2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS) | 978-1-6654-2104-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/COMSNETS53615.2022.9668504

Kub-Sec, an automatic Kubernetes cluster AppArmor profile generation engine

1st Hui Zhu
Department of Electrical and Information Technology
Lund University
Lund, Sweden
hui.zhu@eit.lth.se

2nd Christian Gehrmann
Department of Electrical and Information Technology
Lund University
Lund, Sweden
christian.gehrmann@eit.lth.se

***Abstract*—Kubernetes (K8s) is one of the best options available to deploy applications in large-scale infrastructures. Security has been a big concern for all practitioners in the K8s eco-system. Almost all cloud vendors have their security solution for K8s cluster, pods, workloads, etc. In recent years, a large number of open-source tools and projects related to K8s security have emerged to meet the increased demand for enhanced security in these systems. Following this general need and trend, we propose a new design for automatic K8s cluster AppArmor profile generation. Our design is based on a most recent work of automatic AppArmor policy generator for Docker containers called Lic-Sec. The system collects the behavioral data of application containers in all worker nodes distributively, then centrally transforms the data to AppArmor policies for each application container, and enforces the policies without interrupting the service. We present a prototype of the system using Google K8s environment and with an AppArmor profile for a WordPress personal blog. We show that the security policies generated by the system can defend one typical kind of attack which targets all WordPress's XML-RPC interface.**

***Index Terms*—Kubernetes, security, AppArmor, cloud**

I. Introduction

Kubernetes or K8s is an orchestration system that automates container deployment, scaling, and management[1]. K8s is a future-proof solution designed to support large distributed systems [1]. Almost all major cloud vendors have out-of-the-box solutions to support K8s. The ecosystem is growing rapidly with new products on top of the K8s platform being released every day. According to the CNCF survey 2020[2], K8s use in production has increased to 83%, up from 78% 2019. During the release cycle of version 1.19 between April and August, 382 companies and over 2,464 individuals contributed to K8s. Many global organizations have migrated to K8s such as Spotify, IBM, Nokia, and Yahoo[3].

However, security is a big concern that inhibits companies from migrating their existing workloads to K8s [2], [3]. According to the Fall 2020 edition of StackRox's "State of Container and K8s Security" report[4], around 90% of survey respondents have experienced a security incident in their K8s and container environments during the past 12 months. A pod is the minimum deployment and management unit of K8s so that enforcing strict security policies on pod level is important for the cluster's overall security. K8s officially provides several policy-based protection schemes such as pod security policy (PSP) which securely configures the pods from cluster level, container security context (CSC), and pod security context (PSC) which defines privilege and access control settings for a pod or container from pod level. K8s also allows the user to utilize AppArmor[5], a Linux kernel security module, to restrict a container's access to resources, and Seccomp[6] to restrict a container's syscalls. However, configuring these policies is complex, to some degree error-prone, and labor-intensive. Hence, auto-generation can significantly decrease the time spent configuring security contexts and policies. Further, auto-generation of security configurations based on behavior profiling of the pods strengthen security by reducing the privilege to a minimum. An open-source tool called Sysdig's K8s Policy Advisor[7] realizes auto-generation of pod security policies. It generates a PSP based on the CSC and PSC of a live K8s environment, or from a single YAML file containing a pod specification. Another tool called Lic-Sec can automatically generate AppArmor profiles for Docker containers based on behaviors [4]. It collects the behavioral data of the running containers during a training period and transforms them into AppArmor capability rules, network access rules, and file access rules. The tool is further implemented in a profile generation cloud service which helps to reduce the burden on end-users of using the tool [5].

However, the K8s Policy Advisor cannot generate PSPs based on the runtime container behavior. Lic-Sec only works in a local environment for Docker containers, but people rarely use purely containerization without an orchestration system in a real production environment. To fill this gap, we propose in this paper runtime behavior-based auto-generators for clusters.

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.

[1]https://K8s.io/

[2]https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf

[3]https://K8s.io/case-studies/

[4]https://security.stackrox.com/rs/219-UEH-533/images/State-of-container-and-K8s-security-report-fall-2020.pdf

[5]https://wiki.ubuntu.com/AppArmor

[6]https://kubernetes.io/docs/tutorials/clusters/seccomp/

[7]https://github.com/sysdiglabs/kube-psp-advisor

Our solution offers automatic AppArmor policy generation for K8s clusters based on runtime container behavior. The AppArmor policy is different from the PSP generated by Sysdig's K8s Policy Advisor. PSP is a K8s cluster-level resource that enables authorization of pod creation and updates through pre-defined policies, while AppArmor as described earlier is a Linux security module that restricts the resources for the containerized process. A pod profile can be generated by observing the runtime behavior of its internal containers, thereby helping security engineers customize AppArmor policies for different pods, and enhance the cluster security level.

In summary, we make the following contributions:

- We design and implement an AppArmor policy generation system which can automatically generate AppArmor rules for containers in a K8s cluster.
- We evaluate the performance of the system by enforcing the AppArmor rules on a WordPress service and show that the generated rules will not block the functionality of the microservice and can help to defend against real-world attacks.

The rest of this paper is organized as follows. In section II, we give a background description of the K8s cluster and AppArmor. In section III, we describe the research problem discussed in this study. Then we show the requirements and the design for the proposed system in section IV and V. In section VI, the implementation details are discussed. Next, in section VII, the evaluation results are presented, and a detailed analysis of the results is given. In section VIII, we present and discuss related work. In section IX, we conclude this research and identify future work.

II. Background

Next, we introduce basic K8s security concepts and technologies used in our AppArmor profile generation solution.

