SN Computer Science (2023) 4:748
https://doi.org/10.1007/s42979-023-02186-1

ORIGINAL RESEARCH

# Access Security Policy Generation for Containers as a Cloud Service

Hui Zhu[1] · Christian Gehrmann[1] · Paula Roth[1]

Received: 4 October 2021 / Accepted: 24 July 2023
© The Author(s) 2023

**Abstract**

The rapid development of containerization technology comes with remarkable benefits for developers and operation teams. Container solutions allow building very flexible software infrastructures. Although lots of efforts have been devoted to enhancing containerization security, containerized environments still have a huge attack surface. Completely avoiding severe security issues have so far not been possible to achieve. However, the security problems due to vulnerabilities in for instance kernels, can be largely reduced if the container privileges are as restricted as possible. Mandatory access control is an efficient way to achieve this using for instance AppArmor. As manual AppArmor generation is tedious and error prone, automatic generation of protection profile is necessary. In previous research, a new tool for tight AppArmor profile generation was presented. In this paper we show how, in a system setting, such tool can be combined with container service testing, to provide a cloud based container service for automatic AppArmore profile generation. We present solutions for profile generation both for centrally collected and generated container logs and for log collection through a local agent. To evaluate the effectiveness of the profile generation service, we enable it on a widely used containerized web service to generate profiles and test them with real-world attacks. We generate an exploit database with 11 exploits harmful to the tested web service. These exploits are sifted from the 56 exploits of Exploit-db targeting the tested web service's software. We launch these exploits on the web service protected by the profile. The results show that the proposed profile generation service improves the test web service's overall security a lot compared to using the default Docker security profile. This together with the very user friendly and robust principle for setting up and running the service, clearly indicates that the approach is an important step for improving container security in real deployments.

**Keywords** Security-as-a-service · Docker · Container · AppArmor

This article is part of the topical collection "Cloud Computing and Services Science" guest edited by Donald Ferguson, Markus Helfert and Claus Pahl.

✉ Christian Gehrmann
christian.gehrmann@eit.lth.se

Hui Zhu
hui.zhu@eit.lth.se

Paula Roth
dic15pro@student.lu.se

[1] Department of Electrical and Information Technology, Lund University, Box 118, SE-221 00 Lund, Sweden

Published online: 28 September 2023

## Introduction

Full virtualization was often used in earlier cloud deployments. Currently, containerization is the dominating solution when building software services, both in large and small installations. According to a survey by Dell commissioned Aberdeen Strategy and Research (ASR) in 2021, 50% of all applications were containerized[1] and the adoption is still increasing. However while enjoying the significant benefits brought by containerization technology such as portability, efficiency, and agility, several security issues also arise by the kernel-sharing property of containerization [1].

To meet these issues, many different behavior-based solutions have appeared in the industry. The goal of these solutions is to monitor a container in real-time with respect to the container state changes and external interfaces interactions. This we here refer to as *runtime behavior*. The different container security products' can monitor the runtime behavior and detect malicious activities by using rule-based or machine-learning-based approaches. For example, the TwistLock runtime offers both static analyses and machine-learning-based behavioral monitoring [2]. The TwistLock monitoring and profiling defense work on four levels: the file system [3], the processes, the system calls, and the network [4]. Similarly, Aqua's runtime security for Docker restricts privileges for files, executables, and OS resources based on a machine-learned behavioral profile to ensure that only necessary privileges are given to the container.[2] NeuVector,[3] StackRox[4] and Sysdig[5] also provide similar products. Another solution in the same direction is a British Telecommunication patent for software container profiling, which can generate runtime profile for the container in execution [5]. Worth mentioning in this respect are also two open-source projects for runtime behavioral monitoring of containers: Falco[6] and Dagda[7]. Falco is a cloud-native runtime security tool that can detect and alert on any behavior that involves making system calls such as running a shell inside a container or unexpected read of sensitive files. Dagda adds build-time analysis on top of Falco's runtime analysis. The academic works in the area are not as many. Some researchers propose novel design ideas but lacking implementation details and experimental results. In the work of [6], a new security layer with extra security features on top of the container architecture is proposed to secure the cloud container environment. The proposed layer has two features: a Container Security Profile (CSP) and the Most Privileged Container (MPC) feature. CSP is responsible for access control enforcement. It describes the minimum resource requirements, runtime behavior, and extra privileges for the container. The MPC is monitoring the system and detects any attempt to act against assigned permissions. The MPC alerts the container engine when suspicious processes are detected. This in turn allows the engine to halt a potentially dangerous process.

A different strategy to increased security that can be used as a stand-alone or together with a container behavior analysis tool, is to *restrict* the access privileges of a container to a minimum with Mandatory Access Control (MAC) [7]. By using this, the possibility for a malicious container to utilize a platform vulnerability is reduced. In line with this direction, previous work have address the issue of how to generate suitable MAC profile for a particular container. Two early approaches in this direction were LicShield [8] and DockerSec [9]. More recently, as an extension and considerable enhancement of these tools, LicSec [10] was presented. LicSec is a command-line tool called which utilizes Linux tracing tools: SystemTap[8] and Auditd[9] to trace the behavior of the container runtime and generates customized a Linux security module AppArmor[10] profile. Docker container security is significantly enhanced by restricting the privileges of capabilities, network accesses, file accesses, and executables based on an automatically generated AppArmor profile. The tool was experimentally evaluated and proved to be efficient to real-world attacks, especially against several privilege escalation attacks.

LicSec requires the profiling tool to be run *together* with the container, which in turn needs to be triggered under extensive test conditions, such that as much as possible of the container behavior can be catch by the tracing tools. This is needed to avoid generating a too restrictive profile that otherwise will give false blockings. However, this does not work well for real deployment scenarios were a profile better is generated under extensive test or in early deployment. To address this, we in this paper investigate how to realize a cloud profile generation service, which works under realistic deployment scenarios. We have designed both a service that allows the user to upload a container to a profile generation environment as a cloud service and a principle where the administrator sets up a local client connected to the Docker engine, which is collecting behavioral data. This data can then be fed (in real-time) to the policy generation. Our novel cloud tool for AppArmor profile generation, which utilizes

1 https://www.dell.com/en-us/blog/container-adoption-trends-why-how-and-where/.

2 https://blog.aquasec.com/topic/runtime-security.

3 https://neuvector.com/products/container-security/.

4 https://www.stackrox.com/use-cases/threat-detection/.

5 https://sysdig.com/products/kubernetes-security/runtime-security/.

6 https://github.com/falcosecurity/falco.

7 https://github.com/eliasgranderubio/dagda#monitoring-running-containers-for-detecting-anomalous-activities.

8 https://sourceware.org/systemtap/.

9 https://linux.die.net/man/8/auditd.

10 https://www.openhub.net/p/apparmor/.

**Table 1** A summary of category for Docker official images

| Category | Sub-category | Amount | Percentage |
|---|---|---|---|
| Database | Database and Storage System | 15 | 30% |
| Application service | Service and Tool | 14 | 28% |
| Application infrastructure | Web Server | 5 | 20% |
| | Reverse Proxy | 3 | |
| | Frontend | 1 | |
| | Service discovery | 1 | |
| Programming | Programming Language | 8 | 16% |
| Base image | Operating System | 5 | 10% |

Lic-Sec, allows dynamic and automatic AppArmor profile generation.

Furthermore, we also wanted to evaluate, how strong the resulting profiles are with respect to security and we investigate the strength of a set of profiles generated for some typical containers. The strength can be verify if a set of known Docker vulnerabilities, are applicable to the container running with the generate profile or not. In particular, we evaluate the strength of the profiles by benchmarking against running the sample services with default Docker AppArmor profiles.

