SN Computer Science (2022) 3:80
https://doi.org/10.1007/s42979-021-00939-4

ORIGINAL RESEARCH

SecDocker: Hardening the Continuous Integration Workflow

Wrapping the container layer

David Fernández González[1] · Francisco Javier Rodríguez Lera[1] · Gonzalo Esteban[1] · Camino Fernández Llamas[1]

Received: 31 March 2021 / Accepted: 13 October 2021 / Published online: 23 November 2021

Abstract
Current Continuous Integration (CI) processes face significant intrinsic cybersecurity challenges. The idea is not only to solve and test formal or regulatory security requirements of source code but also to adhere to the same principles to the CI pipeline itself. This paper presents an overview of current security issues in CI workflow. It designs, develops, and deploys a new tool for the secure deployment of a container-based CI pipeline flow without slowing down release cycles. The tool, called *SecDocker* for its Docker-based approach, is publicly available in GitHub. It implements a transparent application firewall based on a configuration mechanism avoiding issues in the CI workflow associated with intended or unintended container configurations. Integrated with other DevOps Engineers tools, it provides feedback from only those scenarios that match specific patterns, addressing future container security issues.

Keywords Containerization · Continuous integration · Docker

Introduction

There are plenty of tools to analyze and secure the creation of container images. Besides that, several organizations have developed guidelines to assist developers in the creation of such images with a certain degree of security. For instance, focusing on Docker [18], it is possible to find the *Docker Benchmark tool* released by the Center for Internet Security (CIS) [9] or the *Ultimate Benchmark for Container Image Scanning* (UBCIS) [2]; both containing guides that analyze every single dangerous step involved during the image-building process.

However, there are different local exploitation issues associated to the continuous integration (CI) workflow that needs to be secured. The problem here is that some containerization solutions (like Docker) have different exploits that allow an attacker to override some of the image specifications; which is done by providing new ones at the very moment the container is created. Furthermore, opening ports are always a security risk if it is controlled by a low level user in the system (like a developer in a DevOps server).

Thus, this study focuses on developing a tool called *SecDocker* that enhances the cybersecurity pipeline when integrating containerization in the CI workflow [27]. *SecDocker* is a wrapper, specifically an application firewall for Docker, that allows system administrators to block the capabilities offered by Docker in the `run` command. By doing so, any dangerous actions performed during the creation or execution of a container, such as deploy container images with malicious code, download malicious payload at runtime within the container of the host, or get sensitive information from the Docker log (to name a few), can be blocked before they even get executed.

But CI environments are, by definition, completely automated, which suggests a security approach that can deal with the underlying workflow. This creates a need for security administrators to apply tools that perform security checks at the processes that shape the CI. *SecDocker* adds a layer of security to CI environments, allowing the complete use

This article is part of the topical collection "New Paradigms of Software Production and Deployment" guest edited by Alfredo Capozucca, Jean-Michel Bruel, Manuel Mazzara and Bertrand Meyer.

✉ David Fernández González
dferng@unileon.es

[1] University of León, Campus de Vegazana, 24071 León, Spain

of Docker to developers and users, regardless of whether they use the system correctly. Right now, a CI user could potentially create security threats if they use the platform incorrectly. *SecDocker* aims to solve such cases by simply removing the possibility of creating them from the users. It controls every request performed by the CI to the container platform, providing a secure use for Docker pipelines.

Below, the research question and main contribution are presented. The remainder of the paper presents the elements for answering the Research Question. Section "Background" overviews the state of the art in containerization and continuous integration workflow. Section "Container Layer in CI" presents the developer's and attacker's schemes to the containerized CI layer. Section "Proposed Solution" presents *SecDocker* tool: its design, architecture and usage. Section "Empirical Validation" validates with the results of *SecDocker* from two different experiments carried out in this research measuring performance and the operational flow. Section "Discussion" discusses enumerating pros and cons of *SecDocker* and finally sect. "Conclusion" provides conclusions about the processes and solutions presented in this paper.

Research Question and Contribution

CI is a cornerstone methodology that automatically addresses several processes previously faced by software developers. However, the CI workflow also needs to meet the security mechanisms that will guarantee flexibility, productivity, and efficiency during the Software Development Life Cycle. Thus, this paper aims to frame a set of elements that are addressed within the next Research Question (RQ): RQ: *Which are the mechanisms for avoiding and minimizing cybersecurity and misconfiguration issues in a CI container-based deployment system?*

This RQ scales to a new level when the containerization tool is Docker. Most parts of current automation servers and processes are supported on Docker containers. However, its engine shows critical points that can lead to a crashed CI pipeline due to malicious users or unaware DevOps engineers. Working with an erroneous configuration would promote a bad system behavior, with the manpower and economical costs associated. There are three main phases when using Docker containers in the CI workflow: (1) issues associated with image retriever; (2) issues associated with image builder; and (3) issues generated when the image is deployed.

This study presents an overview to the latter step, as well as the design and development of a tool called *SecDocker* for minimizing issues associated with container deployment. Besides, the tool introduces expansion capabilities for solving the first and second steps using plugins.

Background

CI is one of many software development practices aimed at helping organizations to accelerate their development and delivery of software features without compromising quality [11]. According to Fitzgerald and Stol [8], it can be defined as "a process which is typically automatically triggered and comprises inter-connected steps such as compiling code, running unit and acceptance tests, validating code coverage, checking compliance with coding standards and building deployment packages". For Shahin et al. [22], alongside Continuous Delivery or CDE (ensure the package is always at a production-ready state after tests) and Continuous Deployment or CD (deploy the package to production or customer environments), CI is considered part of the continuous software engineering paradigm which includes the popular term "DevOps" [8].

DevOps is a mix of the words *Development* and *Operations* and, although there is no common definition for it, some literature reviews exist to date that addresses this point [5, 12, 23]. For instance, Jabbari et al. [12] define it as "a development methodology aimed at bridging the gap between Development (Dev) and Operations (Ops), emphasizing communication and collaboration, continuous integration, quality assurance and delivery with automated deployment utilizing a set of development practices". To enable such concepts or practices, and thus aid developers in materializing them, DevOps relies on using a range of tools [5, 16], from source code management to monitoring and logging, as well as configuration management. Together, these tools allow the creation of a pipeline that automates the processes of compiling, building and deploying the source code into a production platform [11].

