Summer Student Report

Static Code Analysis with Gitlab-CI

Author:
Szymon Tomasz Datko

Supervisors:
Stefan Lueders
Hannah Short

Summer Student Programme 2016

Geneva, 26th of August 2016

I would like to thank here both my supervisors and the whole CERN Computer Security Team for their help and patience.

This internship was not only the best and most amazing experience in my life, but also a unique opportunity to feel like a serious member of the world-famous science institute.

For a long time I will remember all the security meetings, technical discussions about tracking security issues, cutting edge lectures on Physics and Computer Science, a lot of astonishing experiments during the workshops and many fascinating people from all over the world.

Saying *“good bye”* was never so hard...

Table of Contents

Abstract

Static Code Analysis is a simple but efficient way to ensure that application's source code is free from known flaws and security vulnerabilities. Although such analysis tools are often coming with more advanced code editors, there are a lot of people who prefer less complicated environments.

The easiest solution would involve education – where to get and how to use the aforementioned tools. However, counting on the manual usage of such tools still does not guarantee their actual usage. On the other hand, reducing the required effort, according to the idea "setup once, use anytime without sweat" seems like a more promising approach.

In this paper, the approach to automate code scanning, within the existing CERN's Gitlab installation, is described. For realization of that project, the Gitlab-CI service (the "CI" stands for "Continuous Integration"), with Docker assistance, was employed to provide a variety of static code analysers for different programming languages. This document covers the general system architecture as well as introduces its configuration and usage examples.

1. Introduction

Nowadays the presence of computers in our life is so common, that the ability of programming may be considered as one of the essential skills. One may even barely find any field of science/study that does not involve parts of computer science in any way. Multi-dimensional calculations, simulations and processing of huge amount of data – this is how the science looks today! And it is even required to use more and more complex solutions to achieve desired results!

However, there is always the other side of the coin. Among all the benefits from introducing computer science, there are few problems that need to be taken in account. First of all – how you can be sure that your code is good? Usage of the programming language usually does not make someone an expert in programming. In the ideal world, you would have spent at least few months learning about the basics and fundamental rules. Do you have a time for that?

The second thing is – how you can be sure that your code is secure? Every day there are new vulnerabilities found in the software we are using. Sometimes the issues are even discovered in the very fundamental libraries that are used by other applications. In a perfect world, you would check and follow the news about cyber security every day. Can you afford it?

There is nothing wrong about being more concerned about the goal – at least as long as there is a willingness to do things right. Would not it just be nice if there would be someone that would check, tell and warn about all the flaws? And how about not even asking for this every time? Is it even possible?

Introducing additional, automatic Static Code Analysis inside a code repository is a very simple and efficient way to ensure that your code is clean from known security issues and bad practices. It is especially useful when working in groups or big teams, because it allows one to focus more on the task, rather than on which tools everyone should use and how. Since you do not need to prepare a testing environment every time, you are saving time. Also, you are free from platform-dependent excuses – say “good bye!” to phrases like “hmm, it was working fine / there were no errors on my desktop”.

Long story short – it is simply worth to introduce Continuous Integration with Static Code Analysis into projects! On the next few pages the elementary knowledge about the infrastructure will be introduced with example use cases and description of steps required to start.

2. System components

In this chapter the used tools and general system architecture is briefly described.

For a short overview of the system, you can referrer to the recorded presentation, from the Student Session 2016, about automatization of code scanning with Gitlab-CI, which is available under the following web page: `https://cds.cern.ch/record/2206413`.

2.1. Static Code Analyzers

As the name suggests, Static Code Analyzers are tools that are checking the source code for known security issues, bad practices and general mistakes. They are commonly present in some more advanced code editors or development environments, where they appear as (usually) yellow or red underscores, saying that in some particular line/part of code there is a flaw.

The word “static” refers here to a fact that the analysis is being done without actual execution of the program. However, although the execution of program is not necessary, there are some tools that may require compilation of the code, because these tools prefer to check the object files, rather than the raw source code.

In general, Code Analyzers are working in two basic ways. One approach is simple, maybe even very naive, looking through the code for some unsafe function calls or usage of deprecated libraries, as well as checking if all referred variables/functions/objects are defined somewhere and accessible.

The second approach is related to a more sophisticated analysis of the structure of the code. This allows to detect some more complex flaws, like unreachable code, infinite loops, unpredicted boundary conditions, possible memory leaks or uncaught exceptions and some ambiguous, non-optimal expressions.

Apart from all technical details, there is a lot of such applications and solutions ready to use. This includes both open source / free tools, as well as expensive commercial suites; single-language dedicated checkers, as well as various multi-language analyzers.

The list of code analysis tools, concerned in this project, originates from the CERN Computer Security Team’s recommendations that involves well-known applications, like PyLint, PMD or RATS. The full list can be found under the following web page:
`https://cern.ch/security/recommendations/en/code_tools.shtml`

2.2. Gitlab and Gitlab-CI

Gitlab[1] is a web-based manager platform for git repositories – a simple, but powerful version controlling system created by Linus Torvalds to make maintenance of changes in Linux source code easier. It is written in Ruby and widely used by many world-wide institutes, organizations and corporations to manage their private repositories. It is often called as “Github, that you can set on your own computer” – although it is only partially true, it is a good analogy to imagine. One may just call the Gitlab as a web frontend for git, but this would be a big misrepresentation, as it offers much more features – like a code reviewing toolkit, issue reporting system, wiki-like document suite and automation engine.

Gitlab-CI is a part of Gitlab since version 8.0, that allows one to introduce **Continuous Integration** in a very easy way. This means, an automatization of some periodic tasks and executing them each time the code changes. In general, these tasks may be divided into three main stages:

- Building – code compilation, for example
- Testing – running unit tests or some code analyzers
- Deploying – sending a program to package repository

From this project’s point of view, only the testing stage is considered.

Gitlab-CI offers a git-based Continuous Integration (commonly abbreviated with “CI”) with execution of the CI jobs each time when there is new commit pushed into repository. What is very special about it, is not only a great integration with Gitlab itself, but especially a simple and intuitive configuration, that is done per-repository by adding one single file to the project. That single file, named .gitlab-ci.yml, contains the definition of CI jobs to execute and some executor-dependent settings. It shall be placed in the repository’s root directory.

For CI job execution, Gitlab-CI uses a dedicated service, called Gitlab-CI-Multi-Runner. It is recommended to install this service on a separate machine and to associate the runner service with main Gitlab installation using token-like authorization. This service can execute the defined CI jobs directly on a local host where it is installed, or on another host accessible through ssh connection – although it is not a very secure option and therefore not the recommended way for production environments.

An alternative is to use one of a few backends supported by runner service. These involve usage of virtual machines, where VirtualBox[2] and Parallels[3] software are supported; or containers, where Docker[4] only is currently supported. The last mentioned was chosen as it offers the best compromise between speed of execution and node’s safety.

1 – https://gitlab.com/
2 – https://www.virtualbox.org/
3 – http://www.parallels.com/
4 – https://www.docker.com/

The Gitlab installation[5], with dozens of runner nodes, was available and maintained at CERN by the git administrators team.

