CARIAD | docker

Using Docker Desktop in Large-Scale Enterprises

A practical guide for a compliant setup and usage of linux and windows containers on windows workstations within complex corporate environments using docker desktop.

Written By CARIAD SE
Valentin Pravtchev
Julius Pravtchev

Reviewed By Docker Inc.
Gabriela Georgieva
Jason Bivins
Marc Sherwood

Introduction

In 2013, Docker Inc. revolutionized the container landscape by releasing its first version of the Docker container engine. Characterized by an intuitive user interface and a robust container ecosystem, Docker simplified the complex task of setting up containers.

Since 2013, the use of container manager solutions has become highly diverse. Kubernetes has become the primary choice for targeting productional server platforms or managing heavy workloads within Continuous Integration and Continuous Deployment (CICD) environments. Regarding the inner development loop on local developer machines, Docker Inc. has introduced another container manager known as Docker Desktop, which, in turn, has proven to be an excellent workstation solution. It strongly emphasizes facilitating a seamless human-machine interaction and is shipped with many enterprise-grade security controls, making it highly suitable for corporate environments.

Large enterprises typically maintain a heterogeneous development stack, targeting both Windows and Linux platforms. This diversity arises from various factors, including certification of specific tools for a single operating system, lack of support for certain tools on alternative operating systems, or contractual obligations with vendors. Consequently, Docker Desktop becomes an invaluable asset in such complex setups, as it provides a unique solution to seamlessly operate with both Windows and Linux containers on a single (Windows) machine.

While it can be easily installed on personal workstations, deploying and operating Docker Desktop within complex enterprise environments alongside proxies, file encryption, or virtual private networks (VPN) is challenging. The strict compliance that many enterprises must guarantee further intensifies this complexity.

This whitepaper addresses typical challenges when setting up and working with Docker Desktop inside huge-scale enterprises. It aims to support companies in securely deploying and maintaining Docker Desktop within their compliant environments.

Contents

Understanding the system landscape

Comparative look at Docker Desktop against Docker CE

Docker Desktop, developed and released by Docker Inc., is a multi-component application serving as an alternative container management solution alongside the conventional Docker CE. Contrary to a widespread misconception that Docker Desktop, formally known as *Docker On Windows*, is exclusively available on Windows, it has extended full support on various Linux distributions and MacOS.

As illustrated in **Figure 1**, Docker Desktop encompasses all features present in Docker Engine while providing a comprehensive user interface alongside various enterprise-grade features. Especially, its ability to seamlessly switch the development context between the Linux and Windows container engines from a single instance is an invaluable asset for large-scale enterprises requiring containerization on both Windows and Linux development tools.

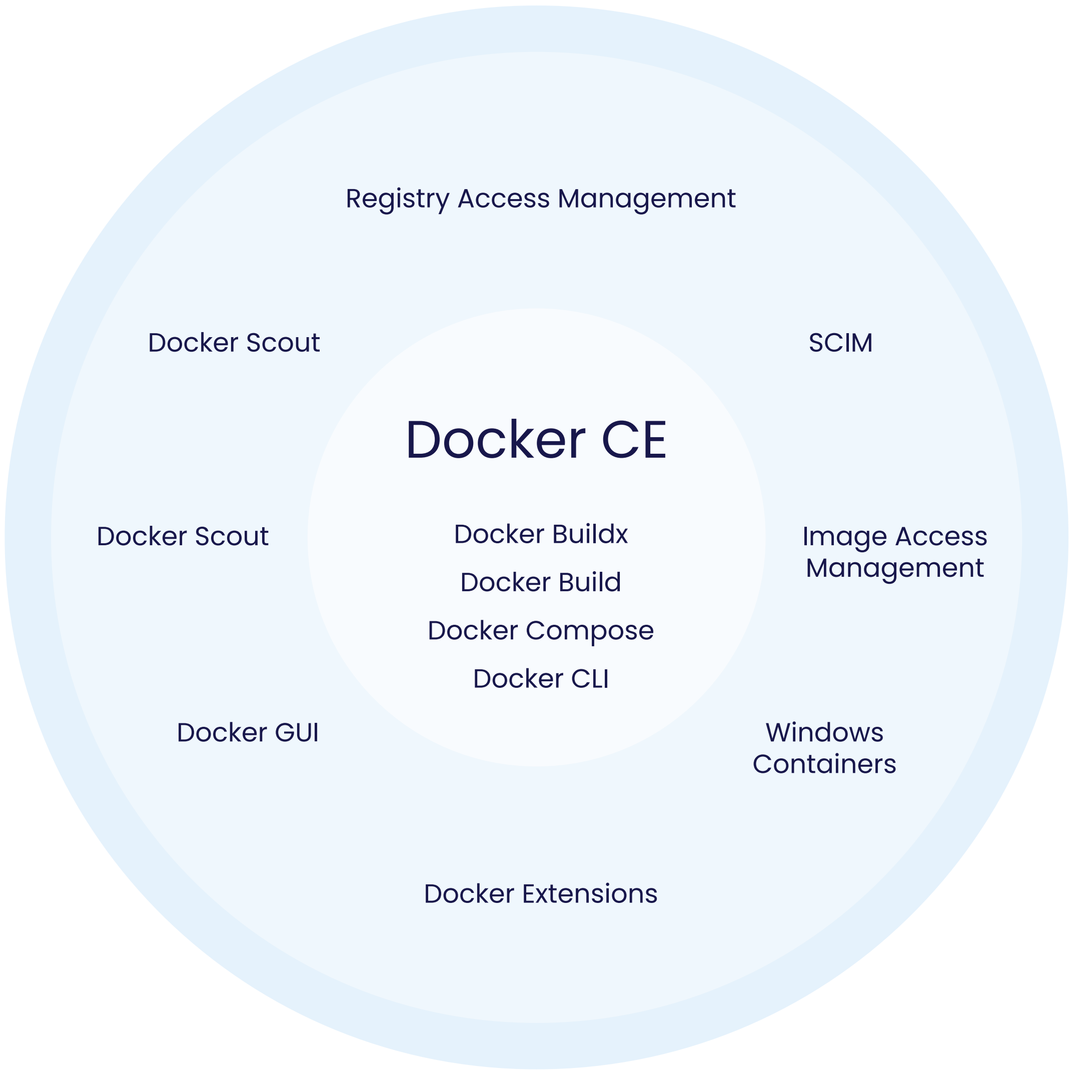


Figure 1: Overview of the capabilities of Docker CE in comparison to Docker Desktop. Docker Desktop encompasses all essential features of Docker CE while notably enhancing its functionalities, particularly enterprise-grade features.

Exploring container manager solutions on Windows workstations

Linux containers vs. Windows containers

On Windows workstations, developers have various options beyond Docker Desktop for implementing containerization. Integrating a full-featured Linux kernel through the so-called Windows Subsystem for Linux 2 (WSL2) in a highly optimized Hyper-V virtual machine significantly enhanced the seamless utilization of Linux containers directly from Windows. However, containers are not an exclusive Linux feature, as Windows incorporated containerization into its NT kernel with the release of Windows Server 2016. All major Windows platforms currently support the containerization of Windows applications, mainly following the principles of Linux containers. Nevertheless, Windows containers differ significantly in terms of version compatibility, runtime-level architecture, image size, and security considerations.

