Designing an Intrusion Detection System for a Kubernetes Cluster

Pavel Hristov, University of Twente, The Netherlands

Recently, our lives have become increasingly dependent on digital platforms. This leads to a growing demand for automation and high-reliability systems. Consequently, the popularity of cloud-based architectures using containerization and container orchestration platforms increased. The result is that more tools are being used, therefore, the number of security concerns also grows. While Kubernetes offers robust security to prevent external attacks, it's internal protection has limitations. For this reason, this research aims to propose design solutions for implementing an Intrusion Detection System (IDS) for a Kubernetes cluster, alongside its tools and the implementation of one of the proposed designs.

Keywords: Intrusion detection system, Signature-based, Kubernetes, Containerization, Virtual machine, Network monitoring

1 INTRODUCTION

Going back in time, all organizations ran their applications directly on physical servers, either rented or bought. A significant disadvantage is that it was impossible to isolate the applications and set resource limitations for each of them. In the case of multiple applications deployed, this could easily lead to resource exhaustion and make other applications underperform. A partly solution was to deploy a single application on a single server, which still results in underutilization of resources and unscalability. This was expensive for companies due to the high prices of physical servers and inflexible maintenance processes.[2]

Virtualization came to the rescue. It enabled running multiple virtual machines (VMs) on a single server and isolate the applications. Moreover, the resource allocation to VMs on creation allows better utilization. Scalability is also improved because of the ease of starting and shutting down virtual machines. Still, this option has downsides as VMs run an operating system and all its accompanying components, which are redundant for running an application and waste additional resources. Overall it still had better scalability than physical servers deployment.[11][2]

Lastly, the containerization era offered a prerequisite for continuous integration and delivery (CI/CD) because of its fast deployments, rollbacks, and easy-to-run nature. The ease and efficiency of creating container images compared to virtual machine images contributed to the success of containerization as well. Another key feature is the portability and environment consistency, which means containers can run on every machine with a container runtime, regardless of the underlying operating system and hardware. Most importantly, they isolate applications and provide maximized resource utilization. [16]

Kubernetes is a container orchestrator that provides features such as managing the lifecycle of containers, service discovery, load balancing, self-healing, autoscaling, rollbacks, rollouts, and secret management and makes the life of software engineers easier by automating a large number of the previously manual tasks. [16]

Kubernetes has many built-in security features for different use cases and informs upon best practices in its documentation. It provides role-based access control to manage who can access services and network policies to isolate pods and limit the traffic inside the cluster. [10]

Overall, it would scale and perform great to prevent security issues, but Kubernetes does not have any native solution to assist if there is already an intruder inside the cluster. A recent example of such vulnerability is CVE-2019-5736 [17]. The user can access the containers' runtime and modify the environment variables. Using this exploit, an attacker can inject code, causing risk to the entire cluster. Therefore, the importance of intrusion detection systems (IDS) that is compatible and easily integrable with Kubernetes is rising. To fulfill this requirement, the research paper will propose several design options for implementing an intrusion detection system inside a Kubernetes cluster.

2 BACKGROUND

2.1 Intrusion detection system (IDS)

An intrusion detection system is a software that constantly monitors systems and their network communication to report malicious activity. There are two main types: signature-based and anomaly-based. The former inspects traffic and classifies the data by testing for known patterns, while the latter creates a model of the regular traffic by machine learning and can detect any abnormal activity. IDS is often confused with firewall, but they are two completely different tools. The difference is that firewalls prevent attacks from outside, while the IDS detects attacks from inside. A simple example of an intrusion attack is an attacker stealing an employee's credentials and trying to access protected company files to which they have no authority [7].

2.2 Snort

Snort is an open-source intrusion detection system and intrusion prevention system using signature-based packet inspection. It can spot attacks such as Distributed Denial of Service (DDoS), port scans, buffer overflows, Common Gateway Interface attacks, etc.

2.2.1 Why Snort?

A tool with similar functionality to Snort is Suricata, being also a signature-based IDS that is developed much more recently than Snort. It offers more features than Snort, such as

multithreading [18]. A paper by Syed Ali Raza Shah and Biju Issac indicates the difference in speed, drop rate, and resources needed by each tool. Overall, Snort performs better in resource consumption, while Suricata could process a more significant amount of network traffic faster with a lower drop rate [12]. Suricata has more to offer, but Snort is chosen for this research because of its straightforward setup and easily comprehendible nature.

2.2.2 Features

- Real-time Traffic Monitor
  Snort can run as a daemon and inspect every packet that goes through a specific network interface in real-time. If a malicious packet or threat is discovered, it will generate an alert. This is useful when you are interested in securing a single device or application.
- Packet logging
  Snort can listen to a network interface and collect all the packets that go through it. It will generate log files on the host machine. This can be used to capture the traffic and analyze it at a later point in time.
- Analysis of Protocol
  Snort can capture data in different protocol layers of the packets. This can be useful to analyze malicious packets in TCP, as it contains most of the needed information for inspection.
- Content Matching
  Snort can inspect packets for content matching by using the specified rules in its configuration. This can be used for capturing intrusion attacks.
- OS Fingerprint
  Since all operating systems have a unique TCP/IP stack, Snort uses this to determine the OS of the system that tries to access the host. Using this feature, a malicious user, who uses uncommon OS can be caught, for example, Kali Linux (a Linux distribution containing many hacking tools).
- Portability
  Snort can be installed on every network environment and every operating system. This makes it highly portable across many systems and easy to deploy.
- Different modes
  Snort has many different modes in which it can run. Such examples are packer logger, packet sniffer, or only as an intrusion detection system.

2.2.3 Architecture

Snort architecture consists of four main components. In the following sections, each of them will be described. A visual representation can be seen in Fig 1.

