AppArmor Profile Generator as a Cloud Service

Hui Zhu and Christian Gehrmann
Department of Electrical and Information Technology, Lund University, Lund, Sweden

Keywords: Security-as-a-Service, Docker, Container, AppArmor.

Abstract: Along with the rapid development of containerization technology, remarkable benefits have been created for developers and operation teams, and overall software infrastructure. Although lots of effort has been devoted to enhancing containerization security, containerized environments still have a huge attack surface. This paper proposes a secure cloud service for generating a Linux security module, AppArmor profiles for containerized services. The profile generator service implements container runtime profiling to apply customized AppArmor policies to protect containerized services without the need to make hard and potentially error-prone manual policy configurations. To evaluate the effectiveness of the profile generator 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 generator service improves the test web service's overall security a lot compared to using the default Docker security profile.

1 INTRODUCTION

Containerization is by far the most eye-catching technology as an alternative or companion to virtualization. Gartner predicts that by 2022, more than 75% of global organizations will be running containerized applications in production[1]. 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 (Casalicchio and Iannucci, 2020). Containers and microservices architectures are different from the traditional virtual machines with monolithic applications (Martin et al., 2018). DevSecOps is a set of practices that combines software development (Dev), security (Sec), and IT operations (Ops), which means built-in security in application development through the whole service life-cycle (Myrbakken and Colomo-Palacios, 2017). Cloud Security Alliance (CSA) points out that DevSecOps are created as a response to resolve security issues that have risen from microservices-based architectures[2]. CSA defines six focus areas critical to integrating DevSecOps into an organization, one of which is automation. The security for microservices in containers should be automated to protect the environment and data. Several security controls for containers have been embedded into a continuous integration and delivery pipeline to ensure the automated end-to-end security of containers. One of such controls is called behavior-based control securing the container runtime.

Many different behavior-based solutions have appeared in the industry. The top container security products' typical way is to monitor the container's 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 (Stopel et al., 2020). The TwistLock monitoring and profiling defense work on four levels: the file system (Levin et al., 2020a), the processes, the system calls, and the network (Levin et al., 2020b). 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

[1]https://www.gartner.com/smarterwithgartner/6-best-practices-for-creating-a-container-platform-strategy/

[2]https://cloudsecurityalliance.org/artifacts/six-pillars-of-devsecops/

Zhu, H. and Gehrmann, C.
AppArmor Profile Generator as a Cloud Service.
DOI: 10.5220/0010434100450055
In *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*, pages 45-55
ISBN: 978-989-758-510-4

container[3]. NeuVector[4], StackRox[5] and Sysdig[6] also provide similar products. Besides the container security companies, British Telecommunication also has a patent called software container profiling, which can generate runtime profile for the container in execution (Daniel and El-Moussa, 2019). Two open-source projects secure containers based on the runtime behavioral monitoring: Falco[7] and Dagda[8]. 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.

However, not many similar solutions have been introduced in academic works. Some researchers propose novel design ideas but lacking implementation details and experimental results. In the work of Sarkale et al. (2017), 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: 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.

Most recently, in the work of Zhu and Gehrmann (2020), a command-line tool called Lic-Sec was proposed which implements Linux tracing tools: SystemTap[9] and Auditd[10] to trace the behavior of the container runtime and generate a Linux security module AppArmor[11] 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 is experimentally evaluated to be efficient to real-world attacks, especially the privilege escalation attacks. However, the original Lic-Sec work does not take profile generation during container usage into account, which may cause the generated profiles to be too restrictive to function in the final container deployment environment. In this paper, we provide a novel cloud tool for AppArmor profile generation, which utilizes Lic-Sec but allows dynamic and automatic AppArmor profile generation following the DevSecOps automation principle. Furthermore, we evaluate the profile generator's strength by running a typical set of containerized web services against a filtered set of relevant known exploits. The evaluation is based on a designed microservice, and we show the significant security improvements achieved with our automated cloud-based profile compared to using the default Docker security configuration
In summary, we make the following contributions:

- We propose, design, and implement a novel, dynamic, AppArmor Profile Generator as a Cloud Service.
- 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 Section 2, we give a background description of Lic-Sec and the classification for containers. In Section 3, 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 Section 4, we introduce the cloud service approach of the profile generator in detail. In Section 5, we describe the implementation details of the cloud service. In Section 6, we introduce how the microservice used in the evaluation is designed and how the exploit database is generated. In Section 7, the profile generator cloud service's primary evaluation results are presented, and a detailed analysis of the results is given. In Section 8, we present and discuss related work. In Section 9, we conclude this research and identify future work.