Introducing Linux container solutions

According to Linux containers, users can work with powerful alternatives such as *Podman Desktop* by Red Hat Inc. or *Rancher Desktop* by SUSE. Both tools follow the same approach, like Docker Desktop, where the command line and desktop interface are installed directly on Windows while deploying the container engine inside the WSL2 environment. *Podman Desktop* is specifically designed to work seamlessly with the Podman container engine, which deploys 'rootless' containers known for their pivotal security benefits. Rancher Desktop, in turn, wraps various container tools such as `docker-cli` or `nerdctl` on the Windows side, along with `dockerd` or pure containerd as the container engine inside WSL2.

Finally, users can install any container manager, such as the Docker or Podman engine, directly within a WSL2 Linux distribution. This approach comes with the notable drawback that this kind of installation is limited to a single WSL2 distribution.

Introducing Windows container solutions

Regarding Windows containers, *Rancher Desktop* and *Podman Desktop* currently lack support. In this scenario, the Docker CE / Moby for Windows, the *Mirantis* container runtime, or the containerd / nerdctl toolchain are possible alternatives to Docker Desktop. The latter claims to be a compatible option with Docker; however, it only provides experimental Windows support and, thus, is not suitable for commercial usage. In contrast, the enterprise-focused Mirantis container runtime, formerly known as Docker Enterprise Edition (EE), proves to be a robust choice for working with Windows containers.

Unveiling the Docker Desktop system architecture

Exploring complexities of Linux container engine Architecture

As illustrated in **Figure 3**, Docker Desktop implements a hybrid system architecture. The command-line interface, `docker.exe`, and the graphical user interface, `docker-desktop.exe,` are directly deployed on the host alongside crucial helper processes like `com.docker.backend.exe`, `com.docker.service.exe`, etc. At the same time, the container engine `dockerd` operates within the WSL2 virtual machine. Within WSL2, containerd is employed to launch a container hosting various services to manage containers, including dockerd. The launch and setup of the WSL2 virtual machine is initiated by the helper service `com.docker.service.exe` that interacts with the `wsl.exe`, which, in turn, utilizes the Windows services *Linux Subsystem Manager Service* (LXSS) and *Host Compute System* (HCS) to manage the lifecycle of the WSL2 virtual machine and to provide the general virtualization platform on operating system level.

Besides the resources mentioned above, the Docker Desktop installer also deploys two internal WSL2 distributions: `docker-desktop and docker-desktop-data`. The `docker-desktop` distribution hosts the resources, such as the Docker engine, to work on containers. If Enhanced Container Isolation (ECI) is not enabled, the following commands demonstrate how containerd is leveraged to employ the Docker engine, `dockerd`, within a dedicated container:

```
# boot docker-desktop
PS C:\Users\MYUSER> wsl -d docker-desktop
# get processes
wslguest:/tmp/docker-desktop-root/mnt/host/c/Users/MYUSER# ps aux | grep dockerd
   609 root   0:00 /usr/bin/logwrite -n dockerd /bin/sh -c export LISTEN_PID="$$"; exec /usr/local/
bin/dockerd --containerd /run/containerd/containerd.sock --pidfile /run/desktop/docker.pid --swarm-
default-advertise-addr=eth0 --host-gateway-ip <host-ip>
   615 root   0:00 /usr/local/bin/dockerd --containerd /run/containerd/containerd.sock --pidfile /
run/desktop/docker.pid --swarm-default-advertise-addr=eth0 --host-gateway-ip <host-ip>
```

Figure 2: Sequence of operations involves starting the special-purpose WSL2 distribution, `docker-desktop`, and then inspecting the processes within this distribution to identify those responsible for initiating `dockerd` within a containerd container

In addition, the `docker-desktop-data` distribution hosts containers and images. Both distributions are not extendable as fully featured development environments, which, in turn, provide substantial security advantages; however, they come with significant drawbacks. Because of the missing Linux filesystem, all projects must be hosted on the Windows filesystem and either bind-mounted into Linux containers or persisted through volumes. This arrangement leads to noticeable performance issues due to file system translation tasks, making it unsuitable for large-scale Linux-based projects.

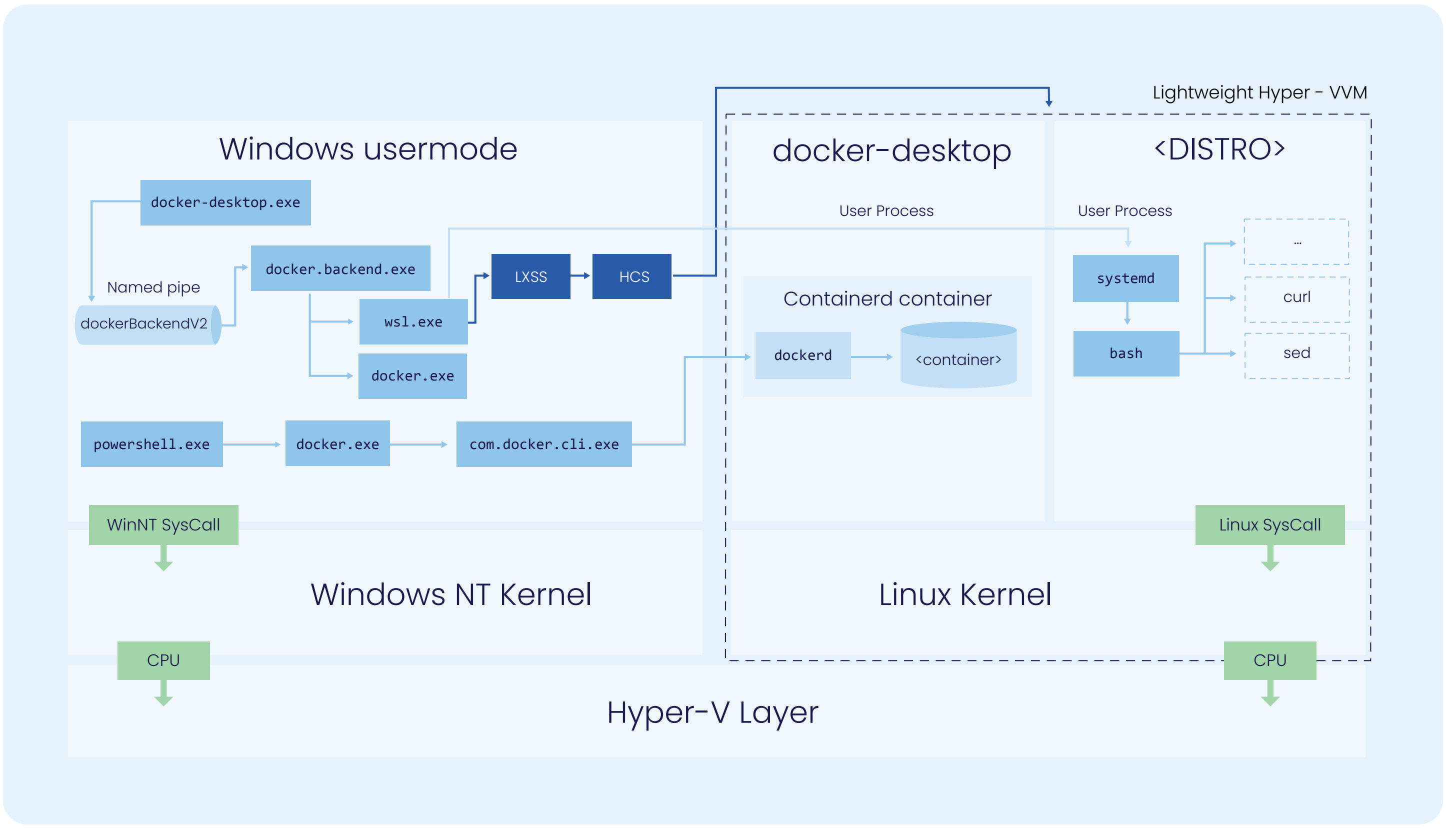


Figure 3: Abstract system architecture of Docker Desktop's Linux container backend using the Windows Subsystem For Linux 2 (WSL2).