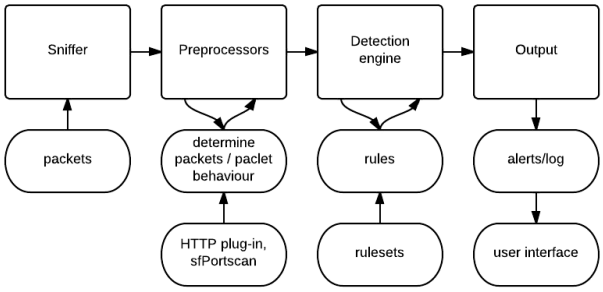


Fig. 1. Snort architecture [19].

2.2.3.1 Sniffer

The network sniffer allows Snort to eavesdrop on a network interface and collect data about the traffic. It can be used for network analysis and the generation of packet capturing files (PCAP). Snort sniffer captures the traffic and sends it to the preprocessors. Another possibility is that Snort is being fed with PCAP files directly, and the step of sniffing packets is omitted.

2.2.3.2 Preprocessors

The preprocessor consists of plugins that determine the packet behavior and intention. For example, there is a plugin responsible for detecting packet fragmentation and DoS attacks. This is important since a particular content matching rule may not match in the next step due to fragmentation. Other examples of plugins are decoders, port scan detectors, etc.

2.2.3.3 Detection engine

The detection engine is responsible for taking a packet and checking it against the Snort rules. In case of a match, it will pass an alert to the output component or drop the packet otherwise.

2.2.3.4 Output

The output component is responsible for processing the alerts generated by the detection engine. It is possible to configure it to generate log files, store logs in a database, pipe them to other systems, or even send an email.

2.2.4 Rules

Snort does its packet analysis based on rules specified in its configuration. They define the signature of a packet and the action that must be taken in response to that signature. All users can use three default rulesets – community, registered user, and subscription ruleset. All of them are GPLv2 Talos certified and are quality assured by Cisco. The difference with the paid subscription is that the rules in it are developed by Talos Security, while the community itself maintains the community and registered ruleset. It is possible to write custom rules, but it would require knowledge of the attack signature the rule will protect from. The format of a Snort rule can be seen in Fig 2.

- Action – the action Snort should take in case the packet matches the signature. It could be alert, log, pass, etc.

- Protocol – the protocol of the packet. Possible values are TCP, ICMP, UDP, etc.
- Networks – the IP of the sender/receiver. It can also be a variable specified in the snort.conf file such as $HOME_NET representing the IP/subnet of the protected network.
- Ports - The port of the sender/receiver.
- Direction, Operator – The direction of the packet. In most cases is "->".

```
Action Protocol Networks Ports Direction Operator Networks Ports
```

Fig. 2. Snort rule format.

An example of a valid rule can be seen in Fig 3.

```
alert tcp !192.168.1.0/24 any -> 192.168.1.0/24 111 (content: "|00 01 86 a5|"; msg:
"external mountd access";)
```

Fig. 3. Valid Snort rule.

The rule in Fig 3 specifies that any packet sent by TCP protocol with the sender's IP is different than the subnet 192.168.1.0/24. The receiver IP is the subnet 192.168.1.0/24 with port 111 and matches the hexadecimal content "00 01 86 a5" which will alert "external mountd access".

2.3 Kubernetes

Kubernetes is an open-source container orchestration platform developed by Google, which automates the manual processes of scaling, managing, and deploying applications. Some of the most common terminology:

- Pods – The smallest deployable units in Kubernetes. A pod consists of one or more containerized applications.
- Nodes – It is a worker machine, which could be a physical or virtual machine that allows you to run pods on it.
- Cluster – A group of nodes managed from the master node.

Kubernetes offers security policies at three levels – Pod, Node, and Kubernetes API. The recommendation at a pod level is to use role-based access control. This allows the administrator to control who can access different pods and services in the cluster. In addition to that, there are network policies. They can be used to limit the network traffic inside the cluster and isolate components that are not supposed to communicate, which will increase the level of security. It is recommended not to run anything with root privilege at a node level and use security-enhanced operating systems such as SELinux. Both pods and nodes should run alpine versions of images as they are minimalistic, therefore less attack surface. For the Kubernetes API level, an administrator can use admission controllers that provide a second level of security by inspecting requests after they are authorized. In the admission controllers, it is possible to set up rules, which they can use to validate or mutate requests. Additionally, there are several global security features such as audit logs, namespaces, and the use of external security software. [5,10].

Kubernetes cluster consists of many components, and communication between them differs on the component and cluster networking type used. There are many types of networking suggested in the Kubernetes documentation. Most of them operate similarly because Kubernetes has provided a concrete implementaton specification. For analysis purposes, the network model of Google Kubernetes Engine (GKE) will be discussed [20].

The most straightforward communication occurs between containers inside the same pod. They can access each other through localhost because they are on the same machine from a container perspective. This can be seen in Fig 4.[14]

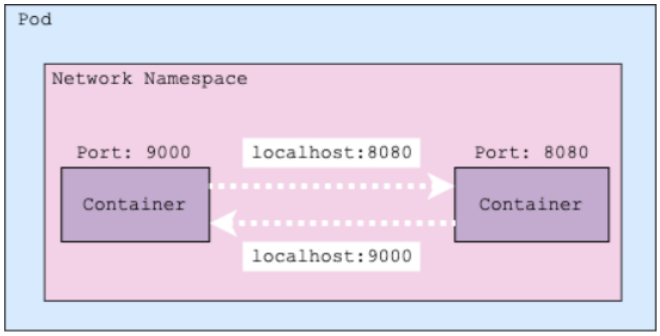


Fig. 4. Two containers communicating inside a Pod.[14]

Communication within the same node occurs through cbr0. This network bridge contains an IP table with IPs and virtual ethernet interfaces allocated for a specific pod. Each packet gets forwarded to the pod after a lookup in the bridge's table. Visualization can be seen in Fig 5.

