




Scanning Best Practices

Version 2019.8.0



This edition of the *Scanning Best Practices* refers to version 2019.8.0 of Black Duck.

This document created or updated on Tuesday, June 25, 2019.

Please send your comments and suggestions to:

Synopsys
800 District Avenue, Suite 201
Burlington, MA 01803-5061 USA

Copyright © 2019 by Synopsys.

All rights reserved. All use of this documentation is subject to the license agreement between Black Duck Software, Inc. and the licensee. No part of the contents of this document may be reproduced or transmitted in any form or by any means without the prior written permission of Black Duck Software, Inc.

Black Duck, Know Your Code, and the Black Duck logo are registered trademarks of Black Duck Software, Inc. in the United States and other jurisdictions. Black Duck Code Center, Black Duck Code Sight, Black Duck Hub, Black Duck Protex, and Black Duck Suite are trademarks of Black Duck Software, Inc. All other trademarks or registered trademarks are the sole property of their respective owners.

Chapter 1: Scanning Best Practices	1
About scanning, scans, and project versions	1
Scans and scan names	2
Scans and project versions	3
Recommended client scan tool - Synopsys Detect	3
Important Synopsys Detect properties	3
Configuring automated scans	4
Where/when in the build process to invoke the scan	4
Asynchronous versus synchronous scans	5
Scan names, project versions, and versioning	5
Scanning very large projects	7
Including library modules	8
Scanning when network connectivity is an issue	11
Maintaining an open source software component-based review process	11
Poor scanning techniques	11
Including the commit ID or build ID in Black Duck version or scan names	11
Keeping a history of versions on a development branch	12
Chapter 2: Troubleshooting scanning issues	13
Accidental scan proliferation by folder paths which include build or commit ID	13
Solution	13
Accidental scan proliferation by a build server farm	13
Solution	13
Intentional scan and project version proliferation due to using the build or commit ID to retain all build versions	14
Solution	14

Black Duck documentation

The documentation for Black Duck consists of online help and these documents:

Title	File	Description
Release Notes	release_notes.pdf	Contains information about the new and improved features, resolved issues, and known issues in the current and previous releases.
Installing Black Duck using Docker Compose	install_compose.pdf	Contains information about installing and upgrading Black Duck using Docker Compose.
Installing Black Duck using Docker Swarm	install_swarm.pdf	Contains information about installing and upgrading Black Duck using Docker Swarm.
Installing Black Duck using Kubernetes	install_kubernetes.pdf	Contains information about installing and upgrading Black Duck using Kubernetes.
Installing Black Duck using OpenShift	install_openshift.pdf	Contains information about installing and upgrading Black Duck using OpenShift.
Getting Started	getting_started.pdf	Provides first-time users with information on using Black Duck.
Scanning Best Practices	scanning_best_practices.pdf	Provides best practices for scanning.
Getting Started with the SDK	getting_started_sdk.pdf	Contains overview information and a sample use case.

Title	File	Description
Report Database	report_db.pdf	Contains information on using the report database.
User Guide	user_guide.pdf	Contains information on using Black Duck's UI.

Black Duck integration documentation can be found on [Confluence](#).

Customer support

If you have any problems with the software or the documentation, please contact Synopsys Customer Support.

You can contact Synopsys Support in several ways:

- Online: <https://www.synopsys.com/software-integrity/support.html>
- Email: software-integrity-support@synopsys.com
- Phone: See the Contact Us section at the bottom of our [support page](#) to find your local phone number.

Another convenient resource available at all times is the [online customer portal](#).

Synopsys Software Integrity Community

The Synopsys Software Integrity Community is our primary online resource for customer support, solutions, and information. The Community allows users to quickly and easily open support cases and monitor progress, learn important product information, search a knowledgebase, and gain insights from other Software Integrity Group (SIG) customers. The many features included in the Community center around the following collaborative actions:

- Connect – Open support cases and monitor their progress, as well as, monitor issues that require Engineering or Product Management assistance
- Learn – Insights and best practices from other SIG product users to allow you to learn valuable lessons from a diverse group of industry leading companies. In addition, the Customer Hub puts all the latest product news and updates from Synopsys at your fingertips, helping you to better utilize our products and services to maximize the value of open source within your organization.
- Solve – Quickly and easily get the answers you're seeking with the access to rich content and product knowledge from SIG experts and our Knowledgebase.
- Share – Collaborate and connect with Software Integrity Group staff and other customers to crowdsource solutions and share your thoughts on product direction.

[Access the Customer Success Community](#). If you do not have an account or have trouble accessing the system, click [here](#) to get started, or send an email to community.manager@synopsys.com.

Training

Synopsys Software Integrity, Customer Education (SIG Edu) is a one-stop resource for all your Black Duck education needs. It provides you with 24x7 access to online training courses and how-to videos.