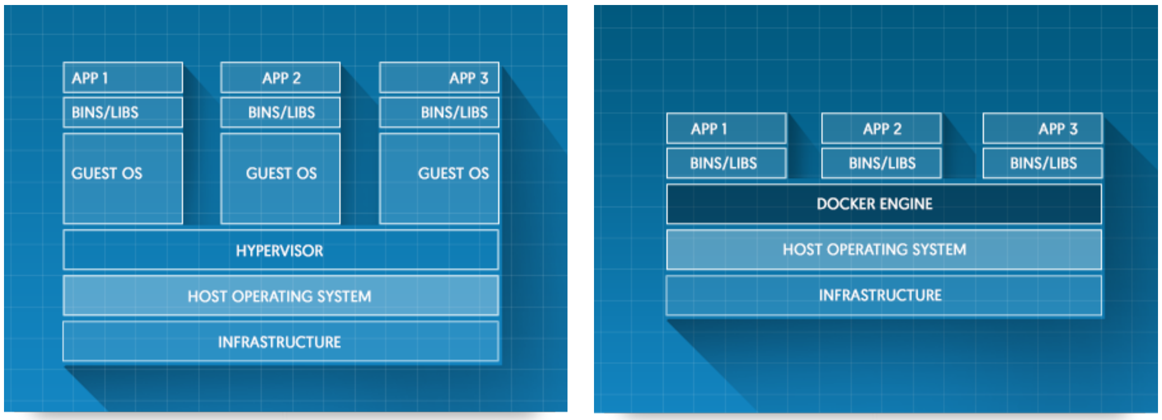
2.3. Docker

Docker[4] is an open source platform that automates deployment of independent (isolated) runtime environments, called containers, inside a Linux host. A container is just a collection of software with all dependencies (libraries and other applications) necessary to run the desired software. From a big picture, it looks like a separated filesystem, that co-exists alongside the original host, similarly to a filesystem of a virtual machine.

The analogy to virtual machines is actually quite good. The main difference is just a technical detail and relates to the fact that single virtual machine simulates also the hardware completely and allows one – more or less to launch completely different operating systems inside; whereas a container uses the original host's hardware and kernel. The Docker engine uses the kernel's namespaces mechanism to separate the processes, filesystem and network traffic between original host and launched containers.

The result is a much greater lightweight of the whole platform due to much smaller overhead. This comes, however, with a price of one smaller limitation: as the original system's kernel is being used, only the Linux-like filesystem with its applications can be launched inside a container.

Such filesystems, ready to use, are distributed in forms of files, called images, similarly to virtual machines. One of the most convenient Docker's feature is the ability to download (pull) new images on demand from repositories, called registries. The default public registry is located under `https://hub.docker.com/explore/`.



Picture 1. The comparison between virtual machine and container architecture
Source: `https://www.docker.com/what-docker`

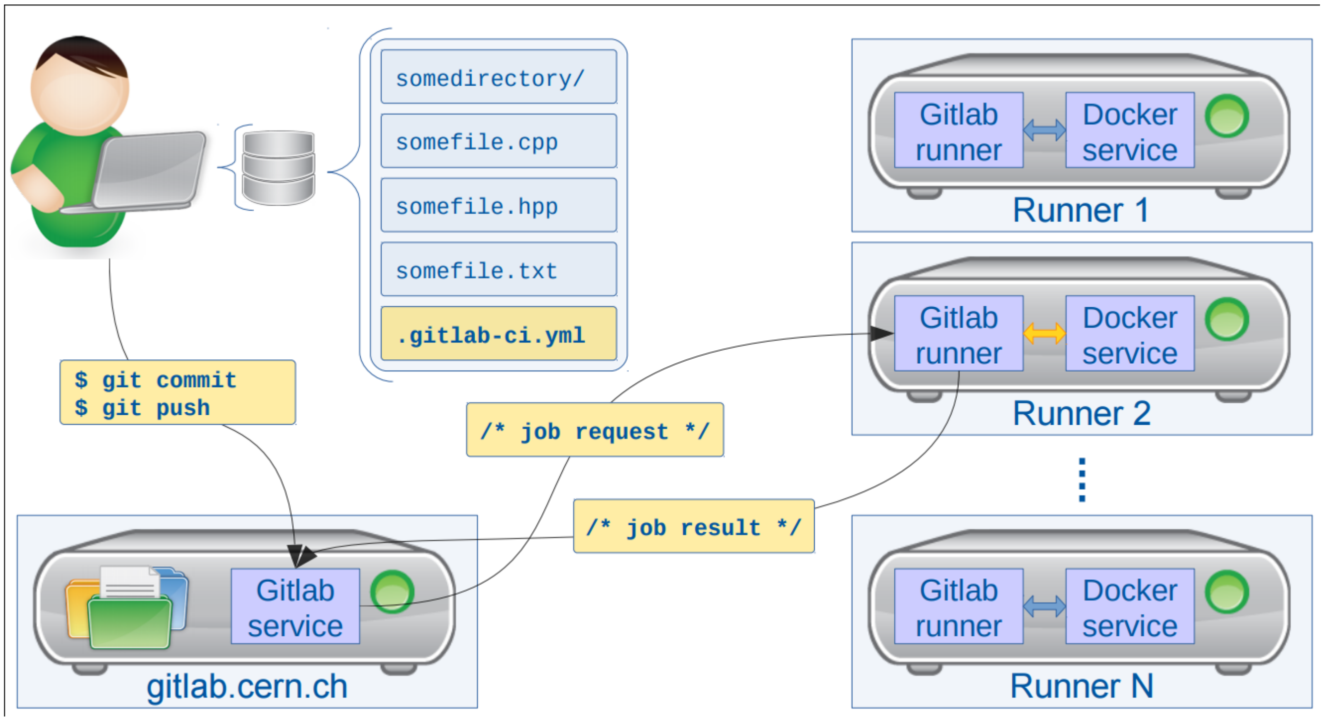
5 – https://gitlab.cern.ch/

2.4. Architecture Overview

On picture 2, the schematic overview of the used architecture is presented.

There are three basic entities in the system:

- User – with a computer and clone of one repository
- Gitlab server – a host, where all the repositories are stored
- Runner node – a special host, capable of running CI jobs



Picture 2. Used architecture overview

Source: `(own creation)`

When a new push is done into the repository, the Gitlab service, present on the main Gitlab server, catches that event and triggers a new CI request to one of the randomly chosen runner node. The Gitlab subservice, present there, will execute this request using a configured executor – Docker, as shown above.

The Docker service on the runner node deploys a container from an image prepared by the CERN Computer Security Team, named 'Security-Services/Code-Checking' and which is containing all necessary code analyzers. This image is pulled by Docker service from CERN's registry.

After executing the CI request, the result is being sent back to the main Gitlab server, where any authorized user may check it using a web browser, under the Pipelines tab in the project/repository's page in the Gitlab web interface.

3. Configuration

This chapter contains the description about content of the **.gitlab-ci.yml** configuration file. As explained in the previous chapter, it is used to set up the Continuous Integration in a repository. The next few pages explain this file's syntax basics and most commonly used keywords. At the end, the example file is presented.

3.1. YAML Syntax

The syntax of the YAML document is pretty intuitive and provided in a human-readable text format with a basic syntax like ~'`<keyword> <colon> <value>`'. It was designed for clear document representation and easy parsing with programming languages. As a value, one may specify:

- just a single string/number

```
keyword: "value"
```

- a list of values (marked by '`-`' character)

```
keyword:
    - "value1"
    - "value2"
    - "value3"
```

- a set of keywords (with further values)

```
keyword1:
    keyword2: "value"
    keyword3:
        - "value1"
        - "value2"
        - "value3"
```

Please note that indentations are obligatory, as they are marking sections. Also, comments (beginning with the `#` character), definitions of own complex datatypes (using the `!` operator) and references mechanism (marked by the `&` and `*` characters) are supported in the YAML language.

Although YAML offers many advanced features that one may use in programming, Gitlab-CI uses only the basic scheme with few predefined keywords to define CI jobs. The next subsection describes some of these keywords.

3.2. Important Keywords

Below, the most common keywords from **.gitlab-ci.yml** file are described.

3.2.1. Image Specification

The `image` keyword is used to specify the target Docker image. For the purposes of Static Code Analysis, described in this paper, the image Security-Services/Code-Checking should be used. A proper image specification should look like this:

```
image: docker.cern.ch/security-services/code-checking:latest
```

Please, note that the image can be specified globally – for all jobs in configuration file; or for each particular job. In this paper only the first option is considered. However, it may be useful for power users to use few smaller images, than one big.