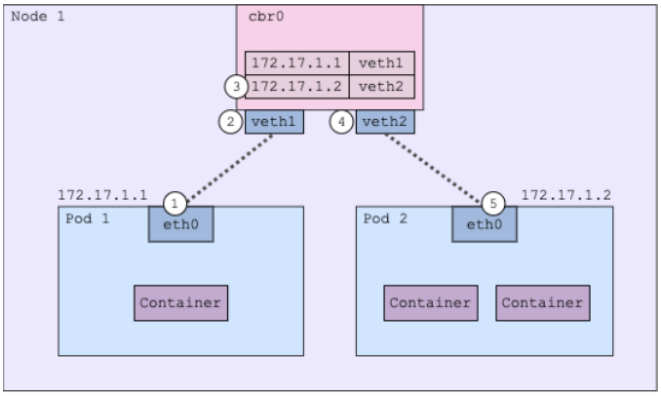


Fig. 5. Two pods communicating inside a node. [14]

Communication within the cluster occurs similarly to inside a node. A cluster has a routing table with a subnet allocated for every specific node. Every incoming packet is forwarded to the correct node after a lookup in the routing table. This can be seen in Fig 6.

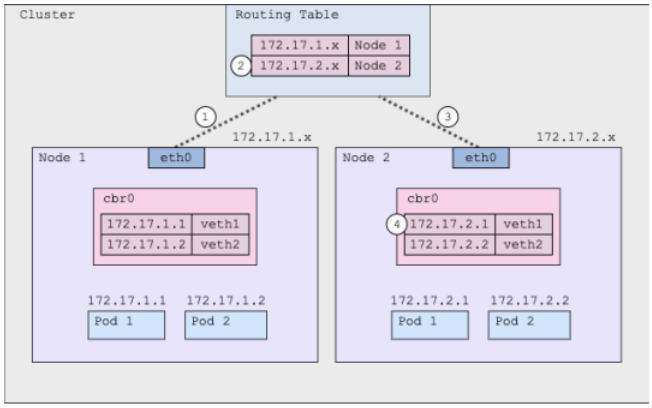


Fig. 6. Two nodes communicating inside a Kubernetes cluster. [14]

3 RELATED WORK

The article by Chirag N. Modi and Kamatchi Acha explains common security issues in cloud nodes. In many cases, VMs have a shared clipboard, which can be misused to transmit malicious programs, or if another VM monitors a VM with shared physical memory, an intruder can execute an ARP attack and sniff over the network. The information from this paper can be extracted to understand the idea of possible attacks on the Kubernetes cluster [9].

Another article explains the potential design of IDS in third-party cloud services. One proposed system uses deep learning and stands between the applications and the firewall. It inspects packets by almost 30 different attributes with corresponding attack categories and protocols. Another solution uses signature-based detection. It must be updated often to classify new attacks, which require knowledge of the system or regular support from its maintainer. The limitation is that such solutions, if not provided with appropriate data, can result in many false positives and negatives [3].

The survey on IDS article is beneficial to future design decisions for this research. It has eight different techniques for IDS implementation with detailed explanations. This article will assist in choosing the best-performing IDS based on the characteristics and limitations explained in it [7].

The 'KubAnomaly' article provides an in-depth description of how to implement anomaly detection using neural networks. The author had made graphs about performance, diagrams of the architecture, and tables for all the features it uses. The same applies to the machine learning IDS thesis of a master's student at the University of Dublin. Both works use machine learning, which requires much data and precise parameter settings to perform well. Many organizations cannot provide this, which would result in worse security [13,14].

A few articles explain unusual designs and implementations like the FCM-SVM algorithm, using data-mining techniques or suggestions for blockchain applications. They are not focused on Kubernetes but cloud-based solutions, making them too generic [1,4,6,8]. The focus of this research is rule-based IDS in Kubernetes because there is a lack of papers, describing how it works and how to implement one.

4 PROBLEM STATEMENT

The main research question that the paper aims to answer is as follows:

- How to implement an intrusion detection system to monitor the network traffic and protect against intrusion attacks inside a Kubernetes cluster?

The question is divided into two sub-questions:

- **RQ1** How to effectively capture the network traffic inside a Kubernetes cluster?
- **RQ2** What are the possible designs for an intrusion detection system to analyze the captured network traffic inside a Kubernetes cluster?

5 DESIGNS

Before the beginning of the design, it is vital to ensure that the Kubernetes cluster implements all security-related best practices mentioned in the documentation. An intrusion detection system would be useless in a non-protected environment. The proposed designs must work equally well in both single and multi-node clusters. In the following sections, possible methods will be discussed.

5.1 CAPTURING NETWORK TRAFFIC

The first step of developing an IDS is to capture the network traffic flowing throughout the cluster. Since Kubernetes is a multi-component system, there are several options where we can extract the packets. Instantly, we can conclude that doing this at a container level would be inefficient due to not protecting the upper components, which are pods and nodes. Therefore, we will omit to discuss this option. This step aims to produce a PCAP (packet capture) file containing all the flowing traffic, which will be passed to the intrusion detection system. The following sections will discuss the possibilities on a pod and a node level.

5.1.1 TOOLS

For proper network capturing, some fundamental knowledge of network capturing tools is needed. For this purpose, a short description of TCP Dump and Ksniff will be given.

5.1.1.1 TCP Dump

TCP Dump is a network capture package based on the libpcap interface. It is a platform-independent system, which makes it portable. One of its primary use-cases is to capture traffic. It provides filtering capabilities such that only a specific part of the traffic can be extracted. An example could be a network interface, specific IP, or protocol.

5.1.1.2 Ksniff

Ksniff is a kubectl plugin that uses TCP dump to capture network traffic for Kubernetes pods. It is helpful as it simplifies choosing the proper network interface and all other settings needed by TCP Dump to extract the correct packets. The

disadvantage of Ksniff is that it only supports pod traffic capturing.

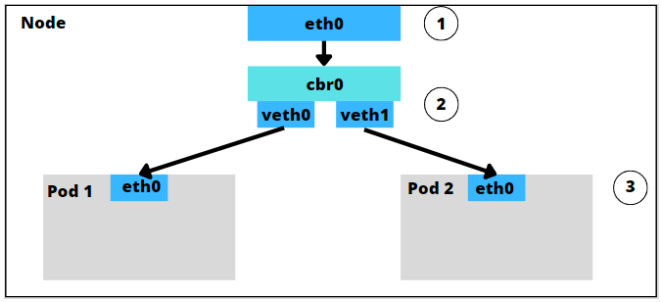


Fig. 7. Simplified network architecture inside a Kubernetes node.