In summary, we make the following contributions:

- We propose, design, and implement a novel, dynamic, AppArmor profile generator as a cloud service, both for local and central behavior monitoring.
- We evaluate the efficiency of the profile generation service by testing, on widely used containerized web services, the generated profile's strength against real-world exploits.

The rest of this paper is organized as follows. In "Background", we give a background description of Lic-Sec and the classification for containers. In "Problem Description", we formulate the main research problem, i.e., the design goal of the profile generator cloud service, and the evaluation goal of the performance of the generated profile. In "Cloud Service Approach", we introduce the cloud service approaches of the profile generator in detail, including the central service-based approach and the client agent-based approach. In "Implementation", we describe the implementation details of those two approaches. In "Experimental Setup", we introduce how the microservice used in the evaluation is designed and how the exploit database is generated. In "Evaluation", the profile generator cloud service's primary evaluation results are presented, and a detailed analysis of the results is given. In "Related Work", we present and discuss related work. In "Conclusion", we conclude this research and identify future work.

## Background

As we discussed in Sect. "Introduction", in this paper we enhance and extend the previous work done on an AppArmor profile generation tool called Lic-Sec [10]. In this section, we give a brief introduction to this tool. In addition, we also evaluate the strength of the generated profile. In order to this, we need a container classification framework, which we also introduce in this section. These classifications will be used throughout the paper.

### Lic-Sec

Lic-Sec is a command-line tool that can automatically generate AppArmor profiles based on container runtime behaviors. Lic-Sec combines LiCShield [8] and Docker-sec [9], both of which enhance container security by applying customized AppArmor policies. Lic-Sec has two primary mechanisms, including tracing and profile generation. SystemTap collects all kernel operations while Auditd collects mount operations, capability operations, and network operations. This information is processed by the rules generator engine, and eventually, the AppArmor profile is generated. The Lic-Sec rules generator engine is the entity responsible for the actual AppArmor policy generation, to generate the set of policies used by AppArmor to do the MAC enforcement on the container. by Rules generated by Lic-Sec include capabilities rules, network access rules, pivot root rules, link rules, file access rules, mount rules, and execution rules.

### Container Classification

A container can support almost any type of application. To design a microservice evaluated by our new AppArmor profile tool, we have searched and classified the major container use cases. We explored the top 50 most popular Docker official images from Docker hub[11] in 2020 and classified them based on their labels. The final classification result is displayed in Table 1. The total amount of images in the

[11] https://hub.docker.com/search?q= &type=image.

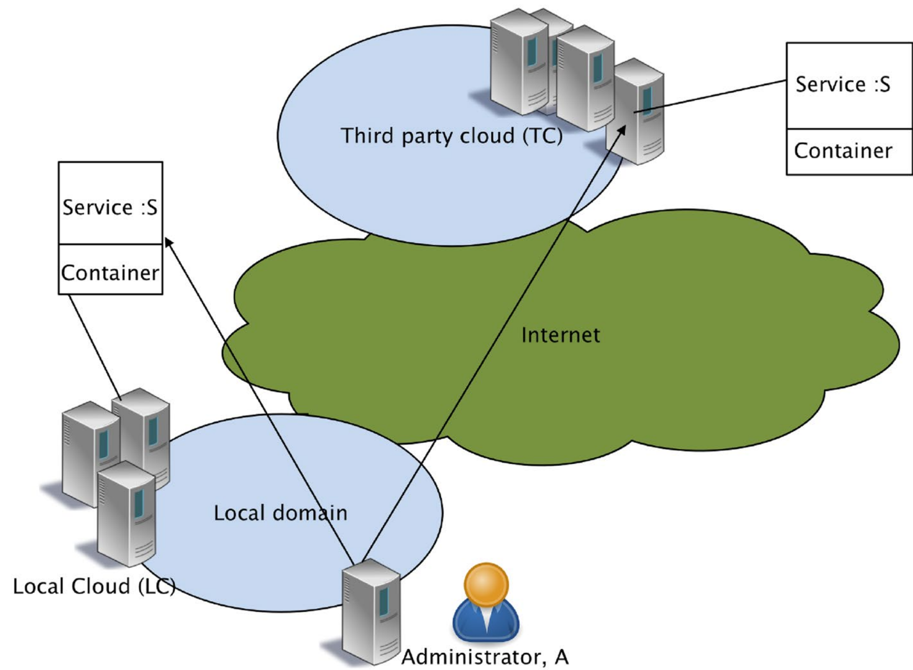


**Fig. 1** Scenario Overview

table is larger than 50 since some images are labeled with multiple categories. From the table we can conclude that the containerized database accounts for the largest proportion, followed by the containerized application services and infrastructures. In the category containerized application infrastructure, 50% constitute web servers Consequently, containerization is widely applied to databases and server-side applications. Other major use cases are containerizing services, programming languages, and operating systems.

## Problem Description

We are considering the scenario in Fig. 1 where an administrator, A, wants to launch an arbitrary service, S, on a container. The service can be launched on a local container infrastructure or a third-party cloud infrastructure utilized by the administrator. In this scenario, the administrator is responsible for preparing S and running it on a suitable container platform. To achieve this goal, the administrator can leverage different protection schemes to enhance the container platform's security. One such scheme is based on AppArmor security architecture, using MAC to protect the container from external threats. However, MAC is complicated to configure manually even if the administrator has good knowledge of the microservices since the MAC rules are directly related to the Linux kernel. Furthermore, even if the administrator can configure it, the rules' scope is still hard to define since it cannot be too strict to blocking the microservice's essential functions nor too generous to open up for attacks on containers.

Therefore, the aim of this research is to provide a cloud service to generate tailored AppArmor profiles for the administrator in order to protect different microservices in the most user-friendly way. We also want to evaluate the efficiency of the generated profiles in a real production environment. To accomplish these goals, we want to solve the following two main problems: (1) find a user-friendly cloud service to generate a tailored AppArmor profile for an arbitrary microservice automatically; (2) find a suitable methodology and test framework for evaluating the strengths of the profiles generated by the cloud service.

## Cloud Service Approach

Next, we describe our cloud access profile generation service system solution. The core of the solution, is that a MAC profile generation is offered as a security service for container administrators. The MAC profile generator is based on Lic-Sec, which has been described in Sect. "Lic-Sec". The proposed profile generation service offloads the administrator of a container service the burden of setting up a protection profile generation environment. We design two approaches for the administrator to use the security service. The primary difference between the two approaches is how the container's behavior data are collected. In the first approach, we set up a profile generation environment in the cloud for the administrator. The administrator's containerized service will be running on the cloud while the containers' behaviors are collected on the cloud as well. However, in the second approach, we separate the behavior collection

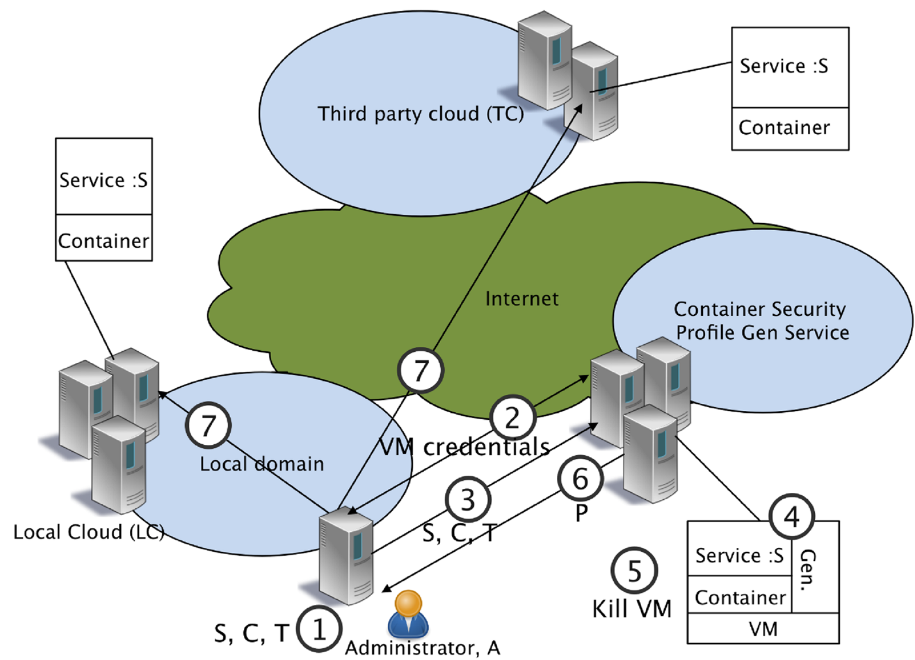


**Fig. 2** Central Service-based Profile Generator as a Cloud Service Solution Overview

module and apply it as a distributed client-side log collection agent that works on the administrator's local environment and continuously collects behavioral data from local containerized services. Those data will be securely sent to the central service for generating the container profile. As a result, we name these two approaches central service-based profile generation and client agent-based profile generation. Below, we describe the details of both approaches.