New videos and courses are added monthly.

At Synopsys Software Integrity, Customer Education (SIG Edu), you can:

- Learn at your own pace.
- Review courses as often as you wish.
- Take assessments to test your skills.
- Print certificates of completion to showcase your accomplishments.

Learn more at <https://community.synopsys.com/s/education>.

Chapter 1: Scanning Best Practices

This chapter describes best practices for scanning, focusing on Synopsys Detect, the recommended scanning tool.

This document includes information on:

- What scanning client tool to use when configuring scans
- When to use asynchronous versus synchronous scanning
- How to configure automated scans (for example, in a build server or in a CI/CD context)
- How to optimize scanning for performance
- How to avoid common pitfalls

This chapter starts with some background information about scanning, scans, and project versions that is important to understand as part of learning about scanning best practices.

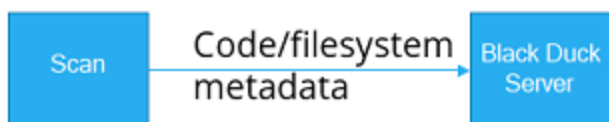
About scanning, scans, and project versions

A scan occurs when a scan client (for example Synopsys Detect) is pointed at a folder.

There are two major modes of scanning: asynchronous and synchronous.

By default, scans are asynchronous. This mode provides the best performance since the scan client does not wait for the results of the scan. An asynchronous scan works as described below

Asynchronous

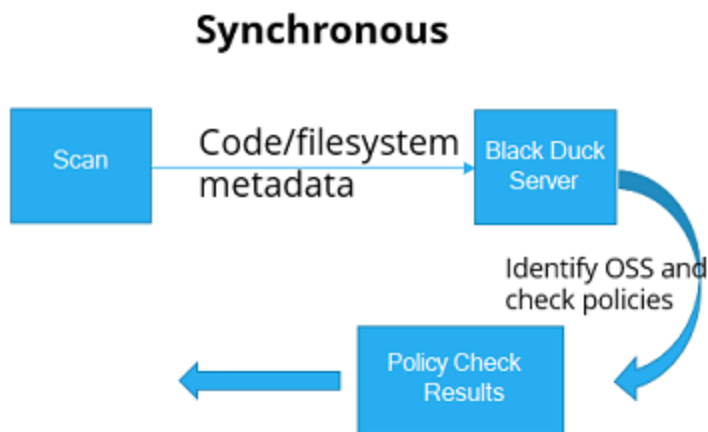


1. Scans generate metadata. Metadata includes:
 - Information from package managers regarding what open source components are used by the project

- File hashes
 - Directory (folder) and file information
2. Metadata is uploaded to the Black Duck server by the scan client and, once the upload is complete, the scan client is finished.
 3. Return/done (check for results later).

Therefore, an asynchronous scan is appropriate when performance is more important than obtaining the results immediately. Users can check for the results later using either the Black Duck UI or REST API. Black Duck Alert can also be used to push notifications of scan results to users.

In some cases, the results are needed immediately, for example, to “fail the build” for policy violations. In these cases, use a synchronous scan, as shown:



When using Synopsys Detect with the **--detect.policy.check.fail.on.severities** property, a synchronous scan is performed which does the following:

1. The scan uploads metadata to the server.
2. The server processes the scan and assesses the results, including checking if any policies are violated.
3. The scan client (Synopsys Detect) waits for the results and exits appropriately, for example, the exit status is set to a non-zero value if any policies are violated.

So, an asynchronous scan simply uploads the scan data and returns which is the best mode when optimizing performance. A synchronous scan uploads the scan and waits for the results which is the right choice if the results are needed immediately.

Scans and scan names

A scan gets a name which, by default, the Black Duck scan client generates. The derived name is formed by combining the hostname of the host running the scan with the full path of the folder being analyzed.

Important: One of the scanning pitfalls is that the path being scanned includes a commit or build ID. In that case, every scan creates a completely new scan when the intent is usually to overwrite the previous scan. This can lead to a proliferation of scans that can degrade system performance and make results in the mapped project version inaccurate. In most cases, it is not a good idea to include the build or commit ID in either the scan name or project version name.

You can override the default, and supply your own scan name by using the **--detect.code.location.name** property in Synopsys Detect (the recommended method) or the **--name** parameter if you are using the Signature Scanner CLI.

If you scan again using the same name, the results of the previous scan are overwritten.

Scans and project versions

A scan is mapped to one, and only one, project version. A project version can have more than one scan mapped to it.

This allows mapping multiple, separate folders and the results of their scans into one aggregated project version. For example, you might have a library in one folder and the application that uses the library in another folder. You can scan the library's folder and the application folder separately and map their scans to the same project version in Black Duck to see the aggregate of both the library and application.