But as a relative young methodology, integrating and maintaining these tools or managing the infrastructure in which they run automatically may pose a challenge [22]; especially for CI and CD. As Leite et al. discuss in their literature review [16], concepts like "infrastructure as code", virtualization, containerization or cloud services are solutions currently known to be used for these types of issues. Among all of them, containerization is perhaps the most popular solution in DevOps environments at the moment. With a platform as a service focus, it is used for delivering software in a portable and streamlining way by providing a platform that allows developing, running and managing applications without worrying about the infrastructure needed [20].

Technically speaking, containerization is a type of lightweight OS-level virtualization technology that allows running multiple isolated systems (in terms of processes, resources, network, etc) while sharing the same host OS. Such systems or *containers*, hold packaged, self-contained applications and, if necessary, binaries and libraries required

to run them [3]. Moreover, they have been around for some time in various forms: from chroot, FreeBSD jails or Solaris zones to Linux-based solutions relying on kernel support like LXC or OpenVZ [3, 7, 20, 26]. But over time, containerization has become a major trend thanks to tools like Docker [16, 19].

Docker is an open-source platform that facilitates the management of containers using a client-server architecture through a CLI tool, a daemon and a REST API [19, 26]. It relies on the concept of *images* to build containers, that is, a specification of the collection of layered file systems, their corresponding execution environment and some metadata; making them portable, shareable and also updatable [20]. Regarding their usage, Docker containers can be used either as a microservice (to host a single service), as a way of shipping complete virtual environments (to reproduce and automate the deployment of applications) or even as a platform as a service (to cope with security and infrastructure integration issues) [7, 18].

From a security perspective, Docker provides different levels of isolation, host hardening capabilities and some countermeasures related to network operations [6, 7, 18]. Nevertheless, it is not exempt from security threats nor vulnerabilities, such as ARP spoofing, DoS attacks, privilege escalation, etc. This is due to the nature of containerization itself because an attack on the host OS may expose all containers and their network traffic. To address these cybersecurity risks, it is necessary to take similar actions to DevOps; especially where pipeline automation is a requirement (as in CI or CD). Such actions can be understood as best practices or recommendations that aim to establish a Secure Software Development Life Cycle. Examples of this can be found in reports like *DevSecOps: How to Seamlessly Integrate Security Into DevOps* [17] or *DoD Enterprise DevSecOps Reference Design* [15], where container hardening is contemplated.

Container Layer in CI

Containers are used in CI processes to isolate and automate the creation of an application into one single self-contained virtual environment. This solution simplifies DevOps manpower, as it allows to split a large application development project into several smaller work units. Having said that, this section describes the role of CI from the point of view of two actors (DevOps engineers and attackers) and also presents the scenarios likely to be vulnerable.

DevOps Engineers Scheme

From a developers perspective, CI is used to guarantee the quality, consistency and viability across different environments [10]. But as CI systems are vulnerable to security attacks and misconfigurations [22], DevOps engineers frequently rely on containers to create such environments as they provide isolation without much effort to them. Generally, this has been achieved by technologies like Docker which allow them to treat infrastructures as code [13].

Regarding CI, Docker has ease DevOps engineers in the replication of environments for building automation pipelines. Particularly, as Boettinger et al. point in their work [4], it has solved common issues encountered by end-users like managing dependencies (through images), imprecise documentation (through scripts to build up such images) or code-rot (with image versioning), along with the adoption and re-use of existing workflows (thanks to features like portability, easy integration into local environments or public repositories for sharing and reusing those images).

But despite the benefits that Docker or other containerization technologies may offer to DevOps engineers in CI environments, the latter still face challenges related to its adoption; particularly associated with introducing any new technologies or phenomena in a given organization [10, 22]. According to Shahin et al. [22], literature shows that, among the common practices for implementing CI workflows, DevOps engineers need to decompose development into smaller units and also plan and document the activities that comprise the automation pipeline. Having said that, it must be noted that there are many ways of approaching the design of such pipelines. But taking into account the use of containers and based on Bass et al. approach [1], any CI workflow must include the following 6 components in such design plan:

1. Automation server. Implements the CI/CD pipeline and creates a local workspace in which its steps take place.
2. Orchestrator. Sequentially triggers each step of the pipeline by communicating with the remaining components. It should be noted that, when using containers, steps may require images to perform their actions. Thus, the same image can be used through the whole pipeline or in specific steps.
3. Code retriever. Pulls source code from repository to local workspace.
4. Unit tester. Runs automated unit tests on source code.
5. Artifact builder. Builds deployable artifacts from source code.
6. Image generator. Builds, verifies, stores and deploys an image to be used within the pipeline.

With this in mind and despite using containers, any standard CI workflow that establishes and defines this components will lower security and increase its functionality risks. To avoid this, different automated continuous tests could be applied to the whole process. However, and particularly for