A. K8s Cluster Overview

Fig. 1 gives a general overview of a K8s cluster. The cluster includes a set of nodes, running as virtual or physical machines[8]. K8s runs the workload by placing containers into Pods to run on Nodes [6]. There are two types of nodes: master nodes and worker nodes.

A master node consists of the following components: an API server, scheduler, controller and etcd. The API server is the entry for all K8s services. The scheduler watches for newly created pods and assigns them to available worker nodes. The controller watches the state of the K8s cluster and makes or requests changes to move the current state to the desired state. The etcd is a consistent and highly-available key-value based storage that saves all configuration information for the K8s cluster as a backing store for all cluster data.

A worker node includes the following components: Kubelet, Kube-proxy, and pod. The Kubelet interacts with the container runtime interface (OCI), manages the life-cycle of the container, and maintains the life-cycle of a pod. It also parses

[8]https://K8s.io/docs/concepts/architecture/nodes/

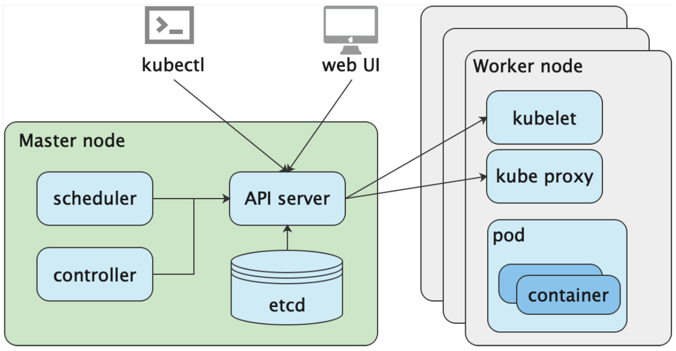

Fig. 1. An overview of the K8s cluster

K8s commands and maps them to Docker commands. The Kube-proxy maintains network rules on node for the internal and external network communication to and from the pods. A pod is the smallest deployable unit of computing that can be created and managed in K8s[9], and generally includes one or more containers.

B. AppArmor

AppArmor is a Linux Security Module (LSM) [7] that confines individual programs to the permission to read, write, or execute files, capabilities, network accesses, and rlimits[10] [8]. It is configured by a profile that is tailored to restrict the access to path-based resources of a specific program or container [9]. If proper restrictive profiles are provided, AppArmor can provide in-depth defense for any application by reducing the potential attack surface [10]. K8s supports AppArmor from v1.4[11]. AppArmor profiles are specified per container by adding an annotation to the Pod's metadata[12]. To enable the AppArmor protection for pods, the AppArmor kernel module must be enabled in the underlying host operating system and the actual AppArmor profile must also be loaded into the kernel.

III. Problem Description

The scenarios considered in this paper are shown in Fig. 2. We conisder the case where an administrator has deployed a K8s cluster including n microservices, in a local private cloud infrastructure or in a third-party cloud infrastructure. Typically a large microservice system can include hundreds to thousands of microservices [11] ($n \gg 1$). The administrator is required to enhance the microservice system security by applying suitable protection schemes. There are several alternatives for the user to choose from. One of them is AppArmor described in section II. However, due to the complexity and dynamic of the microservice system, it's hard to generate those profiles manually since it's nearly impossible for the administrator to understand the concurrent behavior of different microservices and the interactions of the whole system. Without

[9]https://K8s.io/docs/concepts/workloads/pods/
[10]https://wiki.ubuntu.com/AppArmor
[11]https://K8s.io/docs/tutorials/clusters/AppArmor/
[12]https://K8s.io/docs/tutorials/clusters/AppArmor/

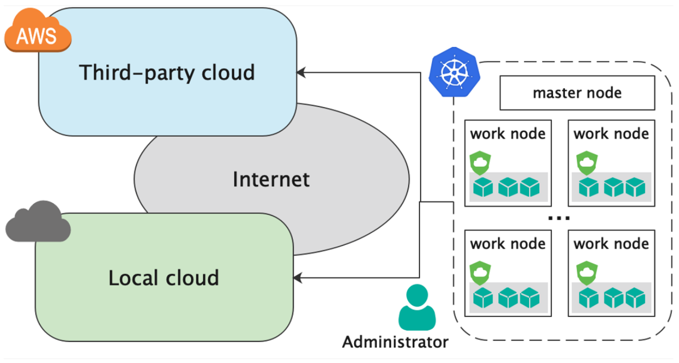

Fig. 2. Scenario overview

that knowledge, it is extremely burdensome to define the AppArmor policies. Furthermore, faced with high-frequency access to various resources in hundreds of microservices in large clusters, manually customizing AppArmor policies is not only time-consuming and labor-intensive but also likely to introduce human errors.

To avoid manual configuration of AppArmor policies, we would like to introduce functionality to K8s that enables the system to automatically restrict per container's access to resources with AppArmor from the system level. To the best of our knowledge, such a solution has not been presented before. As mentioned in Section I, there exist tools like Lic-sec providing this on container level but not for a complete K8s cluster. To solve this, the profile auto-generator should be capable of working with complex applications for a large amount of audience or with high computing resource needs. In addition, the system needs to be able to continuously collect and store the runtime behavior of the containers on all worker nodes in the K8s cluster for a long period in a production environment. In this paper, we address the design of such an automatic profile generation system for K8s clusters fulfilling the expectations listed above. In addition, we address the security and performance evaluation of the designed system.

IV. System Requirements

Large K8s clusters are deployed distributed. However, generating AppArmor policies locally on all nodes will give a lot of overhead and will not be efficient. Hence, we have been working with a solution where a central profile generator service generates the desired profiles for the whole cluster. Under this general system assumption and from the previous problem statement, we have identified the design requirement listed below on the desired solution.

