The Meterian Client

Integrate Meterian security and stability scan to your CI/CD pipeline.

Author:	Bruno Bossola
Date:	30 March 2020
Version:	1.2.9
Client version:	1.2.9

Table of Contents

Introduction

The Meterian client is used to securely execute a scan of your project without the need of providing access to your local source control system. If you want to do an occasional scan on your private project or you want to add the scan to your CI/CD pipeline the client is the most obvious choice.

With the client all the information related to your project will be securely stored and linked to your account: no data will be publicly available at any point in time. To access historical information, status and reports you will always have to log in first.

You will be requested to login to access any private report.

How does the client work?

The client will first authenticate and validate your account: you can review the details of the operation in the related section.

You will need to run the client in the folder where your project is located. The client will scan the folder searching for the files where your dependencies are listed or it will invoke the relevant tool to generate them.

Once the list of dependencies is available it will be then uploaded securely to the Meterian servers, where the analysis will be conducted. Please note that if you decided to run the analysis remotely, which is available for some platforms, then all the build files will be uploaded instead (see the related chapter for more information).

The client will be reporting the progress of the operation in its various stages, and at the end, it will print out the scores , with a link to securely access the full report.

Multiple analyser can be automatically triggered: you can get manual control over this. see the section “Forcing or avoiding specific scans”.

Java, Kotlin, Scala (Maven, Gradle, Sbt)

The Java+ analyser is triggered when one of the relevant manifest files (“pom.xml”, “build.gradle”, “build.sbt”) is found in the root of the project.

When using Maven. Gradle or Sbt, the tool will be actually invoked to generate the dependencies tree, as it's the only one who actually knows the components that, at the end of the day, will be really shipped. If you rely solely on the manifest files you will have to replicate what the tool is doing, which is really unreliable. The logic used is frequently just too convoluted, and only if you ask the tool to generate the dependencies tree you will be obtaining the right one.

For that reason, your project has to build successfully before using Meterian. If this condition is not met then you won't be able to use this tool, so please make sure the project you want to attack is compiling correctly.

Ruby

The Ruby analyser is triggered when a “gemfile.lock” is found in the root of the project.

When operating on a Ruby project the analyser will directly load the dependencies from the lock file, and it may require intervention from the tool (bundler). This analyser cannot work remotely.

.NET

The .NET analyser is triggered by different conditions:

1. a “packages.config” is found in the root folder
2. a “packages” subfolder is found in the root folder
3. a “.csproj” file is found in the root folder

In cases 1 and 2 the client will just collect the dependencies from the locations. In this case it's important that the information is up to date, otherwise, the client will collect stale data that will most probably lead to a wrong analysis.

In case 3 the client will trigger a “dotnet restore” and then collect the dependencies. Please note that, in this situation, the restore command has to succeed. It's important to be aware that in this situation multiple projects may be present under the same main root project folder, so you may need to instruct the client to run multiple times. This is not happening automatically because, at this time, there's no clear way to distinguish test subprojects (which are not shipped) from the non-test elements.

This analyser cannot work remotely

NodeJS

The NodeJS analyser is triggered by two possible conditions:

1. a “package-lock.json” is found in the root folder
2. a “packages.json” is found in the root folder

In case 1 the client will trust the contents of the package-lock.json and will collect the dependencies directly from there. In this case it's important that the information is up to date, otherwise, the client will collect stale data that will most probably lead to a wrong analysis: if you are in doubt just delete the file and the client will fall back to the second option.

In case 2 the client will use the local npm tool: it will first execute an “install” command, to make sure that the dependencies are properly present, and then it will ask npm to generate the dependency tree. For this reason, the project has to build successfully

Vanilla JavaScript (websites)

In case of a JavaScript scan, Meterian will examine all the files that may contain JavaScript code (HTML, JS, PHP and so on) and it will then proceed to send the results to the server, which will provide the final analysis and proceed to update the report. This analyser cannot work remotely Please note that the Vanilla JavaScript is disabled by default, see the section “Forcing or avoiding specific scans” for more information.

PHP

The PHP analyser is triggered when a “composer.json” or “composer.lock” is found in the root of the project.