Note that if you delete a project version, it does *not* delete the scans that were mapped to it; it simply unmaps the scans. If your intent is to clean the system of both the versions and the scans, you need to delete the project version *and* the scan.

Recommended client scan tool - Synopsys Detect

Synopsys recommends using Synopsys Detect, a command line interface (CLI) that integrates with your build jobs to identify package manager dependencies as well as file system matches. Synopsys Detect consolidates the functionality of Black Duck build tools, package managers, and continuous integration plugin tools into a single tool. Synopsys Detect makes it easier to set up and scan code bases using a variety of languages/package managers. When run against built binaries, the match accuracy of Synopsys Detect is very high. False positives and misses are greatly reduced.

Click [here](#) for more information on Synopsys Detect.

Important Synopsys Detect properties

- Check for policy violations.
 - **--detect.policy.check.fail.on.severities**. A comma-separated list of policy violation severities that will fail Synopsys Detect. If this is not set, Synopsys Detect will not fail due to policy violations.
- Disable file system scanning and rely on package manager exclusively.
 - **--detect.blackduck.signature.scanner.disabled**. Set to true to disable the Black Duck Signature Scanner.
- Include and exclude options to tune what gets analyzed.
 - **--detect.blackduck.signature.scanner.exclusion.patterns**. Enables you to specify subdirectories to exclude from scans.

- **--detect.blackduck.signature.scanner.exclusion.name.patterns**. Enables you to specify a list of file name patterns to exclude from the Signature Scanner.
- **--detect.blackduck.signature.scanner.paths**. Enables you to specify that these paths and only these paths will be scanned.

Click [here](#) for more information on Synopsys Detect properties.

Configuring automated scans

Setting up automated scans, for example, in a build server or CI/CD context, requires thinking about:

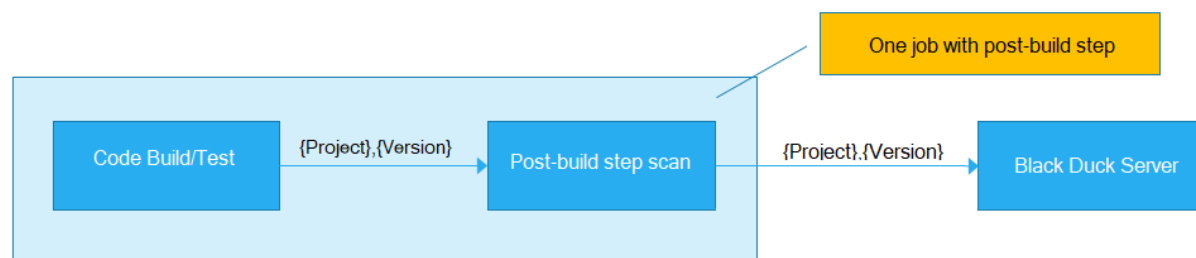
- Where/when in the build process to invoke the scan
- Whether to perform an asynchronous or synchronous scan
- Scan name(s), project version to map the scan (or scans) to, and versioning

Where/when in the build process to invoke the scan

When using a build system such as Jenkins, scans should be run as a post-build step. Performing the scan as a post-build step ensures we can maximize the accuracy of the scan results by scanning the package manager files (if present), the resulting binaries, and the source code.

Another consideration when configuring automated scans is the size and complexity of what is being scanned. It might be advisable to split the scanning into multiple smaller scans if the project being scanned is particularly large (for example larger than 1 GB) or complex (for example, having more than 10,000 files).

For small projects a single post-build step is usually sufficient.