R1. *Integrity of behavioral data:* Behavioral data is fundamental to generate AppArmor policies. Therefore, the system should guarantee the integrity of the behavioral data. First, the behavioral data of the microservices in a K8s cluster should be completely collected and sent to the central service. Any data loss would lead to the lack of corresponding AppArmor policies which further blocks functionalities. Second, the system should be able to prevent behavioral data from being maliciously modified.

R2. *Confidentiality of behavioral data:* The behavioral data is a significant asset of the microservice owner and should be kept confidential. In-transit behavioral data should be encrypted. The sender and the receiver should use mutual authentication to authenticate themselves to prevent malicious attackers from spoofing their identity and obtaining access to sensitive information. Only authenticated and authorized users should have access to behavioral data at rest, which should be kept in persistent, safe storage.

R3. *Efficient processing of behavioral data:* The huge amount of behavioral data requires a high-performance search engine to sanitize data, and a fast policy generation engine to analyze data. The behavioral data shall be recorded in a structured format (such as JSON, CSV) for easy searchability and rapid processing. Context data, such as platform context and runtime context, shall be appended to them to quickly classify data during processing.

R4. *Central management:* The behavioral data are collected distributively but stored and processed centrally. The central service shall be deployed in high availability mode to ensure service reliability and stability.

R5. *Access control:* The central service shall make sure access control is applied on all incoming requests and information exchange with external entities.

R6. *Support for AppArmor policy update without service interruption:* The system shall support the enforcement of updated AppArmor policies to minimize unnecessary service interruptions. Usually, a newly updated AppArmor policy cannot be enforced on a running service unless the service is restarted with the policies. However, restarting a service would lead to service interruption, which is unacceptable in a real production environment.

V. System Design

We now have the requirements in place to define a generic architecture for the AppArmor policy generation system. A high-level picture of the architecture is shown in Fig. 3.

A. Logging Agent

A logging agent captures behavioral data from target microservices running in each worker node on a continuous basis. Daemonset[13], a built-in workload resource supplied by K8s to support node-local facilities, is used to deploy the logging agent. It ensures that whenever a node is added to the cluster, the logging agent is running on this new node. Auditd is used as the container behavior detector[14]. It is the userspace daemon that is in charge of logging the kernel audit subsystem's messages. It's integrated with AppArmor and generates log messages for occurrences that fit rules in the AppArmor profile that include ”audit” as a prefix. We choose to audit the capability events, network events, and file access events inside the container because we believe

[13]https://K8s.io/docs/concepts/workloads/controllers/daemonset/
[14]https://linux.die.net/man/8/auditd

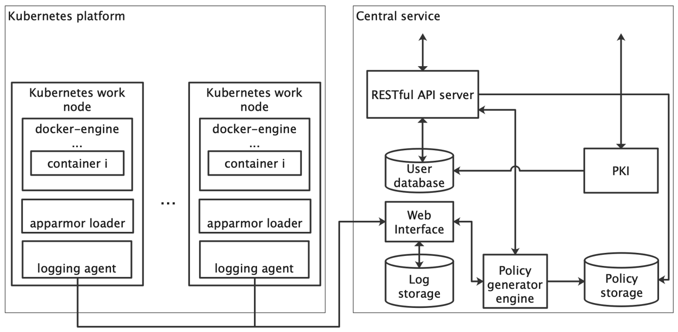

Fig. 3. System architecture

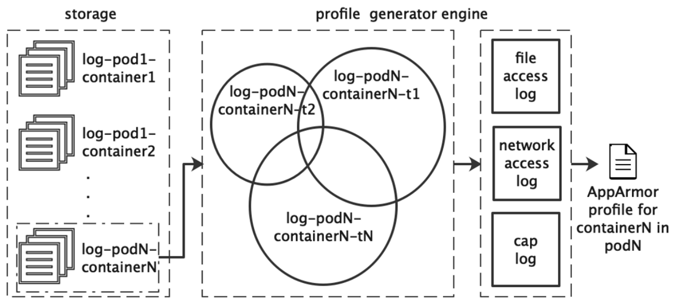

Fig. 4. Converting logs to AppArmor profiles by policy generation engine

they are adequate to accurately describe container behavior. The Auditd service logs its messages to the *audit.log.** file in */var/log/audit/* in a standard format[15] (we call these logs as Auditd logs). When the logging agent is started, it allows the Auditd service to log AppArmor permission check events. After then, the Auditd logs are sent to the central service.

B. AppArmor Loader

The AppArmor loader allows for the cluster's AppArmor profiles to be managed centrally and updated without service interruption. As described in section II-B, one should first load the AppArmor profiles to the kernel before enabling them in the K8s pods. The AppArmor loader makes it easy for users to do so without having to inspect each worker node individually. Daemonset is used for deploying the AppArmor loader.

C. Secure Channel and PKI

Mutual authentication based on TLS[16] (mTLS) is utilized to encrypt log messages sent from the logging agent to the behavioral data storage, which is achieved by an approach built upon a system internal PKI. A Certificate Authority (CA) service that issues both client and server certificates is utilized to realize this[17].

D. Central Service

When a service is set up, the central service collects raw behavioral data sent by the logging agent and keeps it in persistent storage. After enough data has been collected, the policy generation engine starts to create the profile and saves it in a database for the time being. Meanwhile, the central service notifies the end-user that the policy can be obtained via an out-of-band method (such as email or message).