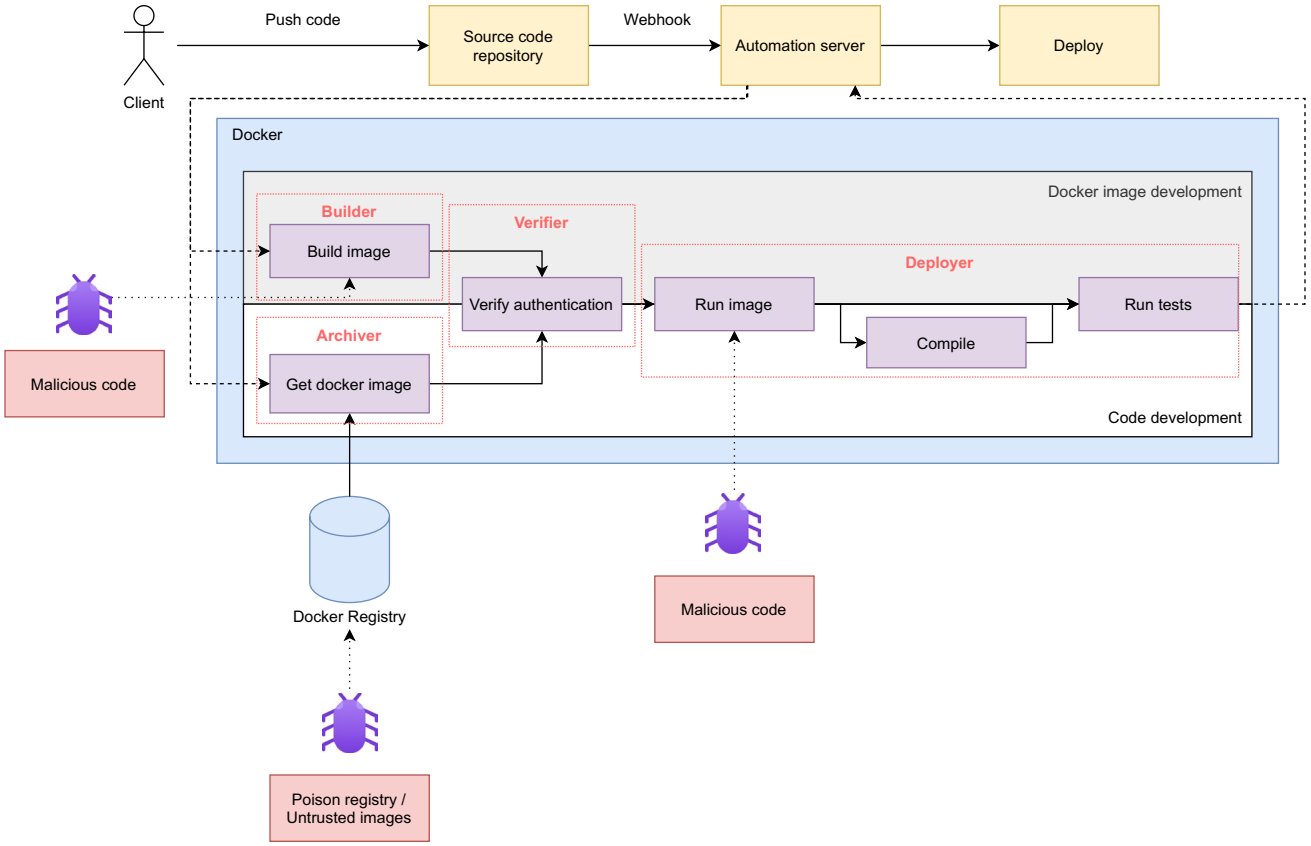

Fig. 1 Attacker scheme: Vulnerability points during container deployment

item 6, some of the tools go toward a specific commercial solution. As a result, there is a need to develop a tool like *SecDocker*.

Attacker Scheme

As mentioned in sect. "Background", containers are the target of different security threats or vulnerabilities. Therefore, a containerized environment—like those created with Docker—may have different potential attack vectors [18]: host OS, network or physical systems, source code repositories, image repositories or the very own containers. Securing these vectors is not a trivial task, but the contributions presented in this paper are framed towards the integrity of container images used by CI (or CD) pipelines.

In such cases, images are frequently used to ship a complete virtual environment where concrete actions from the CI workflow take place (e.g. build, test, run or deploy an application). Such workflow is scripted and usually automated by triggering a webhook from some version control system. But this approach makes pipelines unreliable so, to contribute to its hardening, the image generator component from the CI process (see the previous subsection) is an element that needs to be hardened somehow. Regarding this process and based on Bass et al. approach [1], it is possible to distinguish four components involved in it (see Fig. 1):

1. Builder. Builds a container image according to some specifications. This image comprises the virtual environment or workspace where some or all workflow actions will take place.
2. Verifier. Computes a checksum to verify the authentication of the image was just built.
3. Archiver. Stores the image in a registry or repository so it can be retrieved later.
4. Deployer. Deploys the image into a testing or production environment in order to execute the CI workflow or some of its scripted actions.

This study considers the last component to be one of the most important. The reason is that a correct configuration will minimize the impact of an issue in the previous three components. A container with no root or bounded CPU will guarantee minimal resource exploitation to the host machine. Thus, a runtime check for the detection of common security and configuration weaknesses against a compliance

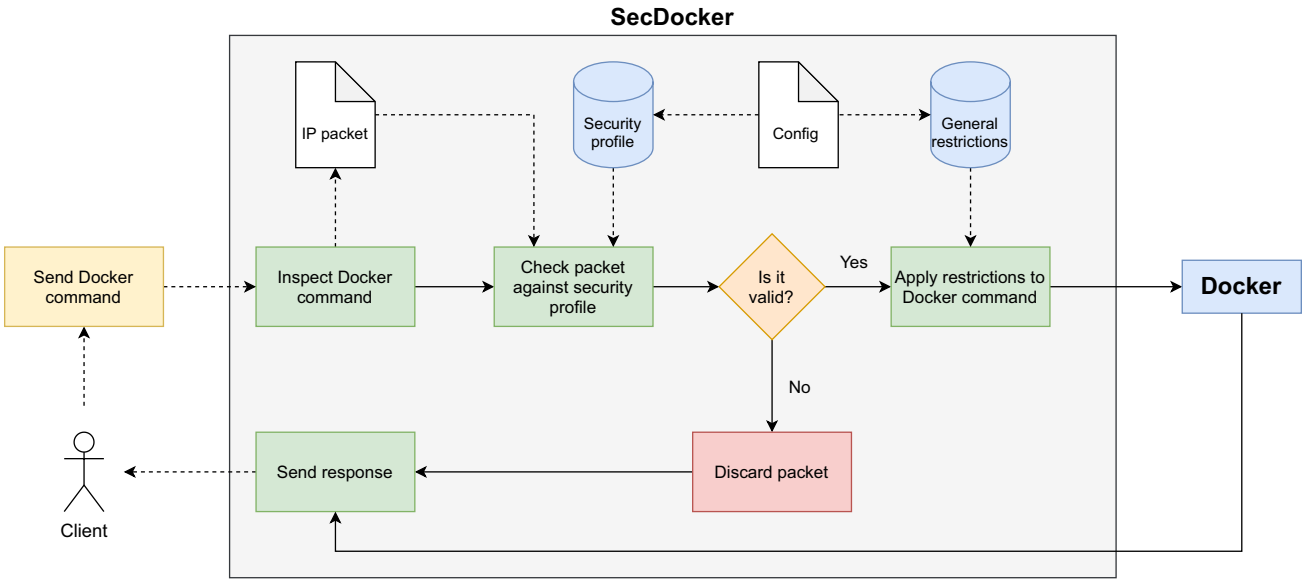

Fig. 2 *SecDocker* general workflow

configuration pattern defined by DevOps engineers seems to meet the requirements for production environments.