It is crucial to emphasize that the Linux container engine, `dockerd`, operates with elevated `root` rights within a WSL2 distribution. While this poses a significant security risk in native Linux platforms, it is more or less negligible in the context of virtual WSL2 environments. WSL2, instead, is designed as a development tool rather than an operating system used for any productional scenario. It allows a seamless `root` login and highly integrates with Windows tools such as VS *Code, Notepad++* or *Windows Explorer*. In addition, these elevated privileges cannot be misused to gain `SYSTEM` rights on the Windows host since WSL2 distributions operate exclusively within the context of the local Windows user and fully honor the permissions. However, this does not imply that WSL2 could not be an entry point for any attack, but it demands substantial effort to exploit `SYSTEM` rights through WSL2 successfully. Furthermore, Microsoft promptly addresses and resolves any security findings regarding WSL2. Large-scale enterprises should consider implementing hardening measures to their WSL2 fleet, as outlined in *Securing Windows Subsystem For Linux 2* (WSL2). These measures are crucial since WSL2 partially lacks enterprise-grade security controls, especially when utilized on Windows 10.

Exploring complexities of Windows container engine architecture

As shown in **Figure 4**, all required resources and processes that manage Windows containers are directly deployed on the Windows host, including the two privileged processes `com.docker.service.exe` and `dockerd.exe`. While the first process functions as the primary Windows container engine, the latter is a privileged helper service running in the background and handling all `SYSTEM` operations necessary for Docker Desktop to maintain the Windows container environment. This, for instance, includes managing the life cycle of the Hyper-V `DockerDesktopVM (DockerVM)`, initiating and terminating `dockerd.exe`, or caching certain security policies. Like setting up Docker CE on Linux and Windows, low-privilege users can seamlessly work with Docker Desktop when provided to the local Windows group `docker-users`. Its members are fully permitted to access the protected named pipe `.//./pipe/dockerBackendV2`, where both `docker-desktop.exe` and `com.docker.service.exe` are directly connected.

As described in *Unveiling Windows Containers best practices*, the Windows container daemon is a potential entry point for a `SYSTEM` exploit. In contrast, the `com.docker.service.exe` provides a minimal attack surface without going into detail and requires at least a *Code Injection Vulnerability*. It's worth noting that specific versions of Docker Desktop disabled the service from permanently running in the background. This must be considered when deploying Docker Desktop at scale without granting elevated rights to the users. Furthermore, there are configuration options to disable the usage of Windows containers entirely.

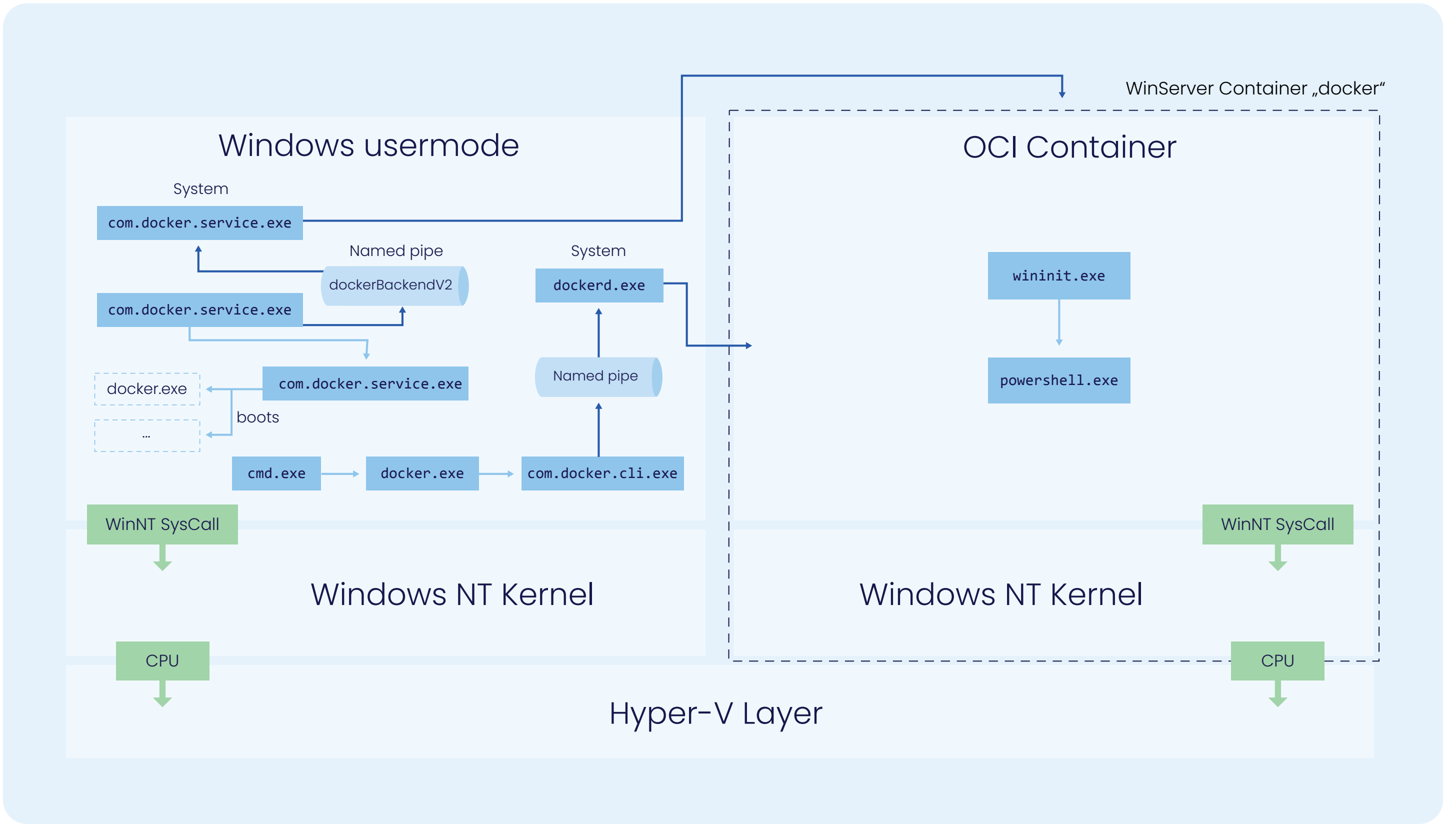


Figure 4: Abstract system architecture of Docker Desktop's Windows container backend in hyperv-isolation mode

Windows containers can operate in two isolation modes: *process-isolation* and *hyperv-isolation*. The first mode isolates containers through traditional namespaces, resource control, etc., and deploys containers directly on the host kernel; this isolation mode is analogous to conventional Linux containers running on a Linux host. The *hyperv-isolation* starts a stripped-down but highly optimized Hyper-V virtual machine named `DockerDesktopVM (DockerVM)` and deploys a Windows-based container into it. This gives each container a separate kernel and, in consequence, provides isolation on the hardware level, which is, in turn, a firm security boundary.