1) ***RESTful API server:*** The central service provides three APIs: **PUT /register** for user authentication, **GET /service** for initializing the policy generation service, and **GET /policy** for obtaining the final security policy. User authentication for the API is based on standard username password basic authentication [12]. The GET/service allows the user to configure the login agent and obtain the necessary certificates. The GET/profile service is used to download the AppArmor profile once it is generated.

2) ***Behavioral Data Storage:*** The central service receives logs from each node. The data is stored in a log storage. Three different Auditd logs are collected corresponding to file access, network, and capability events. To achieve quick data search replies, a distributed and multitenant-capable full-text search engine with an HTTPS web interface is given. Furthermore, the storage is designed to grow, allowing large behavioral data from the entire cluster to be saved without running out of capacity.

3) ***Policy generation engine:*** The engine is designed based on the work done in [4]. In this work, Auditd logs are converted into AppArmor profiles for microservices that aren't orchestrated by K8s. In our design, we add features for log collection through the API and log processing from multiple containers. The layout of the engine is shown in Fig. 4.

The engine's goal is to combine all recordings from the same container, remove duplicate logs, and split them into three different event categories. After that, the engine collects important data from the logs and generates AppArmor rules depending on the rule syntax.

4) ***AppArmor policy generation:*** To create file access rules, the engine pulls the field values *name* and *request_mask* from file access events. Network rules are generated using the field values *family*, *sock_type*, and *protocol* from network events, while capability rules are generated using the field value *capname* from capability events. The produced profiles are saved as a "key-value" pair in the profile storage, with the key being the name in the "[profile name]-[username]" format and the value being the profile.

5) ***File path globbing:*** There are some files or directories named with a combination of random letters and numbers in the file access events, such as "temp-write-test-60d4faf7bc5b11-87466945." Some files and directories are labeled with the date and time of creation, as well as a random string like "20210616_150212_7A9F.mkv." Paths with these unfixed names must be appropriately handled; otherwise, the created file access rules would prevent the creation of a file or directory with this name. We have used a glob pattern file name design that takes a proper trade-off between security

[15]https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/sec-understanding_audit_log_files

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

[17]In our proof of concept we have used this internal PKI strucutre but the solutions can work with any external PKI as well.

and correct file access handling for these cases. We give implementation details in the next section.

VI. Implementation

We have made a full proof-of-concept implementation of the proposed system on Google Cloud. Our prototype generates AppArmor policies for a public WordPress site. In this section we describe the implementation. Fig. 5 shows an overview of the realization, as well as the details of the running workloads of all clusters in this system.

A. Environment Setup

The system is set up on Google Cloud. As illustrated in Fig. 5, we utilize Google K8s Engine (GKE) to create two K8s clusters, one for the user and the other for the central service mentioned in Section V-D. A WordPress with persistent disks and cloud SQL is deployed in the user's cluster. WordPress is a free and open-source content management system. According to a report from W3Techs[18], WordPress powers 40% of all the websites on the Internet. We have used the official Docker image[19] to build WordPress. The four components outlined in Section V-D, are running as containerized applications in different namespaces in the cluster. The implementation details of the component implementations are given in Section VI-C.

B. User Cluster

The user uses the received two K8s resource manifests to deploy two DaemonSets in the namespaces *AppArmor* and *logging*. These two services launch the AppArmor loader of the worker nodes as well as the logging for profile generation:

1) ***AppArmor Loader:*** The responsibility of the AppArmor loader is to allocate AppArmor profiles to worker nodes and to load the profiles into the kernel. An API object called ConfigMap[20] updates AppArmor profiles to the nodes. The creation or update of a profile takes less than 10 seconds.

2) ***Logging Agent:*** The logging agent implements two different services. First, the logging agent applies audit rules related to AppArmor permission checks and enables auditing on the host; second, it forwards logs to the central service. We have implemented the log transfer using the tool Fluentd, an open-source data collector for unified logging layer[21]. The systemd plugin[22] and Elasticsearch plugin[23] for Fluentd are used to read logs from the systemd journal and deliver them to the Elasticsearch service operating on the central service cluster.

C. Central Service Cluster

The central service cluster consists of four pods running the API server, the behavioral storage, the PKI, and the policy generation engine.

[18]https://w3techs.com/technologies/details/cm-wordpress

[19]https://registry.hub.docker.com/_/wordpress/

[20]https://K8s.io/docs/concepts/configuration/configmap/

[21]https://www.fluentd.org/

[22]https://github.com/fluent-plugin-systemd/fluent-plugin-systemd

[23]https://github.com/uken/fluent-plugin-elasticsearch#clienthost-certificate-options

1) ***API Server:*** The API server is exposed to the public as a service through a load balancer. It is primarily in charge of user authentication and user credential management, as well as communication with the policy generation engine.

2) ***Elasticsearch:*** The Elasticsearch is utilized for storing and searching behavioral data. It is managed by a StatefulSet, which is the workload API object used to manage stateful applications[24]. It receives log information from the distributed logging agents over a TLS channel authenticated by certificates. It has a persistent volume backed by a persistent disk with a capacity of 20 GB. The logs are saved in JSON format.

3) ***PKI:*** We have realized the PKI service using OpenSSL[25] and the Python cryptography library[26]. A certificate requrest handlling servie is realized with a combination of Nginx[27], which provides a lightweight HTTPS-enabled web server, and Gunicorn[28] and Falcon[29], which provides the final connection to the certificate-related tasks.