## Central Service-Based Profile Generation

The central as well as the agent-based approach (see Sect. "Client Agent-based Profile Generation") assume that an administrator of a Docker has an account at a profile cloud service. In the central approach, the administrator must prepare the complete Docker images *including* a complete test suit for verifying the image prior to using the service. The more complete the test suite, the better profile in terms of robustness against false blocking, will be the end result. The end-user is required to uploads images, configurations, and test suites to the cloud central service. The central service automates profile generation and provides the user with the ready-to-use profile(s). Below, we give a step-by-step description of the solution (see also overview Fig. 2):

1. An administrator, A, prepares a new service, S, together with configuration information, C, including parameters such as the mounted volumes, the open ports, the needed capabilities, etc., as well as a test suite, T, for S. S will be deployed on a container on local or third-party cloud resources as a new service with the given configurations. T consists of cases for testing all functions of S.
2. A is assumed to have an agreement with a container security provider and set up a secure connection (authenticated, confidentiality and integrity protected) with these providers. The provider evaluates if the requester has an agreement with the provider. If this is the case, the provider launches a new Virtual Machine (VM), including container launch profile and MAC profile generator on an internal cloud resource. Login credentials for the VM running container services are created on the internal resources, and a URL, as well as credentials for accessing the VM, are returned to the administrator machine.
3. A uses the credentials received in step 2) to make a secure connection to the new VM created in the profile generation service cloud. Using the received credentials, A logs in to the VM and uploads S, C, and T to the VM.
4. A script on the VM launches container(s) with the uploaded S and the given C. The functions of S are tested automatically during the tracing period by running T. Then, the script generates a MAC protection profile based on the trace records. T is rerun with profile enforced to verify no function of S is blocked by the profile. If the verification fails, the service provider informs A of the failure and discontinues this service.
5. The profile generated in step 4 that is successfully verified is temporarily stored, and the VM is killed, and all its data is wiped out from memory.
6. P is returned to A by sending a profile download link to A.

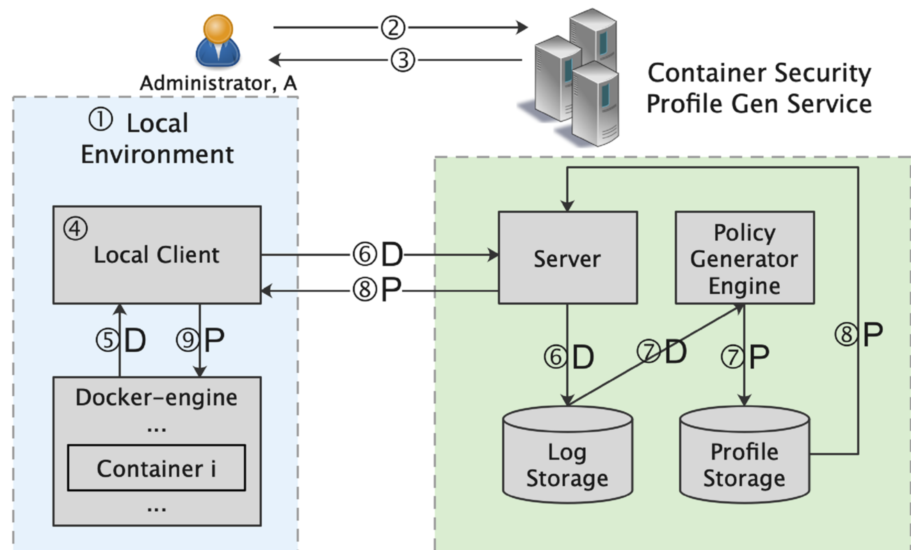


**Fig. 3** Client Agent-based Profile Generator as a Cloud Service Solution Overview

7. A takes the received MAC profile, P, and launches S on a local or remote container service with the profile applied.

## Client Agent-Based Profile Generation

In the client agent approach, the container administrator sets up a client in the local environment. The client takes charge of behavior data collection and transmission, and profile verification. This means that a local agent must be installed on the cloud environment used by the administrator. This we refer to as the "Local Environment". Different from the central design, this means that the agent also needs security configurations to allow secure information transfer between the local client and the central profile generation system. We here have used a simple configuration with a client secret and basic authentication[12] in combination with standard server certificates for server authentication according to the TLS standard.[13] However, the solutions work equally fine using a client certificate or a pre-shared key for instance. The workflow for this approach is shown in Fig. 3. We explain each step in detail below:

1. An administrator, A, has prepared a local microservice environment in which containers run without any security policies. The local environment is required to support AppArmor. It is also required that A is able to test all features of the microservice in the local environment.
2. A registers to the cloud service, meanwhile, a client secret is generated for A and saved in the cloud backend as belonging to A.
3. A downloads the client from the cloud service and fetches the client secret.
4. A sets up the local client in the microservice environment and provides it with the client secret.
5. The client enables the training feature, continuously collecting the behavioral data D generated by the local containers. During this period, A is supposed to test all the functionalities of the local microservice so that the client is able to record the full containers' behavior.
6. The client verifies the server identity and proceeds to validate itself to the cloud service through the client secret. Then the behavioral data D generated in step 5 is securely transmitted to the server by the client. The server keeps D in the log storage.
7. The profile generator engine reads D from the storage, transforms it to the container profile(s) P(s), and saves P(s) in the profile storage.
8. The server fetches P based on A's client ID, and securely sends P back to the local client in A's environment.
9. The client runs P locally and provides A with feedback on the results of the applying the profile, i.e. if the service runs normally or if legal actions are blocked by AppArmor. If the profile is judge to be too restrictive, the client repeats step 5 to 9 until the profile is considered stable.

## Implementation

Here we introduce the implementation details of the two previously mentioned approaches. Citycloud[14] located in Sweden is used as the internal cloud platform of the profile

[12] https://datatracker.ietf.org/doc/html/rfc7617.

[13] https://datatracker.ietf.org/doc/html/rfc8446.

[14] https://citycloud.se/.