3.2.2. Before/After Tasks

Sometimes you may need to perform extra preparations before or after running your jobs. The `before_script` and `after_script` keywords can be used for that. Such actions can be also defined globally and/or for each single job definition. An argument is the list of Linux shell commands to execute in container.

```
before_script:
    - echo “New job started”

after_script:
    - echo “Job just finished”
    - ls -la .
```

3.2.3. Jobs Definitions

Everything that starts without indentation and is not a special keyword (like `image`) is considered by Gitlab-CI as a new job definition. For each job definition, two keywords are obligatory:

- `type`, that specifies a job stage (see below; only the “`test`” stage is used in this paper)
- `script`, that describes what to do (list of Linux shell commands to execute)

It is possible to define multiple jobs with the same stage, but there is one important thing that has to be considered: all jobs from the same stage are executed in parallel. So, in the default stages workflow, at first all jobs with type `build` are executed simultaneously, then all jobs with type `test` and finally all jobs with type `deploy`.

In some examples one may note the `stage` keyword, instead of `type`. Both these options are setting the same thing – they are just acronyms to themselves and can be used alternatively, depending only on personal preferences.

It is also worth to note, that all the commands, defined within the `script` keyword, are executed using a command like '`/bin/bash -c "<cmd>"`', where `<cmd>` is each element from all provided as `script`'s argument. It means that they are run in non-interactive shell mode and some features, due to that, may not work – like aliases, etc.

An example job definition, named **`run_rats`**, looks like this:

```
run_rats:
    type: test
    script:
        - rats -l 'c' ./* >> rats.txt
        - test $(grep -c 'High' rats.txt) -eq 0
    artifacts:
        when: always
        untracked: true
        paths:
            - main.c
```

The last, optional keyword, `artifacts`, can be used to define additional attachments to job results. It is described in next subsection.

3.2.4. Jobs Artifacts

Artifacts are files that shall be extracted from the container and attached to the final result, after jobs execution. A list of such files can be declared using the `artifacts` keyword in the job definition. At least one from two options, described below, is required to provide.

First option is to use the `paths` keyword with a list of files to extract as an argument. Due to security reasons, only files from job's working directory in container can be extracted (only paths like `./something` are allowed in general, not `/bin`, `/usr`, `/root`, etc.).

An alternative is option `untracked: true`, that sets Gitlab-CI to select all the files newly created in the job's directory. Basically it means that only files not known in the `git` repository will be extracted (those without registered history of changes in commits).

Normally the artifacts are extracted only for jobs that succeeded. However, one may want to see the artifacts (i.e. analysis results) even when the job fails! The optional keyword `when: always` tells Gitlab-CI to fetch artifacts no matter if the job succeeded or failed.

3.2.5. Own Stages Workflow

By default, in Gitlab-CI, there are three job stages, executed in the following order:

- `build`
- `test`
- `deploy`

In some specific cases it may be useful to define own stages and the order of their execution. For this purpose, the `types` keyword comes. An expected argument for this keyword is the list of job stages. The order of stages execution will be the same as in the list.

An example workflow definition looks like this:

```
types:
    - static_analysis
    - unit_testing
    - compilation
    - linking
    - deploying
```

Similarly, like for job definitions, there is an alias for `types` keyword, named `stages`. It has exactly the same meaning and can be used alternatively.

3.2.6. Selecting Specific Runner Node

As it was mentioned in the beginning, the runner service supports various executors that can be used to launch jobs. It is possible to have runner nodes configured that way, so a different executor will be used on each node.

However, one may want to select a specific type of node for running the jobs. Abstracting from the reasons, Gitlab-CI offers a tagging mechanism to achieve this goal. After associating a runner node with the Gitlab main service, an administrator can set specific tags for each node. Then, such node may be selected, using `tags` keyword within job definition in **.gitlab-ci.yml** file. An expected argument is a list of tags filter the runner nodes.

In CERN’s architecture the default tag is ’docker‘, but it is not obligatory to define this.

3.2.7. Running Jobs For a Specific Branch Only

By default, all defined CI jobs are executed each time when there is a code push to the repository. However, it is often desirable to run some jobs only for very specific conditions. For example, a deploying job would be only welcome for production branches.

Gitlab-CI provides `only` and `except` keywords for the mentioned purpose. For both keywords, an argument can be a list of expressions matching specific branch, tag (revision) name or API event. Three special values are also supported:

- `branches` – selects all the branches (allow or disable execution for all branches)
- `tags` – selects all the revisions (allow or disable execution for all tags/revisions)
- `triggers` – selects only the Gitlab-CI API events (like ‘rebuild’ click in web panel)

Also, the repository path/namespace may be used as expression, for example to turn off some jobs in project’s forks.

3.3. Example Content

For an always up-to-date example configuration file, please referrer to the following page: `https://gitlab.cern.ch/gitlabci-examples/static_code_analysis/blob/master/.gitlab-ci.yml`.

Also, the example configuration file was provided in the Appendix B of this document.

4. Tools usage

In this chapter an alphabetic list of tools for Static Code Analysis, provided in Security-Services/Code-Checking Docker image, can be found. A short instruction of usage for each tool is also provided.

Most of the tools return non-zero status when they will find and report a vulnerability or other anormality. This will cause break of CI jobs and workflow in normal case. For those programs, that behaves different, a note and example workaround is proposed.

An up-to-date list can be always found in the documentation describing the example configuration, located under `https://gitlab.cern.ch/gitlabci-examples/static_code_analysis`. Visit also `https://cern.ch/security/recommendations/en/code_tools.shtml` for a detailed list.

4.1. CPD

Recommended for: **C**, **C++**, **C#**, **Fortran**, **Go**, **Java**, **JavaScript**, **Matlab, Object-C**, **PHP, Python**, **Ruby**, **Scala**, **Swift**

This simple tool that is shipped with PMD (see below), but can be used standalone, and is meant to find duplicated code. The CPD name stands for *Copy-Paste-Detector*.

Command Line Usage:

```
cpd --minimum-tokens <tokens> [--language <language>] --files <directory/file>
```

Where:

- `<tokens>` - the minimal length of the same set of tokens (the smallest single units of source code recognized by compiler – keywords, variables, operators, etc.) that shall be reported as a duplicated code
- `<language>` - specify the language of code to check; may be c, cpp, cs, ecmascript, fortran, go, java (*default*), jsp, matlab, objectivec, perl, php, python, ruby, scala or swift

Examples:

```
cpd --minimum-tokens 100 --files ./java/src/
```

```
cpd --minimum-tokens 100 --language c --files ./*
```

4.2. CppLint

Recommended for: **C++**

The tool performs checking for compatibility of code with Google's style guide for C++ language, which ensures code to be clean from bad practices and more secure. It also checks for syntax errors and style consistency.

Command Line Usage:

```
cpplint [--exclude=<paths>] <directory/file>
```

Where:

- `<paths>` - comma separated list of paths (files, directories) that should not be checked

Examples:

```
cpplint ./code/*
```

```
cpplint --exclude=./magic/ ./*
```

Warning:

CppLint bases on regular expressions and occasionally may report false-positive warnings; it is possible to suppress scanning in specific parts of code by adding the following comment at the end of each impacted line: `// NOLINT`

4.3. FindBugs

Recommended for: **Java**

This advanced tool works not on source code, but on byte-code (compilation required) for bugs like operating on array with index out of bounds, bad operators for objects comparison, useless object declarations or imports, concurrent accesses and much more.

Command Line Usage:

```
findbugs <directory/file>
```

Examples:

```
findbugs ./java/bin/*.jar ./code/*.class
```

Warning:

FindBugs does not fail even if it will report something, therefore please consider saving the output to a file and counting warnings with command like:

```
test $(wc -l <'findbugs.txt') -eq 0
```

4.4. FlawFinder

Recommended for: **C**, **C++**

Simple program, written in Python, that examines source code for potential security weaknesses. It is designed to be fast, easy to use and to remain compatible with Common Weakness Enumeration[6] – a community flaws catalog, used by many commercial analyzers.

Command Line Usage:

```
flawfinder <directory/file>
```

Examples:

```
flawfinder ./code/*
```

Warning:

Sometimes false-positive warnings may be reported; add `/* Flawfinder: ignore */` at the end of impacted lines in source code to suppress such warnings.