When operating on a PHP project the analyser will directly load the dependencies from the lock file. If a “composer.json” is found but not the lock, this will be generated using “composer install”. This analyser cannot work remotely.

Python

The Python analyser is triggered when a “Pipfile”, “Pipfile.lock” or “requirements.txt” is found in the root of the project.

When operating on a Python project the analyser will either load the dependencies from the lock file, if available, or it will be using Pipenv in order to generate a new one, from the Pipfile or the requirements file. This analyser cannot work remotely.

Golang

The Go analyser is triggered when a “go.mod” is found in the root of the project.

When operating on a Go project the analyser will use the “go” command in order to load the dependencies. This analyser cannot work remotely.

How does the client authenticate me?

The client can be used in interactive and non-interactive mode, depending on the operation being monitored by a human or not, and of course by your choice.

In interactive mode (which is the default) the client will check if it has a valid authorisation and, if not, will open for you a browser window so that you can log in with your credentials. After a successful login, the client will securely store an authorisation token in your home folder, so that you will not need to log in for a certain amount of time. Depending on your configuration, this can vary from hours to day, but by default, a token obtained in this way will last 24 hours.

In non-interactive mode you will need to generate a token on your administrator page and then store it where you prefer on the local file system. When you will launch the client you will have to specify where the token is stored so that the client can authenticate on your behalf. For detailed information please see the instructions section in your administration page.

System requirements

The client is a Java native application written in Java8, so you will need to have a suitable JRE to run it. As it's written in Java it will run on any platform where a Java8 runtime is accessible.
To run the analysis locally the client will need to have access to your local Maven/Gradle/etc. installation. You may also need the JAVA_HOME variable to point to a proper JDK home (this is, for example, a Gradle requirement). If you are using Maven, please set the MAVEN_HOME variable, and if you are using Gradle, please set the GRADLE_HOME variable.

If you are running the analysis over a NodeJS/NPM project and run the client locally, it will need to be running in a working NodeJS/NPM installation compatible with your project. In general, the same applies to all the other platforms.

As the analysis on the dependencies is performed remotely, the client will need to have access to a working internet connection, capable of reaching directly using `https` the domain `*.meterian.com`. In case you need to use proxies, see the section “Connecting through a proxy”.

We plan to release the client also in form of a native application or a Python client: if you are interested in these development please contact our support email to get access to our early access program.

How do I get the client?

Please login from our main page and follow the instructions provided in the admin pages, there is also a permalink you can use for that purpose.

What if I need a previous release?

Please contact our support email.

Using the the client from your command line

From time to time you may want to do the occasional check on your project, or you are just evaluating a project and you want to check how it scores. In that case you may want to run the client in **interactive mode:** the client will assume that a person is present and ready to react to any issue. The quite common case is to provide a login to a Meterian account: as your results will be computed in the cloud, you want to maintain those completely private, and for that reason you will need to provide the client an authorisation token linked to your account.

By default the client runs in interactive mode.

Authorisation in interactive mode

In interactive mode the client will check if you have a valid authorisation and, if not, will open for you a browser window so that you can log in with your credentials. After a successful login, the client will securely store an authorisation token in your home folder, so that you will not need to log in for a certain amount of time. Depending on your configuration, this can vary from hours to a day, but by default, a token obtained in this way it will last 24 hours.

Interrupting the client

The client stores the state of its progress in a hidden folder under your home, and can be interrupted at any time. When you are ready to resume it, you just need to launch it in the same folder and it will start exactly from the point where it stopped: the state of the operation is preserved on the Meterian servers.

Please note that the client will refuse to execute another analysis if the previous one is still in progress: you can override this behaviour passing the **`--clean`** parameter to the command line, which will remove any pending build, starting a new one.