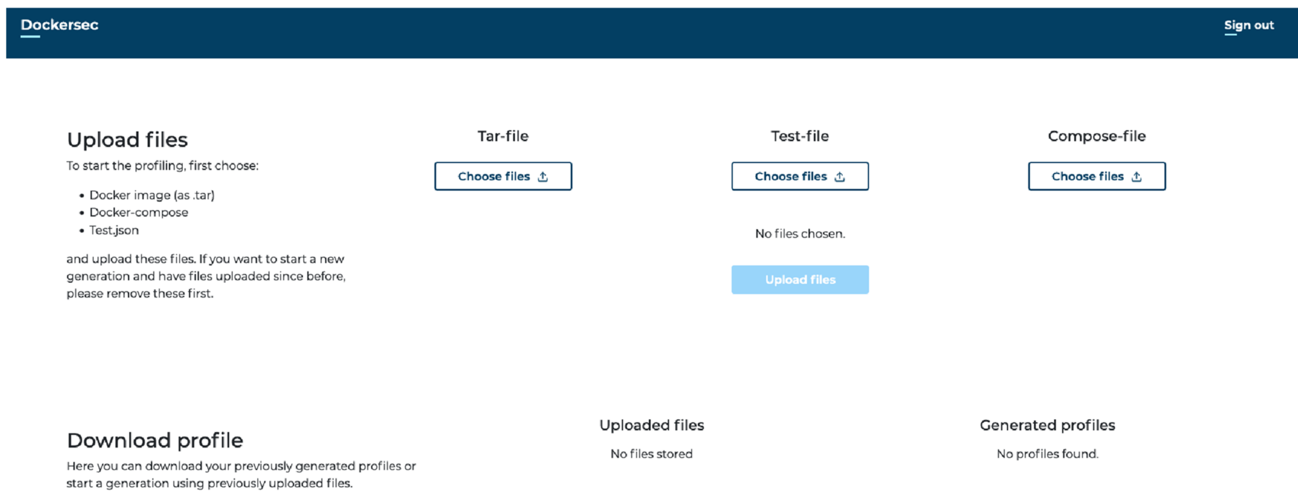


**Fig. 4** The user interface for the profile generation service

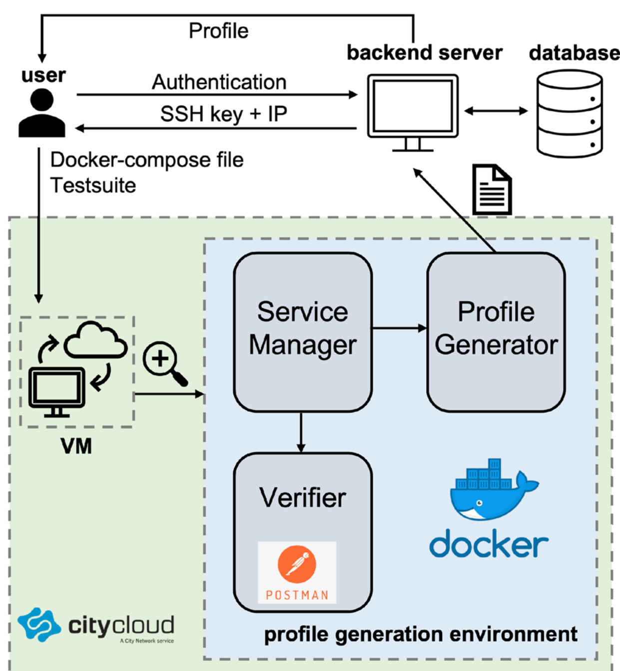


**Fig. 5** The implementation framework of central service-based profile generation service

generation service. Figure 4 shows the end-user interface for the service. The interface is a web graphical user interface where the user can upload images, configuration files, and test suites, replace those files and download ready-to-use generated profiles.

## Central Service-Based Profile Generation

The implementation framework is displayed in Fig. 5. A backend server and a data storage with contract users' information are running on the cloud to provide three main functions: user authentication, service launch, and profile fetch. The detailed description for each function is as follows:

**User Authentication:** the user is authenticated by the backend server (username and password). After successful authentication, the backend server sets up a VM with a ready-to-use profile generation environment on Citycloud. The profile generation environment includes the following pre-installed components:

- **profile generator:** we use the Lic-Sec tool described in Sect. "Lic-Sec" for tracing behaviors and generating AppArmor profile for the uploaded service.
- **service manager:** this is a bash script responsible for discovering newly uploaded service, launching Docker service, and enabling the profile generator and verifier, which automates the profile generation and verification.
- **verifier:** we use Newman[15] as the verifier, which is a command-line collection runner for Postman.[16] It is responsible for running RESTful API tests in the test suite uploaded by the user. The test suite is a JSON file and easy to run with a simple command: $*newman run* < *testsuite.json* >.
- **Docker environment:** the Docker CLI, the Docker daemon, and the docker-compose package constitute the Docker environment, which runs the uploaded service in Docker containers.

[15] https://www.npmjs.com/package/newman.

[16] https://www.postman.com/.

**Service Launch:** to use the profile generation service, the user needs to prepare the service and configurations for running the service in Docker containers and a test suite script created by the service(s) owner. The script tests all the service's functionalities (see also the discussion on test suit preparation below). For the configurations, the user can directly use the Docker Compose file. For the test suite, the user can use the JSON file exported from Postman Collection. Once the service, configurations, and test suite are uploaded, the service manager inside the VM runs the service with docker-compose and starts the profile generator. Simultaneously, the service manager enables the training period and calls the verifier to run the test suite. After the training phase is over, and the profile is successfully generated, the service manager calls the verifier again with the profile enforced. Hence, the two significant phases of service launch are training and verification. Below, we discuss them in more detail.

- **Training:** the profile generator uses Lic-Sec to trace the runtime behavior of the service, which has been described in Sect. "Lic-Sec". At the same time, Newman runs the test suite, and all the functionalities of the service are tested.
- **Verification:** verification ensures that the generated profile does not block any functionality of the service. The service manager first enforces the generated profile and then calls Newman to rerun the test suite. If any test case fails, the service manager restarts the profile generation service and regenerates the profile. If the verification fails three times, the service manager stops the profile generator and sends an error message to the backend server. The backend server then provides a secure link for users to check the failed cases. The users can ask for technical supports from the profile generation service provider.

**Profile Fetch:** once the verification is successful, the profile generated inside the VM is uploaded to the backend server immediately by the service manager. Upon receiving the profile, the backend server requests Citycloud to kill this VM completely. Meanwhile, the backend server temporarily saves the profile locally and provides a secure link for users to download the profile.

**Test Suite Preparation:** postman is a popular API client that has been widely used by developers to create and save HTTP/s requests, read and verify their responses. The Postman Collection is a built-in function that includes a set of pre-built requests. Newman automates the running and test of a Postman Collection. Users create a new collection by merely clicking +*NewCollection* in the Postman GUI and then import all pre-built requests against the same service into this new collection. To run the collection with Newman, users should export the collection as a JSON file. This file is the test suite that will be run by the verifier automatically during the training period. The required permissions and file operations by those requests are traced to generate the profile. If the test suite misses any request, corresponding permissions, and file operations required to handle the request will not be generated in the profile. Therefore, the profile's effectiveness dramatically relies on the test suite's quality, and the generated profile only fits the service that has been trained. It is the users' responsibility to guarantee that the test suite covers all functions of the service. We consider it not an extra effort since an end-to-end test of a service is typically required before publishing the service independently of our cloud profile generation service.

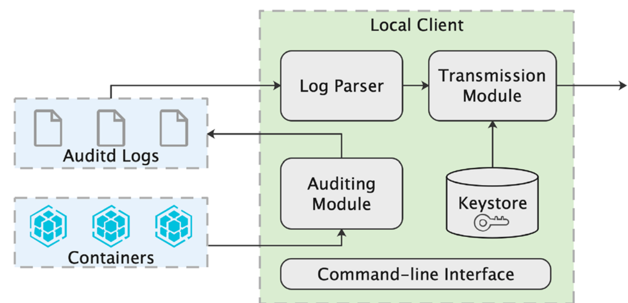


**Fig. 6** The implementation details for the local client

## Client Agent-Based Profile Generation

In order to realize the agent-based profile generation service, we divided Lic-Sec into two new modules. The first module is an auditing module and the second module is the actual profile generator. We accommodate the auditing module to work as a local client. The profile generator engine module is integrated with the cloud service to process the behavioral data transmitted from the client. Below, we give further details of how we realized the local client, the secure log transmission, and the profile generation parts of the solution.