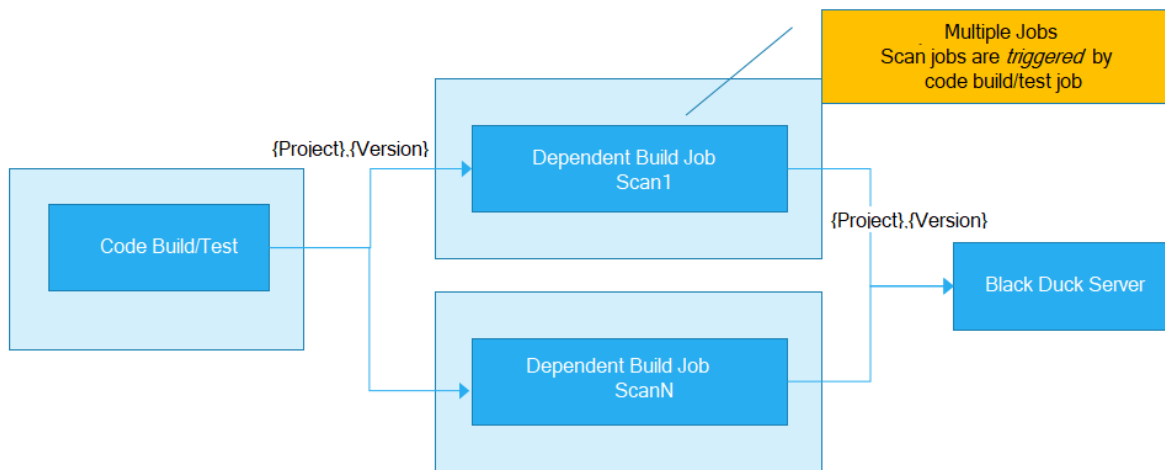


For large projects Synopsys recommends splitting up the scanning into multiple smaller scans that all get mapped to the same project version to aggregate the results. There are multiple ways to split the scanning, and most do not require any change to how the code is built.

For instance, suppose you have a large project that is spread across multiple folders. Instead of having one scan starting from the parent folder for the project, configure scans on each of the subfolders. Each scan is still mapped to the same project version and, as a result, overall scan time will be reduced, and the results will still be aggregated in Black Duck.

As shown, in Jenkins, this can be done by having multiple, downstream jobs scan the subfolders. Or, you could write a shell script that does multiple scans as a post-build step within the parent job.

Note: Alternatively, you can map each of the scans to a separate project version and build a project version hierarchy to aggregate results.



Asynchronous versus synchronous scans

When setting up the automated scans determine if the scans will be asynchronous (default) or synchronous. If the intent is to provide immediate feedback and “break the build” when policy violations occur, use a synchronous scan. However, if performance is more important than providing immediate feedback, then an asynchronous scan is the best choice.

Scan names, project versions, and versioning

We want to configure scans to generate the business results needed and to minimize having to delete or purge items over time. The business results needed usually include:

- Giving developers early warning that the open source they are using has risks associated with it, so they can choose appropriate alternatives to avoid those risks
- Retaining knowledge of software that has been distributed (for example, within a product, or as part of a service run in production)
- Providing the organization with an overall view of the open source in use and the risks associated with it

We want to avoid setting up scans in a way that leads to unbounded growth of data since that can cause system performance to degrade very quickly. And we want to avoid the need to purge or delete items since that is often neglected or forgotten and can be time consuming in any case.

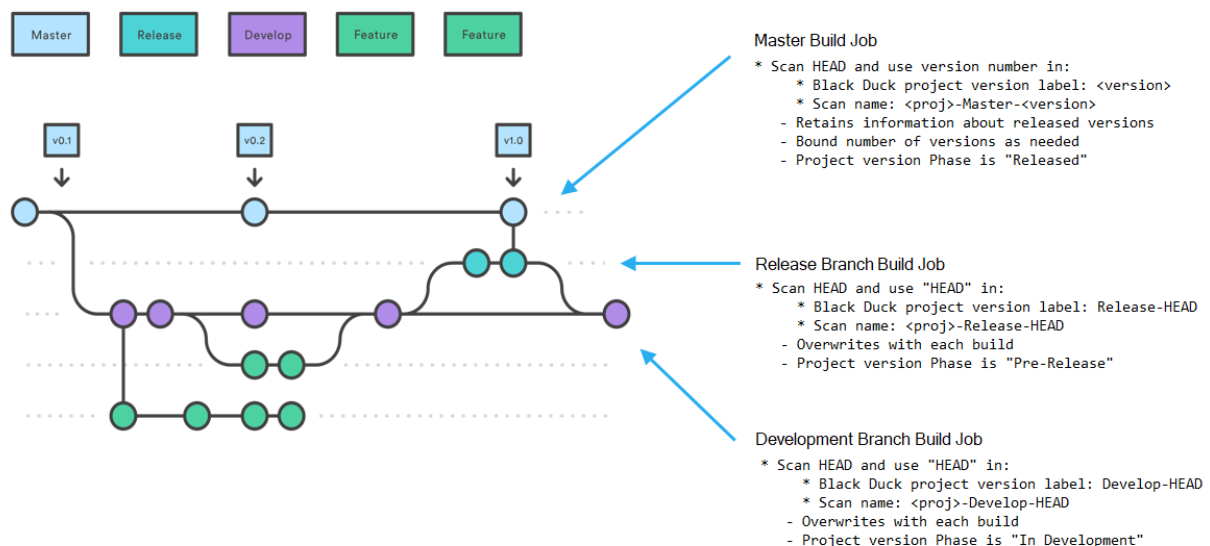
How do we do this?

For projects that use multiple branches

In projects with multiple branches there is invariably one branch that is used to release/distribute software from, and the others are used for development/testing leading up to a release (also known as distribution). We want to configure scanning such that we retain the information corresponding to what is distributed (released). If scans are configured on other branches, for example, to give early warning to developers, those scans should

always overwrite the previous scan results. Setting up the scanning this way allows us to achieve our business goals while minimizing scan/version proliferation.