On Windows 10 and 11, the container engine defaults to running containers in *hyperv-isolation*, whereas on Windows Server, the default isolation mode is *process-isolation*. In addition, only versions of Windows 10 build 17763 or greater provide experimental support for *process-isolation*, with a strong emphasis on not using this setup for any productional scenario.

Tackling enterprise challenges with Docker Desktop

Crafting a seamless user and license management

Choosing the suitable pricing model

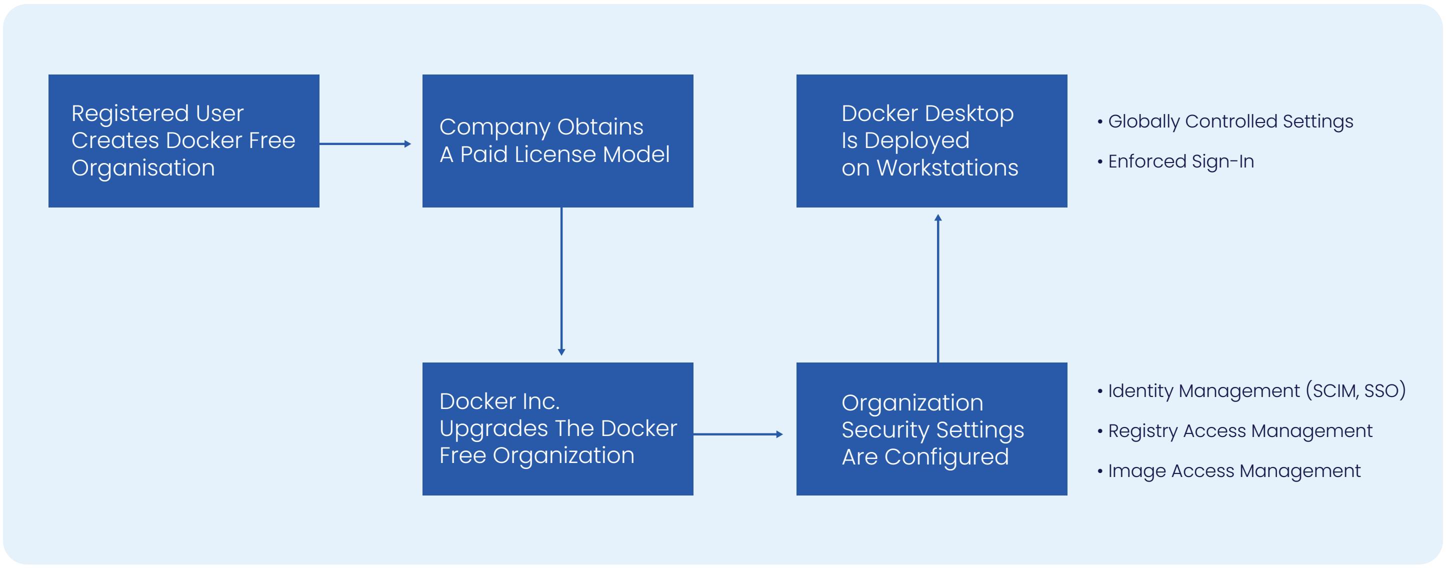
Large enterprises, defined as those with over 250 employees or an annual revenue exceeding $10 million, necessitate a paid subscription for the commercial use of Docker Desktop. Although this is a mandatory requirement, there are no additional restrictions on selecting the specific pricing model. Following **Table 1** outlines available pricing models and the inclusion of essential enterprise-grade security controls.

	Pro	Team	Business
Kubernetes	+	+	+
Commercial Support	(+)	(+)	+
Single Sign-On (SSO)	-	-	+
Cross-Domain Identity Management (SCIM)	-	-	+
Hardened Docker Desktop (incl. Settings Management)	-	-	+
Image Access Management	-	-	+
Registry Access Management	-	-	+

Table 1: Excerpt about critical enterprise-grade security controls that should be considered when choosing a mandatory pricing model for the commercial use of Docker Desktop.

It is obvious that the *Business Subscription* encompasses all essential enterprise-grade security features necessary for a compliant usage of Docker Desktop within large-scale enterprises. It includes support for Single Sign-On (SSO), System for Cross-Domain Identity Management (SCIM), standardized deployment on regulated workstations at scale through Settings Management, and integration possibilities with third-party platforms like *JFrog Artifactory* or *Azure Container Registry* through Registry Access Management to secure the whole container ecosystem of a company. These features contribute to a more secure software supply chain in concert with secure practices across other tools and processes.

While *Handling huge-scale deployments* will describe the typical challenges of a compliant and large-scale deployment of Docker Desktop on tightly regulated workstations, the following sections will outline the main configuration steps of the *Docker Business Organization* itself. **Figure 5** gives an overview of the initial steps required to purchase a paid license, set up a Docker Hub Organization, and securely deploy Docker Desktop on a local developer workstation.



Registered User Creates Docker Free Organisation
Company Obtains A Paid License Model
Docker Desktop Is Deployed on Workstations
• Globally Controlled Settings
• Enforced Sign-In
Docker Inc. Upgrades The Docker Free Organization
Organization Security Settings Are Configured
• Identity Management (SCIM, SSO)
• Registry Access Management
• Image Access Management

Figure 5: Flow chart about the main steps to set up and configure Docker Desktop in large-scale enterprises.

Setting up user access control

Docker Desktop's Business subscription offers the capability to restrict local access to Docker Desktop and its resources exclusively to users who are authenticated through the company's identity providers (IdPs). Those attempting to use personal login credentials are restricted. Consequently, all global security settings are consistently applied, and the user can fully leverage the features provided by the respective pricing model. This is possible by the non-negotiable features *Enforced Sign-In* and *Single Sign-On (SSO)*. While *Crafting* a smooth installation procedure outlines the activation of *Enforced Sign-In* on local workstations, implementing a proper *SSO integration* is required first.

The SSO integration follows the workflow outlined in **Figure 6**. It is important to note that SSO can also be configured via the *Admin Console*, however, following steps refer on the traditional setup in Docker Hub.