**Local Client:** this is a CLI (command-line interface) tool written using Java Spring Boot.[17] In a future version of our solution, the tool should be possible to download from the cloud service when the user has registered the service. In our current solution, we need to manually fetch the tool. The client is responsible for the trace of containers, the parsing, and the secure transmission of the logs. The client consists of five components which are displayed in Fig. 6. The command-line interface is the entry for end-users to communicate with the client. The auditing module has been changed from Lic-Sec to allow the Auditd service to log AppArmor permission check events for containers. The log

[17] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/.

parser collects, formats, and packages the local logs. The transmission module sets up a secure connection with the server using the secret stored in the Keystore. When the client has been set up and running in the users' hosts, they can enable the training feature of the auditing module through CLI to trace containers and generate the Auditd logs. Those logs are saved locally as */var/log/audit/audit.log*. The log parser then reads these logs from the local files, formats them as JSON, and collects the formatted logs during the same training period into the same log file. This file is sent to the server securely by the transmission module according to the principle below.

**Secure Data Transmission:** mutual authentication based on TLS[18] (mTLS) is utilized to encrypt logs sent from the client to the server. Therefore, all communication is over HTTPS (neither client nor server serves "plain" HTTP requests). The local client utilizes a client secret for authentication as well as a certificate for verification of the server. When the user registers to the cloud service, a client secret is generated and saved in the cloud backend. Users can obtain their client secret from the cloud service and provide the secret to the client before running it. The secret is then stored in a Keystore in PKCS12 format and used for establishing a secure transmission.

**Profile Generation:** the profile generating engine module in Lic-Sec has been modified to process the logs sent by the client. The engine performs efficient analysis utilizing Pandas,[19] one of Python's most popular data science modules for big data processing. Those logs are saved in the log storage with the unique client ID as the key. The engine monitors the newly created log files in the storage and transforms them into AppArmor profiles. To be more specific, the following three structures are seen in the Auditd logs, which correspond to file access events, network events, and capability events, respectively:

- structure 1: [AppArmor] [operation] [info*] [profile] [name] [pid] [comm] [requested_mask] [fsuid] [ouid] [target*]
- structure 2: [AppArmor] [operation] [profile] [pid] [comm] [laddr*] [lport*] [faddr*] [fport*] [family] [sock_type] [protocol] [requested_mask] [addr*]
- structure 3: [AppArmor] [operation] [profile] [pid] [comm] [capability] [capname]

The fields with asterisks (*) are optional. In file execution events, the inheritance fallback will be recorded in field *info*, which shows the approach to permission inheritance from the executed binary, and the target profile will be recorded in field *target*, which is the profile for the child process of the executed binary. Fields *faddr* and *fport* in some network events record foreign addresses and ports, while fields *laddr* and *lport* record local addresses and ports.

Since there would be logs for several containers in the same log file, the engine first classifies the logs based on the value in field *profile*. Then, it further categorizes each container's logs by event type. For the capability-related logs, it should have *capable* as the value in the field *operation*. For network-related logs, the field *sock_type* must exist. While searching for the file access-related logs, different requirements apply depending on the existence of the optional fields mentioned earlier. For execution-related operation on files, it requires the existence of fields *fsuid*, *info*, and *target*. Conversely, non-execution operations on files can be searched out by the unique existence of the field *fsuid*. Those sorted logs are further converted to dataframes[20] for efficient processing. Dataframe represents a table of data with rows and columns. In this scenario, each row of the table corresponds to a log, and each column corresponds to a field in the log.

At this point, the engine starts to generate rules of different types. The patterns for producing the rules are listed below. The engine extracts the values and combines them based on the patterns.

- capability rule: capability [capname]
- network rule: network [family] [sock_type] [protocol]
- file access rule:
  - non-execution operations: [name] [requested_mask]
  - execution operations: [name] [info]

In the end, the engine may generate several different profiles. It saves those profiles in the profile storage with the key being the name in the "[profile name]-[client ID]" format and the value being the corresponding AppArmor profile. In this way, it is easy to locate the specific profile based on user and container information.

## Experimental Setup

We have experimentally evaluated the profile generator architecture and design. In this section we describe the details of the implementations and the evaluations. We start by discussing the selection and deployment of the microservice used in our evaluation. Then we describe how we have collected and classified the exploits targeting this microservice, and finally, we explain how the tests were executed.

**Microservice selection and deployment:** we decide to use a web service stack to build the evaluated microservice.

[18] https://datatracker.ietf.org/doc/html/rfc8446.

[19] https://pandas.pydata.org/.

[20] https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html.

**Table 2** Exploit Database Collection

| Object | EDB-ID | CVE-ID | Category |
|---|---|---|---|
| Redis | 48272 | N/A | Execute code Gain information |
| | 47195 | N/A | Execute code Gain information |
| | 40678 | CVE-2016-6663 | Gain Privilege |
| MySQL | 40360 | CVE-2016-6662 | Execute code Gain Privilege |
| | 39867 | CVE-2015-4870 | DoS |
| | N/A | CVE-2012-2122 | Bypass Gain information |
| PHP | 47553 48182 | CVE-2019-11043 | Execute code |
| Linux | 48052 | CVE-2019-18634 | Gain Privilege |
| Docker engine | N/A | CVE-2020-13401 | Gain information DoS |
| phpMyAdmin | 40185 | CVE-2016-5734 | Execute code |
| | 44496 | CVE-2018-10188 | Execute code |

**Table 3** Evaluation Result Overview

| Categories | Software | | | | | |
|---|---|---|---|---|---|---|
| | Redis | MySQL | PHP | Linux | Docker Engine | phpMyAdmin |
| Bypass (Doc/Svc[1]) | \ | 1/1 | \ | \ | \ | \ |
| Gain Privilege (Inside Container) (Doc/Svc[1]) | \ | 2/0 | \ | 1/1 | \ | \ |
| DoS (Doc/Svc[1] | \ | 1/1 | \ | \ | 1/0 | \ |
| Gain Information (Doc/Svc[1]) | 2/0 | 1/1 | \ | \ | 1/0 | \ |
| Execute Code (Doc/Svc[1]) | 2/0 | 1/0 | 1/0 | \ | \ | 2/1 |

[1] "Doc" denotes the number of exploits execute successfully on containers launched with Docker, and "Svc" denotes the number of exploits execute successfully on containers launched with the profile generation service

This stack compiles software that enables the creating and running of complex websites on any computer. It usually includes a web server, a database system, an underlying operating system, and supports for particular programming languages. It is very suitable to be used as the underlying stack for building the containerized service since databases, server-side applications, programming languages, and operating systems are commonly deployed as microservices in Docker containers as concluded in Sect. "Container Classification". Web services are also among the most widely deployed container services. The stack we used for the evaluation includes a backend service, a reverse proxy service, and a database service, each of which runs in a separate container. We used a simple secret management system to test the set-up. The chosen service provides four APIs and safe persistent storage for secret owners to save and manage their secrets. To be more specific, the four APIs are $POST/path_1$ for creating secret and securely saving it to the database, $DELETE/path_1 < sec_{ID} >$ and $PUT/path_1 < sec_{ID} >$ for deleting and updating a specific secret with $sec_{ID}$, and $GET/path_2 < sec_{ID} >$ for fetching a specific secret with $sec_{ID}$.