2 BACKGROUND

In this section, we describe Lic-Sec and the classification for containers.

2.1 Lic-Sec

Lic-Sec (Zhu and Gehrmann, 2020) is a command-line tool that can automatically generate AppArmor profiles based on container runtime behaviors. Lic-Sec combines LiCShield (Mattetti et al., 2015) and

[3]https://blog.aquasec.com/topic/runtime-security
[4]https://neuvector.com/products/container-security/
[5]https://www.stackrox.com/use-cases/threat-detection/
[6]https://sysdig.com/products/kubernetes-security/runtime-security/
[7]https://github.com/falcosecurity/falco
[8]https://github.com/eliasgranderubio/dagda#monitoring-running-containers-for-detecting-anomalous-activities
[9]https://sourceware.org/systemtap/
[10]https://linux.die.net/man/8/auditd
[11]https://www.openhub.net/p/apparmor/

Docker-sec (Loukidis-Andreou et al., 2018), 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. Rules generated by Lic-Sec include capabilities rules, network access rules, pivot root rules, link rules, file access rules, mount rules, and execution rules. This tool has been utilized in our new cloud-based profiling solution.

2.2 Container Classification

A container can support almost any type of application traditionally virtualized or runs natively on a machine. 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[12] 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 table is larger than 50 since some images are labeled with multiple categories. The results clearly show that containerized database accounts for the largest proportion, followed by the containerized application services and infrastructures. Among the category of containerized application infrastructure, web server takes 50 percentage. Consequently, containerization is widely applied to databases and server-side applications. Other major use cases are containerizing services, programming languages, and operating systems.

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%

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

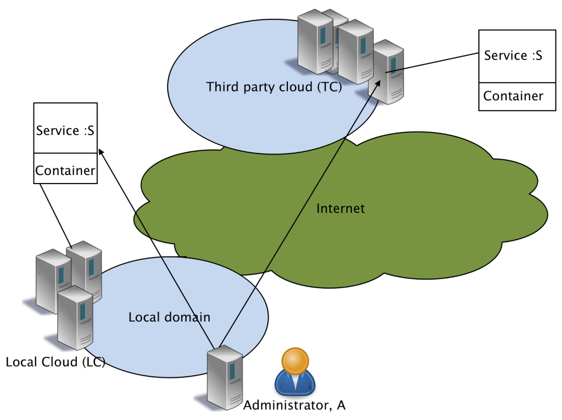

Figure 1: Scenario Overview.

3 PROBLEM DESCRIPTION

We are considering the scenario in Figure 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 the Mandatory Access Control (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, first, 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. Second, 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.

4 CLOUD SERVICE APPROACH

We suggest a solution where 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 Section 2.1. The proposed profile generation service offloads the administrator of a container service the burden of setting up a protection profile generation environment. In particular, the following principles apply (see also overview Figure 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.

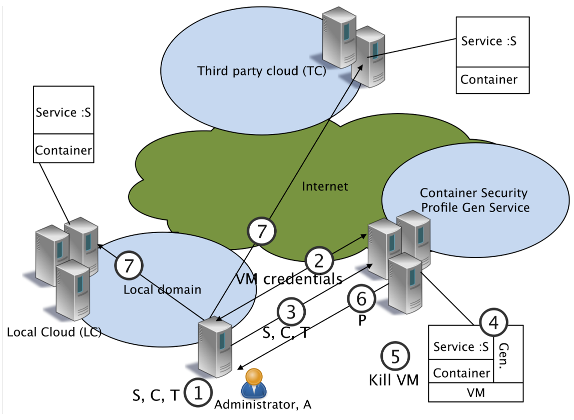

Figure 2: Profile Generator as a Cloud Service Solution Overview.

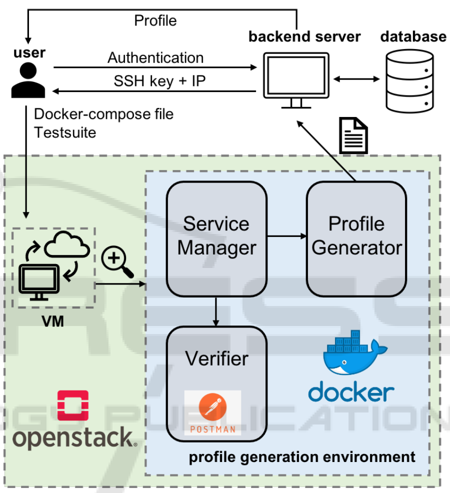

Figure 3: The implementation framework of profile generator service.