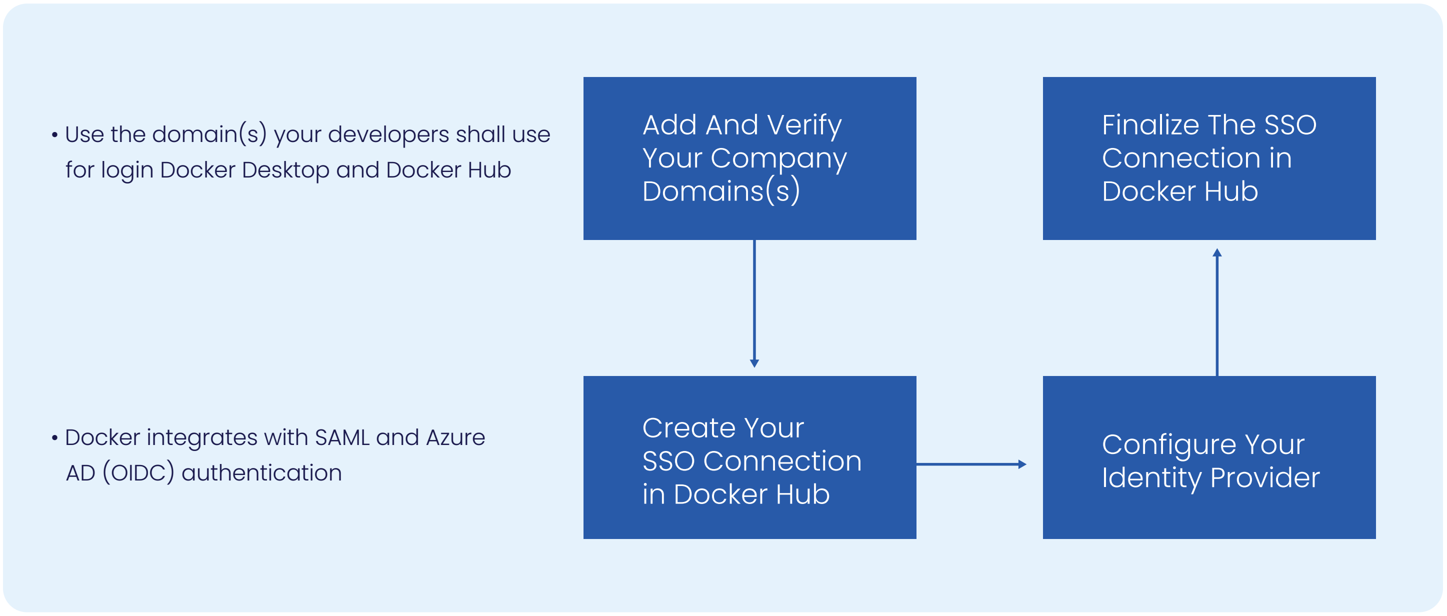
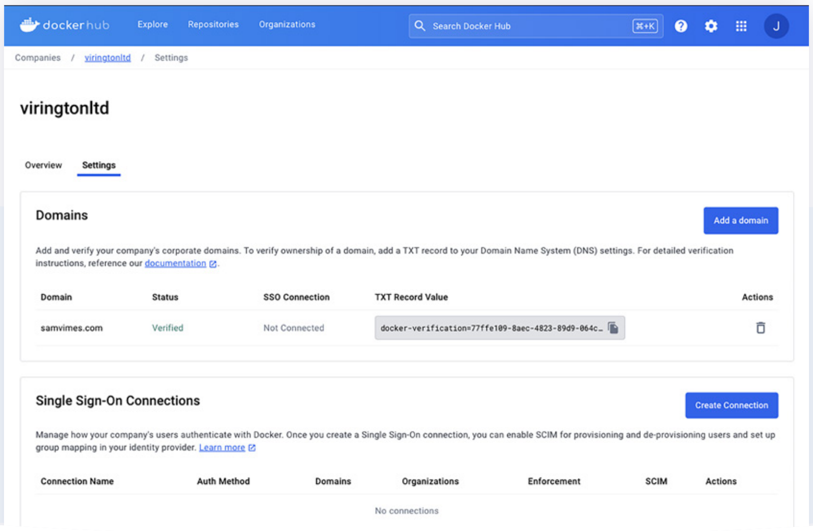

• Use the domain(s) your developers shall use for login Docker Desktop and Docker Hub
• Docker integrates with SAML and Azure AD (OIDC) authentication
Add And Verify Your Company Domains(s)
Create Your SSO Connection in Docker Hub
Configure Your Identity Provider
Finalize The SSO Connection in Docker Hub

Figure 6: Flow chart about the main configuration steps to setup a proper SSO integration with company identity provider(s).

To get started, head to your Docker Hub organization's *Settings* page and navigate to the *Security* tab. Here, you can kick off the SSO setup process by clicking the `Add a domain` button. Next, you'll need to specify a non-public domain without protocol or www information. Docker Hub will then generate a TXT Record Value, which must be added to the TXT record associated with the previously added domain in your company's DNS. Allow 72 hours for DNS changes to take effect and for your domain to be verified.



viringtonltd

Domains

Add and verify your company's corporate domains. To verify ownership of a domain, add a TXT record to your Domain Name System (DNS) settings. For detailed verification instructions, reference our documentation.

Domain	Status	SSO Connection	TXT Record Value	Actions
samvimes.com	Verified	Not Connected	docker-verification=77ffe109-8aec-4823-89d9-064c…	

Single Sign-On Connections

Manage how your company's users authenticate with Docker. Once you create a Single Sign-On connection, you can enable SCIM for provisioning and de-provisioning users and set up group mapping in your identity provider. Learn more

Connection Name	Auth Method	Domains	Organizations	Enforcement	SCIM	Actions
No connections						

Figure 7: Overview about existing domains and there connection status in the Security tab of a Docker Hub business organization

Following this, you can initiate the setup of a SSO connection within your Docker Hub organization by returning to the *Security* tab in the *Settings* menu. As you set up this connection, you will notice that Docker supports SAML 2.0 and Azure AD (OIDC) authentication and will provide you with all the necessary information required for creating the SSO connection within your company IdP, as outlined in **Table 2**.

SAML 2.0	Azure AD (OIDC)
Entity ID: Identifies Docker Hubs SSO system and facilitates secure communication between Docker Hub and company IdP.	**Redirect URI:** Endpoint where users are redirected after they successfully authenticate with Azure AD
ACL URL: Manages access permissions and secure communication between Docker Hub and company's IdP	

Table 2: Overview about the necessitated properties to create a SSO connection for Docker Hub on a company IdP server.

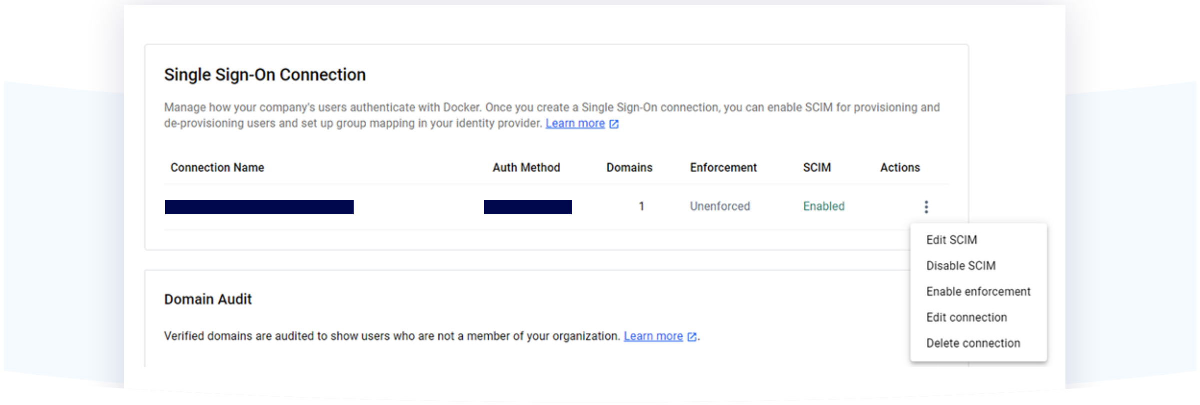
Be aware that Docker retrieves the user's email address and name from the Identity Provider (IdP) when a user attempts to log in. The email address serves as the unique identifier for the user. Since the setup of SSO within your IdP might differ from company to company, it cannot be detailly explained here and must be retrieved from the official documentation of Docker.

You are ready to complete the SSO connection setup within Docker Hub. When users attempt to log in to Docker Hub or Docker Desktop, they'll need to use their company email address. Remember to enable Enforced Sign-In to ensure that colleagues use their company IDs when using Docker Desktop locally.

Docker's SSO implementation uses Just-in-Time (JIT) provisioning by default. This means any users signing in to Docker through their company's IdP will get assigned a license by being added to their Docker Hub organization.