The best way to describe the method is by looking at the branches in a Gitflow model. In the Gitflow model, software is only distributed (released) from the Master branch and each version is given a label which comes from source control (for example, GIT). Other branches, such as Release or Develop, are used for intermediate builds and do not get distributed.



This diagram depicts configuring three build/scan jobs that occur whenever a commit happens on their respective branches. The next sections provide details on how to configure scans for each branch.

Scanning the Develop branch

The scan on the Develop branch uses the name "HEAD" for the project version and scan names to ensure that the latest scan and BOM results are always overwritten. As the diagram shows, we configure the scan of this branch to use the scan name <proj>-Develop-HEAD where <proj> represents the project name. The other sections of the scan name are fixed so that each time a new scan occurs it overwrites the previous scan. The project version label is set to "Develop-HEAD" and the project version phase is set to **In Development** to indicate that this scan is for code still in development.

This gives developers an early warning about components they have chosen which do not conform to policy or perhaps include high risk security vulnerabilities.

We do not care about retaining history, and it would be wrong to keep prior versions here because it will clog your system and does not provide any business benefit.

Scanning the Release branch

The scan on the Release branch uses the name "HEAD" for the project version and scan names to ensure that the latest scan and BOM results are always overwritten. As the diagram shows, we configure the scan of this branch to use the scan name <proj>-Release-HEAD where <proj> represents the project name. The other sections of the scan name are fixed so that each time a new scan occurs it overwrites the previous scan. The

project version label is set to “Release-HEAD” and the project version phase is set to **Pre-Release** to indicate that this scan is for code that is about to be released.

This gives developers and development management an early warning about components they have chosen which do not conform to policy or perhaps include high risk security vulnerabilities.

We do not care about retaining history, and it would be wrong to keep prior versions because it will clog your system and does not provide any business benefit.

Scanning the Master branch

In Gitflow, all releases happen off the Master branch, and we want to retain knowledge of released versions because those are the ones that run in production or are distributed to customers. We therefore use a version number, usually pulled from a file in the source code, and use it as part of the project version and scan names to ensure that the scan and BOM results are retained for all released versions.

By maintaining all the released versions, we ensure that the Black Duck system can notify you about future vulnerabilities that affect components used in those released versions.

By only creating scans for controlled release versions we keep the number of versions retained in the system to a minimum.

This project version has the **Released** phase.

The following table shows the project version and scan location names if your project is named MY-SAMPLE-PROJECT:

Branch	Scan/Code Location Name (--detect.code.location.name property)	Project Version Name (--detect.project.version.label property)
Develop	MY-SAMPLE-PROJECT-Develop-HEAD	Develop-HEAD
Release	MY-SAMPLE-PROJECT-Release-HEAD	Release-HEAD
Master	MY-SAMPLE-PROJECT-\$(version_label_from_source_code)	\$(version_label_from_source_code)

Exceptions

If you are creating a micro-service or a SaaS application where you only run the latest version of the software, then there is no need to keep prior versions; you should delete them from your system. If you do not delete the older versions (which are no longer running anywhere) you will receive notifications for them when new vulnerabilities are published which creates a false alarm.

You should only maintain versions in Black Duck that correspond to things you must support, for example the version running in production or that you have distributed to your customers.

Scanning very large projects

If you have a very large project (for example, a 4.5 GB monolithic project with 500+ files), your initial instinct may be to scan the entire project in one scan. However, this is the incorrect approach as a single scan of a very large project can stress your Black Duck system (you may need to add additional RAM and CPUs to process the scan). If there are any changes – even if only a small portion of the code has changed – you will need to rescan the entire project each time and look at the results for the entire project. Additionally, you lose the ability

to have a flexible policy management system (for example if you want different subcomponents managed differently), which would not be possible with a single scan.

The correct way to manage scanning for a large project is to divide the project into multiple scans by using multiple dependent jobs. These multiple scans can then become subprojects which you aggregate into one parent project. This scanning strategy has these benefits:

- Only need to scan and inspect what has changed
- Can still report on the entire project
- Provides a flexible policy management system:
 - Different sub-components can be managed differently
 - The parent can still be treated as a single entity.

Note: The size limit for an individual scan is 5 GB. An error message appears if you exceed this limit. Contact Customer Support if you receive this message.

Including library modules

Suppose you want a BOM for a library which is contained in one folder to be combined with an application that sits in another folder. This section describes the two methods you can use to accomplish this by:

- Aggregating multiple scans into the same project version
- Creating a BOM hierarchy

Aggregating multiple scans

To aggregate the results of multiple scans, point those scans to the same project and version. For example, in the example of a library that sits in one folder and an application (that uses the library) in a separate folder:

1. Scan the library folder and point the scan to a project (for example, my-application) and version (for example, 1.0).
2. Scan the application folder and point the scan to the same project (my-application) and version (1.0).