4) ***Policy Generation Engine:*** The engine is exposed as an internal service which only communicates with the API server. We tweaked the engine proposed in [4] to make it more compatible with Elasticsearch and more efficient to be able to handle massive amounts of data. The original engine in [4] is only able to work locally in a Linux machine for a small amount of data. The modified policy generator primarily performs two functions: search and analysis. A quick log search is implemented using the Python Elasticsearch Client[30]. The analysis utilizes Pandas[31], one of Python's most popular data science modules. The generator first does a scan search on all logs for a specified container. The search results are then converted to a dataframe[32], a pandas object which is a 2-dimensional labeled data structure. Each column in the dataframe corresponds to a field in that log, and each row corresponds to a log.

The dataframe is then used to conduct further analysis. The duplicated logs are first removed. The rules are then constructed directly by extracting and combining columns named after the fields they belong to. Capability rules, for example, can be created by extracting the column *audit_field_capname* and inserting the keyword *capability* before it.

While generating the file paths, we use glob patterns to include wild cards characters in the path names to solve the random pathname issue mentioned in sectionV-D3. The value of *audit_field_name* of the file access events represents the absolute path for a file. A random path identifier is implemented to check the names for the directory, all subdirectories, and the file. If the name is identified to be random, it will be replaced by wild cards characters. The random path identifier is developed based on an open-source tool

[24]https://K8s.io/docs/concepts/workloads/controllers/statefulset/

[25]https://www.openssl.org/

[26]https://cryptography.io/en/latest/

[27]https://www.nginx.com/

[28]https://gunicorn.org/

[29]https://falcon.readthedocs.io/en/stable/

[30]https://elasticsearch-py.readthedocs.io/en/v7.13.3/

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

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

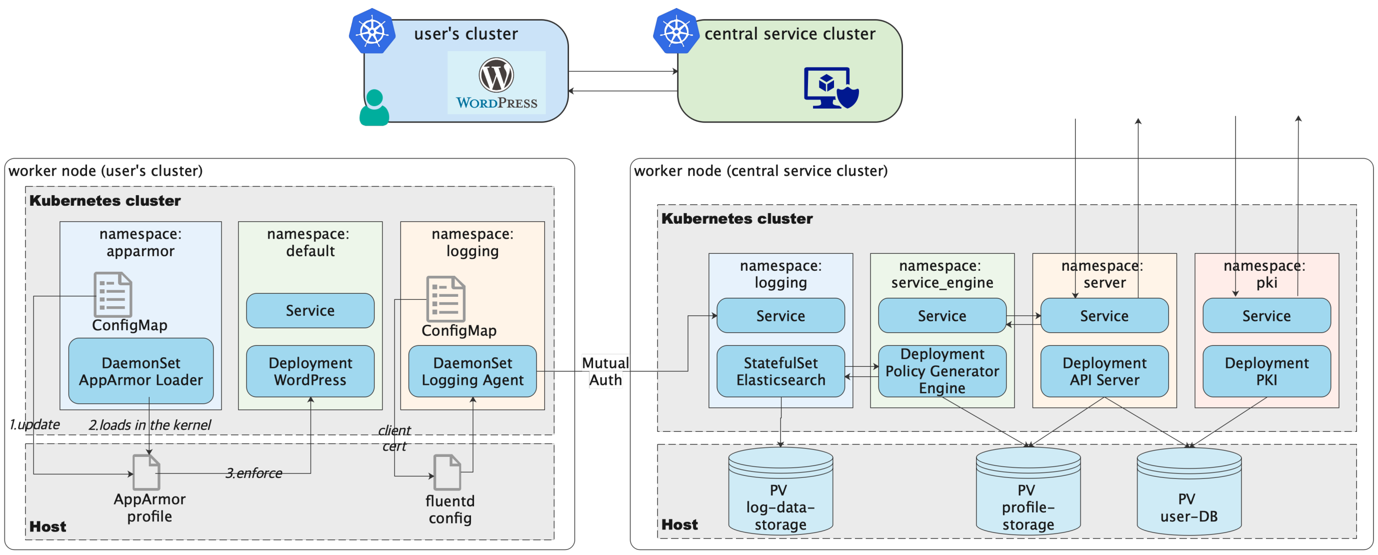

Fig. 5. Implementation of the proposed AppArmor policy generator

called Gebberish-Detector[33]. This tool allows us to distinguish random alphabetic stings from real names. By using this in combination with separate identification of typical Linux file or directory names and treating numerical numbers separately, we achieved a reliable file path name filter to handle also random paths.

To avoid false positives when a non-random path is mistakenly detected as random, the path for a file before and after globbing is generated for the user to inspect. If there are any errors, the user can manually correct them. It should be highlighted that even if the fault positive circumstances exist, they will result in open permission, which will not affect the microservice's functionality.

VII. Evaluation

Next, we evaluate the proposed system design and implementation by revisiting the identified requirement in section IV. We also discuss the performance and security enhancements we obtained in our WordPress pilot.

A. Requirements Assessment

The logging agent responsible for the data collection and transmission to the central service uses TLS with mutual authentication ensuring the integrity and confidentiality of transmitted log data. Even if the logging agent would break down, the Auditd service will still collect all events making it possible to recover without data loss. Furthermore, the central service uses the K8s auto-scaling allowing reliable handling of a huge amount of log data. Access to the stored log data is restricted to the profile generator and central service administrator. Hence, all the integrity (R1) and confidentiality expectations (R2) concerning behavioral data are met (given that the data storage is not sensitive to direct read or physical attacks).

The behavioral data is structured in JSON format with context data ”username” and ”profile name” appended, which is convenient for further analysis. Elasticsearch is implemented for efficient searching and indexing of massive data. The policy generation engine can generate final security policies quickly with the use of Pandas. All of these implementations help to meet R3. Furthermore, we have chosen a design with central processing of the logs in a cloud back-end with high availability demands, which means that we can fulfil also R4.

