRegion is intended to capture the relevant scan data associated with a result along with sufficient surrounding source code to assist in a preliminary review of a code problem (typically in a browser or other lightweight viewer experience). The SARIF SDK can populate the physical.contextRegion object based on expanding the physicalLocation.region data. Some tools may be able to provide an improved context region and should do so in this case (an HTML analyzer, for example, may be able to provide a context region that exactly matches the encapsulating HTML element of a problem, improving its readability when the code is examined in isolation).

## region.Snippet

The region.snippet property renders the actual bytes or text associated with a region. Assuming that a SARIF producer has populated the physicalLocation object and any associated regions properly, a SARIF SDK post-processer should be able to populate all region.snippet instances within the log file.

IMPORTANT:

* The snippet information is an important fallback in cases where code is undergoing significant churn in physical and logical location details. Baselining tools can combine snippet information (along with tool and rule id) in some cases to match fingerprints that would otherwise break when physical and logical location details change. In these cases, a baselined item that appears to be newly produced in the current run, and a result in the previous baseline which appears to have gone absent, may be matched because they share a tool and rule id and snippet.

## file.contents

The SARIF run.files property contains an array of file objects. Each file object provides additional data for a file that is referenced elsewhere in the format (as a physical location), by a result, stack, code flow, attachment, etc. The file.contents property allows the literal contents of a file to be embedded in the SARIF log file. This is required to persist any content (referenced elsewhere with the log file) that cannot by referenced by URL (e.g., because it is not under source control). This mechanism can also be used to persist content that is generated during a scan or build (which may subsequently be overwritten). All files referenced within a SARIF log file may be persisted in this way when the implied increase in log file size is not a concerned. This approach is more typical when attaching SARIF log files to work item reports than in result matching scenarios.

## file.hashes

The SARIF run.files property contains an array of file objects. Each file object provides additional data for a file that is referenced elsewhere in the format, by a result, stack, code flow, attachment, etc. The file object contains a hashes property that allows for SARIF producers to provide a hash signature of the file. For security reasons, the SARIF specification requires that, when populated, the hashes property should minimally contain a SHA256 hash for the file.

The file hashes object can be used as a first attempt to verify whether analysis results match across scan runs. A tool that uses regular expressions to find issues in source files should return identical results run-over-run when scanning the same file. A result matcher may use file hashes, therefore, to efficiently identify results that are common between two runs (obviously, only in cases where files haven’t changed).

IMPORTANT:

* In general, SARIF tools **do not** need to populate file.hashes. Instead, a post-processing tool tool based on the SARIF SDK should inject this information when it is required to drive result matching. The SARIF tools do need to populate the physicalLocation object for the post-processor to create hashes from that content.