Example: running the client in interactive mode

Let's run this example with a simple open source project you can find on GitHub; for this exercise let's assume also you already downloaded and stored the client under `~/apps/Meterian-cli.jar`

First, let's clone the project from GitHub, for example Eclipse Vert.x:

```
$ git clone git@github.com:eclipse/vert.x.git
Cloning into 'vert.x'...
remote: Counting objects: 110122, done.
remote: Compressing objects: 100% (86/86), done.
remote: Total 110122 (delta 81), reused 145 (delta 61), pack-reused 109941
Receiving objects: 100% (110122/110122), 94.48 MiB | 1.20 MiB/s, done.
Resolving deltas: 100% (57513/57513), done.
Checking connectivity... done.
```

Now, let's move into the folder and launch the client with the default configuration:

```
$ cd vert.x
$ java -jar ~/apps/meterian-cli.jar

Meterian Client v0.1
- running locally: yes
- interactive mode: on
- working on folder: /tmp/vert.x

Checking folder...
Folder /tmp/vert.x contains a viable project!
Authorizing the client...
I cannot find a valid authorization token: I will open the browser so
obtain one
Please login as usual with your selected credentials
Created new window in existing browser session.
```

As you can see the client introduced itself and then, as it's the first time it is launched, is opens a new browser window for you to login with your credentials and authorise it.
After a successful login the client will proceed to collect the dependencies, using your local Maven installation (you can also move this task server side, but it's a good idea to execute this step on your local environment). After collecting the dependencies it will upload them to the Meterian servers, where they will be analysed. The client will also output information about the status of the process.

```
Client successfully authorized

Loading build status...
No build running found!

Requesting build...
Build allowed

Running maven locally...
- maven: loading dependency tree...
- maven: dependencies generated...
Execution successful!

Uploading dependencies information - 63 found...
Done!
```

```
Starting build...
Current status: in preparation
Current status: "cleaning" - last updated at "2017-07-01T16:09:02.189"
```

When the work on the server is finished the client will emit the result of the analysis. The client will also output information about the status of the process, and provide you a link to access the report.

```
Final results:
- report:         "OK"
- security:       "0"
- stability:      "88"
- timestamp:      "2017-07-01 16:09:02"

Full report available at:
https://www.meterian.com/projects.html?pid=2fae2c4c-e22b-445c-a2cb-
cc7796d6c579&branch=master&login=true
```

Opening the link will force you through a login page: please make sure to use a set of credentials associated to your account to access, otherwise you will not be able to see the report.

Using the the client in your CI/CD pipeline

When you run the client as part of you CI/CD pipeline everything hopefully will be automated, and for that reason you want the client to work without the need of user intervention. You can achieve that using the **non-interactive** mode of the client that you can activate specifying the command line `--interactive=false` argument.

In non-interactive mode the client will assume it's already authorised (see the details in the following section) and will fail if such authorisation is missing. You will be able to control the exit code of the client using specific arguments, so that you can quickly block your pipeline if the libraries are not up to your required standards.

In case you decide to do so, you can split the client work in two phases: the first one will just kick the analysis, returning the control to the calling shell so that you can run your build in parallel. At the end you can run the client again: it will collect the results from the server and report back accordingly.

Authorisation in non-interactive mode

In order for the authorisation to work in non-interactive mode you will need to generate a token on your administrator page, then download and store it where you prefer on the local file system. When you will launch the client you will have to specify where the token is stored so that the client can authenticate in behalf of you, using the `--auth-file=/path-to-file` argument:
as soon as the token is not revoked or deleted the client will be able to function without any interaction with the user and all the operation will be linked to your account.

Providing the project branch

In modern software development branches are widely used, and Meterian provides a report for each of them. If you use Git, the client usually detects automatically some basic information, including the current branch, but when this is not possible you can simply specify on the command line the parameter `--project-branch=name-of-your-branch` to enforce the branch name. A common scenario is when you are running within a CI system, which may not checkout the whole branch but just detach the head at a certain commit: please refer to the documentation of your CI system. You can also provide the project url (`--project-url`) and the current commit (`--project-commit`) if you want, but usually this information are automatically detected by the client.