5.1.2 POD LEVEL

To extract the packets at a pod level, it would be needed to intercept the flow at point 3 in Fig 7. This is possible in several ways, using TCP dump, Ksniff, or Snort. Ksniff reduces the workload since you need to install it once for kubectl use, while TCP dump or Snort requires installing it on every single pod in the cluster. Choosing Snort as a packet capturing tool would also result in worse start-up time for the pods if it were not included in the base image of the container inside the pod.

Capturing traffic at pod level is not the safest decision for detecting intrusion since the traffic with the node's IP as a destination will stop at point 1 in Fig 7. In contrast, the network capturing is situated at point 3. This can become problematic as the node constantly communicates with the Kubernetes API server through kubelet. Still, this depends on the networking type used in the cluster. Capturing packages at a pod level is only advisable for protecting a small number of pods due to not covering the whole cluster.

5.1.3 NODE LEVEL

To capture the network traffic at a node level, it must be intercepted at points 2 and/or 1 in Fig 7. Point 2 is the network bridge responsible for distributing the traffic to the correct pod, while point 1 is the entry point of the node's networking. Both are required for maximum security since communication between pods always goes through cbr0, while if a packet's destination is the node itself, it will stop at eth0 (point 1). This would ensure that nodes and pods are analyzed for intrusion within a cluster. For this purpose, we can use TCP Dump or Snort on a node by specifying which network interfaces we are interested in capturing the traffic from. The disadvantage of capturing at two points is that this option will generate duplicate traffic in case this network type is chosen for the cluster. Network types directly connect vethX to eth0, which will remove this issue. Therefore, the best option is to forbid communication to node IP addresses in the cluster and focus the network capturing only to point 2. This option will result in better security than pod level because it covers every Kubernetes component. There are two options for automating the installation of a network capturing tool: develop a custom controller, which will continuously check the state of the nodes and update them or run jobs periodically to perform the installation. A disadvantage of ensuring that a packet capturing tool is installed on a node is that the IDS becomes dependant on the node's state in the cluster.

5.2 INTRUSION DETECTION

After the network traffic has been successfully captured, it must be passed to the IDS for analysis. There are two options when it comes up to how to position the IDS inside a Kubernetes cluster. These options will be discussed in the following two sections.

5.2.1 CENTRALIZED

The first option is to have a centralized IDS, which means there will be only a single instance of the IDS service running in the cluster. It consists of three steps: capture the traffic, send it to Snort, and analyze it. Capturing traffic options has been discussed in the previous section. Therefore, we focus on sending the PCAP file. There are many alternatives to directly pipe the file using HTTP, FTP server, or a message queue broker such as ActiveMQ. The first option is easier to implement as it would require one line of code to pipe the data directly to the IDS, while the rest need more setup. Still, the advantage of a message queue and FTP server is that even if Snort is not working, the files will be persisted in the broker and received later. Before all that, it is essential to have Snort deployed in the cluster. For this purpose, a possibility is to install and set up Snort on a container and develop a web service with an endpoint accepting files. The service will trigger Snort, run a complete analysis, and generate an alert upon a malicious packet for every incoming file. The advantages of using a centralized IDS are that it is more maintainable as it is packaged inside a single container and has a more organized alert structure because they are generated from a single source. The disadvantage of centralized IDS is that it is a single point of failure, which can partly affect real-time analysis until Kubernetes spins up a new pod. Still, the omission of traffic can be prevented by using more than one replica or using a message broker. A visualization of a centralized IDS can be seen in Fig 8.

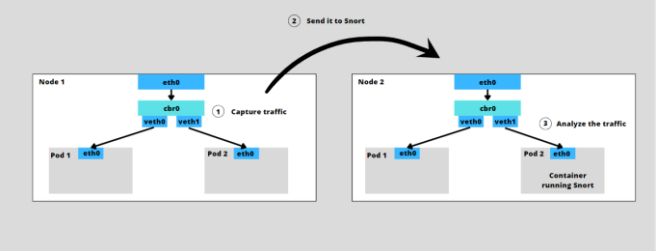


Fig. 8. Centralized design visualization of network capturing and analysis by snort.

5.2.2 NON-CENTRALIZED

The second option regarding IDS placement is non-centralized. This will involve installing Snort at every node in the cluster to enable capturing of all the traffic in the cluster. Installing it on a pod would not provide an optimized security level. Therefore, node must be the preferred choice. Since Snort provides network capture, it is not necessary to use TCP Dump. The advantages are that it has a straightforward setup, requiring only basic knowledge of Snort configuration. The non-centralized alternative will secure only the components that have Snort

installed. Therefore, it is advisable to use it only for a single-node cluster or if there are only specific components that the IDS must protect, which would be a rare occasion. Otherwise, the maintenance of numerous Snort instances would be a challenging task to perform.

5.3 ATTACK MATRIX

Snort manages to perform well and actively protect against intrusion attacks again, but after all, it is not explicitly designed for Kubernetes. Therefore, a few common problems will be discussed in this section with potential rules that can protect against them.

5.3.1 MALICIOUS CONTAINERS

To create a Docker image of an application, a base image must be used to be built upon. If the base image is compromised or has security issues, the whole cluster is in danger. This will allow an attacker to request full permissions and take over the entire cluster. This can be prevented by using network policies and requiring multi-factor authentication, but these are not always applicable in every situation. This is why Snort rules can be added that identify containers accessing the Kubernetes's control-plane and tampering with the host node's permissions.[15]

5.3.2 COMPROMISED USERS

A Kubernetes user's credentials can always get stolen. This can result in a malicious attacker using the account's permission and tampering with the cluster. Therefore Role-based access control must be enforced in Kubernetes, but this is insufficient. For this reason, a Snort rule that alerts upon attempted access from outside the organization's subnet to the kube-api must always be added.