FlawFinder does not fail even if it will report something, so please consider saving the output to a file and checking the content with command like:

```
awk '/^Hits /{ if($3 > 0) exit(1) }' flawfinder.txt
```

4.5. Perl::Critic

Recommended for: **Perl**

Code analyzer with policies mostly based on "Perl Best Practices"[7] book by Damian Conway, but not only – also some security-related policies are included into tool.

Command Line Usage:

```
perlcritic [--severity <level>] [--count] <directory/file>
```

6 – https://cwe.mitre.org/
7 – http://shop.oreilly.com/product/9780596001735.do

Where:

- `<level>` - select how cruel the tool shall be; may be **`1`** (*brutal*), **`2`** (*cruel*), **`3`** (*harsh*), **`4`** (*stern*) or **`5`** (*gentle*); both number or name can be used; **`5`** is default

Examples:

```
perlcritic ./code/*.pl

perlcritic --severity 1 ./*
```

4.6. PMD

Recommended for: **Java**, **JavaScript**

PMD is a static source code analyzer that finds common programming mistakes and flaws - like unused variables, empty catch blocks, unnecessary object creation, boundary cases, unreachable parts and so forth. The acronym can be expanded as “*Project Mess Detector*”.

Command Line Usage:

```
pmd -d <directory/file> -rulesets <rules> [-l <language>]
```

Where:

- `<language>` - specify the language that shall be checked; may be `ecmascript` or `java` (*default*)
- `<rules>` - identifiers of rule sets that shall be used during the code checking, comma separated; the list of possible sets can be found under CERN Computer Security Team’s web page with description of PMD tool: `https://cern.ch/security/recommendations/en/codetools/pmd.shtml`

Examples:

```
pmd -d ./code/src/ -rulesets java-basic,java-imports,java-unusedcode

pmd -d ./code/src/ -rulesets java-design,java-typeresolution

pmd -d ./code/src/ -rulesets java-sunsecure
```

4.7. PyChecker

Recommended for: **Python**

Quite a simple tool that aims to find some bad practices in Python's code that would be reported by the C/C++ compiler – for example wrong number of arguments to a function, usage of uninitialized variables or redefinition of functions within the same scope.

Command Line Usage:

```
pychecker [--limit <number>] <directory/file>
```

Where:

- `<number>` - max number of flaws that shall be reported; default 10

Examples:

```
pychecker ./src/*
pychecker --limit 100 ./*
```

4.8. PyLint

Recommended for: **Python**

The tool performs checking for compatibility of code with general Python's PEP8 style guide[8], as well as for bad practices, unused variables or modules and other common errors. Following PyLint advises makes your code clean, easy to read and secure.

Command Line Usage:

```
pylint [-d <msg-ids>] [-e <msg-ids>] <directory/file>
```

Where:

- `<msg-ids>` - identifiers of reported status messages, comma separated, that shall be ignored (-d) or checked (-e); may be either message name (like '*bad-continuation*') or code (like '*C0330*') - you can find a list under `https://pylint.readthedocs.io/en/latest/features.html`

Examples:

```
pylint ./*
pylint -d bad-continuation ./*
pylint -d all -e C0330,C0301 ./*
```

8 – https://www.python.org/dev/peps/pep-0008/

4.9. RATS

Recommended for: **C**, **C++**, **Perl**, **PHP**, **Python**

Multi-language tool that scans source code for common security related errors, including cases like buffer overflows, race conditions and references to risky built-in functions. The name stands for “*Rough Auditing Tool for Security*” and as first word suggests (rough), some false-positives may be reported.

Generally it should be used with mixed-language repositories with caution. It is recommended to specify the language when running RATS to reduce the chances for false-positives.

Command Line Usage:

```
rats [-l <language>] [-w <level>] <directory/file>
```

Where:

- `<language>` - specify the language that shall be checked; may be c, perl, php or python; by default try guessing by file extension
- `<level>` - set how rough shall the tool be: 1 (*less*), 2 (*default*), 3 (*more*); please, note that for higher level there may be more false-positives

Examples:

```
rats ./*

rats -l c ./codes/*.c

rats -l perl -w 3 ./*
```

Warning:

As RATS does not fail even if it will find something, please consider saving the output to a file and checking the content with command like:

```
test $(grep -c 'High' rats.txt) -eq 0
```

5. Jobs results

In this part of document, the results-related mechanisms for CI jobs are presented.

5.1. Badges

This is the most basic mechanism that basically tells one if the CI jobs from specific repository and branch succeeded or failed. They may be used on project's `readme` pages or in some sort of monitoring page to ensure the general status of project.

Below the most common badges with meaning are presented:

- build unknown - there is no `.gitlab-ci.yml` file in repository
- build success - all CI jobs succeeded
- build pending - CI jobs are waiting for free runner node
- build running - CI jobs are currently running
- build failed - at least one of the CI jobs has failed

Badges are always available and can be fetched by referring to the following URL scheme: `http://<gitlab-server>/<user>/<project>/badges/<branch>/build.svg`.

In CERN's case the `<gitlab-server>` is `gitlab.cern.ch`, a `<user>` represents project's namespace – a owner's username or group like `ComputerSecurity`; `<project>` stands for project name – for example `Security-Services-Code-Checking`; finally there is a branch name, for example `master`, under `<branch>`. Therefore, an example URL for a badge describing status of repository which is used to build our Docker image for Static Code Analysis, look like the following:
`gitlab.cern.ch/ComputerSecurity/Security-Services-Code-Checking/badges/master/build.svg`

Alternatively, an URL for your own repository, as well as HTML or the Markdown code to put directly on a website or in a `readme` page, can be obtained by selecting **Badges** menu position in project's settings on the Gitlab web interface.

The last section of this chapter presents where to obtain badges and few similar statuses that one may encounter in Gitlab web interface, after introducing the **.gitlab-ci.yml** file.

5.2. Artifacts

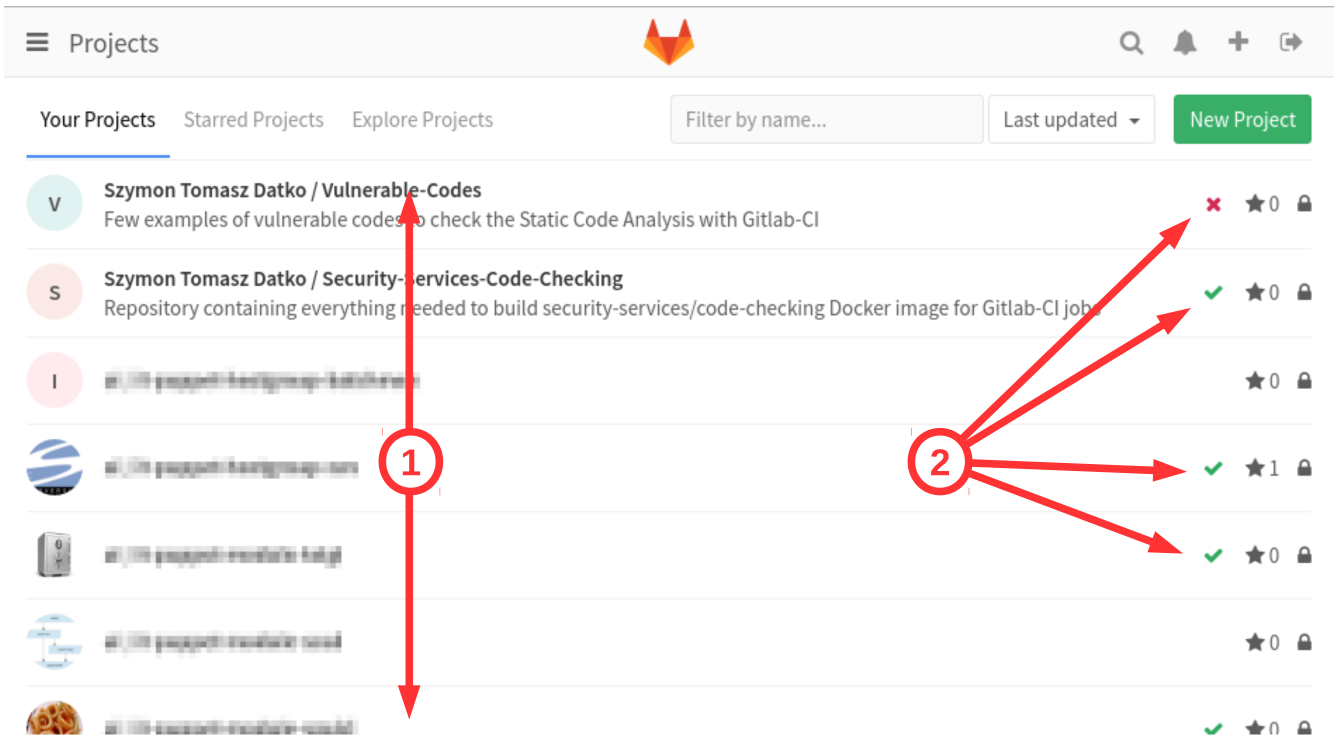
After declaring artifacts in the **.gitlab-ci.yml** file, it is possible to download an archive containing all the specified files and directories or browse them directly in the web interface. At the time when this paper is being written, there is not any dashboard mechanism to parse and preview the artifacts – there is, however, a feature proposal for that, reported on Gitlab's forum[9], so maybe in the future...