To control this, company admins can enable the System for Cross-domain Identity Management (SCIM) 2.0 for their business. SCIM provides automated user provisioning and de-provisioning for your Docker Hub organization through your IdP. Once you enable SCIM in Docker and your IdP, any user assigned to the Docker application in the IdP is automatically provisioned in Docker and added to the organization or company. Similarly, if a user gets unassigned from the Docker application in the IdP, this removes the user from the organization or company in Docker.

SCIM also synchronizes changes made to a user's attributes in the IdP, for example the user’s first name and last name. You can enable SCIM from your Docker Hub organization or Admin Console by navigating to the SSO settings page and selecting *'Setup SCIM'*. Here, you will then be able to obtain the SCIM Base URL and API Token required by your IdP to enable SCIM.



Single Sign-On Connection

Manage how your company's users authenticate with Docker. Once you create a Single Sign-On connection, you can enable SCIM for provisioning and de-provisioning users and set up group mapping in your identity provider. Learn more

Connection Name	Auth Method	Domains	Enforcement	SCIM	Actions
[redacted]	[redacted]	1	Unenforced	Enabled	⋮

Domain Audit

Verified domains are audited to show users who are not a member of your organization. Learn more.

Figure 8: To activate SCIM, you need to access your Docker Hub organization's *Security* tab in the *Settings* menu. From there, simply right-click on the relevant SSO connection and select the *"Enable SCIM"* option.

Controlling image and registry access

The *Log4Shell* vulnerability, part of the *Log4J* Library within the version range of 2.0-2.14 and labeled with *CVE-2021-4428*, caused significant issues for numerous enterprises since they lacked an overview of where this high-risk vulnerability was shipped together with their products. To mitigate the risk of such *Supply Chain Vulnerabilities, a Software Composition Analysis* (SCA) process is recommended throughout the entire life-cycle of an application. Regarding containers, the SCA has to encompass the whole ecosystem, starting with centralized quality gates for the build and upload process and regular monitoring and scanning of container registries.

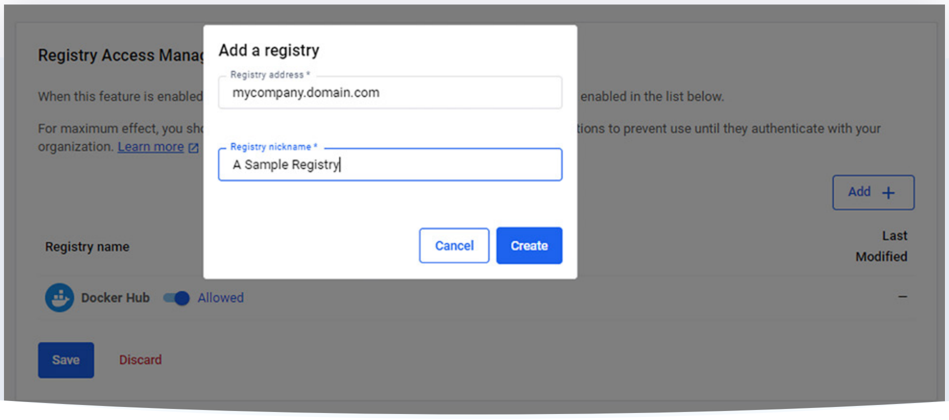
While the risks introduced by such kinds of vulnerabilities are significantly low for developer workstations that typically operate within internal networks, systematic control of available images should also be integrated into the SCA process. Additionally, large-scale enterprises incorporate well-defined processes before introducing any new development tool on local workstations, underscoring the significance of centralized control over the local deployment of containers.

Docker Desktop offers two integration possibilities to gain centralized control of OCI image access for a fleet of enterprise workstations. The first feature, called Image Access Management (IAM), can be activated within the Settings space of a *Docker Business Organization* and is automatically applied to the local workstations by activating *Enforced Sign-In*. IAM is suitable for companies that use Docker Hub as their primary container registry platform. IAM operates on the image level and empowers enterprises to finely configure global permissions, allowing developers to pull images based on specific OCI image types. **Table 2** lists the available image types on Docker Hub and their corresponding level of reliability. It is strongly recommended that restrictions be enforced on accessing *Community* Images, as they pose high risks for *Malicious Container* or *Supply Chain* attacks.

Settings	Description	Reliable?
Docker Official	Respective type for those OCI images that Docker Inc. publishes.	Yes
Verified Publisher	Respective type for those OCI Images that commercial vendors verified by Docker Inc. publish.	Yes
Organization	Respective type for those OCI Images created by members of the Docker Hub organization and published to private Docker Hub repositories.	Yes*
Community	Respective type for those OCI Images published by unknown and not officially verified image vendors.	No

Table 3: Overview of OCI image types that are hosted on Docker Hub. The kind of OCI image gives a precise conclusion about the reliability of the image source and can be facilitated by Image Access Management (IAM). *) Huge-scale enterprises must incorporate security measures and a centralized way of building and uploading OCI images to ensure the reliability of their internally developed OCI images.

In the case of a company using another platform, such as *JFrog Artifactory*, to primarily host their OCI Images, Docker Desktop provides so-called *Registry Access Management* (RAM) to have a central control about which specific container registry can be used to pull an image to the local workstation. RAM can also be activated and configured within the *Settings* space of a *Docker Business Organization* and, luckily, works with regular expressions, as shown in **Figure 9.**


Registry Access Manag
Add a registry
When this feature is enabled
enabled in the list below.
For maximum effect, you sh
tions to prevent use until they authenticate with your
organization. Learn more
Add +
Registry name
Last Modified
Docker Hub
Allowed
Save
Discard
Cancel
Create

Registry address *
mycompany.domain.com

Registry nickname *
A Sample Registry

Figure 9: Facilitating Registry Access Management (RAM) for a particular container registry within the Settings space of a Docker Business Organization. The input value for the Registry address integrates with regular expressions.

In line with full-featured RAM, it is highly recommended that enterprises update Docker Desktop to a version equal to or beyond 4.21, as earlier versions lack RAM support for Windows-based container registries. To apply RAM on Windows-based containers, the Windows container daemon has to use Docker Desktop's internal proxy, as shown in **Figure 14**. Alternatively, companies can exclude Windows containers from their local workstations.

Handling huge-scale deployments on regulated workstations

Crafting a smooth installation procedure

Although installing Docker Desktop is straightforward by executing the released '`Docker Desktop Installer.exe`', large enterprises encounter a significant challenge. Typically, those enterprises apply tight regulations to their local workstations, which include the absence of local `SYSTEM` rights for their developers. However, those elevated permissions are essential for executing the installer and activating Docker Desktop's system prerequisites. These prerequisites encompass Optional Windows NT Kernel features, detailed in **Table 3**, and the *Store Version* of WSL, recommended since recent features of WSL, such as GPU or GUI support and security updates, are urgently accessible that way.

Optional NT Feature	Description	Required For
Containers	Containers feature enables the OCI-compliant Host Compute Service (HCS), a Windows API used to operate Windows containers.	Windows Container Engine
Hyper-V	Hyper-V is a Windows feature that provides hardware virtualization (isolation), allowing multiple operating systems to run on a single host machine.	Windows Container Engine
Virtual Machine Platform	Windows Machine Platform enables the creation and management of virtual machines with the aid of Hyper-V.	Linux Container Engine
Windows Subsystem For Linux (WSL)	Inbox version of WSL that provides a Linux-compatible kernel interface that supports running Linux applications on Windows.	Optional*
Windows Hypervisor Platform	Windows Hypervisor Platform provides a Windows API to level 2 virtualizers such as VirtualBox to allow them to spawn VMs on the host despite enabled Hyper-V.	Optional

Table 4: Overview of the Optional Windows NT Kernel features that contribute to the containerization of Linux and Windows Containers on the local developer workstations. *`The Windows-Subsystem-For-Linux` optional NT Kernel feature is no longer required when operating on Windows 11, since the so-called *Store Version* is employed.