**Exploit database collection and classification:** we used the exploit collection and classification method suggested in [11]. According to this methodology, we first generated a universe exploit database by collecting the latest 100 exploits of each category from Exploit-db.[21] Then we filtered out the exploits which may probably fail on the container platform and used a two-dimensional method for classifying the final set. We generated the final exploit dataset and classified the exploits based on the method discussed above but modified it to suit the study's evaluation goal. We implement the method from [10] to obtain the exploits which were effective on the evaluated microservice discussed before. We first generated the initial universe set of exploits by searching out the exploits that mainly target the microservice's software. Based on this set, we filtered out exploits that can be defended by default Docker security mechanisms

[21] https://www.exploit-db.com.

by analyzing the exploit codes and launching the exploits in the Docker containers with default security configurations. Eventually, we obtained the final exploit dataset with 11 exploits published after 2016 out of 56 exploits, which were harmful to the containerized web service. We classified these exploits using the targeting object and its impact. The exploit details and their categories are shown in Table 2.

**Test setup:** the microservice was set up on a host running the Linux distribution Ubuntu 18.04.5 LTS with kernel version 4.15.0-72-generic. This Linux version was chosen to guarantee that the host is vulnerable to the Linux vulnerabilities in the selected exploit collection. Docker 19.03.1-ce was used for the microservice. This version was released on 25th July 2019 and supported Linux kernel security mechanisms, including Capability, Seccomp, and MAC. We implemented the Redis and MySQL database services. While implementing MySQL, we also deployed phpMyAdmin as the administrator. Nginx was implemented as the reverse proxy. PHP was used for the backend service.

## Evaluation

Here we present the evaluation results. We start by summarizing the overall results, and then we make a detailed analysis of the successful and failed defenses, respectively.

### Test Results Overview

The evaluation results are listed in Table 3. The results indicate that, first, among all the rules generated by the cloud service, the file access rules play a much more significant role in defending exploits than the other rules. Second, the AppArmor profile-based container protection scheme is more effective against attacks with a high level of sophistication, which requires many file manipulations than the simple attacks, which directly exploit targets' innate flaws with limited privileges in the profile. We will explain it in detail by analyzing the attacking principle of the exploits, the defending principle of the enforced profiles, and the reasons for the failed defenses in the following subsections. It should be noted that limits exist for the evaluation: first, the test profile is generated based on the designed microservice discussed in Sect. "Experimental Setup". It gives the least privileges for running the service without blocking any functionality of this service only. Therefore, the exploits defended in this evaluation setup may not be defended anymore in another setup. Second, the generated profile cannot remediate the vulnerability but *prevent* attacks exploiting the vulnerability.

### Successful Defenses

In total, the generated profile successfully defends 7 out of 11 exploits. Among these defenses, 6 defenses are due to the restriction of permissions to file resources, and only 1 defense is due to the lack of specific capability.

**Redis**: two exploits targeting Redis are proved to be vulnerable to the tested microservice. These exploits take advantage of an unauthorized access vulnerability of Redis version 4.x and 5.x. It uses the Master-Slave replication to load remote modules from a Rogue Redis server to a targeted Redis server. It executes arbitrary commands on the target.[22] Successful launch of the exploit requires to create a malicious exploit module written by the attacker in the Redis server's '/data' directory. After loading the module, the attacker can execute arbitrary commands. The exploit can be launched with the default security mechanism since the file access rules for '/data' directory is quite generous with no restrictions. However, the exploits are successfully defended by the enforced profile. Since the profile only grants 'read' permission to '/data' directory, no files can be created inside this directory.

**MySQL**: two exploits ( EDB-ID-40678[23] and EDB-ID-40360)[24] aiming to gain privilege inside the container are successfully defended by the generated profile. These two privilege escalation exploits take advantage of two critical vulnerabilities (CVE-2016-6662[25] and CVE-2016-6663)[26] in Oracle MySQL. The former one is a race condition that allows local users with certain permissions to gain privileges. The latter creates arbitrary configurations and bypasses certain protection mechanisms to perform arbitrary code execution with root privileges. The successful launch of EDB-40678 needs to create a table named 'exploit_table' in directory '/tmp/mysql_privesc_exploit'. Since the profile does not grant any 'write' permission to this directory, the launch of the exploit fails. Similarly, to launch EDB-40360, the attacker must write to the file 'poctable.TRG' in directory '/var/lib/mysql/demo,' which also requires 'write' permission to the directory and the file. The profile defends the exploit since there is no rule giving such permissions to the directory and the file.

**PHP**: there is one attack targeting PHP-fpm exploiting CVE-2019-11043,[27] which is a bug in PHP-fpm with specific configurations. It allows a malicious web user to get code execution. We used an open tool to reproduce the

[22] https://2018.zeronights.ru/wp-content/uploads/materials/15-redis-post-exploitation.pdf.

[23] https://www.exploit-db.com/exploits/40678.

[24] https://www.exploit-db.com/exploits/40360

[25] https://nvd.nist.gov/vuln/detail/CVE-2016-6662.

[26] https://nvd.nist.gov/vuln/detail/CVE-2016-6663.

[27] https://nvd.nist.gov/vuln/detail/CVE-2019-11043.

vulnerability of this tool.[28] A web shell is written in the background of PHP-fpm, and any command can be executed by appending it to all PHP scripts. This attack cannot be performed with the profile in force since the exploit needs 'write' permission to directory '/tmp' to create new files in this directory, which is not granted in the profile. The reason is that the evaluated microservice does not provide an API for users to upload files to the server. Consequently, no permissions are granted to the directory '/tmp'.

**Docker Engine**: a vulnerability, CVE-2020-13401,[29] is discovered in Docker Engine before 19.03.11. An attacker inside a container with the CAP_NET_RAW capability can craft IPv6 router advertisements to obtain sensitive information or cause a denial of service. The enforced profile perfectly defends this attack since the profile discards the CAP_NET_RAW capability.

**phpMyAdmin**: CVE-2016-5734[30] is an issue of phpMyAdmin which may allow remote attackers to execute arbitrary PHP code via a crafted string. The attack is written in Python and uses the function 'system()' to execute command after exploiting. This function's call needs the execution permission of '/bin/dash' to prompt a terminal. The enforced profile successfully defends this attack since it denies the execution of '/bin/dash.'

### Failed Defenses

In total, the generated profile fails to defend 4 out of 11 exploits. The attacks we could not prevent are generally not very complicated and do not rely on any specific capability or network access.

**MySQL**: Two exploits are targeting on MySQL that cannot be defended by the profile. One is a DoS attack exploiting vulnerability CVE-2015-4870[31] to crash the MySQL server by passing a subquery to function PROCEDURE ANALYSE(). The attack does not require any extra capability to launch. The required network access is only 'network inet stream', which is also necessary for running the MySQL database. Regarding the file accesses, the attack needs 'read' permission to the directory '/var/lib/mysql/mysql', which has been granted by the profile as it is needed to run the service.

The other uses vulnerability CVE-2012-2122[32] to log in to a MySQL server without knowing the correct password. The vulnerability comes from the incorrect handling of the return value of the memcmp function, which is an innate flaw of the software. Hence, the AppArmor profile will not help here. The first attack's impact is more severe than the second one since it completely disrupts the database service. For the second attack, even if the attacker bypasses authentication and logs in as an authenticated user, his/her behavior is still restricted by the enforced profile.

**Linux**: CVE-2019-18634[33] is a bug in Sudo before 1.8.26. Pwfeed-back option is used to provide visual feedback while inputting passwords with sudo. The option is disabled by default, but in some systems, users can trigger a stack-based buffer overflow in the privileged sudo process if this option is enabled. The stack overflow may allow unprivileged users to escalate to the root account.[34] The enforced profile fails to defend this attack since overflowing the buffer does not require extra file manipulation or extra capabilities. However, the attack's impact is limited since the attacker gets root privilege only inside the compromised container. The profile is still effective to the container so that the attacker is still under supervision.