Proposed Solution

SecDocker is an application firewall for Docker. It must be noted that, nowadays, such firewalls are frequently used to control the traffic of web applications [21]; for instance, as a reverse HTTP proxy that decides whether a token requires to replace any suspicious parts found in requests [14]. Bearing this in mind, *SecDocker* shares the same purpose as any web application firewall: prevent users from performing dangerous or unexpected actions on the application.

Docker is commonly used in a local environment when configured and managed by end users. But for those Docker platforms set up in a system different from the one where commands are executed, network traffic is a topic to be discussed. In this context, it is important to highlight that the Docker CLI sends an HTTP request to the Docker daemon and the latter processes it and answers with the corresponding results.

Therefore and, broadly speaking, *SecDocker* filters TCP traffic and works by monitoring the Docker commands. Its main goal is to evaluate all the requests meant for the Docker daemon by standing between it and the user (see Fig. 2). This workflow can be described in four steps:

1. Send Docker command. The workflow starts when the end user sends a new HTTP request through a Docker command. As the CLI is configured to send commands to a Docker daemon located in a different system, the CLI just crafts the HTTP request and sends it to the daemon.
2. Inspect Docker command. Whenever a new HTTP request aiming for the run API endpoint reaches the firewall, the IP packet is intercepted, opened and the request parameters from the Docker command (i.e. ports request, user, image name, etc) are loaded from its data section. If IP packets are encrypted the same actions are applied, but in this case, *SecDocker* needs to be configured with the same TLS/SSL certificates used by the Docker daemon in order to be able to decrypt and inspect their content; otherwise, the IP packet will not even be intercepted by *SecDocker* as its contents cannot be read. Finally, if the request contains a command different than *run*, it simply forwards it to the Docker daemon.
3. Check packet against security profile. After the inspection, the request parameters are checked against a security profile (previously configured by the DevOps engineer) in order to prevent unauthorized actions coming from the container itself. This profile is part of a configuration file and contains a set of constraints for the parameters of the Docker command; for instance, the list of banned ports, forbidden mounted volumes or restricted container images. If at least one of the parameters in the packet contains a value present within the security profile, the packet is considered as "not valid". Hence, it is discarded and a new one is created and sent back as a response to the end user, notifying him about the use of a forbidden option.

4. Apply restrictions to Docker command. If the packet is valid (i.e. no matches were found in the security profile during the packet verification), *SecDocker* is able to append or modify the requested parameters to suit some general purpose restrictions for creating or running any container from the server hosting the Docker daemon. These restrictions (also previously specified by the DevOps engineer as part of the same configuration file containing the security profile), are meant to limit all containers with settings such as: memory or CPU usage limit, users forbidden to run containers or environment variables meant to omit. Having a single configuration file allows *SecDocker* to define an additional security layer in the server, ensuring that all containers run under the same settings. Once the restrictions have been applied, the packet is recreated and sent back to the Docker daemon to finally perform the requested action.

In addition, it should be noted that since *SecDocker* runs in parallel to Docker, this workflow is pseudo-transparent in terms of performance to commands other than `run`. This ensures that the tool acts as a web application firewall and only filters traffic from processes in isolated containers.

Software architecture

SecDocker is written in Go and is publicly available in GitHub.[1] It features a modular and extensible design composed of 5 components at its core:

1. *Security*. Performs validation against the user-supplied options.
2. *Config*. Loads user's information into the firewall in real-time.
3. *Docker*. Performs tasks related to how Docker processes information.
4. *HTTPServer*. Manages and performs actions against HTTP data (e.g. loading the body of the requests, crafting new requests/responses, etc.).
5. *TCPIntercept*. Handles packages at the TCP level, so the communication looks transparent for the end-user. It also maintains the communications and gathers data for the `HTTPServer` module. Additionally, it must be noted that this module is based on Trudy,[2] a transparent proxy that can modify and drop TCP traffic.

Its functionality can also be expanded by third-party applications thanks to a dedicated component named `Plugins`. For its basic workflow, *SecDocker* delegates some extra functionality to two plugins:

- Anchore.[3] Inspects, analyzes and applies user-defined acceptance policies.
- Notary.[4] Ensures the integrity of a trusted collection of Docker images.

Likewise, an accountability component based on logs is also included with *SecDocker*. This logging component relies on Logrus,[5] an external logger package for Go that provides structured logs.

Usage

As mentioned at the beginning of this section, *SecDocker* workflow involves routing TCP packages in a similar way to a firewall. In a nutshell, it listens to all incoming TCP traffic and only monitors those packets involving HTTP data and which body contains requests to the Docker Engine API (e.g. list containers, create containers, start a container, etc). Consequently, it should be placed in top of the server responsible for handling requests to Docker. This is done either to maintain the original destination port of the Docker daemon or to perform some alteration to redirect the traffic to the right port by applying some firewall rules so the traffic can be intercepted by *SecDocker*.

Once installed, a configuration file written in YAML is used to filter the HTTP data (see Listing 1). This file contains a list of plugins to be enabled, the location of the Docker daemon and a security profile. With regard to the latter, an aggregation of rules must be specified. These rules define a set of parameters that allow DevOps engineers to set up some security features related to the Docker image and its execution. On the one hand, there are restrictions or rules that forbid the use of specific parameters; that is, if a packet contains one parameter listed there, the packet will be dropped. On the other hand, there are general rules that apply to all requests; for example, if we want to restrict the amount of RAM to 1GB per container, we can set a rule to force it (as in Listing 1).

[1] https://github.com/uleroboticsgroup/Secdocker/.

[2] https://github.com/praetorian-inc/trudy/.

[3] https://anchore.com/.

[4] https://docs.docker.com/notary/getting_started/.

[5] https://github.com/sirupsen/logrus/.