5.3.3 ABNORMAL COMMUNICATION

In Kubernetes, it is a best practice to expose deployments through services because they can distribute the load equally to all pods. Consequently, there are cases of illogical communication, which may represent a potential intrusion. An example of such would be communication directly between two pods. They are expected to communicate through their respective services, not directly. A Snort rule must protect against such cases if they apply to the cluster.

6 IMPLEMENTATION

The following implementation will use a centralized design with node-level packet capture. This choice is because this design offers the best security level inside the Kubernetes cluster and analyzing it further would give more insights into its behavior and performance.

6.1 CLUSTER STRUCTURE

For the creation of a Kubernetes cluster locally, the tool 'kind' is used. It is mainly designed for locally testing clusters, allowing many configuration options, and choosing the number of nodes you want. Kubernetes support different container runtimes such as containerd, CRI-O, and Docker. Kind's default container runtime is containerd, which will be used for this implementation. The visualization of the cluster architecture can be seen in Fig 9 and will be explained in the following two paragraphs.

The cluster is made of two applications – echo-service and snort-wa. Echo-service is a simple Spring Boot application containing a single endpoint that returns the path variable extracted from the URL. It is going to be used as a traffic endpoint for testing purposes. Snort-wa is a Python web application containing an endpoint that accepts PCAP files. After receiving a file, it triggers Snort, runs analysis on it, and logs in case of malicious packets.

The cluster contains two deployments, responsible for ensuring that there is always at least one instance of the application running. They are exposed using ClusterIP services, which assign an internal IP and are only accessible from the cluster itself. The echo-service application is exposed to the outside world using an ingress controller, which provides external access to the cluster.

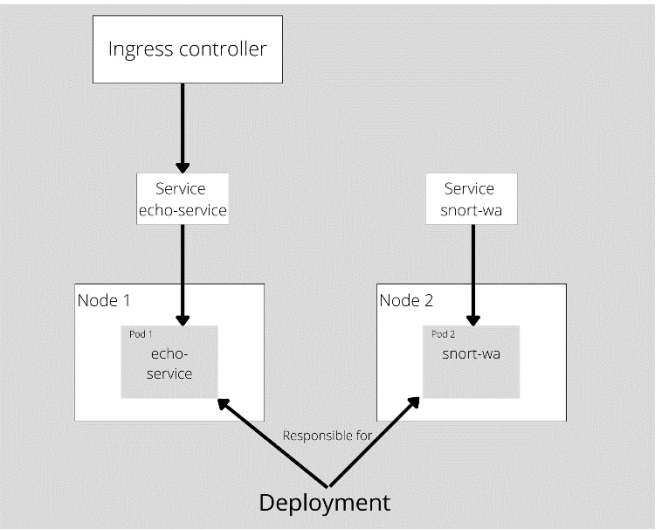


Fig. 9. Kubernetes testing cluster structure

6.2 SNORT

Snort is installed and set up during the build of the Docker image of the snort-wa service, making it immediately available after deploying the pod in Kubernetes. This also allows for quickly replacing or auto-scaling pods because Snort is preinstalled on the container, and it will not increase the time for starting it up. In the installation, all libraries and packages required by Snort are installed. Also, a separate group is created not to run Snort as a root user. Afterward, a custom configuration and rules are copied to the /etc/snort subdirectory, which contains all the information needed for Snort to run. The following configuration options must be adapted, which are dependent on the cluster parameters:

- $HOME_NET – variable containing the IP/subnet of the cluster, used by Snort to discard packets that are not pointing to or from it.
- Rule paths – the path to the rule files used by Snort to locate them.
- Other – It is possible to customize the preprocessors, include or discard rules by name, change the log format, etc.

In the current implementation, the registered user ruleset will be used [21]. It is possible to test whether Snort is set up correctly by attaching to the pod/node and running "snort -T -c /etc/snort/snort.conf". A disadvantage of this decision is that all the setup happens during the build of the Docker image. This would require a new build for changing configuration options.

6.3 NETWORK CAPTURE

To handle the network traffic, a script that pipes the PCAP file using the HTTP protocol is used. The script will run TCP dump in rotation mode to capture traffic for a specific time and pipe the generated PCAP file to snort-wa. Since an intruder can get into the network capturing component and try to disable it purposely, the script will use signal trapping. For a process to be killed, a signal must be sent. These signals responsible for terminating a process are SIGINT and SIGTERM. In case of receiving such a signal, the script will ping the Snort endpoint to trigger an alert and then kill itself. This enables an additional layer of security.

7 RESULTS

7.1 TESTING

After all the components of the IDS are set, it must be assured that the interoperation between them is problem-free. Firstly, a rule will be created that alerts every TCP packet containing the content 'test' to test whether the network capture and Snort work. The rule can be seen in Fig 10. Afterward, we access the node hosting echo-service and execute the network capture script. In the meantime, we access the echo-service using 'test' as a path variable, and the output from snort-wa returns several test alerts. This can be seen in Fig 11.

```
alert tcp any any -> any any (msg:"Test Alert"; content:"test"; sid:1000001; rev:1;)
```

