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 managed by a Cloud administrator. He sets up the physical servers, manages VM’s, allocates resources etc.

We have what I will call the multiple truths problem: What if the cloud and Container layer have different policies that contradict each other. 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 Cloud manager might want to box the cluster in as much as possible to minimise the risk of malicious attacks spreading to other VM’s. The Cluster manager however might want all of it’s nodes 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-privilige 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, TenantGuard and Kano. I will go slightly deeper into two of these and also show their 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.

# NFVGuard

***Image from NFVGuard paper (TODO REFERENCE)***

***Diagram

Description automatically generated***

So let us scope away from the container level and look at NFVGuard next. While Kano verifies security policies on the container level NFVGuard verifies policies on the Cloud level.   
For those unfamiliar, NFV is a network architecture concept that virtualizes network functions like routers, load balancers, etc to enable the hosting of multiple VM’s on the same physical network and devices. The NFV Stack consists of four levels as indicated in the figure:

Level 1 is the Service Orchestration and it holds the specification and lifecycle management of network services.  
Level 2, the Resource Management supports the instantiation of these services and manages compute, storage and network resources  
Level 3 is the Virtual Infrastructure which hosts the virtual resources to support the higher levels.  
Level 4 is the final level and is the Physical Infrastructure like the router, ethernet kabels etc.

A naïve solution to verify security properties would be to verify all four levels against each other. Ofcourse this could become exponentially computationally expensive by increasing number of security properties.   
NFVGuard proposes to verify every property at the level where it is specified and extend the verification results to other levels by verifying the consistency between adjacent levels.

Diagram, engineering drawing

Description automatically generated

NFVGuard does this in 3 main steps:

First It will identify all the security and consistency properties. Because of the NFV stack nature it collects this from multiple sources all throughout the stack: logs, configuration files etc.

After this the data needs to be correlated within each level based upon their relationships. The goal is to make sure all the data that is left is grouped and transformed into usable information.

The last step is verification itself. For this NFVGuard uses a Constraint Satisfaction Problem (CSP) solver called Sugar. Relationships between system entities are encoded as CSP relations, and security properties as predicates to from a CSP constraint. Negation of these constraints allow us to use Sugar to solve them. If no solution is found the properties are reported to hold.   
  
One of the biggest limitation of NFVGuard is that the current implementation is limited to Openstack/Tacker. The authors however have stated the desire to extend this to other open-source NFV platforms like OPNFV and OSM. Once again it is also still limited to the cloud layer.

# Microsegmentation

So now we have seen a solution for security policy verification on the Container level, namely Kano, and a solution with the same goal but on Cloud level, namely NFVGuard. There is however 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 different layers 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: VM’s and interfaces 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 all 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 parts of both NFVGuard and Kano, and combine them with my own ideas. But this also brings up another roadblock. One of the biggest weaknesses of NFVGuard is that is specific to the Openstack software. To make my solution viable I might need an abstraction layer so it is reusable with different technologies in the future.

So summarised I first need to delve deeper into the most common microsegmentation tools to find the gaps they leave. Afterwards I need to find a way to compare the policies between the different levels of the Technology stack by leveraging the techniques used in Kano and NFVGuard. The solution needs to be abstract enough to be used in combination with different Cloud provider technologies and Kubernetes, while also keeping the computational and time overhead to a minimum. Not a small task but let’s look at my milestones for this plan.

# The plan:

1. **Literature study (finished)**

I read up about Kubernetes, Cloud architecture, micro-segmentation tools, security tools and papers to get a understanding about the topics. Notable sources are kept for later reference.

1. **Define scope of the thesis**This is the current step as I am writing this. Deciding exactly where I will focus on and what my next steps will be. This will be a red wire throughout the rest of my thesis.
2. **Define research questions and thesis title**

If stap 2 does not yield unexpected results I will continue with defining my thesis title and research questions

Proposal title: “Verifying security policies throughout the Kubernetes and VNF stack”

Proposal questions:

TODO

1. **Construct the algorithms for my solution**

This will probably consist of a few steps:

* **Create a list of Containers/Nodes/Security groups and the VM’s they run on**

It will be important to know the location of our Kubernetes components to compare the Container level policies with the correct Cloud level policies later on.

* **Generate inter-VM reachability communication matrix by policies on cloud level**Now I generate a NxN matrix that shows all the communication ingress and egress between the existing VM’s. Of course we are not interested in VM’s where none of our pods are deployed. So the size of N is defined by the amount of unique VM’s. I will look into the NFVGuard implementations for reusable patterns and at Grashopper to possibly create security groups. This will hopefully help in making this matrix generation algorithm general enough and not software specific for e.g. Openstack.
* **Compare Kano Container reachability matrix with inter-VM communication matrix** Taking the previously made inter-VM matrix and Kano’s reachability matrix we can start looking for inconsistencies.   
  For example: if containers don’t allow communication between them (shown in Kano matrix), then the VM’s upon which the pods of the containers are deployed should not contradict this by allowing communication between them nonetheless (shown in inter-VM matrix).   
    
  The results of this could possibly be shown once again in a matrix. This NxM matrix would need the N container policies and M inter-VM policies and indicate whether or not they contradict. However, if multiple policies are involved in a single inconsistency this might not be viable. Looking for a way of representing the solutions while keeping in mind the aforementioned ‘2 truths’ problem which changes the definition of desirable and undesirable policies will be a big part of my thesis.
* **Define events that force recalculation of (part of) the solutions algorithm**As mentioned before running the entire algorithm again every time something changes anywhere in the stack between the Cloud and the Container layers will lead to many executions. I need to define a minimum set of events that will trigger (partial) execution of the solutions algorithms.
* **Code an implementation for Kubernetes and e.g. Openstack**To fully test the solution I will need a benchmarking setup on which to run experiments and an implementation for those platforms of the algorithms.

1. **Time complexity**

Testing the algorithms with the implementation of the benchmarking setup and finding the best possible time complexity in order to verify feasibility will be necessary. Due to the continuous deployment of pods the solution will have to be run often, meaning a minimum of time and computational overhead is needed.

1. ***Propose solutions to fix the found policy contradictions***

Only mentioning that a problem has arisen does not solve the problem itself. The end user will need to know which policies are conflicting and would have great benefit on a list of possibilities on how to fix them.

1. **Answer the research questions**

I will now need to answer my previously created thesis questions.

1. **Thesis paper**

Hopefully I have been able to keep my final paper up to date as I went trough the previous steps. This is however when I need to finalise this. The final deadline for this step is 4th of June.