Concurrent mode

If you are running your build process on the same CI machine, sharing between different builds the same environment, you will need to run the client in concurrent mode using the command line parameter `--concurrent-mode.` In this mode the client will allow different builds to run concurrently on the same environment. Please note that in concurrent mode you cannot execute two phases builds, and that concurrent mode is enabled by default

Two phases build

Sometimes, especially if you run server side analysis, the process can take several minutes: in that situation you may want to start the process, go back to your build and collect the results later. This can be done passing the **`--start-only`** argument to the client. In that situation the client will collect the required data, send them to the server and return to the shell. You can then run your build, executing it in parallel to the work done on the Meterian servers. At the end of your build you just need to launch again the client with the same configuration, removing the **`--start-only`** argument: it will pick up the same analysis again, contact the server and return the results. Of course, if the server is still not ready, you may need to wait until everything is ready, but the client will report about the progress.

In concurrent mode the two phases build will not work.

Controlling the exit code

Specific arguments are at your disposal to control the exit code of the client based on the score, `--`**`min-security`** and **`--min-stability`** (plus `--`**`min-licensing`** if the feature is enabled on your account). These are the minimal scores: if not met, the build will have a positive exit code , which will be reported as a failure to the shell and will, most probably, stop your pipeline to progress. In case of error the code will be calculated using a bitmask over the exit code: +1 for a fail on the security score, +2 for a fail on the stability score, +4 for a fail on the licensing score.

The default values for these scores are **`90`** for security and **`80`** for stability

Example: running the client in non-interactive mode

Let's run the previous example, but now in non-interactive mode. First of all you will need to generate a token on the admin pages, download it and save it a well know location on the filesystem of the machine you intend to use.

Let's assume you stored the token under `~/token.json`, the is available at `~/apps/meterian-cli.jar,` and your shell current working directory is the one of the project. You can now launch the client:

```
$ cd vert.x
$ java -jar ~/apps/meterian-cli.jar --interactive=false \
--auth-file=~/token.json

Meterian Client v0.1
- running locally: yes
- interactive mode: off
- working on folder: /tmp/vert.x


[...]
```

The client will continue as usual, and if the token is valid it will proceed as in the interactive scenario.

Sample commands for common CI systems

Jenkins / Hudson (shared builder):

```
java -jar ~/meterian-cli.jar --auth-file=~/meterian-cli-auth.json
--interactive=false --min-security=95 --min-stability=95 --project-
branch=${GIT_BRANCH#*/} --project-commit=${GIT_COMMIT} --concurrent-mode
```

The settings we use:

- **`--min-security=95 –min-stability=95`**
  We will fail the build if any of the two scores goes below 95
- **`--auth-file=~/meterian-cli-auth.json --interactive:false`**
  We are running in non-interactive mode specifying an authorisation file
- **`--project-branch=${GIT_BRANCH#*/} --project-commit=${GIT_COMMIT}`**
  We are using the environment variables that are provided to specify the branch and the commit id so that they will appear correctly in the report
- **`--concurrent-mode`**
  We are sharing the slave across different and potentially concurrent jobs we request to run in concurrent mode, so allowing multiple builds to happen at the same time.

General client operation

You will find in this chapter some general information about the client operation, which can be applied to both operational modes (interactive and non-interactive).

Running the analysis remotely

The default operation mode of the client is running the dependency discovery locally, using the exact same environment used in your build. This is the default, and it's a very good idea as it forces the client to use the same environment, configuration and setup that the build tool (Maven, Gradle, etc.) is using for his own build. However it's possible to have this step executed remotely, on the Meterian servers, passing the **`--local=false`** argument to the client. In this situation the client will upload all your build files (`build.xml`, `build.gradle`, etc.) to the Meterian servers, as a necessary mean to compute your project's library dependencies. Please be sure that such files will be stay on the server only for the time required for the computation and will be removed securely when no longer necessary. Please also note that at the moment remote scanning is possible only for Java and NodeJS.

Interrupting the client

The client stores the state of its progress in a hidden folder under your home, and can be interrupted at any time. When you are ready to resume it, you just need to launch it in the same folder and it will start exactly from the point where it was stopped. If you want to systematically exploit this behaviour to run the build in two separate stages please refer to chapter titled “Two phases build”.

Generating a report on the file system

A report can be generated on the file system using the parameter **`--report-file=`**`/path/to/file`. Please note that such file is just a “pointer” to the online report, so you will need internet access to view it using your browser.