Fig. 10. Snort rule that alerts on every TCP packet containing 'test'.

```
PS C:\Users\Pavel Hristov> kubectl attach -i -t -n default snort-wa-5469966bd5-7sjxf -c snort-wa
error: Unable to use a TTY - container snort-wa did not allocate one
If you don't see a command prompt, try pressing enter.
06/04-16:35:09.522166  [**] [1:1000001:1] Test Alert [**] [Priority: 0] {TCP} 10.244.0.5:38398 -> 10.244.2.4:8081
06/04-16:35:09.524682  [**] [1:1000001:1] Test Alert [**] [Priority: 0] {TCP} 10.244.2.4:8081 -> 10.244.0.5:38398
06/04-16:35:09.691587  [**] [1:1000001:1] Test Alert [**] [Priority: 0] {TCP} 10.244.0.5:38398 -> 10.244.2.4:8081
06/04-16:35:10.839779  [**] [1:1000001:1] Test Alert [**] [Priority: 0] {TCP} 10.244.0.5:38398 -> 10.244.2.4:8081
06/04-16:35:10.841896  [**] [1:1000001:1] Test Alert [**] [Priority: 0] {TCP} 10.244.2.4:8081 -> 10.244.0.5:38398
06/04-16:35:10.926597  [**] [1:1000001:1] Test Alert [**] [Priority: 0] {TCP} 10.244.0.5:38398 -> 10.244.2.4:8081
06/04-16:35:12.287064  [**] [1:1000001:1] Test Alert [**] [Priority: 0] {TCP} 10.244.0.5:38398 -> 10.244.2.4:8081
06/04-16:35:12.289211  [**] [1:1000001:1] Test Alert [**] [Priority: 0] {TCP} 10.244.2.4:8081 -> 10.244.0.5:38398
06/04-16:35:12.371332  [**] [1:1000001:1] Test Alert [**] [Priority: 0] {TCP} 10.244.0.5:38398 -> 10.244.2.4:8081
172.18.0.2 - - [04/Jun/2022 16:35:21] "POST / HTTP/1.1" 200 -
```

Fig. 11. Test alerts, using a snort rule that checks for the keyword 'test'.

The next step is ensuring that snort-wa will generate an alert if we cancel the network capturing. For this purpose, we will start the network capturing and kill the process using CTRL+C. This must send a SIGINT signal to the process and terminate it. The output of the snort service can be seen in Fig 12.

```
The network traffic capture of node with machine id 2e302cd720994da980609aeb999148d7 has been exited!
172.18.0.2 - - [04/Jun/2022 16:55:28] "POST / HTTP/1.1" 200 -
```

Fig. 12. Alert upon killing the network capturing process.

Now, we will try a real attack attempt by running a port scan on the network capturing node. The attacker can use this to find out which ports are open and whether anonymous logins are possible. For this purpose, we must enable the preprocessor called sfportscan, responsible for capturing such behavior in packets. After running nmap from the other worker node, Snort generates a log file that can be seen in Fig 13. The file gives the time of occurrence, the IPs involved, and other helpful information for the performed attack.

```
root@snort-wa-9c556f9f5-m6k6q:/var/log/snort# cat alert
Time: 06/04-18:29:18.184041
event_ref: 0
172.18.0.3 -> 172.18.0.2 (portscan) TCP Portscan
Priority Count: 9
Connection Count: 10
IP Count: 1
Scanner IP Range: 172.18.0.3:172.18.0.3
Port/Proto Count: 10
Port/Proto Range: 22:5900

Time: 06/04-18:32:06.013739
event_ref: 0
172.18.0.3 -> 172.18.0.2 (portscan) TCP Portscan
Priority Count: 9
Connection Count: 10
IP Count: 1
Scanner IP Range: 172.18.0.3:172.18.0.3
Port/Proto Count: 10
Port/Proto Range: 22:8080
```

Fig. 13. Port scan log file.

7.2 PERFORMANCE

The IDS would be wasteful if it overutilizes resources and forces applications to underperform. Therefore, an analysis of its CPU and memory footprint will be conducted. Considering the implementation, the components included in the analysis are the worker node executing the network capturing script and the pod running Snort.

The initial metric statistics for the node show that it uses approximately 0.16 CPU, having a maximum of 12 CPUs allocatable and 4 GB of memory, having a maximum of 24 GB allocatable. After running the network script, it averaged at 1.5 CPU and 4.7 GB memory and spiked once at 2.8 CPUs and 6.4 GB memory.

The pod which runs Snort has initial metrics that display 0 CPU usage and 0.049 GB memory. This is presumably, as it does not execute any tasks. When it started receiving traffic, it averaged at 2.5 CPU and 2.4 GB of memory, spiking to 3.1 CPU and 3.3 GB of memory.

Considering the number of packets that must be analyzed, resource utilization is reasonable. It can be significantly reduced if snort-wa is set up to scale horizontally, splitting the work across several pods.

7.3 OUTCOME

The paper discussed two options for capturing the network traffic inside the Kubernetes cluster – at pod level and node level. Both offer different security levels based on the analysis of how Kubernetes networking functions. The tools that can help with this task are Ksniff, TCP Dump, or Snort. Each of them simplifies capturing traffic and is suitable for different component level. Therefore, one must be chosen according to the situation and needs of the administrator.

The intrusion detection system can be either centralized or decentralized. The former represents a Kubernetes service, which will receive the traffic from all pods or nodes and run an analysis on it. The transfer of the network traffic can happen using HTTP, message queue, or FTP server. The latter option will require the

installation of the IDS on every component, which must be secured. The network capturing and packet analysis will happen on the same machine. Consequently, the need for communication is omitted.

The most optimal and efficient option is to use node-level network capturing as it offers the best level of security and centralized IDS placement because it will provide the opportunity to scale it horizontally. Moreover, it will use fewer resources than a non-centralized since there will be several pods running the IDS service instead of running it on every node in the cluster.

8 DISCUSSION

The results concluded from this paper give insight into what possibilities for IDS in Kubernetes exist and will add value to the existing research. The main findings include what alternatives exist to capture network traffic in Kubernetes, which may be helpful for other purposes, such as analytics or performance metrics. Moreover, the intrusion detection system design is not applicable only for Snort since most of them work in a similar matter or offer the option to feed them network packets directly. Since both the network capturing and the IDS are considered as a separate components, there is the freedom to choose better tools for doing the respective tasks. The implementation gave the idea of how performant the proposed solution is and how it behaves, which offers a base for improvement. Overall, some Kubernetes principles are violated by the proposed designs. A new topic to be investigated is how to avoid this and properly automate the network capturing and tool installation on nodes.