**phpMyAdmin**: CVE-2018-10188[35] is a Cross-Site Request Forgery issue in phpMyAdmin 4.8.0, which allows an attacker to execute arbitrary SQL statements. The vulnerability comes from the failure in 'sql.php' script to properly verify the source of an HTTP request, which is also an innate flaw of the software. Similarly, the profile privileges are enough to launch the attack, which leads to the failed defense. The impact is relatively high since 'write' and 'read' permissions generally should be granted to ensure the regular operation of a database's essential functions; the attacker is unfortunately still able to drop, read or modify an existing database even if the profile is enforced.

## Related Work

There are some researches addressing profiling to enhance runtime security for containerization environment. In the work of [12], a security control map, including rate limit, memory limit, and session limit, as well as a malware detection system with profiling, is proposed to harden the security of runtime containers. All of the limit thresholds in this control map are derived from lab experiments and customer use case scenarios. The malware detection system is responsible for detecting malware behavior events, conveying semantic information about malicious behaviors, and predicting malware intentions. Based on the intentions, corresponding security policies are created automatically. The proposed control map is experimentally evaluated to improve container security significantly, especially when the attacker is inside the container. The main difference compared to our work is that this security control map is profiling the

[28] https://github.com/neex/phuip-fpizdam.

[29] https://nvd.nist.gov/vuln/detail/CVE-2020-13401.

[30] https://nvd.nist.gov/vuln/detail/CVE-2016-5734.

[31] https://nvd.nist.gov/vuln/detail/CVE-2015-4870.

[32] https://nvd.nist.gov/vuln/detail/CVE-2012-2122.

[33] https://nvd.nist.gov/vuln/detail/CVE-2019-18634.

[34] https://www.sudo.ws/alerts/pwfeedback.html.

[35] https://nvd.nist.gov/vuln/detail/CVE-2018-10188.

malware behavior but not the container runtime behavior. Hence, the created security policies will only protect the container from malware attacks that have been detected by the malware detection system and no other attacks.

Many commercial products are providing container runtime profiling, as mentioned in Sect. "Introduction". In the academic area, LiCShield [8] and Docker-sec [9] mentioned in Sect. "Lic-Sec" are two such solutions. Both aim to secure Docker containers through their whole life-cycle by automatically generating AppArmor profiles based on container runtime behavior profiling. The main difference is that Docker-sec uses Auditd as the tracing tool and generates capability rules and network access rules, while LiCShield uses SystemTap and generates rules other than the ones generated by Docker-sec such as file access rules and mount rules. However, both are command-line tools to be used locally and do not provide full dynamic profiling with verification for the target application. LicSec [10] was created as a continuation of LicShield and DockerSec. As we have explained above, this paper is built upon LiSec but extend it to build a full cloud system profile generation service and also evaluate the security performance of the created profiles provided by the generator.

Apart from solutions based on profiling, researchers are exploring other ways to enhance container security. One direction is to apply customized LSM modules. Bacis et al. propose a solution that binds SELinux policies with Docker container images by adding SELinux policy module to the Dockerfile. In this way, containerized processes are protected by pre-defined SELinux policies [13]. This approach requires the system administrator to have good knowledge of the service running inside the containers to define the most suitable SELinux policy. Consequently, it is not an automatic process. [14] propose the design of security namespace, which is a kernel abstraction that enables containers to utilize virtualization of the whole Linux kernel security framework to achieve autonomous per-container security control rather than relying on the system administrator to enforce the security control from the host. The experimental results show that security Namespaces can solve several container security problems with an acceptable performance overhead. An architecture called DIVE (Docker Integrity Verification Engine) is proposed by [15] to support integrity verification and remote attestation of Docker containers. DIVE relies on a modified version of IMA (Integrity Measurement Architecture) [16], and OpenAttestation, a well-known tool for attestation of cloud services. DIVE can detect any specific compromised container or hosting system and request to rebuild this single container and report to the manager.

Another direction is to protect containers from the kernel layer by providing a secure framework or wrapper to run Docker containers. Charliecloud, which is a security framework based on the Linux user and mount namespaces, is proposed by [17] to run industry-standard Docker containers without privileged operations. Charliecloud can defend against most security risks such as bypass of file and directory permissions and chroot escape. A secure wrapper called Socker is described by [18] for running Docker containers on Slurm and other similar queuing systems. Socker bounds the resource usage of any container by the number of resources assigned by Slurm to avoid resource hijacking. Furthermore, Socker enforces the submitting user instead of the root user to execute on containers to avoid privileged operations.

Similar client agent-based approaches have been used in certain studies to enhance container security. Generally, containers are distributively deployed and centrally managed by an orchestration platform such as Kubernetes. As a result, client agent-based approaches fit perfectly in this scenario. KubAnomaly [19] is such a system that utilizes an agent service to collect monitor logs from Docker-based containers through Sysdig[36] and Falco,[37] and sends these monitor log data back to the center for anomaly detection based on neural network techniques. The service monitors the container behavior based on a designed event list including 4 categories of system calls (file I/O, network I/O, scheduler, and memory). In [20], an ADS (anomaly detection system) is designed to detect and diagnose anomalies in microservices. It employs a monitoring module to collect the real-time performance data of containers such as CPU metrics, memory metrics, and network metrics. The agent is also utilized in [21] to collect the monitoring data including container performance metrics, host metrics, and workloads. Those data are sent to a real-time data storage for detection of anomalies and identification of possible causes of the anomalies through Hidden Markov Models. Besides the agent-based solutions, container monitoring based on third-party tools is another major way of detecting container anomalies and monitoring container integrity. In the work of [22], Prometheus,[38] an open-source monitoring and alerting tool, is employed to help microservice administrators to catch and predict container anomalies earlier. Falco is used in [23] as the system level monitor for containers to monitor the system calls. And a novel mechanism is proposed to filter out expected system calls and detect abnormal mutations to avoid false alarms in container integrity monitoring.

Besides proposing general security solutions for containers, many different research works focus on proposing container security countermeasure or algorithm against a particular attack category, which includes special investigations on some common attacks such as DoS attacks [24], application level attacks [25] and covert channels attacks [26], as well as some attacks with severe impacts such as

[36] https://sysdig.com/.

[37] https://falco.org/.

[38] https://prometheus.io/.

container escape attacks [27] and attacks from the underlying compromised higher-privileged system software such as the OS kernel and the hypervisor [28]. In the work of [24], a three-tier protection mechanism is applied to defend against DoS attacks. The mechanism is designed with memory limit assignment, memory reservation assignment, and default memory value setting to limit the container's resource consumption. Regarding the application-level attacks, [25] propose DATS, a system to run web containerized applications that require data-access heavy in shared folders. The system enforces non-interference across containers of data accessing and can mitigate data-disclosure vulnerabilities. Covert channel attacks against Docker containers are analyzed by [26]. They identify different types of covert channel attacks in Docker and propose solutions to prevent them by configuring Docker security mechanisms. They also emphasize that deploying a full-fledged SELinux or AppArmor security policy is essential to protect containers' security perimeters. [27] make a thorough investigation of Docker escape attacks and discover that a successful escape would create different Namespaces. Therefore, they propose a defense based on Namespaces status inspection, and once a different Namespaces tag is detected, the affiliated process is killed immediately, and the malicious user is tracked. The test results show that this defense can effectively prevent some real-world attacks. SCONE is proposed by [28], which is a secure container environment for Docker utilizing Intel Software Guard eXtension (SGX) [29] for running Linux applications in secure containers.

Some researches aim to provide secure connections for Docker containers. In the work of [30] and [31], both of them propose solutions to build secure and persistent connectivities between containers. The work of Secure Cloud proposed by [30] is realized with the support of Intel's SGX. While the SynAPTIC architecture from [31] is based on the standard host identity protocol (HIP). Cilium[39] is open-source software for securing the network connectivity between containerized application services.