Table 1 Configurable features supported by *SecDocker*

Feature type	Setting	Description
Docker options	Ports	Port number(s) used by the container
	Users	Username(s) or uid(s) running the container
	Mounts	Mounted volume(s) (file or directory) from the host filesystem
	Environment	List of environment variables (KEY=VAL) used within the container
	Security policies	Adds or drops Linux capabilities
	Images	Name(s) of container image(s) to monitor
	Privileged	Whether the containers has granted capabilities in its host machine
General restrictions	Memory	Maximum amount of memory usage
	CPU	Number of CPU resources
	User	Username or uid (in host) allowed to run the container
	Environment	Environment variables of the container

```
plugins:

dockerapi: "/var/run/docker.sock"

restrictions:
  ports:
    - 22
    - 25
  mounts:
    - /root
    - /
  users:
    - root
  environment:
    - USER=0
  images:
    - ubuntu:16.04
  privileged: true

general:
  memory: "1g"
  cpu: 0.25
  user: "1000"
  environment:
    - "MY_ENV=true"
```

Listing 1 Example of a *SecDocker* configuration file

In addition, Table 1 collects a list of the current available parameters supported by *SecDocker*. The definition of these features (or parameters) resembles those from the official Docker Compose tool, meaning that a minimal understanding of Docker parameters is required. With this in mind, writing a security profile like the one shown in Listing 1 is relatively easy. Thus, any DevOps engineer may create one attending to company policies, NIST suggestions for resourcing allocation [24] or even for performance as suggested by Tesfatsion, Klein and Scarfone [25].

Regarding its output, *SecDocker* starts to listen on port 8999 and logs all packets and their related data to the standard output by default. A separated log file is also created containing all the requests; whether they were allowed or not and why. Furthermore, any external plugins can have their logs to output their own results.

Empirical Validation

This section presents *SecDocker* software metrics and the results from three experiments that were conducted to assess its performance, its own workflow and also its role in the CI workflow.

Table 2 Statistics related to time taken to execute three different Docker commands (100 times each) when *SecDocker* is both enabled and disabled

	Time elapsed (secs)					
	Without SecDocker			With SecDocker		
	image ls	container ls	run	image ls	container ls	run
Mean	0.134	0.044	0.301	0.163	0.066	0.479
Median	0.130	0.040	0.300	0.160	0.070	0.485
Mode	0.130	0.040	0.300	0.160	0.060	0.490
Std. deviation	0.007	0.005	0.013	0.005	0.006	0.030
Minimum	0.120	0.030	0.270	0.150	0.060	0.410
Maximum	0.160	0.060	0.350	0.180	0.080	0.560
First quartile	0.130	0.040	0.290	0.160	0.060	0.460
Third quartile	0.140	0.050	0.310	0.170	0.070	0.500
Sum	13.380	4.370	30.120	16.320	6.570	47.910

Software Implementation

SecDocker has 1834 lines of code (LOC) distributed among the different functions of four files: `tcpintercept` (*tcpproxy.go*), `commandline` (*command.go*) `docker` (*security.go*) and `httpserver_test` (*server_test.go*). Moreover, a set of software metrics is presented to provide some sort of assessment to the tool implemented in this study. These metrics can be used to define its maintainability and code quality but also can give details about how easy is to debug, maintain or integrate new functionalities to it. They were measured against version v0.1-beta of the application and using SonarQube[6] and Golint[7] as code quality tools. Additionally, *SecDocker* has a total number of 35 test cases—aggregated in 13 test functions that are grouped by table-driven tests—, defining an 87% of test coverage. Lastly, and regarding code quality, Golint detects 31 issues (28 related to naming and comments and 3 to coding structures) while SonarQube detects only 12 code smells and no bugs, vulnerabilities nor security hotspots.

Experiment Description

Two experiments were carried out to both measure *SecDocker*'s performance and check its functionality. The experiments were conducted on two PCs connected to the same LAN. Both systems were configured using Elementary OS 5.1.7 and had different specifications: one with an Intel(R) i5-3570 CPU @ 3.40 GHz and 16.0 GB memory, and the other with an AMD Ryzen 5 3500U CPU @ 3.60 GHz and 8.0 GB memory. The first PC was used as a server for running Docker and *SecDocker* and the second as a client to connect to the latter and execute different Docker commands.

Performance Testing

The first experiment was carried out to evaluate *SecDocker*'s performance as timing behavior, that is, to run transparently from a running Docker server. The test consisted of running 100 times each of the following commands from the client's PC:

```
# docker image ls
# docker container ls
# docker run -d -p 1000:1000 --rm ubuntu:18.04
```

It must be noted that the purpose of the first two commands was to test the command overhead for the third one; that is, to measure the processing time required by the system prior to executing the Docker `run` command. To measure these times, the standard Unix `time` tool was used. That said, Table 2 summarizes the experiment results after its execution without and with *SecDocker*.

On the one hand, the mean times for running the `image ls` and the `container ls` commands without *SecDocker* in the server PC were 0.134 ± 0.007 s and 0.044 ± 0.005 s, respectively, with an interquartile range for both commands of 0.01 s. Likewise, the mean times for running those same commands with *SecDocker* in the same server PC was 0.163 ± 0.005 s and 0.066 ± 0.006 s, respectively, with an interquartile range of 0.01 s again for both commands. Due to the fact that the differences between the mean times are relatively low (0.029 s for `image ls` and 0.022 s for `container ls`) and the interquartile range does not change, it is possible to assert that *SecDocker*'s overhead effect is negligible to commands other than `run`.

On the other hand, the mean time for running the `run` command when *SecDocker* was disabled in the server PC was 0.301 ± 0.013 s, with an interquartile range of 0.02 s. Meanwhile, the mean time when *SecDocker* was enabled in the same server was 0.479 ± 0.030 s, with an interquartile

[6] https://www.sonarqube.org/.