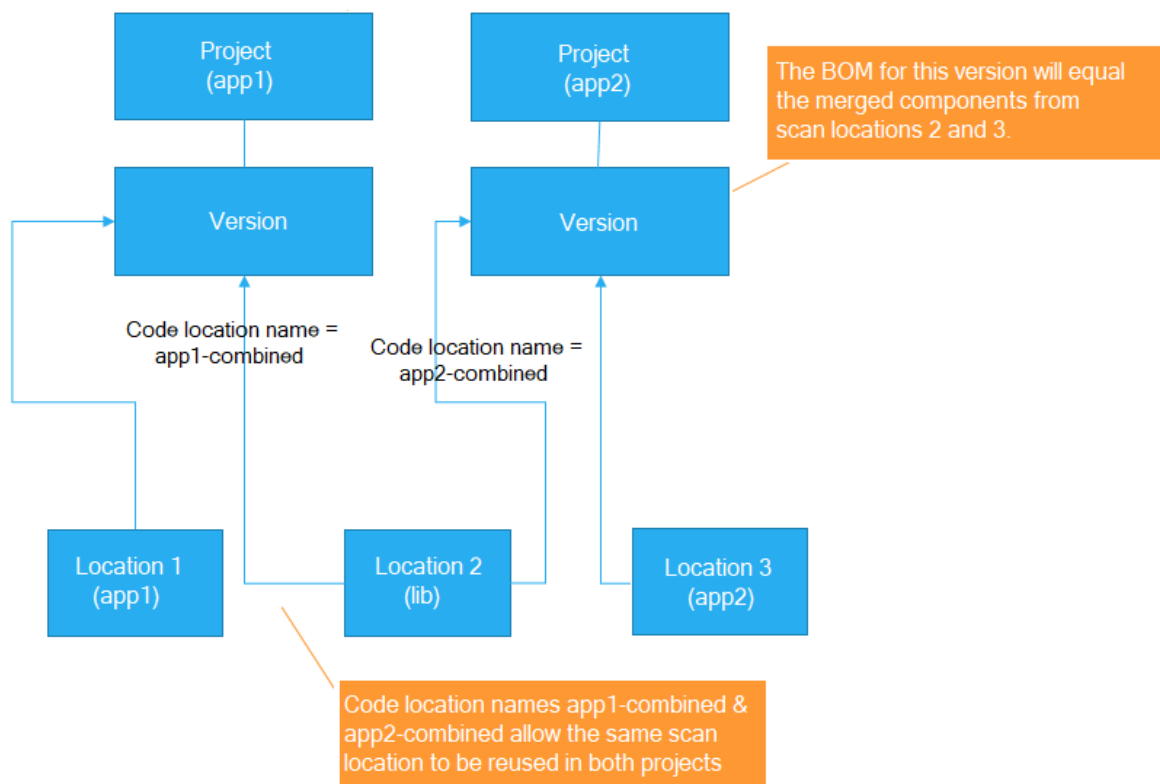
Note:

- The resulting aggregate BOM represents both the library and the application.

Note that the BOM page displays a "flat" view of components – all components found during the scans – regardless of the directory where the component was found – are listed at the same level on the BOM page.

- Use the **--detect.code.location.name** property to avoid scan name conflicts. This allows reuse of the same physical location into multiple projects. This is useful if the library is used by more than one application and you want the contents of the library reflected in all the applications that use it. In that case, scan the same library folder multiple times giving each scan a unique name and map the scan to the application that uses it.
- You must rescan to update the combined BOMs.

For example:



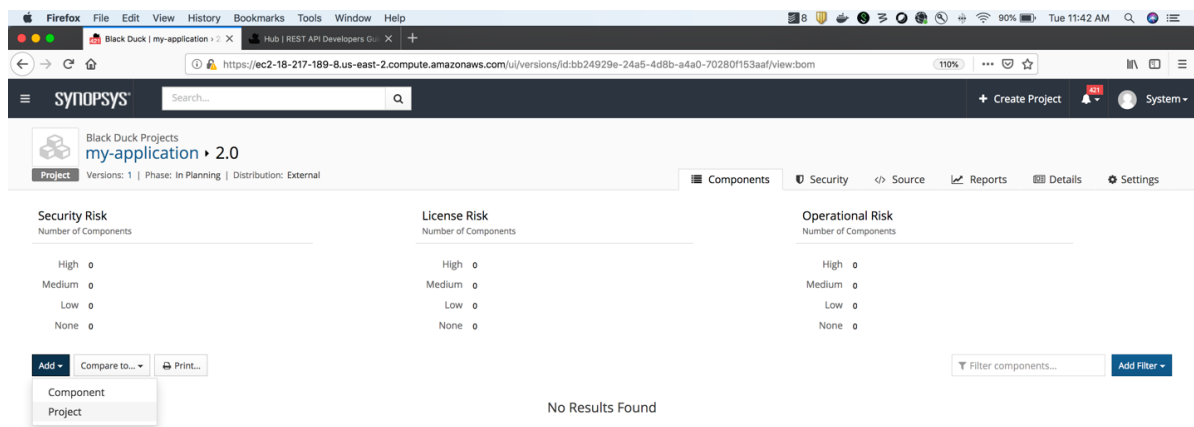
Using a project hierarchy

In this option, use a different project and version name in Synopsys Detect for each scan. In the parent BOM, add the library module project version as a subproject. In this scenario, subsequent scans/changes to the subproject are immediately reflected in the parent BOM.

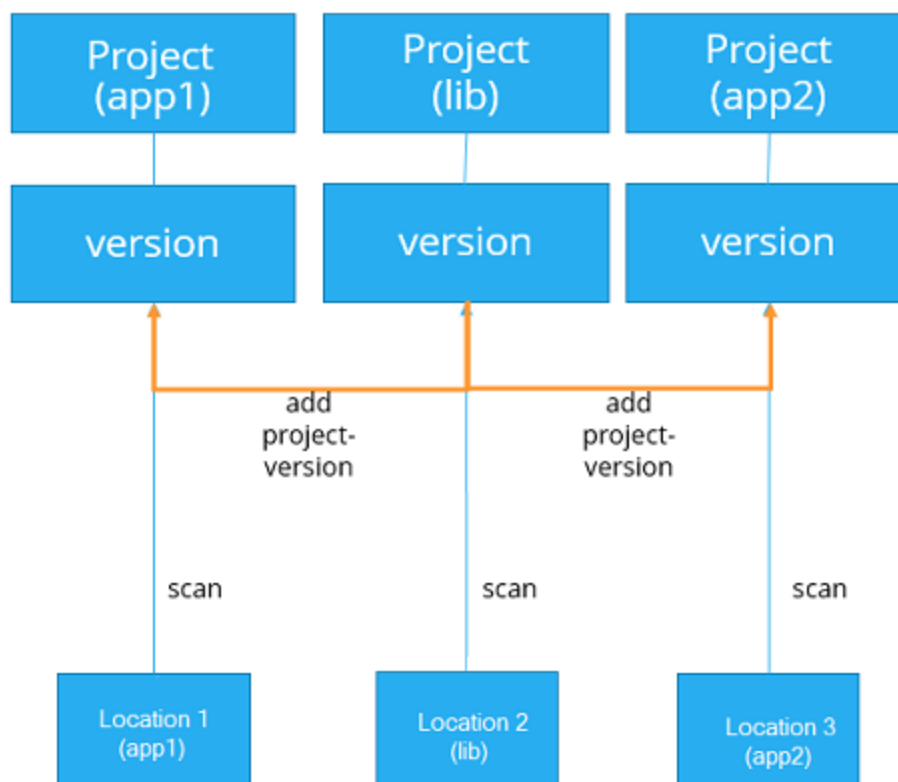
So, for instance, in our example of a library in one folder and an application in a separate folder, do the following:

1. Scan the library folder and map it to its own project (for example, my-library) and version (for example, 1.0).
2. Scan the application folder and map it to a separate project (for example, my-application) and version (for example, 2.0).
3. In the Black Duck UI, navigate to the BOM for application project (my-application) version (2.0).
4. Add the library's project (my-library) to the BOM of the application project (my-application, version 2.0): click **Add** and select **Project** from the list. The UI prompts you for the version (2.0) and after you confirm

the addition, the BOM from the library project version will be included in the application's BOM:



The diagram below depicts the case where a library is used by two applications. Project (lib) is added as a subproject to the BOM of Project (app1) BOM and/or Project (app2).



Using a project hierarchy in this way isolates the scans of the library (child project) from the applications

(parent projects) that use it. It also encapsulates the BOM for the library from the projects that use it, so you can view and interact with the library and its contents separate from the application BOM which simply includes the library's results.

Scanning when network connectivity is an issue

If network connectivity is an issue, use a store and forward strategy:

1. Use the Synopsys Detect--**blackduck.hub.offline.mode=true** property. This disables all communication with Black Duck but still generates metadata.
2. Forward and then upload to Black Duck separately.

Maintaining an open source software component-based review process

Black Duck was designed and is optimized for scanning projects, because that is where an open source software (OSS) component is being used. Components are not run in isolation in production, nor are they distributed. Instead, you run a project in production or you distribute a project that contains components.