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

5 IMPLEMENTATION

The implementation framework is displayed in Figure 3. Openstack is implemented as the internal cloud platform of the profile generator service. 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 Openstack. Openstack generates the SSH key and the floating IP address for this VM. The backend server collects this information from Openstack and sends it back to the user. The profile generation environment includes the following pre-installed components:

- **Profile Generator:** We use the Lic-Sec tool described in Section 2.1 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[13] as the verifier, which is a command-line collection runner for Postman[14]. 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.

Service Launch: A user uses the received SSH key and IP address to build a secure connection with the VM. 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 Section2.1. 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 generator 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 Openstack 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.

[13]https://www.npmjs.com/package/newman
[14]https://www.postman.com/

6 EXPERIMENTAL SETUP

Here, we describe the experimental set-up used in our evaluation. First, we discuss 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. 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 Section 2.2. Besides, web services are the most popular services which have been widely deployed. Based on this stack, the evaluation's microservice 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 reported by Lin et al. (2018). The authors first generated a universe exploit database by collecting the latest 100 exploits of each category from Exploit-db[15]. Then they 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 Zhu and Gehrmann (2020) 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 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.

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

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.

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
MySQL	40678	CVE-2016-6663	Gain Privilege
	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

7 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.

7.1 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 Section 6. 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.

7.2 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[16]. 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[17] and EDB-ID-40360[18]) 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[19] and CVE-2016-6663[20]) 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 [21], 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 vulnerabilitythis tool[22]. 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[23], 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[24] 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

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

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

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

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

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

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

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

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

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

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.'

7.3 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 targetting on MySQL that cannot be defended by the profile. One is a DoS attack exploiting vulnerability CVE-2015-4870 [25] 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[26] 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 behavior is still restricted by the enforced profile.

Linux: CVE-2019-18634[27] 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[28]. 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.

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

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

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

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

phpMyAdmin: CVE-2018-10188[29] 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.

Table 3: Evaluation Result Overview.

Software / Categories	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 generator service.

8 RELATED WORK

There are some researches addressing profiling to enhance runtime security for containerization environment. In the work of Pothula et al. (2019), 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 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 sys-

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

tem and no other attacks.

Many commercial products are providing container runtime profiling, as mentioned in Section 1. In the academic area, LiCShield (Mattetti et al., 2015) and Docker-sec (Loukidis-Andreou et al., 2018) mentioned in Section 2.1 are two such solutions. Both aim to secure Docker containers through their whole lifecycle 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.

Other than 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 (Bacis et al., 2015). 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 according to the DevSecOps methodology. Sun et al. (2018) 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 De Benedictis and Lioy (2019) to support integrity verification and remote attestation of Docker containers. DIVE relies on a modified version of IMA (Integrity Measurement Architecture) (Sailer et al., 2004), 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 Priedhorsky and Randles (2017) 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 Azab (2017) 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.

Besides proposing general security solutions for containers, many pieces of research 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 (Chelladhurai et al., 2016), the application level attacks (Hunger et al., 2018) and covert channels attacks (Luo et al., 2016), as well as some attacks with severe impacts such as container escape attacks (Jian and Chen, 2017) and attacks from the underlying compromised higher-privileged system software such as the OS kernel and the hypervisor (Arnautov et al., 2016). In the work of Chelladhurai et al. (2016), 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, Hunger et al. (2018) 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 Luo et al. (2016). 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. Jian and Chen (2017) 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 Arnautov et al. (2016), which is a secure container environment for Docker utilizing Intel Software Guard eXtension (SGX) (Hoekstra et al., 2013) for running Linux applications in secure containers.

Some researches aim to provide secure connections for Docker containers. In the work of Kelbert et al. (2017) and Ranjbar et al. (2017), both of them propose solutions to build secure and persistent connectivities between containers. The work of Secure Cloud proposed by Kelbert et al. (2017) is realized with the support of Intel's SGX. While the SynAPTIC architecture from Ranjbar et al. (2017) is based on the standard host identity protocol (HIP). Cilium[30] is open-source software for securing the network connectivity between containerized application services.