A JSON formatted report can also be generated, assuming your company is entitled of an “enterprise” plan. using the parameter **`--report-json=`**`/path/to/file`.

A PDF formatted report can also be generated, assuming your company is entitled of an “enterprise” plan. using the parameter **`--report-pdf=`**`/path/to/file`.

Connecting through a proxy

The client can work behind a proxy, which is a common situation in large enterprises. The client will read the standard variable “http_proxy” in order to detect such configuration automatically. The configuration however can also be done through these system properties:

`-Dhttp.proxy.host=<host>`

`-Dhttp.proxy.port=<port>`

`-Dhttp.proxy.user=<username>`

`-Dhttp.proxy.pass=<password>`

Both “host” and “port” properties need to be defined in order for the proxy to be used. When that happens, the client will echo to you that a proxy is being used while he tries to authorise itself, in a format similar to this one:

```
$ java -Dhttp.proxy.host=proxy.acme.org -Dhttp.proxy.port=3128 -jar
~/apps/meterian-cli.jar -

Meterian Client v0.4.6
All rights reserved
[…]

Authorizing the client...
Using http proxy: http://proxy.acme.org:3128
[…]
```

Furthermore, when you launch the client with the “--help” parameter it will also test the connectivity to the servers, as in this example:

```
$ java -jar ~/apps/meterian-cli.jar --help

Meterian Client v1.2.6, build 09ce5de-283
All rights reserved

 --help                 Displays this help end exits(0)
[...]

Using http proxy http://proxy.acme.org:3128
Using authentication for proxy username:***

Meterian servers are reachable from this system
```

You also have a couple of fine grained controls over your http stack, which we honestly think will be rarely used but we list here as a matter of completeness, with their default:

`-Dhttp.connect.timeout.millis=9000`

The timeout in milliseconds to estabilish a connection over http/https

`-Dhttp.socket.timeout.millis=8000`

The timeout in milliseconds before declaring a connection over http/https dead

Forcing or avoiding specific scans

The current client will automatically select the correct set of scanners based on the content of your project folder. You can, however, force or disable a scanner using these parameters:

- **`--scan-java=false`** will disable the Java scanner
- **`--scan-nodejs=false`** will disable the NodeJS scanner
- **`--scan-javascript=true`** will enable the Javascript/Web scanner (disabled by default)
- **`--scan-ruby=false`** will disable the Ruby scanner
- **`--scan-dotnet=false`** will disable the Dotnet scanner
- **`--scan-scala=false`** will disable the Scala scanner
- **`--scan-php=false`** will disable the PHP scanner
- **`--scan-python=false`** will disable the Python scanner
- **`--scan-golang=false`** will disable the Golang scanner

A value of “true” will force the scanner to run (and fail if conditions are not met).

Self downloading and updating the client

The client can be easily self downloaded (and updated) with a two liner script using **curl** like this one:

```
#!/bin/sh -
curl -s -o "~/meterian-cli.jar" -z "~/meterian-cli.jar"
"https://www.meterian.com/downloads/meterian-cli.jar"
java -jar ~/meterian-cli.jar $*
```

If **wget** has to be preferred, we explicitly suggest to use bash and pushd/popd like in this example

```
#!/bin/bash
pushd ~ > /dev/null
wget -q -N https://www.meterian.io/downloads/meterian-cli.jar > /dev/null
popd > /dev/null
java -jar ~/meterian-cli.jar $*
```

Maven specific controls

System properties can be used to trick some specific Maven behaviours when launching the client:

`-Dmaven.profile=<profile_name>`

The profile that Maven will use while executing the scan. Please note that this will also cause an automatic change in the branch name, that will be now called *branch-profile_name*

`-Dmaven.binary=`

Allows to specify the exact location of the Maven binary. It's not advisable however to use this property: please use the standard MAVEN_HOME variable to specify where Maven is located.

Dotnet specific controls

System properties can be used to avoid the execution of the “dotnet restore” command from the dotnet analysis specifying:

`-Ddotnet.restore=false`