That said, you may want to establish and maintain an OSS component-based review and approval process. To use Black Duck for such a process, do the following:

1. Scan the component or use the Black Duck UI to search for it, so that you can assess whether it complies with your policies or if it includes high risk security vulnerabilities.
2. Use Component Management to change the status of the component to either Approved, Limited Approval, Rejected, or Deprecated so you can use the component status as a condition in your policies.
3. Produce and export a project version CSV report to obtain additional details about the component. This report can also serve as an audit trail to indicate that you scanned the component and assessed the results.
4. Delete the scan and project version created in step 1, as appropriate.

When the component appears in a build or project, the component will be evaluated in accordance with your policies and, if it does not comply, a policy violation will occur, and project members will be notified.

Thereafter, if the component gets used in a build or a project that gets released, you will get a notification anytime a new vulnerability is announced, without having to rescan.

Poor scanning techniques

Including the commit ID or build ID in Black Duck version or scan names

Some customers have been reporting poor performance of their Black Duck server. There are several ways that this can happen, but for many of these customers their Black Duck server/database was filled with hundreds or even thousands of spurious versions and scans because they used either a commit ID or build ID in the project version name or scan name when they configured their scans for their automated builds. In some cases, without realizing it, they had configured a system of continually creating new versions and scans with no thought to purging old ones or had no effective automated solution for keeping the system from clogging up.

Once their system got to the point of having all those hundreds or in some case thousands of unwanted

versions or scans, cleanup becomes a huge issue since deletion is slow in an overloaded system and deleting scans/versions is a scary prospect when you are perhaps uncertain what is important to keep and what you can discard. It is much easier if you never get to this point in the first place.

This is more than a performance degradation issue. This also makes the resulting project version BOM inaccurate since the builds occur over time and the resulting BOM is the aggregation of all the scans.

Keeping a history of versions on a development branch

Some users say they want to keep a history of the versions on a development branch so that they can determine when the development team introduced an issue.

However, Black Duck was not designed to be a data warehouse, keeping all information forever. If you are trying to use the Black Duck system in this manner, your server/database will be overwhelmed with too much data. Performance will begin to severely degrade, and the system will become unusable.

If your goal is to understand forensically what has happened over time, the appropriate way to approach this is to export the scan results from the system (for example, using the Black Duck reporting database or REST APIs) to a data lake or data warehouse that is designed to accumulate large amounts of data and supports the slicing/dicing of the data. That capability is outside of the scope of what the Black Duck server was designed to do.

Chapter 2: Troubleshooting scanning issues

This chapter describes some scanning problems and provides solutions.

Accidental scan proliferation by folder paths which include build or commit ID

Suppose your build server scans the same project repeatedly. Each time it creates a build, the workspace folder where the scan occurs includes a build or commit ID.

The scan is configured to place the results into the same project version, however since a scan name was not provided, the scan client creates one. Since the scan name is derived from the full folder path, which includes the new build or commit ID, the result is a growing list of scans that all point to the same project version.

If you look at the BOM for this project version, you may not see much changing, so you may believe that everything is working as planned. However, looking at the scans associated with this project version you would see a very long list that grows with each build.

This is more than a performance degradation issue as the resulting project version BOM is inaccurate as the builds occur over time and the resulting BOM is the aggregation of all the scans.

Solution

Supply a scan name using the **--detect.code.location.name** property. Click [here](#) for more information on Synopsys Detect properties.

Accidental scan proliferation by a build server farm

If you have a build server farm creating your builds, then each time a build occurs there is potentially a new host (up to the number of hosts in the farm). If you do not supply your own scan name, Synopsys Detect derives one that includes the hostname and therefore, you will get a different scan for each host. All scans will be mapped to the same project version in Black Duck and the result is multiple redundant scans mapping to the same project version.

This is more than a performance degradation issue as the resulting project version BOM is inaccurate since the builds occur over time and the resulting BOM is the aggregation of all scans.

Solution

Supply a scan name using the **--detect.code.location.name** property. Click [here](#) for more information on Synopsys Detect properties.

Intentional scan and project version proliferation due to using the build or commit ID to retain all build versions

You may have intentionally used the build or commit ID (or some other variable) for scan names and project version names with the idea that you want to keep an audit history of all your builds. The rationale is that by keeping all this information you can, for example, compare versions of previous builds and see when a new component was introduced.

While this may sound like a good idea, in practice this will quickly clog up the system, leading to performance degradation. Furthermore, changes to open source software components are infrequent and it is far more effective to use policies to alert you immediately if a component is not acceptable.

Solution

Follow the instructions described [here](#) for setting up scans so that developers and development teams will be warned early if they have an issue they need to deal with.