9 CONCLUSION AND FUTURE WORK

The need for robust security is increasing with the emerging world of technology. The outcome of this research gives initial insights into what possibilities exist in embedding an intrusion detection system inside Kubernetes.

The paper provided three different ways of capturing network traffic – TCP Dump, Ksniff, or Snort itself. The advantages and disadvantages of every choice were precisely discussed for optimal security during the implementation phase. Furthermore, an example of a potential implementation was given, which delivers a better sense of what would be more suitable in different situations.

The same applies to the usage and placement of intrusion detection systems inside a Kubernetes cluster. Centralized and non-centralized options were researched, supplying guidelines in which one would be more flexible and maintainable. In the end, running and successfully testing the signature-based IDS Snort in a multi-node cluster shows that the research has fulfilled its purpose.

Although this research gives a good impression of what is achievable in Kubernetes, it has some limitations. For a start, there are no design proposals for logging management in the case of centralized IDS with several instances of the IDS running. The current solution will result in a chaotic log structure spread across several pods and even lost logs in case of terminated pods. Next, it can be investigated how suitable it would be to include more security measures for intruders, such as honeypots and fake pods that trigger alerts when accessed.

10 ACKNOWLEDGEMENTS

I want to express my most profound appreciation to my supervisors Leon de Vries and Chakshu Gupta for helping me to choose the right topic, guiding me throughout the project and providing valuable feedback to extract the best of my work.

11 REFERENCES

[1] Osama Alkadi, Nour Moustafa, and Benjamin Turnbull. 2020. A Review of Intrusion Detection and Blockchain Applications in the Cloud: Approaches, Challenges and Solutions. *IEEE Access* 8, (2020), 104893–104917. DOI:https://doi.org/10.1109/ACCESS.2020.2999715

[2] Abhineet Anand, Amit Chaudhary, and M. Arvindhan. 2021. The Need for Virtualization: When and Why Virtualization Took Over Physical Servers. *Lect. Notes Electr. Eng.* 668, (2021), 1351–1359. DOI:https://doi.org/10.1007/978-981-15-5341-7_102/FIGURES/4

[3] Wisam Elmasry, Akhan Akbulut, and Abdul Halim Zaim. 2021. A Design of an Integrated Cloud-based Intrusion Detection System with Third Party Cloud Service. *Open Comput. Sci.* 11, 1 (January 2021), 365–379. DOI:https://doi.org/10.1515/COMP-2020-0214/PDF

[4] Mohamed Idhammad, Karim Afdel, and Mustapha Belouch. 2018. Distributed intrusion detection system for cloud environments based on data mining techniques. *Procedia Comput. Sci.* 127, (2018), 35–41. DOI:https://doi.org/10.1016/J.PROCS.2018.01.095/DISTRIBUTED_INTRUSION_DETECTION_SYSTEM_FOR_CLOUD_ENVIRONMENTS_BASED_ON_DATA_MINING_TECHNIQUES.PDF

[5] Md Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, and Akond Rahman. 2020. XI Commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices. *Proc. - 2020 IEEE Secur. Dev. SecDev 2020* (September 2020), 58–64. DOI:https://doi.org/10.1109/SECDEV45635.2020.00025

[6] Aws Naser Jaber and Shafiq Ul Rehman. 2020. FCM–SVM based intrusion detection system for cloud computing environment. *Cluster Comput.* 23, 4 (December 2020), 3221–3231. DOI:https://doi.org/10.1007/S10586-020-03082-6

[7] Uttam Kumar and Bhavesh N. Gohil. 2015. A Survey on Intrusion Detection Systems for Cloud Computing Environment. *Int. J. Comput. Appl.* 109, 1 (January 2015), 6–15. DOI:https://doi.org/10.5120/19150-0573

[8] Zhi Li, Haitao Xu, and Yanzhu Liu. 2017. A differential game model of intrusion detection system in cloud computing. *Int. J. Distrib. Sens. Networks* 13, 1 (January 2017). DOI:https://doi.org/10.1177/1550147716687995

[9] Chirag N. Modi and Kamatchi Acha. 2017. Virtualization layer security challenges and intrusion detection/prevention systems in cloud computing: a comprehensive review. *J. Supercomput.* 73, 3 (March 2017), 1192–1234. DOI:https://doi.org/10.1007/s11227-016-1805-9

[10] Aneta Poniszewska-Marańda and Ewa Czechowska. 2021. Kubernetes cluster for automating software production environment. *Sensors* 21, 5 (March 2021), 1–24. DOI:https://doi.org/10.3390/S21051910

[11] Rosehosting. 2016. Physical vs Virtual server. *Blogs* (2016).

[12] Syed Ali Raza Shah and Biju Issac. 2018. Performance comparison of intrusion detection systems and application of machine learning to Snort system. *Futur. Gener. Comput. Syst.* 80, (March 2018), 157–170. DOI:https://doi.org/10.1016/J.FUTURE.2017.10.016

[13] Chin-Wei Tien, Tse-Yung Huang, Chia-Wei Tien, Ting-Chun Huang, and Sy-Yen Kuo. 2019. KubAnomaly: Anomaly detection for the Docker orchestration platform with neural network approaches. *Eng. Reports* 1, 5 (December 2019), e12080. DOI:https://doi.org/10.1002/ENG2.12080

[14] Irene Ann Tony. 2021. Application of Machine Learning with Traffic Monitoring to Intrusion Detection in Kubernetes Deployments. (2021).