[7] https://github.com/golang/lint/.

```
❯ docker run --privileged ubuntu:18.04
docker: Error response from daemon: Option forbidden.
See 'docker run --help'.
```

Fig. 3 *SecDocker* response when running the proposed Docker command

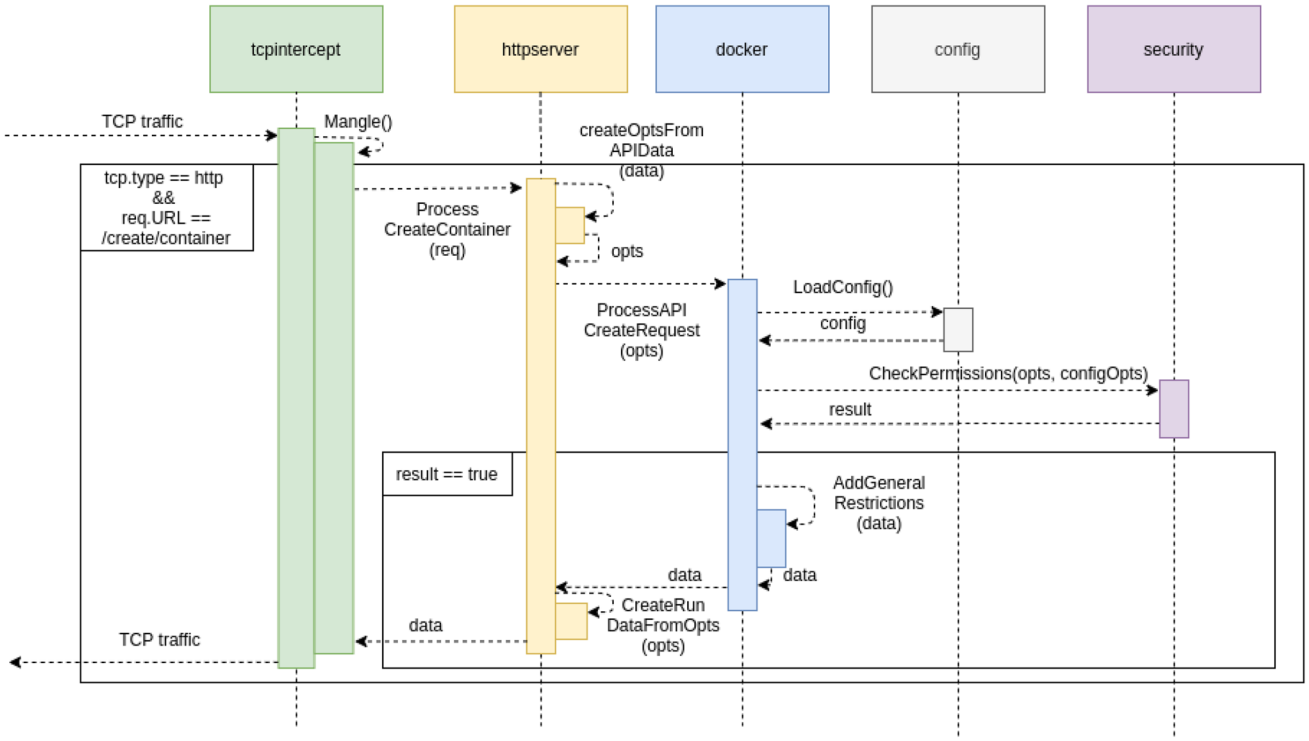

Fig. 4 Sequence diagram describing how *SecDocker* blocks a `docker run` command

range of 0.04 s. Since the differences are only 0.178 s between the mean times and 0.02 s between the interquartile ranges, it can be considered valid to state that, apparently, *SecDocker* runs transparently from Docker.

Functional Testing

The second experiment was carried out to test *SecDocker* functionality. This time, the goal was to send the following command from the client's PC to perform a hypothetical privilege escalation attack:

```
# docker run --privileged ubuntu:18.04
```

To prevent this potential threat, the server PC used the same configuration file shown in Listing 1; which includes the `privileged` option set to *true* in order to drop commands like the one previously mentioned.

Figure 3 shows that running the proposed command for this test fails as expected. From *SecDocker*'s point of view, the command is processed as represented in the sequence diagram shown in Fig. 4. When the HTTP request derived from the command arrives at *SecDocker*, it extracts all parameters and checks them against the security configuration loaded. In the test environment, the privileged option is met, so a response is sent to the user stating that it has a forbidden option.

SecDocker in the CI Flow

SecDocker is a standalone tool not meant to replace current state-of-the-art solutions. Instead, it is meant to run in parallel with such tools. Hence, its impact in the CI ecosystem needs to be discussed and, for such task, this section compares *SecDocker* to other tools, such as hadolint[8] or Docker `scan`[9].

To this end, an experiment that evaluates the system resources used by each of the above-mentioned tools was

[8] https://github.com/hadolint/hadolint/.

[9] https://docs.docker.com/engine/scan/.

Table 3 Statistics related to the performance of the time taken to run hadolint, `scan` and *SecDocker*100 times

	User process time (secs)		
	Hadolint	Scan	SecDocker
Mean	0.019	0.254	0.034
Median	0.020	0.255	0.030
Mode	0.020	0.270	0.030
Std. deviation	0.007	0.052	0.009
Minimum	0.000	0.130	0.010
Maximum	0.030	0.370	0.050
Sum	1.940	25.380	3.420
First quartile	0.020	0.210	0.030
Third quartile	0.020	0.290	0.040
	System process time (secs)		
	Hadolint	Scan	SecDocker
Mean	0.010	0.059	0.023
Median	0.010	0.060	0.020
Mode	0.010	0.060	0.020
Std. deviation	0.007	0.015	0.009
Minimum	0.000	0.030	0.010
Maximum	0.030	0.120	0.040
Sum	1.020	5.940	2.280
First quartile	0.010	0.050	0.020
Third quartile	0.010	0.070	0.030
	Time elapsed (secs)		
	Hadolint	Scan	SecDocker
Mean	1.145	4.165	0.485
Median	1.040	4.000	0.490
Mode	1.030	3.940	0.490
Std. Deviation	0.235	0.937	0.027
Minimum	0.870	3.740	0.410
Maximum	1.930	13.170	0.560
Sum	114.540	416.490	48.500
First quartile	0.970	3.930	0.470
Third quartile	1.290	4.190	0.500

carried out. The goal was to measure the time and cpu usage taken by each tool to execute 100 times. The Unix `dstat` tool was used for such task and, particularly, the following data were assessed: (1) user process time or amount of CPU time spent by the tool in user mode; (2) system process time or amount of CPU time spent by the tool in the kernel; (3) time elapsed or total time spent to finish each execution; and (4) percentage of CPU used during each execution.