With respect to user authentication and access control (R5), when the end-user requests service from the central service, authentication using standard user authentication together with TLS server authentication applies as we described in section V. This gives rudimentary authentication, but the client authentication can easily be upgraded to a two-factor mechanism if higher security is required. Only users having an agreement with the service supplier can request a service. The Role-based access control (RBAC)[34] authorization of K8s is enforced on the logging agent and the storage in order to mitigate unauthorized access and enforce the principle of the least privilege.

The implementation of the AppArmor loader helps to synchronize any AppArmor profile update or creation to the node and enforce them to the microservice without service interruption, which fulfills R6.

B. Performance Evaluation

We measure the performance of the system by comparing the CPU utilization of the host, and the transmitted bytes from the host, with the logging agent enabled and disabled. They are monitored every minute for 12 hours as shown in Fig. 6 and Fig. 7.

According to Fig. 6, the logging agent does not result in a significant increase in CPU usage. Most of the time,

[33]https://github.com/rrenaud/Gibberish-Detector

[34]https://kubernetes.io/docs/reference/access-authn-authz/rbac/

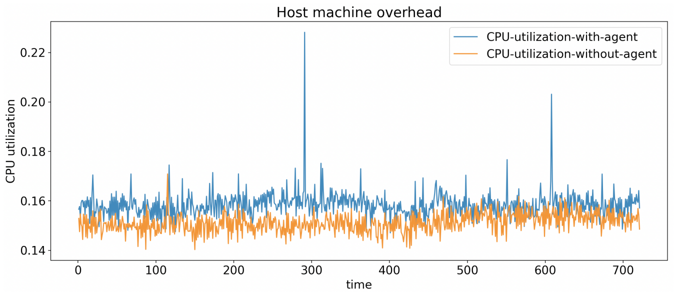

Fig. 6. Host overhead during 12 hours

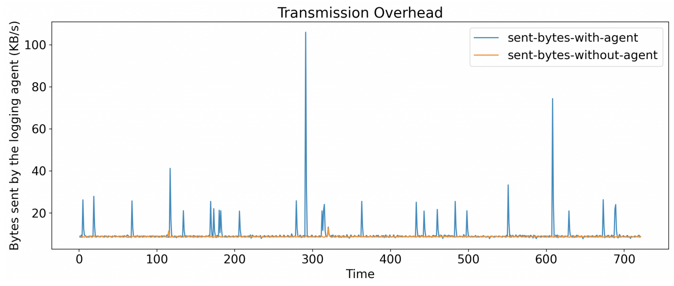

Fig. 7. Transmission Overhead during 12 hours

CPU utilization is raised by approximately 4%. However, the logging agent results in two peak values of about 20% and 22%. These peaks corresponds to increased blog activities. The transmission overhead ranges between 20KB/s and 100KB/s, as seen in Fig. 7. The two peak points in Fig. 7 match the two peak points in Fig. 6, implying that a rapid increase in the produced behavioral data would result in a large increase in CPU usage, but in this case, only for a short period.

C. Security Evaluation

Our system is designed based on the work done in [4]. We integrated the engine into K8s clusters and improved it to work with massive microservices without changing the underlying working mechanism. This profile generation principle, which we adopt, performs very well compare to using a default Docker profile. It was shown in [4] 10 out of 40 exploits were prevented by the generated AppArmor profiles. This gives an evidence of the security strength of the profiles for K8s according to our design as well.

We also evaluated the security of the system by enabling it in a WordPress service for one week as shown in Section VI-A. We made the WordPress service public to a safe intranet environment instead of the public internet. The reason is that the attacks targeting the WordPress server happen frequently. If the service is traced on the public internet, the evil behavior of attackers will also be traced and converted to corresponding AppArmor policies, which means we will have a too wide and vulnerable profile applied in the end. Finally, we got 4 capability rules which are *capability setuid*, *capability setgid*, *capability kill* and *capability net_bind_service*. No network rules were generated since it is the external load balancer instead of the WordPress container itself that interacts with the external traffic. 997 rules were generated for file accesses, most of them are for files or sub-directories under /var/www/html/, which is the working directory of WordPress. After the file path globbing, the number was reduced to 793 file access rules.

The enforced profile cannot defend against two typical WordPress attacks: brute force attacks on the login page (/wp-login.php) and DDoS attacks on the wp-cron interface (/wp-cron.php). The reason is that actions performed on those interfaces are recorded during the training period as necessary functions of the service so that the files /var/www/html/wp-login.php and /var/www/html/wp-cron.php get read permission in the AppArmor profile, which grants enough permission for the attacker to launch the attack.

XML-RPC is an API that allows developers who make third-party applications and services to interact with the WordPress site. This API opens two kinds of attacks: pingback attack and brute force attack. Hackers can use the pingback method to send pingbacks launching a DDoS attack to the WordPress site[35]. They can also access the site using the getUserBlogs method by trying various username and password combinations. Hackers must send a POST request to launch those attacks, which requires at least read permission of the API. However, no action performed against XML-RPC interface is recorded during the training period, so no permission is granted to the file /var/www/html/xmlrpc.php in the generated profile and the API is entirely blocked, which is not the case for the default Docker AppArmor profile[36].

