Verifying the consistency of security policies across the Kubernetes Stack

*By Jasper Goris*

# Kubernetes basics

Kubernetes is an open-source system that allows automatic deployment, scaling and management of containerized applications. The Kubernetes installation as a whole is called a cluster, which consists of a set of nodes. There are 2 types of nodes: master (also called control plane) nodes, which are responsible for making global decisions for the cluster, and worker nodes. Within Nodes we have Pods which in turn hold containers that run the containerized applications itself.

Diagram

Description automatically generated

*Image from XI commandments paper (****Todo Reference****)*

The users interact with the cluster trough Kubectl (the command line interface) or the Kubernetes dashboard. The user input then arrives at the API server which orchestrates the operations that come in. The Controller watches the API server and updates the current states accordingly. The Scheduler will handle all scheduling of the pods across the worker nodes. The last component in the master node is the ETCD database. This is a simple key-value database to store configurations.

A worker node consists of three main components: firstly we have the Kube-proxy which will maintain the network rules. The Kubelet’s functionality is ensuring that containers are not going rogue and are running outside their assigned Pods. The Pods are the smallest Kubernetes entity and contain one or more containerized applications (e.g. a database built with a Docker image).

The biggest strengths of Kubernetes are its autoscaling and deployment of pods depending on the application’s needs. When the application needs more memory and/or processing power pods can be replicated to meet these needs. When the needs are scaled down again so are the pods by deleting some of them. This way the availability and response times of the application can be kept within certain thresholds.

# The 4 C’s of Kubernetes security

Table

Description automatically generated with medium confidence

***Image from Kubernetes website (TODO REFERENCE)***

Kubernetes itself talks about the 4 C’s of security on their website: The code, container, cluster and cloud layer.

The code layer falls outside of the scope for my project, but it goes without saying that these are essential for safe practices. If you have an insecure codebase the consequences might be dire.

The Container layer is the responsibility of the Application manager. This is the person that develops, configures, controls or monitors an application that runs in the container. The application manager decides how many nodes need to be available, what databases we might use (including backups) etc.

The Cluster however is managed by the Cluster manager who installs, configures, controls and monitors the cluster (aka the Kubernetes installation). A handy tool for the Cluster manager is namespaces: These allow the grouping of resources within the cluster in order to isolate them using policies and define a limit to their allowed resources.

The cloud layer is out of scope in a sense: My thesis will stop at the cluster level with its nodes.

We have what I will call the multiple truths problem: What if the Cluster and Container layer have different policies that contradict each other, defined by different managers. Which one is right and wrong, which one has priority? This is hard to determine since the policies might be made with different reasons in mind. For example the Cluster manager might want to box the nodes in, in terms of security policies being as restricting as possible, to minimise the risk of malicious attacks spreading to other Nodes. The Application manager however might want all of it’s pods to freely communicate at all times and be reachable by any users.

# Container security policy example

# Diagram Description automatically generated

***Figure 1 from Kano paper: TODO REFERENCE***

Kubernetes allows to define network policies on the container level that define how a pod is allowed to communicate with other entities. There are two sorts of isolation that we can apply to a pod: egress and ingress isolation. By default a pod is non-isolated for all ingress and egress, in other words all communication is allowed to go in and out of the pod. It is possible to set these to isolated while creating a whitelist of allowed communications. This enables us to use least-privilege principles in communication: only what is necessary should be non-isolated and all the rest isolated. Misconfiguration however might still happen and have negative consequences. Let me show you an example:

Here we have an example cluster with multiple containers to show the importance of aligning security policies. The example comes from the Kano research paper. (***TODO REFERENCE***).\*

In this example we have 2 users: Bob and Alice, running within the same container cluster. Alice wants her Nginx container to be available to everyone, while her Tomcat container is only reachable through the nginx container. Therefore, she might set the rule that Nginx containers can reach Tomcat containers. However, she did not think about the Nginx container from Bob, who now can also access Alice’s Tomcat container. Bob now might even access Alice her Database without Alice knowing. Further so other users might do the same through Bob his Nginx container.

Solving this issue is trivial when we know exactly which policies are involved and luckily some solutions exist for verification of security policies. notable mentions would be NFVGuard (Cloud level), TenantGuard and Kano. I will go slightly deeper into Kano and also show its limitations.

# Kano: The solution and it’s limits

Let’s talk about Kano: What it does, how it works and where it’s limits lie.

Diagram

Description automatically generatedDiagram

Description automatically generatedA picture containing text, calculator, electronics

Description automatically generated

As mentioned before Kano verifies security policies while also maintaining a good time complexity. It does so on the Container level of the 4 C’s   
It starts with modelling a container network as bipartite graphs with two sets of independent nodes representing Egress and Ingress of all containers as defined by the policies. The edges represent the allowed network policies, which we can then split up in two sets: A set of allowed egress policies and a set of allowed ingress policies. We than create the join connection set E (E1 ∩ E2) which shows the connections allowed by both policies. To make computations easier this is not saved as graphs but transformed into a matrix. The egresses are rows while ingresses are columns. The values remain binary, with a 1 in row x (Egress) and column y (ingress) representing x is allowed to reach y. This is called the reachability matrix

A picture containing text, receipt

Description automatically generated

With this it is easy to check whether the policies that were defined are violated or not.   
We can read the matrix to find all reachable (allows all ingress), all isolated (denies all ingress), reachable between 2 users (they can connect) and system isolations (meaning the container is isolated with a container, usually the kube-system container). Policy shadow shows us policies that are entirely covered by other policies and are thus possibly redundant. Policy conflict on the other hand shows how 2 policies are contradicting of each other.

Kano also allows to define new violations to be checked, includes algorithms to scale down the time complexity and offers solutions with a fix advisor. However Kano is limited to the Container level of the 4 C’s of Kubernetes security.

# Microsegmentation

So now we have seen a solution for security policy verification on the Container level, however there is a missing link: aligning the policies throughout the different layers of the 4C’s model of Kubernetes.

This is exactly where I want to put my focus on and where the existing solutions, often referred to as the microsegmentation tools come short. These so-called micro-segmentation tools come in a wide variety with slightly different approaches. But as I just said they fail to correctly synchronise the security policies on all the different levels thoroughly. The idea of microsegmenation is looking at the resources and giving them all a unique ID based on name, location and other variables. These Id’s are then used to define the policies instead of using IP’s, ports etc.   
  
After consideration with my promotor we decided that, even though my thesis will lean on the idea of microsegmentationm I will not go into depth around the microsegmentation tools.

# The thesis

So what would be part of my thesis? I would focus on the consistency issues between the application and cluster layer in the security stack of Kubernetes. This is not a trivial issue due to the autoscaling nature of Kubernetes where the environment might change often due to for example pod deletion and replication. The cloud layer might have less changes: nodes and node ports might remain quite static, but If changes occur nonetheless verification might be necessary again. A Naïve solution might be to fully verify all the policies on both the different layers every time a change happens somewhere in the technology stack. But this will result in a terrible computational overhead that could impact the availability of the applications. To find a solution I would like to use the matrix generated by Kano, and combine it with my own code solution. But this also brings up another roadblock.