Table 3 collects the user and system process times along with the time elapsed to execute each tool studied 100 times. First, in terms of user process time, the mean time for hadolint was 0.019 ± 0.007 s, for `scan` was 0.254 ± 0.052 s and for *SecDocker* was 0.034 ± 0.009. Similarly, their total time for executing 100 iterations of each tool was 1.940 s

Table 4 Statistics related to CPU percentage usage from executing 100 iterations of hadolint, `scan` and *SecDocker*

	CPU (%)		
	Hadolint	Scan	SecDocker
Mean	2.970	7.370	13.110
Median	3.000	7.000	13.000
Mode	3.000	8.000	13.000
Std. deviation	0.758	1.468	0.952
Minimum	1.000	2.000	11.000
Maximum	5.000	10.000	16.000
First quartile	2.000	6.750	13.000
Third quartile	3.000	8.000	14.000

for hadolint, 25.380 s for `scan` and 3.420 s for *SecDocker*. Their interquartile ranges are 0 s, 0.080 s and 0.010 s, respectively. These data suggest that `scan` is the tool which takes the longest time to execute, followed by *SecDocker* and then hadolint.

Second, with regard to system process time, the mean time for hadolint was 0.010 ± 0.007 s, for `scan` was 0.059 ± 0.015 s and for *SecDocker* was 0.023 ± 0.009. Likewise, the total times for running each tool were 1.020 s (hadolint), 5.940 s (`scan`) and 2.280 s (*SecDocker*). Their interquartile ranges are 0 s, 0.020 s and 0.010 s, respectively. That said, the data suggest once more that `scan` is the tool which takes the longest time to execute, followed by *SecDocker* and then hadolint.

Lastly, for the time elapsed, the mean times were: 1.145 ± 0.235 s for hadolint, 4.165 ± 0.937 s for `scan` and 0.485 ± 0.027 s for *SecDocker*. Also, the sum of time for hadolint was 114.540 s, for `scan` 416.490 s and for *SecDocker* 48.500 s; with interquartile ranges of 0.32 s, 0.26 s and 0.03 s, respectively. In view of the results obtained, *SecDocker* is the fastest tool followed by hadolint and then `scan`.

Continuing with the analysis, Table 4 shows the CPU percentage usage from executing the aforementioned tools. The mean percentages were $2.970 \pm 0.758\%$ for hadolint, $7.370 \pm 1.468\%$ for Docker scan and $13.110 \pm 0.952\%$ for *SecDocker*. Furthermore, their interquartile ranges were 1%, 1.25% and 1%, respectively. These results show that all tools have a low impact on the CPU, with *SecDocker* being the most "demanding" (its median is 13% versus the 3% from hadolint and the 8% from `scan`). However, this is due to being an application that runs in real time to filter all Docker's traffic.

Discussion

SecDocker is a tool meant to be used for a wide variety of members from CI and DevOps communities. Thus, this section explains what this paper has presented from two different points of view: research and software.

Research Perspective

This work makes the assumption that the RQ proposed in sect. "Research Question and Contribution" (see below) is appropriate, meaningful, and purposeful when facing cybersecurity issues during the CI workflow.

RQ: *Which are the mechanisms for avoiding and minimizing cybersecurity and misconfiguration issues in a CI container-based deployment system?*

As discussed in sect. "Attacker Scheme", four points are vulnerable to attacks during the container CI workflow: (1) when building a container image according to some specifications; (2) when verifying the authentication of the image just build; (3) when storing the image in a registry or repository; and (4) when deploying the image into some environment. *SecDocker* is a tool meant to secure the latter point by acting as an application firewall in order to prevent users from dangerous or unexpected actions. However, its design also allows to indirectly securitise other vulnerable points of the CI workflow. Thanks to its design, not only *SecDocker*'s functionality can be easily expanded with external tools like Anchore and Notary, but it also can complement Docker's security at the same time as tools like hadolint or Docker scan without impacting their performance.

Moreover, previous sections mentioned some of the common strategies used to solve CI issues associated to this RQ, mainly focused on the Image Generator step. Hence, it is possible to present a subset of scenarios for identifying *SecDocker* validity. Some of the answers extracted from this work are:

- Even though is commonly accepted that the CI workflow relies on DevOps engineers experience, it is necessary to avoid unaware behaviors using a transparent and automated mechanism such as *SecDocker*.
- *SecDocker*, which works as an application firewall, has no impact compared with regular Docker use.
- Using a deployment engine based on YAML configuration files minimizes unaware deployments, simplifies repetitively tasks and makes more comprehensible automated monitoring process.
- *SecDocker* allows to track and audit all commands sent to Docker. Its logging capabilities could be used as a tracking system having in mind the timestamp.

These points summarize the goal of the work presented and, at the same time, provide a concise and clear way to answer the RQ posed.

Software Perspective

SecDocker offers many potential benefits regarding the CI process. Some of these are:

- Publicly available. It is an open source tool released under the MIT license. The tool is presented in a way that makes deployment easier for the DevOps community. It is written in Go, a popular programming language, and offers a middleware solution for Docker, a mainstream containerization solution.
- Flexibility, scalability and security. It should be noticeable from DevOps and CI engineers that the current

release of *SecDocker* brings simplicity to CI and CD processes. Likewise, relying on different configuration files, makes easier to define all the requirements needed for an infrastructure and thus prevent misconfiguration issues related to last minute fixes, to reduce performance issues associated with lack of hardware resources or even software incompatibilities between versions.
- Installation costs. The process of downloading, compiling and deploying is performed with exactly three commands, as indicated in the documentation available in the GitHub repository.
- Assurance. *SecDocker* users do not need to consider the trade-off between speed and certainty. Results presented in Sect. "Performance Testing" show similar performance and negligible differences when using Docker with or without *SecDocker*.

However, *SecDocker* also has certain shortcomings, including:

- The solution is only applicable to the deployment part of a CI/CD workflow; it does not cover previous steps. However, *SecDocker* architecture favors the use of plugins (like Anchore and Notary) in order to support such features.
- *SecDocker* works as an application proxy. Each time a client makes a Docker request, *SecDocker* only intercepts it and checks its IP and port (which need to be the ones associated to Docker). Currently, *SecDocker* does not route these packages, which, on the other hand, would add a new level of security allowing to hide connection elements to the user.
- The image provided in *SecDocker*'s configuration file is not validated. More precisely, *SecDocker* does not check if the image provided by the Docker server is legit.
- Unusual launch parameters (like those related to DNS or Input/Output) are also not checked by *SecDocker*.
- Once a container is running, *SecDocker* does not perform additional actions to test whether such container is executing under the defined specifications or is being used for the intended purpose.