[15] Xinchen Xu, Aidong Xu, Yixin Jiang, al -, Falk Herwig, Robert Andrassy, Nic Annau, Gengsheng Zheng, Yao Fu, Tingting Wu -, Vipin Jain, Baldev Singh, Medha Khenwar, and Milind Sharma. Static Vulnerability Analysis of Docker Images You may also like Research on Security Issues of Docker and Container Monitoring System in Edge Computing System Cyberhubs: Virtual Research Environments for Astronomy Research on Docker Cluster Scheduling Based on Self-define Kubernetes Scheduler Static Vulnerability Analysis of Docker Images. DOI:https://doi.org/10.1088/1757-899X/1131/1/012018

[16] What is Kubernetes? | Kubernetes. Retrieved June 19, 2022 from https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[17] CVE-2019-5736- Red Hat Customer Portal. Retrieved May 3, 2022 from https://access.redhat.com/security/cve/cve-2019-5736

[18] 9.1. Runmodes — Suricata 6.0.0 documentation. Retrieved June 2, 2022 from https://suricata.readthedocs.io/en/suricata-6.0.0/performance/runmodes.html

[19] Understanding the Snort architecture. Retrieved June 25, 2022 from https://truica-victor.com/snort-architecture/

[20] Network overview | Kubernetes Engine Documentation | Google Cloud. Retrieved June 19, 2022 from https://cloud.google.com/kubernetes-engine/docs/concepts/network-overview

[21] Snort - Network Intrusion Detection & Prevention System. Retrieved May 28, 2022 from https://snort.org/rules_explanation

APPENDIX

A. NETWORK CAPTURING SCRIPTS

Script executing TCP dump with PCAP generation every two seconds.

```
#!/bin/sh

# using this kubectl command you can find the machineId of every node in the cluster
# kubectl get nodes -o json
exit_script() {
  echo "Exiting capturing traffic"

  # pass the machineId of the current node
  machineId=$(cat /etc/machine-id)

  Ping snort-wa that somebody stopped this process
  curl -d "machineId=$machineId" 10.96.133.99

  trap - INT TERM EXIT
  kill -- -$$
}

trap exit_script INT TERM EXIT

tcpdump -s 0 -i eth0 -G 2 -w %H%M%S.pcap -z ./sendpcap.sh -Z root

```

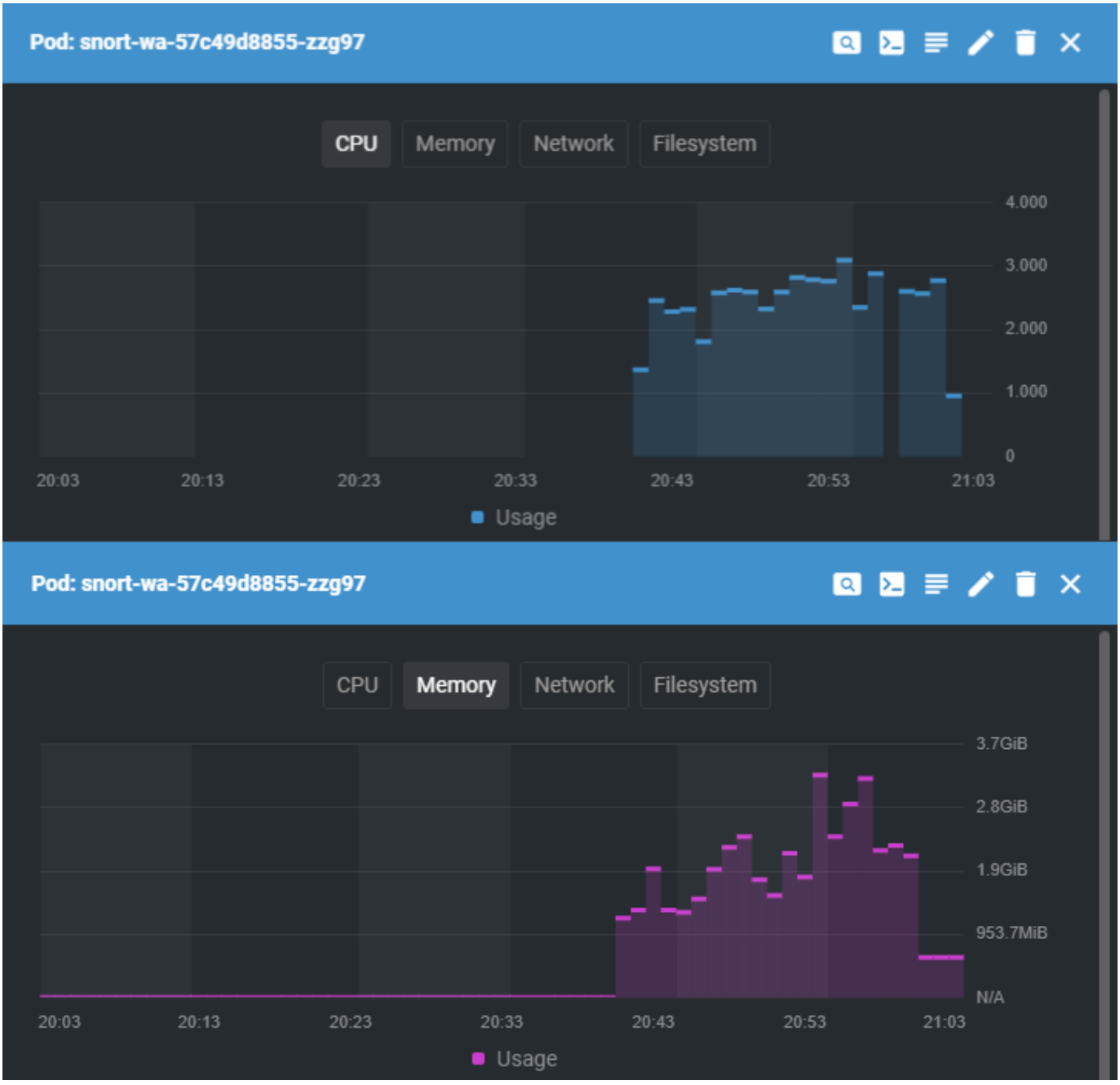
Script to pipe the PCAP file to the Snort service

```
#!/bin/sh

curl -F "pcap=@$1;filename=output.pcap" 10.96.133.99
rm $1
```

B. PERFORMANCE METRICS OF NETWORK CAPRUTING AND IDS ANALYSIS

Snort pod executing network traffic analysis CPU (top) and memory (bottom) statistics.



Worker node executing network traffic capturing CPU (top) and memory (bottom) statistics.