[2018-10-30] Change Log

New version of CQM, Kiuwan Engine and Kiuwan Insights

Main features of this release are:

1. ***Kiuwan CQM (v1.2.20) and Engine***
   - Support for **.Net Xamarin** platform
   - Increased support for **security** in **Swift** (18 new security rules)
   - ***Performance Optimization Guide***
2. ***Insights*** - new ***Component Dependency Tree***

1. Kiuwan CQM and Engine

A ***new version of CQM*** has been released that incorporate ***new rules*** *(*as detailed below).

CQM is the default Model (i.e. a concrete set of active and pre-configured rules):

- If you are using ***CQM***, ***new rules will automatically become active*** and will be applied to new analyses.
- If you are using your own **custom model, your model remains unchaged,** but *you can modify it and activate the new rules* (in case you want to be applied to your code).

You can find new rules by comparing this release of CQM against previous version. A detailed description of the behavior of these new rules is available in rule's description.

A ***new version of Kiuwan Engine*** has been released that incorporates ***bug fixes, performance and reliability improvements in rules and parsers***.

Kiuwan Engine is the binary code executed when an analysis is run.

- ***If the engine is not blocked*** in your Kiuwan account, ***the engine will upgrade automatically*** to the last version of Kiuwan Engine once a new analysis is run
- ***If the engine is blocked***, your kiuwan ***engine will not be modified***.

Support for .Net Xamarin platform

This new Kiuwan Engine provides support for **.NET Xamarin** platform (Microsoft framework for developing multi-device mobile apps in C# for Android, iOS, MacOS and Windows mobile platforms).

Kiuwan engine is now aware of security-relevant items in Xamarin APIs, providing mappings for input elements in user-defined interfaces (via XAML in Xamarin.Forms) so they are properly considered as user-controlled input.

New Swift security rules

- OPT.SWIFT.SECURITY.PasteboardCachingLeak

- OPT.SWIFT.SECURITY.PasswordInConfigurationFile
- OPT.SWIFT.SECURITY.PotentialInfiniteLoop
- OPT.SWIFT.SECURITY.URLSchemeHijacking
- OPT.SWIFT.SECURITY.HardcodedUsernamePassword
- OPT.SWIFT.SECURITY.PlaintextStorageInACookieRule
- OPT.SWIFT.SECURITY.SerializableClassContainingSensitiveData
- OPT.SWIFT.SECURITY.ThirdPartyKeyboardAllowed
- OPT.SWIFT.SECURITY.UncheckedInputInLoopCondition
- OPT.SWIFT.SECURITY.ExecutionAfterRedirect
- OPT.SWIFT.SECURITY.SensitiveSQL
- OPT.SWIFT.SECURITY.SensitiveNoSQL
- OPT.SWIFT.SECURITY.InsecureTemporaryFile
- OPT.SWIFT.SECURITY.ConnectionStringParameterPollution
- OPT.SWIFT.SECURITY.SensitiveCoreData
- OPT.SWIFT.SECURITY.PasswordInCommentRule
- OPT.SWIFT.SECURITY.HttpParameterPollutionRule
- OPT.SWIFT.SECURITY.NoSQLInjection

Performance Optimization Guide

If you need to optimize the performance of your local analyses, please read Performance Optimization Guide , a practical how-to ***guide to optimize the performance and memory consumption of Kiuwan local Analyzer***.

2. Insights - Component Dependency Tree

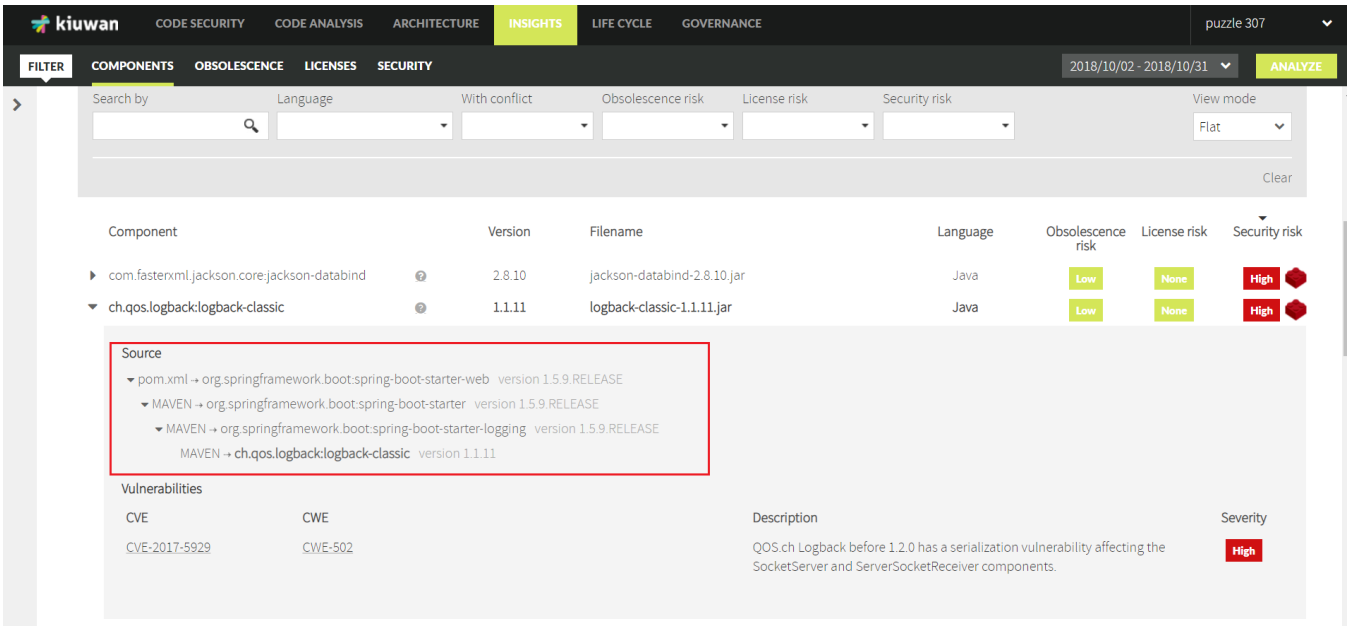
Kiuwan ***Insights*** incorporates a new view to better understand the external dependencies of your app.

Insights' ***Component tab*** let's you now to select a ***Flat*** or **Tree** view of the external components of your application.

By selecting ***Flat*** view, you will be able to see the *full list of external dependencies* (as before), but *opening a component you will see the "**source**" of the dependency*, i.e. what's the path that Insights has followed to discover your component.

This Flat view will help you to identify the origin (source) of your dependencies.

For example, in this image you can view that the discovered component (*ch.qos.logback:logback-classic*) has its origin in a dependency within your *pom.xml* file which, in turn, contains other transitive dependencies (discovered in Maven repository) until the selected component.



The Flat view, then, displays the whole list of components, directly or indirectly used by your application.

Butm what if you need to know your "direct" dependencies? That is, to know what components are directly used (called) by your application, and be able to drill-down to view the components indirectly used by your application.

The ***Tree*** view allows to view your *"**direct**" dependencies as well as the "**indirect**" dependencies*.

The example below shows how the directly-used component "*org.springframework.boot:spring-boot-starter-web*" (Level 1) uses directly other components (level 2) which, in turn, uses other components (Level 3), and so on.