Conclusion

In conclusion, it is important to harden CI workflow. We knew from previous experiences that corporations refuse to deploy new tools given the cost associated (training, deployment, etc). Thus, the idea of providing a firewall app that allows maintaining the current workflow was a key for designing *SecDocker*.

It is critical for every DevOps engineer to secure as much as possible their containers platforms. By developing *SecDocker*, we have learned the possible threats of a CI system running containers, in particular the mainstream tool Docker. Performing a close analysis of the user input hardens the systems to minimize the possible attack surface and the capabilities the users can access to.

Acknowledgements This work has been partially funded by the "Universidad de León-Instituto Nacional de Ciberseguridad (INCIBE) Convention Framework about *Detection of new threats and unknown patterns*" (Spain).

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Bass L, Holz R, Rimba P, Tran AB, Zhu L. Securing a deployment pipeline. In: 2015 IEEE/ACM 3rd International workshop on release engineering; 2015, pp. 4–7 https://doi.org/10.1109/RELENG.2015.11.
2. Berkovich S, Kam J, Wurster G. UBCIS: Ultimate benchmark for container image scanning. In: 13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20). USENIX Association (2020). https://www.usenix.org/conference/cset20/presentation/berkovich. Available online March, 2021.
3. Bernstein D. Containers and cloud: from LXC to docker to kubernetes. IEEE Cloud Comput. 2014;1(3):81–4. https://doi.org/10.1109/MCC.2014.51.
4. Boettiger C. An introduction to docker for reproducible research. ACM SIGOPS Oper Syst Rev. 2015;49(1):71–9. https://doi.org/10.1145/2723872.2723882.
5. Bou Ghantous G, Gill A. Devops: concepts, practices, tools, benefits and challenges. In: Proceedings of the 21st Pacific-Asia conference on information systems (PACIS2017). AIS Electronic Library (AISeL) 2017
6. Chelladhurai J, Chelliah PR, Kumar SA. Securing Docker containers from Denial of Service (DoS) attacks. In: 2016 IEEE International Conference on Services Computing (SCC), pp. 856–859. IEEE 2016. https://doi.org/10.1109/SCC.2016.123.
7. Combe T, Martin A, Di Pietro R. To docker or not to docker: a security perspective. IEEE Cloud Comput. 2016;3(5):54–62. https://doi.org/10.1109/MCC.2016.100.

8. Fitzgerald B, Stol KJ. Continuous software engineering: a roadmap and agenda. J Syst Softw. 2017;123:176–89. https://doi.org/10.1016/j.jss.2015.06.063.
9. Goyal P. CIS docker community edition benchmark. PDF. https://www.cisecurity.org/benchmark/docker. Available online March, 2021.
10. Hilton M, Nelson N, Tunnell T, Marinov D, Dig D. Trade-offs in continuous integration: assurance, security, and flexibility. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017;197–207 https://doi.org/10.1145/3106237.3106270.
11. Humble J, Farley D. Continuous delivery: reliable software releases through build, test, and deployment automation. London: Pearson Education; 2010.
12. Jabbari R, bin Ali N, Petersen K, Tanveer B. What is DevOps? a systematic mapping study on definitions and practices. In: Proceedings of the Scientific Workshop Proceedings of XP2016, 2016;1–11 https://doi.org/10.1145/2962695.2962707.
13. Kang H, Le M, Tao S. Container and microservice driven design for cloud infrastructure DevOps. In: 2016 IEEE International Conference on Cloud Engineering (IC2E), pp. 202–211. IEEE 2016. https://doi.org/10.1109/IC2E.2016.26.
14. Krueger T, Gehl C, Rieck K, Laskov P. Tokdoc: A self-healing web application firewall. In: Proceedings of the 2010 ACM symposium on applied computing, SAC '10, p. 1846–1853. Association for computing machinery, New York, NY, USA 2010. https://doi.org/10.1145/1774088.1774480.
15. Lam T, Chaillan N, Ranks P. DoD enterprise DevSecOps reference design version 1.0. Tech. rep., Department of Defense, Chief information officer (2019). https://dodcio.defense.gov/Portals/0/Documents/DoDEnterprise DevSecOps Reference Design v1.0_Public Release.pdf. Accessed Mar 2021
16. Leite L, Rocha C, Kon F, Milojicic D, Meirelles P. A survey of devops concepts and challenges. ACM Comput Surv. 2019. https://doi.org/10.1145/3359981.
17. MacDonald N, Head I. DevSecOps: how to seamlessly integrate security into DevOps. Tech rep Gartner Tech Rep 2016
18. Martin A, Raponi S, Combe TRD. Docker ecosystem-vulnerability analysis. Comput Commun. 2018;122:30–43. https://doi.org/10.1016/j.comcom.2018.03.011.
19. Merkel D. Docker: lightweight linux containers for consistent development and deployment. Linux J. 2014;2014(239):2.
20. Pahl C. Containerization and the PaaS cloud. IEEE Cloud Comput. 2015;2(3):24–31. https://doi.org/10.1109/MCC.2015.51.
21. Prandl S, Lazarescu M, Pham DS. A study of web application firewall solutions. In: Jajoda S, Mazumdar C, editors. Information systems security. Cham: Springer; 2015. p. 501–10.
22. Shahin M, Babar MA, Zhu L. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access. 2017;5:3909–43. https://doi.org/10.1109/ACCESS.2017.2685629.
23. Smeds J, Nybom K, Porres I. DevOps: A definition and perceived adoption impediments. In: International conference on agile software development. Springer; 2015. pp 166–177 https://doi.org/10.1007/978-3-319-18612-2_14.
24. Souppaya M, Morello J, Scarfone K. Application container security guide. National Institute of Standards and Technology: Tech Rep; 2017.
25. Tesfatsion SK, Klein C, Tordsson J. Virtualization techniques compared: performance, resource, and power usage overheads in clouds. In: Proceedings of the 2018 ACM/SPEC international conference on performance engineering; 2018. pp. 145–156
26. Turnbull J. The Docker book: containerization is the new virtualization. James Turnbull 2014
27. Vase T. Integrating Docker to a Continuous Delivery pipeline: a pragmatic approach. Master's thesis, University of Jyväskylän (2016). https://jyx.jyu.fi/handle/123456789/52756. Accessed Mar 2021