How to obtain artifacts is presented in the subsection below.

5.3. Web Interface Overview

In Picture 3 there is main page of Gitlab presented – containing a list of all repositories that an user is allowed to view **(1)**. For each repository that contains **.gitlab-ci.yml** configuration file, there is an icon that represents status of the CI jobs for main branch in each particular repository **(2)**, similarly to the badges mechanism.

Picture 4 shows the project's main page, with CI jobs status for the last pushed commit **(3)**. Also, under the project's settings button **(4)** an URL to the badges may be obtained.



Szymon Tomasz Datko / Vulnerable-Codes
Few examples of vulnerable codes to check the Static Code Analysis with Gitlab-CI

Szymon Tomasz Datko / Security-Services-Code-Checking
Repository containing everything needed to build security-services/code-checking Docker image for Gitlab-CI jobs

[redacted]

[redacted]

[redacted]

[redacted]

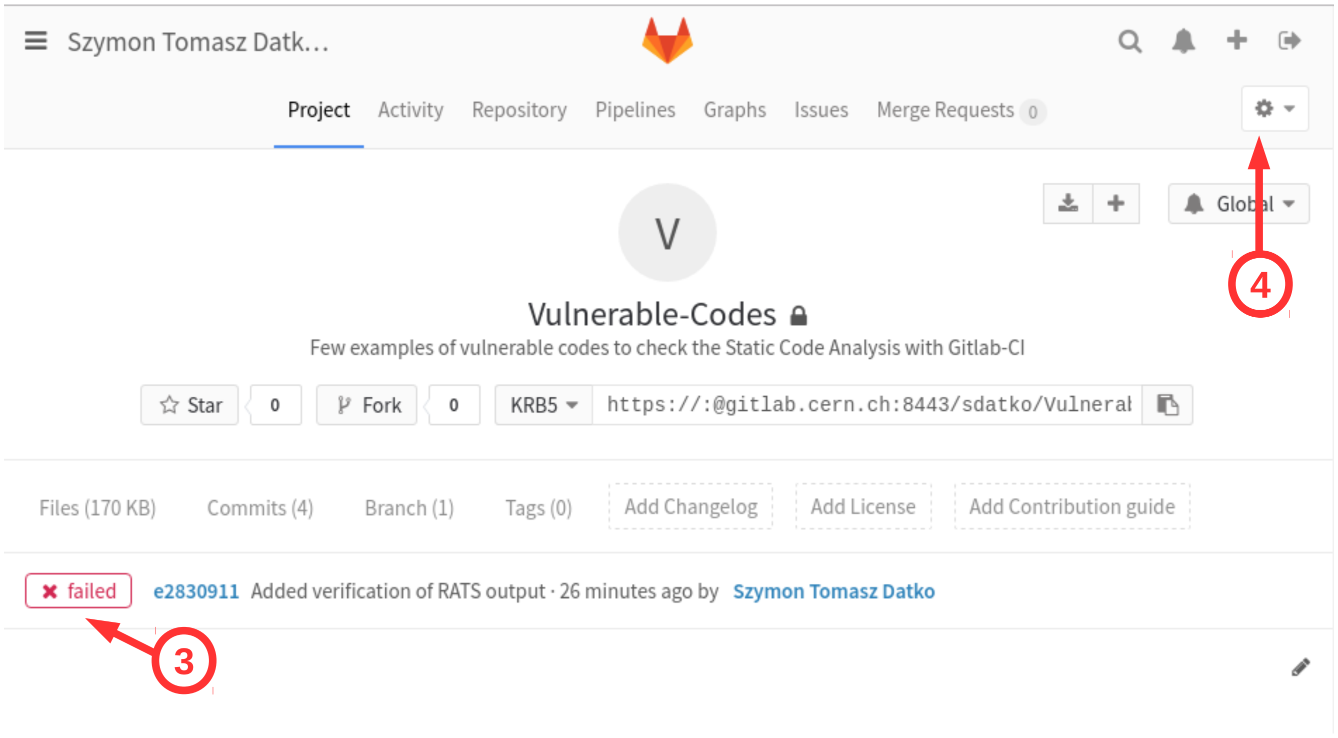
[redacted]

Picture 3. Gitlab main page – list of repositories

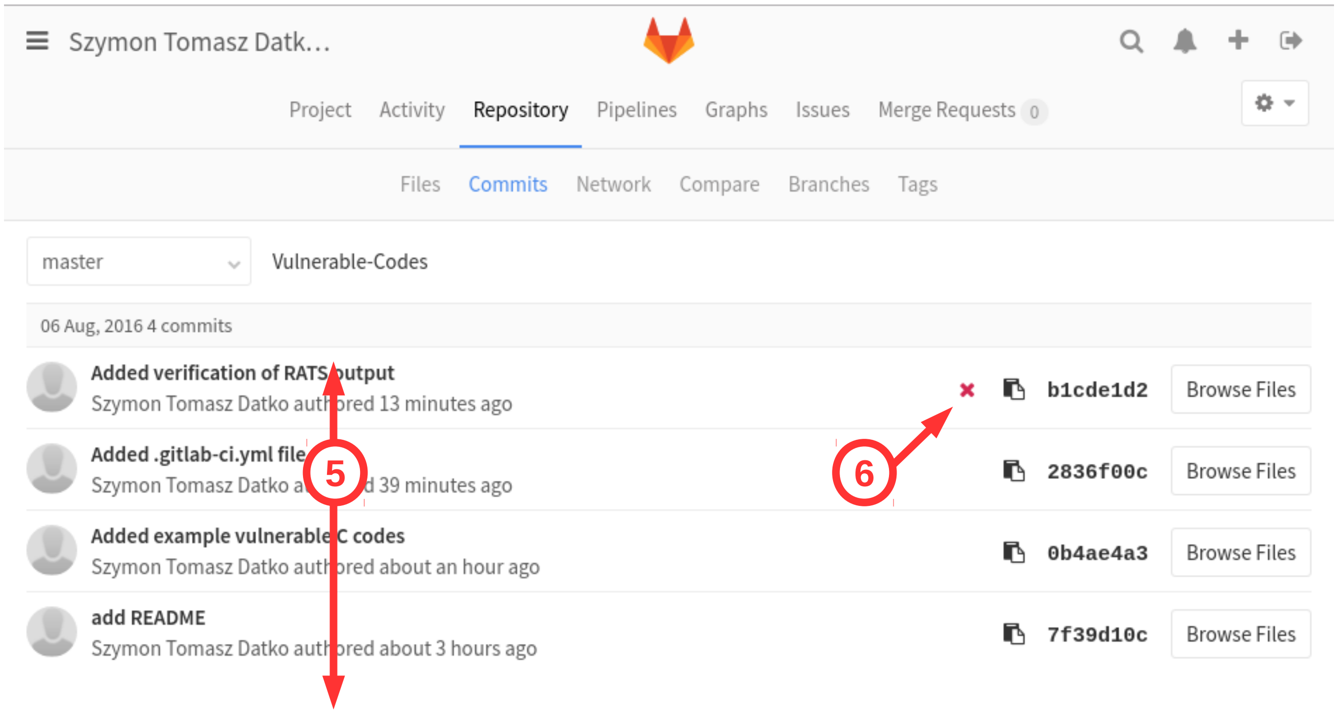
Source: `(own creation)`

9 – https://gitlab.com/gitlab-org/gitlab-ce/issues/13227

On picture 5 the list of commits in a repository is visible **(5)**. There are also icons representing statuses of CI jobs on the right side **(6)** for each commit.