VIII. Related Work

There is rising competition for K8s security products in the industry. K8s security products are provided by almost all well-known cloud suppliers, including Aqua[37], Sysdig[38], StackRox[39], Paloalto[40], and Portshift[41], and they integrate security and compliance into the K8s lifecycle and provide full-stack protection for K8s. Users can pay for those managed K8s services to protect their cluster. If someone wants to build their infrastructure, they can use one of several popular open-source tools developed by major cloud security companies to assist to configure difficult security configurations by monitoring and predicting K8s cluster security-related activities and risks. Aqua's Kube-bench[42] runs K8s CIS benchmark tests and detects misconfigurations that could lead to security incidents. Aqua also releases Kube-hunter[43], a well-known tool that automates penetration testing of a user's K8s cluster and looks for security flaws. Sysdig's kube-AppArmor-manager[44] attempts to manage the AppArmor profiles of each node in a K8s cluster in a user-friendly manner, including

[35]https://www.a10networks.com/blog/wordpress-pingback-attack/
[36]https://github.com/moby/moby/blob/master/profiles/apparmor/template.go
[37]https://www.aquasec.com/products/K8s-security/
[38]https://sysdig.com/products/K8s-security/
[39]https://www.stackrox.com/platform/
[40]https://www.paloaltonetworks.com/prisma/environments/K8s
[41]https://www.portshift.io/blog/k8shield-mitre-attck-framework/
[42]https://github.com/aquasecurity/kube-bench
[43]https://github.com/aquasecurity/kube-hunter
[44]https://github.com/sysdiglabs/kube-AppArmor-manager

AppArmor status checks and profile synchronization from database to nodes. AccuKnox's[45] KubeArmor[46] is a container-ware runtime security enforcement system designed for K8s environments. It operates with Linux Security Modules (LSM) and uses appropriate LSMs to enforce policies to restrict the behavior of containers at system level. A review of the open-source tools is given in production [13].

The results in this paper is build upon the results and design of AppArmor profile generation for Docker containers first suggested in [4] and then extended in [5]. The design we present in this paper reuses the basic profile generation principles from these previous works, but adapt them to the K8s environment. Furthermore, we here show how to collect logs from a whole cluster and then apply the generated profile to this complete cluster.

K8 security research is a rapidly expanding academic research field. Lots of research is directed towards making security advice for K8s clusters or to propose mitigations for specific K8s attacks. To assist practitioners in safeguarding their K8s installations, systematization of knowledge related to K8s security practices is proposed in [14]. The authors conducted a qualitative analysis of 104 Internet artifacts and identified 11 security practices, including the use of RBAC and the implementation of pod and network AppArmor policies. Artem et al. [15] perform research into privilege escalation in containers running on K8s pods. Based on the results, a solution is given to eliminate this type of attack, emphasizing the importance of specifying *runAsUser* directives and *mustRunAsNonRoot:true* directives in pods. The authors in [16] investigate the security defects that appear in K8s manifests and finds that the number of vulnerabilities are under-reported.

Other academic work is more practical oriented. One idea is to enhance K8s security based on container monitoring. In [17], Kithara et al. propose a novel mechanism, able to handle large K8s clusters, for filtering out expected behavioral events of containers and detect abnormal mutations events to avoid false alarms in container integrity monitoring and free up resources. The authors in [18] explore the design of an automated threat mitigation architecture for K8s based on user container scanning. In their solutions, user policies are used to define vulnerability levels and to quarantine containers that are considered potentially hostile.

Anomaly detection is widely utilized on K8s. One such tool is KubAnomaly [19]. KubAnomaly introduces a container monitoring module that uses neural network techniques to detect abnormal behavior. The results [19] show that KubAnomaly can detect anomalies with an overall accuracy of up to 96% while having a low-performance overhead. In [20], another anomaly detection system (ADS) is designed for anomaly detection and diagnosis in microservices using real-time performance data such as CPU metrics, memory metrics, and network metrics. Further improvements on anomaly detection on K8s are investigated in [21], [22]. The authors in [21] propose a novel method employing forecasting to catch and predict abnormalities in K8s clusters based on the tool Prometheus[47]. In [22] an anomaly detection framework is suggested which can identify the reason for the anomalies by employing Hidden Markov Models and link the observed anomalies to hidden resources. The proposed framework can detect and identify anomalous behavior with a 96% accuracy rate.

[45]https://accuknox.com/

[46]https://kubearmor.com/

Another subject that has gotten a lot of interest from academia is K8s network security [23]–[25]. In [26] an investigating of K8 network security policies are presented, security issues are identified and mitigation techniques are suggested. BASTION [27] is a high-performance security enforcement network stack that gives containers in a K8s cluster least privileged network access. It ensures that a container can only connect to containers that are required for the composition of a service based on inter-dependencies. The authors in [28] introduce a so-called Zero Trust architecture and validate that it can improve security at all OSI model layers. Another novel network security framework is proposed in [29]. The authors suggest principles for how to automatically and intelligently assign and configure security functions in virtualized network services. Some research is attempting to identify attacks on K8s. In [30], the YoYo attack, a novel type of Burst attack against K8s cluster, is studied, and a machine learning model is tested to accurately identify this attack. A machine learning crypto-mining detection engine, which monitors Linux kernel system calls for K8s clusters is proposed in [31].

IX. Conclusion and Future Work

In this paper, we have introduced an automatic AppArmor policy generation system for K8s clusters. The system aims to enhance the cluster's overall security by protecting the workloads at runtime with AppArmor. The approach largely reduces the need for manual configuration of security policies for microservices. First, we identified 6 basic security, performance, and architecture design requirements for the system. Next, we proposed an architecture and design meeting the requirements. Our system design includes a central service, which mainly takes responsibility for security policy generation and management, and several logging agents, which work in the worker node and that continuously forward behavioral data of the application containers to the central service. We implemented the prototype of the system in GKE and simulated a user who requests to generate security policies for the WordPress blog. The system finally generated 793 rules for file accesses and 4 rules for capability after one week's collection of service behavioral data. Those generated security policies help to successfully defend a typical attack targeting the WordPress XML-RPC interface. The robustness of the method against false blockings will depend on the profile generation time in combination with how intensive the service is tested during the training period. We leave a deeper and more comprehensive evaluation with a wider range of microservices for future work.