## Conclusion

In this paper, we have proposed a secure cloud service to generate runtime AppArmor profiles for Docker containers. The cloud service is user-friendly and offloads the administrator of a container service the burden of setting up a protection profile generation environment. We have provided both a solution where the administrator of the container uploads the complete container including a test-suit to our profiling tool, as well as a version where the container executes at the administrator domain and sends access logs to the cloud profiling generator. We evaluated the approach by running a set of typical microservices on the cloud profile generator solution. We manually collected 11 most relevant real-world exploits from Exploit-db, which target the selected microservice's software. Even if the number of exploits is not very large, it still gives us a good view of our approach's efficiency compared to the strength of the default Docker profile. The results show that the profile successfully defends 7 out of 11 exploits not covered by the default profile, a considerable improvement based on the evaluation set-up. By analyzing the defending principles, we found that the profile is more efficient against complicated exploits that require many file manipulations. The results also indicate that among all kinds of rules generated in the profile, the file access rules play a much more significant role in defending exploits than other rules.

It is left to future work to compare our profile generator cloud service with other commercial products mentioned in Section "Introduction" to get a comprehensive understanding of the proposed service's strengths and weaknesses.

**Acknowledgements** Work supported by framework grant RIT17-0032 from the Swedish Foundation for Strategic Research as well as the EU H2020 project CloudiFacturing under grant 768892.

**Funding** Open access funding provided by Lund University.

**Data availability** The profiles generated for this research study are available on https://github.com/kikoashin/kubesec. The profiling service have since the research study was concluded, been further developed and transferred to a new start-up company, https://quritis.com/, which offer free of charge AppArmor container profile test generation.

## Declarations

**Conflict of interest** On behalf of all authors, the corresponding author states that there is no conflict of interest.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

## References

1. Casalicchio E, Iannucci S. The state-of-the-art in container technologies: application, orchestration and security. Concurrency and Computation: Practice and Experience. 2020;5668.
2. Stopel D, Levin L, Yankovich L. Profiling of container images and enforcing security policies respective thereof. Google Patents. US Patent 10,586,042 (2020).

[39] https://github.com/cilium/cilium.

3. Levin L, Stopel D, Yanay E. Filesystem action profiling of containers and security enforcement. Google Patents. US Patent 10,664,590 (2020).
4. Levin L, Stopel D, Yanay E. Networking-based profiling of containers and security enforcement. Google Patents. US Patent 10,599,833 (2020).
5. Daniel J, El-Moussa F. Software container profiling. Google Patents. US Patent App. 16/300,169 (2019).
6. Sarkale VV, Rad P, Lee W. Secure cloud container: Runtime behavior monitoring using most privileged container (mpc). In: 2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud), IEEE; 2017. p. 351–356.
7. Stallings W, Brown L. Computer security: principles and practice. 3rd ed. USA: Prentice Hall Press; 2014.
8. Mattetti M, Shulman-Peleg A, Allouche Y, Corradi A, Dolev S, Foschini L. Securing the infrastructure and the workloads of linux containers. In: 2015 IEEE conference on communications and network security (CNS), IEEE; 2015. p. 559–567.
9. Loukidis-Andreou F, Giannakopoulos I, Doka K, Koziris N. Docker-sec: A fully automated container security enhancement mechanism. In: 2018 IEEE 38th international conference on distributed computing systems (ICDCS), IEEE; 2018. p. 1561–1564.
10. Zhu H, Gehrmann C. Lic-sec: An enhanced apparmor docker security profile generator. J Inform Secur Appl. 2021;61.
11. Lin X, Lei L, Wang Y, Jing J, Sun K, Zhou Q. A measurement study on linux container security: Attacks and countermeasures. In: Proceedings of the 34th annual computer security applications conference, ACM; 2018. p. 418–429.
12. Pothula DR, Kumar KM, Kumar S. Run time container security hardening using a proposed model of security control map. In: 2019 Global Conference for Advancement in Technology (GCAT), IEEE; 2019. p. 1–6.
13. Bacis E, Mutti S, Capelli S, Paraboschi S. Dockerpolicymodules: mandatory access control for docker containers. In: 2015 IEEE conference on communications and network security (CNS), IEEE; 2015. p. 749–750.
14. Sun Y, Safford D, Zohar M, Pendarakis D, Gu Z, Jaeger T. Security namespace: making linux security frameworks available to containers. In: 27th {USENIX} security symposium ({USENIX} security 18), 2018. p. 1423–1439.
15. De Benedictis M, Lioy A. Integrity verification of docker containers for a lightweight cloud environment. Future Generat Comput Syst. 2019;97:236–46.
16. Sailer R, Zhang X, Jaeger T, Van Doorn L. Design and implementation of a tcg-based integrity measurement architecture. In: USENIX Security Symposium, 2004. vol. 13, p. 223–238.
17. Priedhorsky R, Randles T. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In: Proceedings of the international conference for high performance computing, networking, storage and analysis, 2017. p. 1–10.
18. Azab A. Enabling docker containers for high-performance and many-task computing. In: 2017 Ieee international conference on cloud engineering (ic2e), IEEE; 2017. p. 279–285.
19. Tien C-W, Huang T-Y, Tien C-W, Huang T-C, Kuo S-Y. Kubanomaly: anomaly detection for the docker orchestration platform with neural network approaches. Eng Rep. 2019;1(5):12080.
20. Du Q, Xie T, He Y. Anomaly detection and diagnosis for container-based microservices with performance monitoring. In: International conference on algorithms and architectures for parallel processing, Springer; 2018. p. 560–572.
21. Samir A, Pahl C. Anomaly detection and analysis for clustered cloud computing reliability. Cloud Comput. 2019;2019:120.
22. Mart O, Negru C, Pop F, Castiglione A. Observability in kubernetes cluster: Automatic anomalies detection using prometheus. In: 2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), IEEE; 2020. p. 565–570.
23. Kitahara H, Gajananan K, Watanabe Y. Highly-scalable container integrity monitoring for large-scale kubernetes cluster. In: 2020 IEEE international conference on big data (Big Data), IEEE; 2020. p. 449–454.
24. Chelladhurai J, Chelliah PR, Kumar SA. Securing docker containers from denial of service (dos) attacks. In: 2016 IEEE international conference on services computing (SCC), IEEE; 2016. p. 856–859.
25. Hunger C, Vilanova L, Papamanthou C, Etsion Y, Tiwari M. Dats-data containers for web applications. In: Proceedings of the twenty-third international conference on architectural support for programming languages and operating systems. 2018. p. 722–736.
26. Luo Y, Luo W, Sun X, Shen Q, Ruan A, Wu Z. Whispers between the containers: high-capacity covert channel attacks in docker. In: 2016 IEEE Trustcom/BigDataSE/ISPA. IEEE; 2016. p. 630–637.
27. Jian Z, Chen L. A defense method against docker escape attack. In: Proceedings of the 2017 international conference on cryptography, security and privacy, ACM; 2017. p. 142–146.
28. Arnautov S, Trach B, Gregor F, Knauth T, Martin A, Priebe C, Lind J, Muthukumaran D, O'Keeffe D, Stillwell ML, et al. {SCONE}: Secure linux containers with intel {SGX}. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16). 2016. p. 689–703.
29. Hoekstra M, Lal R, Pappachan P, Phegade V, Del Cuvillo J. Using innovative instructions to create trustworthy software solutions. HASP@ ISCA. 2013;11(10.1145):2487726–8370.
30. Kelbert F, Gregor F, Pires R, Köpsell S, Pasin M, Havet A, Schiavoni V, Felber P, Fetzer C, Pietzuch P. Securecloud: Secure big data processing in untrusted clouds. In: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017. IEEE; 2017. p. 282–285.
31. Ranjbar A, Komu M, Salmela P, Aura T. Synaptic: Secure and persistent connectivity for containers. In: 2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID), IEEE; 2017. p. 262–267.


**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.