Picture 4. Project's main page in Gitlab
Source: `(own creation)`


Szymon Tomasz Datk...
Project
Activity
Repository
Pipelines
Graphs
Issues
Merge Requests 0
Files
Commits
Network
Compare
Branches
Tags
master
Vulnerable-Codes
06 Aug, 2016 4 commits
Added verification of RATS output
Szymon Tomasz Datko authored 13 minutes ago
b1cde1d2
Browse Files
Added .gitlab-ci.yml file
Szymon Tomasz Datko authored 39 minutes ago
2836f00c
Browse Files
Added example vulnerable C codes
Szymon Tomasz Datko authored about an hour ago
0b4ae4a3
Browse Files
add README
Szymon Tomasz Datko authored about 3 hours ago
7f39d10c
Browse Files
5
6

Picture 5. List of commits in project
Source: `(own creation)`

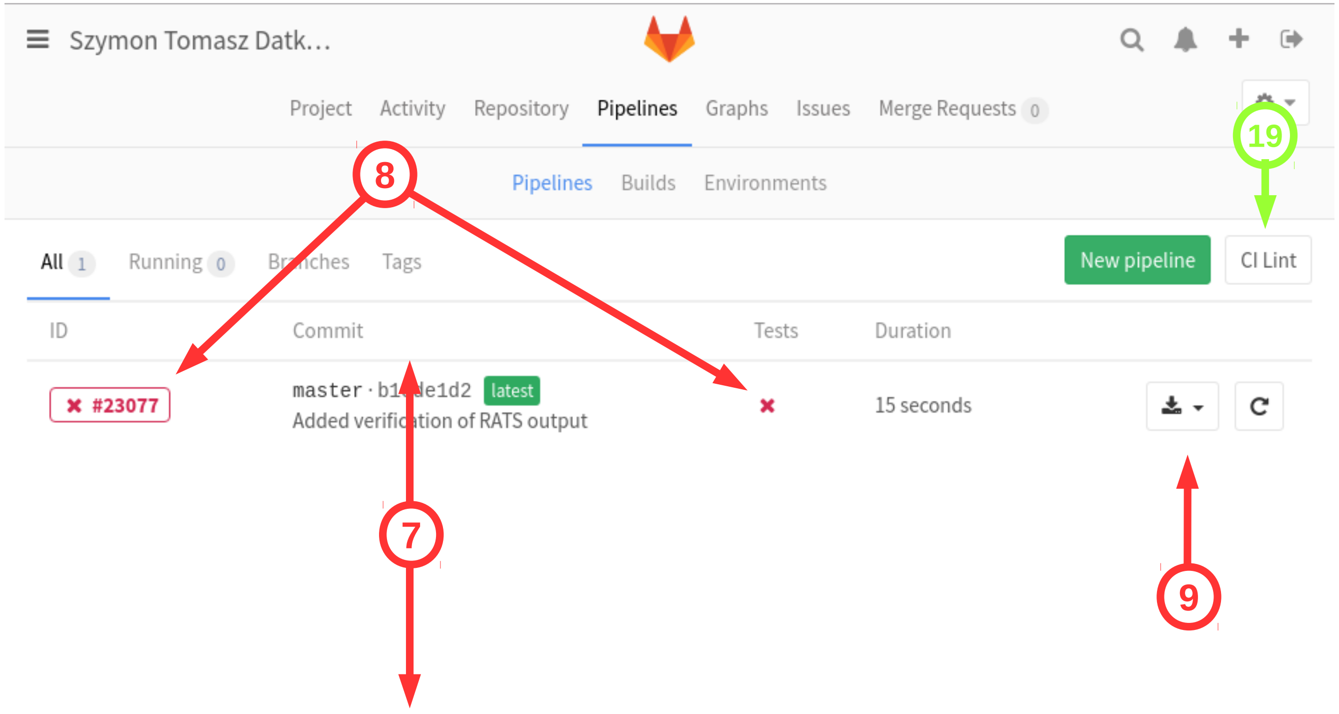
Picture 6 shows the pipelines page. Each pipeline is a single job request sent by the Gitlab service to the runner node **(7)**. The status with job ID is presented in the first column **(8)** – by clicking on it, the details for this request will be displayed. Also, it is possible to download artifacts from this page **(9)**.

Picture 7 shows detailed view of single job requests (pipeline). There is list of single jobs that have been run within the request **(10)**, with status for each job **(11)** and an overall one **(12)**. With the button **(13)** all failed jobs can be re-launched. Under **(14)** there are buttons to download artifacts and re-run single jobs. By clicking on the single job status, all its details are displayed.

The final picture, number 8, presents the details about a single job. On the left side, there is the raw output from backend displayed **(15)**. Looking through it may be very useful for debugging purposes. Above the logging console, at position **(16)**, there may be tabs with other jobs that were executed in parallel – to switch between them fast in the interface (not visible in the picture).

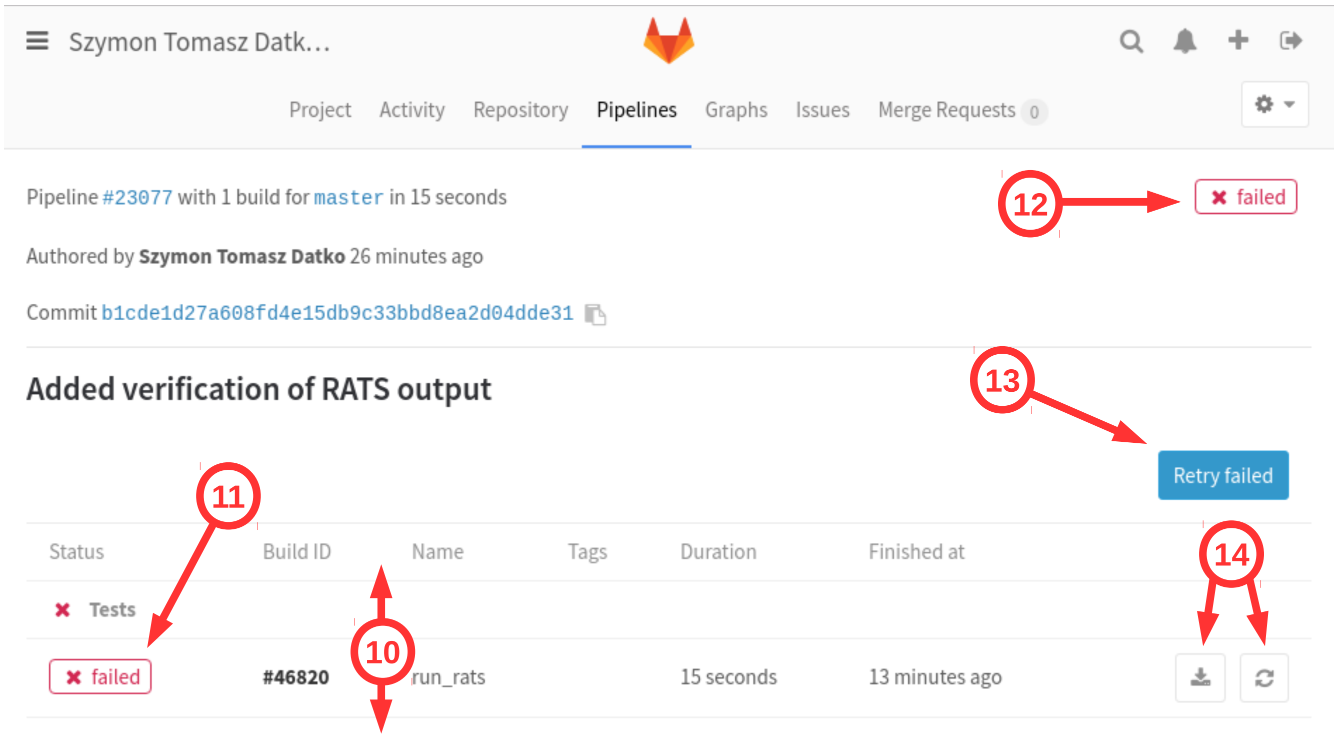
The right bar contains some details about job execution **(17)**, as well as two buttons that can be used to download the artifacts or check them directly in web browser **(18)**. There is also a `Retry` button, that will cause re-launch of the job.