[47]https://prometheus.io/

References

[1] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[2] A. Modak, S. Chaudhary, P. Paygude, and S. Ldate, "Techniques to secure data on cloud: Docker swarm or kubernetes?" in *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. IEEE, 2018, pp. 7–12.

[3] M. Souppaya, J. Morello, and K. Scarfone, "Application container security guide," National Institute of Standards and Technology, Tech. Rep., 2017.

[4] H. Zhu and C. Gehrmann, "Lic-sec: an enhanced apparmor docker security profile generator," *Journal of Information Security and Applications*, vol. 61, p. 102924, 2021.

[5] ——, "Apparmor profile generator as a cloud service." in *Proceedings of International Conference on Cloud Computing and Services Science (CLOSER)*, 2021, pp. 45–55.

[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[7] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman, "Linux security module framework," in *Ottawa Linux Symposium*, vol. 8032. Citeseer, 2002, pp. 6–16.

[8] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. D. Gligor, "Subdomain: Parsimonious server security." in *LISA*, 2000, pp. 355–368.

[9] Z. C. Schreuders, T. McGill, and C. Payne, "Empowering end users to confine their own applications: The results of a usability study comparing selinux, apparmor, and fbac-lsm," *ACM Transactions on Information and System Security (TISSEC)*, vol. 14, no. 2, pp. 1–28, 2011.

[10] M. Bélair, S. Laniepce, and J.-M. Menaud, "Leveraging kernel security mechanisms to improve container security: a survey," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, 2019, pp. 1–6.

[11] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, 2018.

[12] P. J. Franks, P. Hallam-Baker, L. C. Stewart, J. L. Hostetler, S. Lawrence, P. J. Leach, and A. Luotonen, "HTTP Authentication: Basic and Digest Access Authentication," RFC 2617, Jun. 1999. [Online]. Available: https://rfc-editor.org/rfc/rfc2617.txt

[13] S. Shaikh, M. Khan, N. B. Mirza, and A. Ansari, "Kubernetes security: A quick reference," 2021. [Online]. Available: https://www.ijarsct.co.in/Paper1152.pdf

[14] M. Shamim, S. Islam, F. A. Bhuiyan, and A. Rahman, "Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," *arXiv preprint arXiv:2006.15275*, 2020.

[15] L. Artem, B. Tetiana, M. Larysa, and V. Vira, "Eliminating privilage escalation to root in containers running on kubernetes," *Scientific and practical cyber security journal*, 2020.

[16] D. B. Bose, A. Rahman, and S. I. Shamim, "'under-reported'security defects in kubernetes manifests," in *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*. IEEE, 2021, pp. 9–12.

[17] H. Kitahara, K. Gajananan, and Y. Watanabe, "Highly-scalable container integrity monitoring for large-scale kubernetes cluster," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 449–454.

[18] N. Bila, P. Dettori, A. Kanso, Y. Watanabe, and A. Youssef, "Leveraging the serverless architecture for securing linux containers," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2017, pp. 401–404.

[19] C.-W. Tien, T.-Y. Huang, C.-W. Tien, T.-C. Huang, and S.-Y. Kuo, "Kubanomaly: Anomaly detection for the docker orchestration platform with neural network approaches," *Engineering Reports*, vol. 1, no. 5, p. e12080, 2019.

[20] Q. Du, T. Xie, and Y. He, "Anomaly detection and diagnosis for container-based microservices with performance monitoring," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2018, pp. 560–572.

[21] O. Mart, C. Negru, F. Pop, and A. Castiglione, "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, pp. 565–570.

[22] A. Samir and C. Pahl, "Anomaly detection and analysis for clustered cloud computing reliability," *CLOUD COMPUTING*, vol. 2019, p. 120, 2019.

[23] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization: a review," *IEEE Access*, vol. 7, pp. 152 443–152 472, 2019.

[24] S. Sultan, I. Ahmad, and T. Dimitriou, "Container security: Issues, challenges, and the road ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019.

[25] F. Minna, A. Blaise, F. Rebecchi, B. Chandrasekaran, and F. Massacci, "Understanding the security implications of kubernetes networking," *IEEE Security & Privacy*, no. 01, pp. 2–12, 1900.

[26] G. Budigiri, C. Baumann, J. T. Mühlberg, E. Truyen, and W. Joosen, "Network policies in kubernetes: Performance evaluation and security analysis," in *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*. IEEE, 2021, pp. 407–412.

[27] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "{BASTION}: A security enforcement network stack for container networks," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 81–95.

[28] D. D'Silva and D. D. Ambawade, "Building a zero trust architecture using kubernetes," in *2021 6th International Conference for Convergence in Technology (I2CT)*. IEEE, 2021, pp. 1–8.

[29] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Towards a fully automated and optimized network security functions orchestration," in *2019 4th International Conference on Computing, Communications and Security (ICCCS)*. IEEE, 2019, pp. 1–7.

[30] R. B. David and A. B. Barr, "Kubernetes autoscaling: Yoyo attack vulnerability and mitigation," *arXiv preprint arXiv:2105.00542*, 2021.

[31] R. R. Karn, P. Kudva, H. Huang, S. Suneja, and I. M. Elfadel, "Cryptomining detection in container clouds using system calls and explainable machine learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 674–691, 2020.