It is typically advised to employ common package managers like *Microsoft WinGet* or *Chocolatey* for deploying applications like the WSL Store Version or Docker Desktop and to leverage centralized administration tools such as *Microsoft Intune* or *System Center Configuration Manager* (SCCM) for configuring these tools on a large scale. If companies do not employ such tools, it becomes vital to have a fundamental understanding of low-level terminal commands to configure the environment effectively. To begin with, the above-listed Optional Windows NT Kernel features can be enabled by running the command shown in **Figure 10** within an elevated PowerShell.

```
# Please replace <FEATURE> with following features:
# 1) Microsoft-Windows-Subsystem-Linux, 2) VirtualMachinePlatform
# 3) Microsoft-Hyper-V-All, 4) Containers, 5) HypervisorPlatform
PS C:\Windows\System32> dism.exe /online /enable-feature /featurename:<FEATURE>
/all /norestart
```

Figure 10: Activating required Optional Windows NT Kernel features on a local workstation within an elevated PowerShell terminal

To finally apply these changes, you need to restart the system. Following this, you have the option to install the WSL Store Version. This version can be installed through the Microsoft Store or the command line. The latter option is the recommended approach for large-scale tool deployments. Microsoft offers the `wsl --update or wsl --install` command to facilitate installation. However, it's important to note that both commands have notable drawbacks, as they do not support installing a fixed WSL version and only update to its latest release. This may lead to inconsistencies across local workstations and result in additional support efforts for the internal IT department. Additionally, by default, the `--install` command deploys a standard Ubuntu distribution on the local WSL platform, lacking essential security measures, especially when deployed on Windows 10, as described in *Securing Windows Subsystem For Linux 2* (WSL2). Consequently, enterprises can download the WSL installer for a specific release from Microsoft's official GitHub repository and execute the following command:

```
# Install WSL Store Version from it's sources
PS C:\Windows\System32> Add-AppxProvisionedPackage -Online -PackagePath "\path\to\
installer\Microsoft.WSL_1.2.5.0_x64_ARM64.
```

Figure 11: Install the WSL Store Version by executing the MSI bundle of the sample release 1.2.5.0 within an elevated PowerShell

It is recommended to use the `Add-AppxProvisionedPackage` command, as it installs a Windows application for all users of the respective Windows image. In contrast, the `Add-AppxPackage` command only installs the application for a particular (current) user, which is only suitable for large-scale deployments where users have elevated permissions.

Finally, Docker Desktop can be installed by executing `'Docker Desktop Installer.exe'` within an elevated PowerShell, as shown in **Figure 12**. Docker Desktop version 4.27 is strongly recommended as it includes crucial features for complex enterprise environments. Notable enhancements include support for RAM on Windows containers, compatibility with the `--always-run` installation command, and resolution of persistent configuration settings issues disallowing `docker login` or `docker pull` inside a WSL2 distribution in case `interop` feature is disabled as described in *Securing Windows Subsystem For Linux 2* (WSL2).

```
# Install Docker Desktop from command-line
PS C:\Windows\System32> Start-Process -FilePath 'Docker Desktop Installer.exe'
-ArgumentList 'install', '--quiet', '--accept-license', '--allowed-org=myorgname',
'--always-run-service' -Wait
```

Figure 12: Install execution for Docker Desktop on Windows PowerShell.

Besides the `install` argument, huge-scale deployments should provide the following options:

- `--quiet`
  Suppress any output during installation.
- `--accept-license:`
  Accepts Docker Desktop business agreement and, thus, prevents any installation prompt
- `--allowed-org=myorgname:`
  Deploys `%ProgramData%/DockerDesktop/registry.json` on the local workstation, which applies *Enforced Sign-in* and, in consequence, the security measures of the Docker Hub Organization `--always-run-service`:
  Allows low-privileged users to switch between Windows and Linux container engine without UAC prompt as the `SYSTEM` level service `com.docker.service.exe` always runs in the background
- `--no-windows-containers:`
  Organizations that do not require Windows-based containers are strongly advised to provide this install option since Windows containers significantly increase the attack surface of a system. That way, the Windows container integration is disabled.

Regarding large-scale deployments, it is crucial to note that an admin account, distinct from the Windows user of the local workstation, typically takes care of the installation process. In this scenario, the command shown in **Figure 13** must be executed to grant permissions to the local user for utilizing the `SYSTEM` level workload of Docker Desktop, as elaborated in `Exploring complexities of Windows container engine architecture.`

```
# Apply elevated permissions for user through membership of group
'docker-users'
PS C:\Windows\System32> net localgroup docker-users <user> /add
```

Figure 13: Add a local Windows user to the docker-users group to effectively authorize this user to work with the `SYSTEM` level workload of Docker Desktop

Apply globally controlled settings management

Docker Desktop version 4.13 or later introduces an essential enterprise-grade security control, namely improved *settings management*. This capability provides a centralized approach for applying standardized and consistent configuration options to multiple workstations running Docker Desktop. This feature is applied with *Enforced Sign-In* and deploying `%ProgramData%/DockerDesktop/admin-settings.json` on the local workstation, which remains unalterable without local admin rights. This feature becomes highly valuable in large-scale deployments when aiming to limit elevated employee permissions.

The recommendations outlined in **Figure 14** propose the recommended settings to activate and restrict the local workstation. *Enhanced Container Isolation* (ECI) facilitates the seamless integration of the OCI runtime `sysbox-runc`, known for its heightened security control, enabling a more or less secure execution of system-level workloads such as *Docker-in-Docker* or *Kubernetes-in-Docker* within a container. It is important to note that Docker Desktop 4.20 or higher is required, as earlier versions do not support ECI within WSL2. While activating this feature is advisable, it is recommended not to restrict it for developers, as certain use cases may not be supported by `sysbox-runc`.

Furthermore, it is crucial to disable and restrict the exposure of the REST interface of the Docker Engine on TCP port 2375 without proper TLS protection, as this poses significant risks to the local system. While this setup is critical in a production environment, it may not result in substantial issues when employed in the inner development loop, where host protection measures such as firewalls offer additional security for the machine. However, there are limited use cases where this feature is truly beneficial.

Enterprises with finely defined approval processes for installing development tools can employ the option to deactivate the transmission of analytics to Docker Inc. and prevent the installation of any Docker Desktop Extensions. *Unveiling Windows Containers best practices* will discuss essential settings related to the Domain Name Server (DNS) setup of the Windows container engine.

```
{
  "configurationFileVersion": 2,
  "exposeDockerAPIOnTCP2375": {
    "locked": true,
    "value": false
  },
  "proxy": {
    "locked": true,
    "mode": "system",
    "http": "",
    "https": "",
    "exclude": [],
    "windowsDockerdPort": 65000
  },
  "enhancedContainerIsolation": {
    "locked": false,
    "value": true
  },
  "analyticsEnabled": {
    "locked": true,
    "value": false
  },
  "extensionsEnabled": {
    "locked": true,
    "value": false
  },
  "windowsContainers": {
     "dockerDaemonOptions": {
       "locked": false,
       "value": "{\"debug\": false, \"default-network-opts\": { \"nat\": { \"com.docker.network.
windowsshim.disable_gatewaydns\": \"true\" } } }"
     }
  }
}
```

Figure 14: Proposed content for `admin-settings.json` that can be installed on local workstations to deploy consistent settings for large-scale deployments.