One useful hint at the end – on the pipelines page (Picture 6) there is also a button that leads to page with a service able to check the **.gitlab-ci.yml** content/syntax. It is marked on the picture as **(19)**.



Picture 6. Pipelines page – list of CI requests with overall statues

Source: `(own creation)`



Pipeline #23077 with 1 build for master in 15 seconds

Authored by **Szymon Tomasz Datko** 26 minutes ago

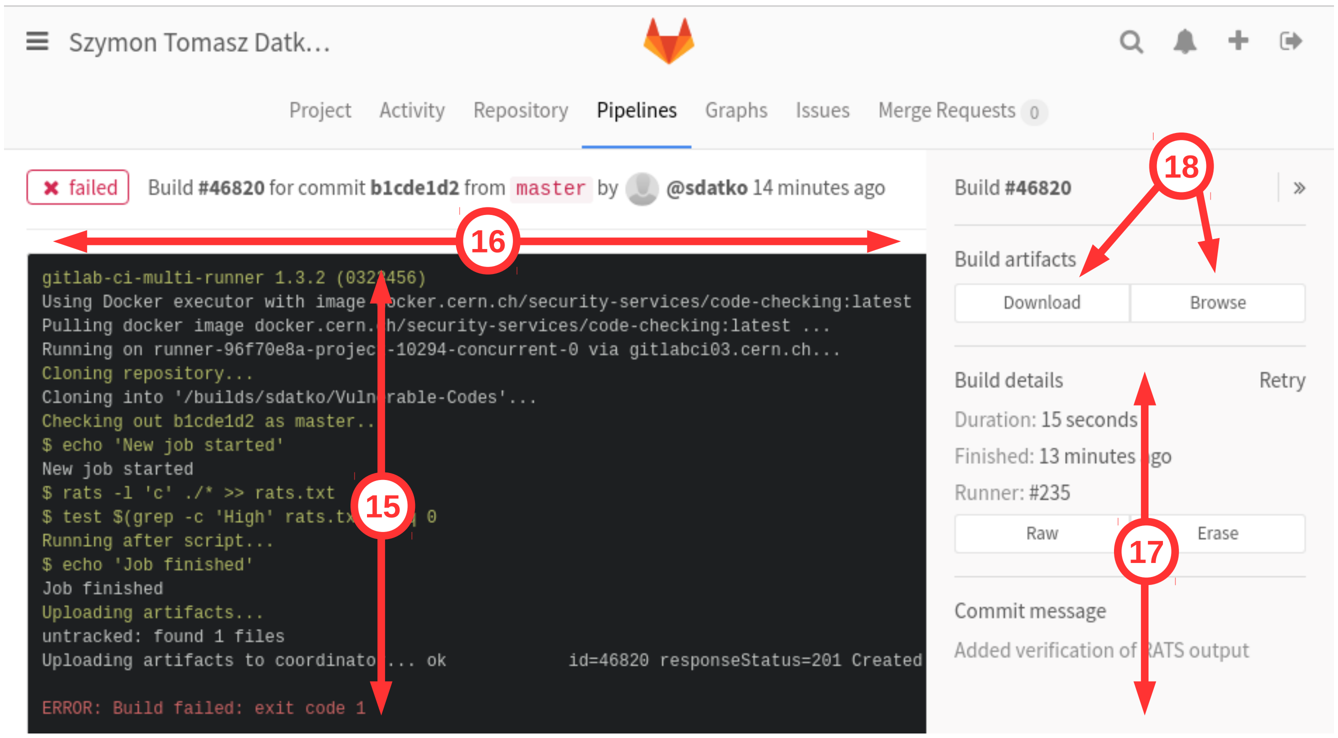
Commit b1cde1d27a608fd4e15db9c33bbd8ea2d04dde31

Added verification of RATS output

Status	Build ID	Name	Tags	Duration	Finished at
✖ Tests					
✖ failed	#46820	run_rats		15 seconds	13 minutes ago

Picture 7. Pipeline details – list of jobs inside selected request

Source: `(own creation)`



✖ failed Build **#46820** for commit **b1cde1d2** from master by **@sdatko** 14 minutes ago

```
gitlab-ci-multi-runner 1.3.2 (0322456)
Using Docker executor with image docker.cern.ch/security-services/code-checking:latest
Pulling docker image docker.cern.ch/security-services/code-checking:latest ...
Running on runner-96f70e8a-project-10294-concurrent-0 via gitlabci03.cern.ch...
Cloning repository...
Cloning into '/builds/sdatko/Vulnerable-Codes'...
Checking out b1cde1d2 as master...
$ echo 'New job started'
New job started
$ rats -l 'c' ./* >> rats.txt
$ test $(grep -c 'High' rats.txt) -eq 0
Running after script...
$ echo 'Job finished'
Job finished
Uploading artifacts...
untracked: found 1 files
Uploading artifacts to coordinator... ok          id=46820 responseStatus=201 Created

ERROR: Build failed: exit code 1
```

Build **#46820**

Build artifacts

Build details

Duration: 15 seconds

Finished: 13 minutes ago

Runner: #235

Commit message

Added verification of RATS output

Picture 8. Single CI job details in Gitlab

Source: `(own creation)`

6. Summary

In this chapter the done work and conclusions are described. Also, some possible future steps, to improve the project, are proposed.

6.1. Conclusions

During the project a usable platform for Static Code Analysis was defined, tested and described. It may be successfully integrated into existing repositories, hosted on CERN’s Gitlab installation in order to improve quality of code.

In a prepared image, the basic development tools and code analyzers for programming languages most commonly used at CERN were installed. Its current state allows, more or less, to achieve the project’s goals. There is, however, still a lot of details that are worth to check (see next subsection).

It was a great opportunity to put hands on Gitlab, Docker and Static Code Analyzers. Although Gitlab-CI is a pretty new part of Gitlab and its configuration is much simpler, when compared to other CI toolkits, a great integration with Gitlab makes it very powerful. Gitlab remains under active development, which means that in the nearest future we may expect a lot of improvements in existing features and possibly some new mechanisms.

This paper contains only the very basics mechanisms described, but still it should allow every interested person to start with Continuous Integration and hopefully even continue researching for further improvements of CERN’s architecture for Static Code Analysis.

6.2. Future Work

Most obvious part of future work would involve continuous updating of the Docker image `Security-Serices/Code-Checking` with the newest versions of code analyzers. The process of building an image and sending to CERN’s Docker registry was automated, however changes in Dockerfile, that describes the image, requires manual changes in the `Computer-Security/Security-Services-Code-Checking` repository.

There are also few limitations of current architecture and things to consider.

First limitation is that yet there is no dashboard for cool, eye-catching preview of test results like similar ones that are available in Bamboo[10] and in Jenkins[11] as a plugin. However, it was requested and marked as a feature proposal for Gitlab this year, so maybe in the nearest future it will become available. A temporary alternative may be to use some other tool for that and define, in example, a job, that will send a copy of the results to remote dashboard service.