9 CONCLUSION AND FUTURE WORK

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 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 1 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.

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

REFERENCES

Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., et al. (2016). {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 689–703.

Azab, A. (2017). Enabling docker containers for high-performance and many-task computing. In *2017 ieee international conference on cloud engineering (ic2e)*, pages 279–285. IEEE.

Bacis, E., Mutti, S., Capelli, S., and Paraboschi, S. (2015). Dockerpolicymodules: mandatory access control for docker containers. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 749–750. IEEE.

Casalicchio, E. and Iannucci, S. (2020). The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience*, page e5668.

Chelladhurai, J., Chelliah, P. R., and Kumar, S. A. (2016). Securing docker containers from denial of service (dos) attacks. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 856–859. IEEE.

Daniel, J. and El-Moussa, F. (2019). Software container profiling. US Patent App. 16/300,169.

De Benedictis, M. and Lioy, A. (2019). Integrity verification of docker containers for a lightweight cloud environment. *Future Generation Computer Systems*, 97:236–246.

Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., and Del Cuvillo, J. (2013). Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA*, 11(10.1145):2487726–2488370.

Hunger, C., Vilanova, L., Papamanthou, C., Etsion, Y., and Tiwari, M. (2018). Dats-data containers for web applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 722–736.

Jian, Z. and Chen, L. (2017). A defense method against docker escape attack. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, pages 142–146. ACM.

Kelbert, F., Gregor, F., Pires, R., Köpsell, S., Pasin, M., Havet, A., Schiavoni, V., Felber, P., Fetzer, C., and Pietzuch, P. (2017). Securecloud: Secure big data processing in untrusted clouds. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 282–285. IEEE.

Levin, L., Stopel, D., and Yanay, E. (2020a). Filesystem action profiling of containers and security enforcement. US Patent 10,664,590.

Levin, L., Stopel, D., and Yanay, E. (2020b). Networking-based profiling of containers and security enforcement. US Patent 10,599,833.

Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., and Zhou, Q. (2018). A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 418–429. ACM.

Loukidis-Andreou, F., Giannakopoulos, I., Doka, K., and Koziris, N. (2018). Docker-sec: A fully automated container security enhancement mechanism. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1561–1564. IEEE.

Luo, Y., Luo, W., Sun, X., Shen, Q., Ruan, A., and Wu, Z. (2016). Whispers between the containers: High-capacity covert channel attacks in docker. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 630–637. IEEE.

Martin, A., Raponi, S., Combe, T., and Di Pietro, R. (2018). Docker ecosystem–vulnerability analysis. *Computer Communications*, 122:30–43.

Mattetti, M., Shulman-Peleg, A., Allouche, Y., Corradi, A., Dolev, S., and Foschini, L. (2015). Securing the infrastructure and the workloads of linux containers. In *2015 IEEE Conference on Communications and Network Security (CNS)*, pages 559–567. IEEE.

Myrbakken, H. and Colomo-Palacios, R. (2017). Devsecops: a multivocal literature review. In *International Conference on Software Process Improvement and Capability Determination*, pages 17–29. Springer.

Pothula, D. R., Kumar, K. M., and Kumar, S. (2019). Run time container security hardening using a proposed model of security control map. In *2019 Global Conference for Advancement in Technology (GCAT)*, pages 1–6. IEEE.

Priedhorsky, R. and Randles, T. (2017). Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10.

Ranjbar, A., Komu, M., Salmela, P., and Aura, T. (2017). Synaptic: Secure and persistent connectivity for containers. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 262–267. IEEE.

Sailer, R., Zhang, X., Jaeger, T., and Van Doorn, L. (2004). Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security symposium*, volume 13, pages 223–238.

Sarkale, V. V., Rad, P., and Lee, W. (2017). Secure cloud container: Runtime behavior monitoring using most privileged container (mpc). In *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 351–356. IEEE.

Stopel, D., Levin, L., and Yankovich, L. (2020). Profiling of container images and enforcing security policies respective thereof. US Patent 10,586,042.

Sun, Y., Safford, D., Zohar, M., Pendarakis, D., Gu, Z., and Jaeger, T. (2018). Security namespace: making linux security frameworks available to containers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1423–1439.

Zhu, H. and Gehrmann, C. (2020). Lic-sec: an enhanced apparmor docker security profile generator. preprint on webpage at https://arxiv.org/abs/2009.11572.