The second limitation is that there is no support for the compilation of C/C++ applications on the Windows operating system platform yet. Although it should be possible just by installing additional compilers, like MinGW, it was not done due to lack of time.

The third thing is related to code analyzers. There are a lot of really decent commercial applications, better that the free ones in many aspects. In this project, the possibility of using them inside Docker image was not considered, as there are many potential licensing limitations. But due to many advantages, it may be worth to verify that.

The fourth limitation relates to running Windows applications inside a container. There is at least one very good code analyzer distributed as Windows executable, the Visual Code Grepper[12]. Although it supports launching from a CLI, it requires also a .NET 4.0 library. An attempt to it setup, using Wine libraries, was made, but due to encountered difficulties (related to 32 vs 64 bit architecture) and lack of time, it was dropped. However, it may be very worth to research this field.

An alternative here may be to use Windows-based runner nodes. Although the CERN Git administrators have provided only the Linux-based runners, it is possible to provide properly configured Windows-based nodes and associate them with main Gitlab installation after (either as a global ‘shared nodes’, available for all users, or ‘private nodes’, available only for specific projects).

Last thing to consider is the splitting of the Docker image into smaller ones. Currently the solution is not 100% compatible with the Art of Docker, as it is one big image to rule all the tools and in Gitlab-CI to bind them. Splitting and creating, for example, an image per tool may result in more lightweight solution and slightly smaller execution times for jobs. This needs, however, further research.

10 – https://www.atlassian.com/software/bamboo
11 – https://jenkins.io/
12 – https://github.com/nccgroup/VCG

Appendix A. Setup Own Installation

In this paper, the usage and configuration of Gitlab-CI for Static Code Analysis purposes is described. However, to test some nasty development modifications, or for some other reason, one may want to deploy its own infrastructure.

In order to set up a basic infrastructure, it is generally required to install and run just two independent components – the **Gitlab** service and the **Gitlab-CI Runner** service. Although installing them on separated hosts is highly recommended, for development or personal purposes both may be set on the same machine. The guidelines below are considered for the **CentOS 7** operating system.

A.1. Setup Gitlab

Basically, there are two required dependencies to install for Gitlab – it is the `postfix` mail server and the `openssh-server` package. This will allow receiving e-mails from any Gitlab installation and cloning repositories over the “ssh” protocol. Finally, just launching the services is required:

```
yum update -y
yum install postfix openssh-server -y
systemctl enable sshd
systemctl start sshd
systemctl enable postfix
systemctl start postfix
```

After that, system is ready for Gitlab. Now it is required to get the package in the latest version for CentOS7 and install it in the system. It may be downloaded from following website: https://packages.gitlab.com/gitlab/gitlab-ce/packages/el/7/gitlab-ce-8.9.6-ce.0.el7.x86_64.rpm/download.

The final set of commands looks then like this:

```
curl -LJO ‘https://packages.gitlab.com/gitlab/gitlab-ce/packages/el’\
          ‘/7/gitlab-ce-8.9.6-ce.0.el7.x86_64.rpm/download’
rpm -i gitlab-ce-8.9.6-ce.0.el7.x86_64.rpm
```

The final thing is to launch the automating configuration, which is based on the Chef[13] service. In order to do that, it is simply enough to execute the following command:

```
gitlab-ctl reconfigure
```

13 – https://www.chef.io/

After that last step (which may take few minutes), a Gitlab main page should be accessible via any web browser. As a first thing, one should open that web page and set the administrator's password.

Also, for development infrastructure, it may be good idea to turn off account creations on its main page. It is doable by clicking the '`Admin area`' button, after logging to Gitlab as administrator, then '`Settings`' and disabling the '`Sign-up`' option.

A.2. Setup Gitlab-CI Runner

The installation of the runner service should be preceded by the installation of Docker engine. In order to do that, the easiest option is to execute the following commands:

```
curl -sSL https://get.docker.com/ | sh
sh -c 'sleep 3; yum -y -q install docker-engine'
```

Then remember to launch the Docker service:

```
systemctl enable docker
systemctl start docker
```

For other backends (e.g. VirtualBox) the previous steps shall be adapted accordingly. When the desired backend is properly configured, an installation of the runner service, called gitlab-ci-multi-runner, may be performed. The easiest method is to use a script from: `https://packages.gitlab.com/install/repositories/runner/gitlab-ci-multi-runner/script.rpm.sh`.

Execution of this script will add the repository in system:

```
curl -L ‘https://packages.gitlab.com/install/repositories/runner/’\
        ‘gitlab-ci-multi-runner/script.rpm.sh’ | sudo bash
```

Then just install the package with yum manager:

```
yum install gitlab-ci-multi-runner -y
```

Now it is required to associate the runner service with Gitlab installation:

```
gitlab-ci-multi-runner register
```

At this step, the configuration script will ask few important questions:

- Gitlab installation URL – generally it is `http://<gitlab-server>/ci`
- Token for authorization – can be found under '`Admin area`' > '`Overview`' > '`Runners`'
- Runner name – by default it is the node's hostname, which should be enough
- Runner tags – does not really matter for now (useful if there is more than one runner)
- Executor – '`docker`' in this case, but feel free to experiment
- Default Docker image – '`centos:latest`' is always good option

Finally, make sure that the gitlab-ci-multi-runner service is running:

```
gitlab-ci-multi-runner restart
```

After all those steps, the new runner service shall be visible in the Gitlab web interface, under ‘`Admin area`’ > ‘`Overview`’ > ‘`Runners`’ page. Additional, more advanced configurations of the runner service can be performed by setting values in `/etc/gitlab-runner/config.toml` file. For example, the section `[runners.docker]` allows specifying parameters for Docker containers – in case of DNS problems (unable to clone repository in CI jobs), a solution may be to add there a line like:

```
extra_hosts = ["<gitlab-server-hostname>:<gitlab-server-IP>"]
```

For detailed debugging, from container’s point of view, it may be useful to deploy a container on runner node. If Docker service is running, the following command should work:

```
docker run -i -t --entrypoint /bin/bash \
       docker.cern.ch/security-services/code-checking:latest
```

This command will create a container and attach an interactive session.

Appendix B. Example Configuration File

```
#
# This is an example .gitlab-ci.yml file for C++ and Python code checking
#

image: docker.cern.ch/security-services/code-checking:latest

before_script:
    - echo 'New job started'

after_script:
    - echo 'Job finished'
    - ls -la

types:
    - test

run_cpd:
    type: test
    script:
        - cpd --minimum-tokens 100 --language cpp --files ./codes/Cpp/*
        - cpd --minimum-tokens 100 --language python --files ./codes/Python/*

run_pylint:
    type: test
    script:
        - pylint -d bad-continuation ./codes/Python/*

run_cpplint:
    type: test
    script:
        - cpplint ./codes/Cpp/*

run_rats:
    type: test
    script:
        - rats -l 'c' ./codes/Cpp/* >> rats.txt
        - test $(grep -c 'High' rats.txt) -eq 0
    artifacts:
        when: always
        untracked: true
        paths:
            - ./codes/Cpp/*
            - ./.gitlab-ci.yml
```

Bibliography

[1] How to use Gitlab-CI for Continuous Integration of Gitlab projects
(`https://cern.service-now.com/service-portal/article.do?n=KB0003690`)

[2] Gitlab Documentation - Gitlab-CI Quick Start
(`http://docs.gitlab.com/ce/ci/quick_start/README.html`)

[3] Static Code Analysis with Gitlab-CI -- brief usage description
(`https://gitlab.cern.ch/gitlabci-examples/static_code_analysis`)

[4] Static Code Analysis with Gitlab-CI -- example configuration file
(`https://gitlab.cern.ch/gitlabci-examples/static_code_analysis/blob/master/.gitlab-ci.yml`)

[5] Automate your life with Gitlab-CI -- Students Session 2016 (video)
(`https://cds.cern.ch/record/2206413`)

[6] List of Static Code Analysis Tools at CERN Computer Security site
(`https://security.web.cern.ch/security/recommendations/en/code_tools.shtml`)