Securing Windows Subsystem for Linux 2 (WSL2)

Exploring the WSL2 environment

As outlined in *Exploring Complexities Of Linux Container Engine Architecture,* WSL2 is a Windows feature that deploys a Linux environment inside a highly optimized Hyper-V virtual machine, resulting in running Linux applications 'natively' on Windows. Docker Desktop, in turn, installs two special-purpose WSL2 distributions to the local WSL environment that are utilized to facilitate the Linux container backend.

Nonetheless, users have the option to deploy fully-featured WSL2 distributions like *Ubuntu, Debian, openSUSE,* etc. to their local environment and seamlessly integrate Docker Desktop into them. Microsoft Corp. has recently introduced several enterprise-grade security controls to enhance the security of WSL in enterprise environments. These include a new networking stack, support for *Microsoft Intune*, a WSL plug-in framework with current support for *Microsoft Defender for Endpoint* (MDE), and automatic inheritance of host firewall rules. Nevertheless, the inclusion of fully-featured WSL2 distributions significantly expands the company's attack surface, especially since many of these features are exclusive to Windows 11. While applying those features is more than recommended, organizations that rely on alternative tools like *Trellix* for anti-malware support or *System Center Configuration Manager* (SCCM) for tool administration may not benefit from these enhancements. This underscores the need for additional hardening measures to ensure compliance and security.

WSL2 distributions are built with common container technologies and adhering to the *Open Container Initiative* (OCI) standard and, thus, offer a lightweight solution for a local virtual Linux environment. By default, the WSL2 environment employs a NAT-based network architecture, that employs a virtual private network named 'wsl' on the Windows host. This network provides network access to each WSL2 distribution through a virtual Ethernet adapter with a DHCP-NAT address. Since WSL2 distributions share a common network namespace, they all receive the same IP address within this network. The host also gains an IP address in this network, serving as the network gateway and DNS server.

Fortunately, the 'wsl' network is only accessible by the local Windows host, offering a significant security benefit. To expose WSL2 distributions to remote hosts, local admin rights are required to establish a port proxy linking a port on the local WSL2 guest to a port on the Windows host. Additionally, the Windows Firewall needs to be relaxed for this specific port to allow access from other network clients. In enterprise environments where Group Policies are typically applied, domain admin rights are required for this purpose.

In case the WSL2 environment is used within a company Virtual Private Network (VPN), WSL2 distributions may encounter significant networking connectivity issues, even when deployed with the mirrored networking mode. This necessitates to utilize the so-called 'wsl-vpnkit' that generates an additional network interface for WSL2 guests named 'wsltap', reconfigures their default routing behavior and launches the wsl-gvproxy.exe process on the host side to receive and forward networking traffic. While this introduces an additional third-party dependency, that demands additional security assessments, it establishes a reliable network setup that ensures networking connectivity from office and home.

Additional security considerations for WSL2 guests

For Windows 11, Microsoft introduced a local Hyper-V firewall for WSL2, which is activated by default but can be deactivated within the `.wslconfig` located in the Windows user home directory. On Windows 10, this feature is not available and, additionally, the Windows host firewall rules are not honored by local WSL2 guests. Furthermore, the Linux file system lacks in terms of anti-malware support, by default, and standard anti-malware solutions safeguarding the Windows host only extend their protection to the mounted C drive / / / / / / within WSL but not to the Linux file system, as demonstrated by running the standard EICAR test shown in **Figure 15**.

```
# Executing the standard EICAR test on Linux user home
~$ echo 'X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*' > eicar-test.txt
```

Figure 15: Running the standard EICAR test within the (WSL) Linux user home, that will not be recognized by host anti-malware protection.

While WSL2 seamlessly grants elevated `root` capabilities within its environment, it is fortunate that these permissions do not extend to `SYSTEM` rights on the Windows host. However, if both registry and image access management are absent by the Docker Desktop setup, the lack of firewall and anti-malware protection could introduce a potential *Malicious Container Attack and a Local Privileged Escalation*, as illustrated in **Figure 16**.

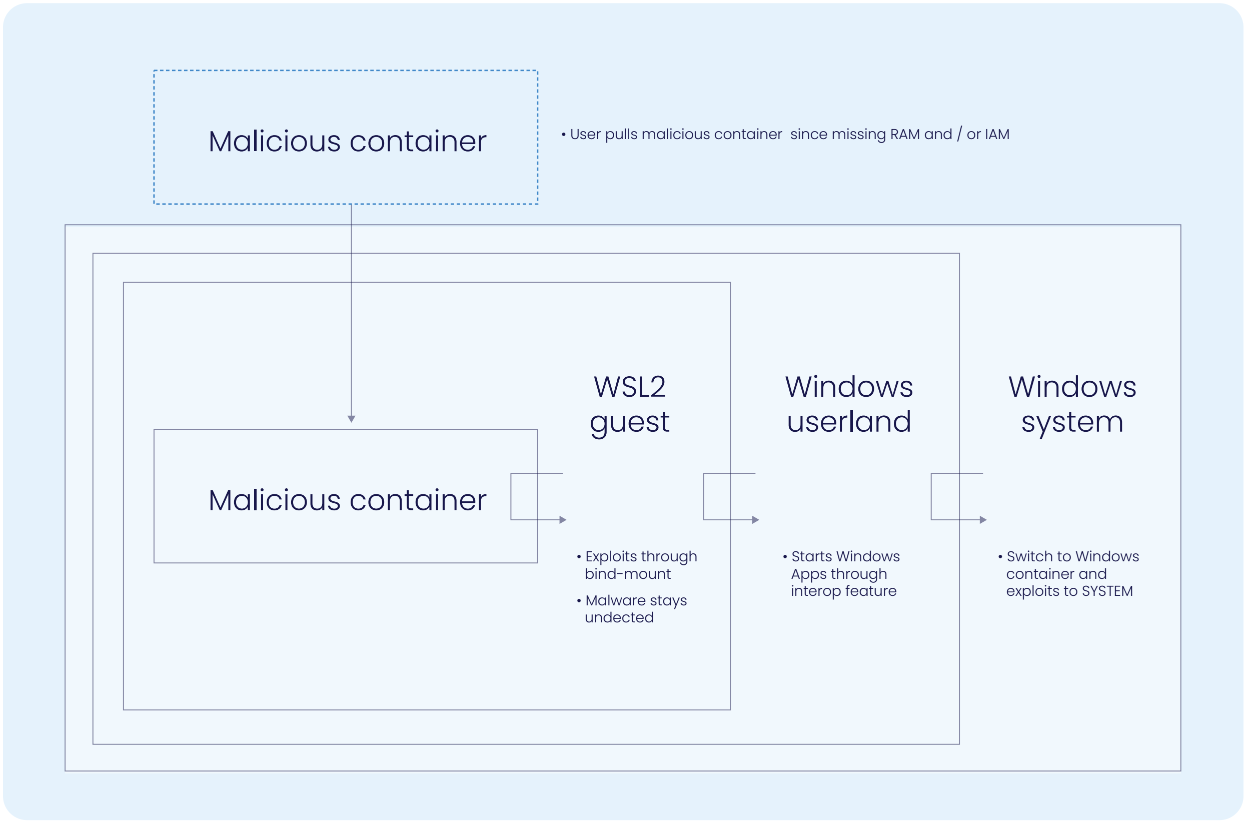
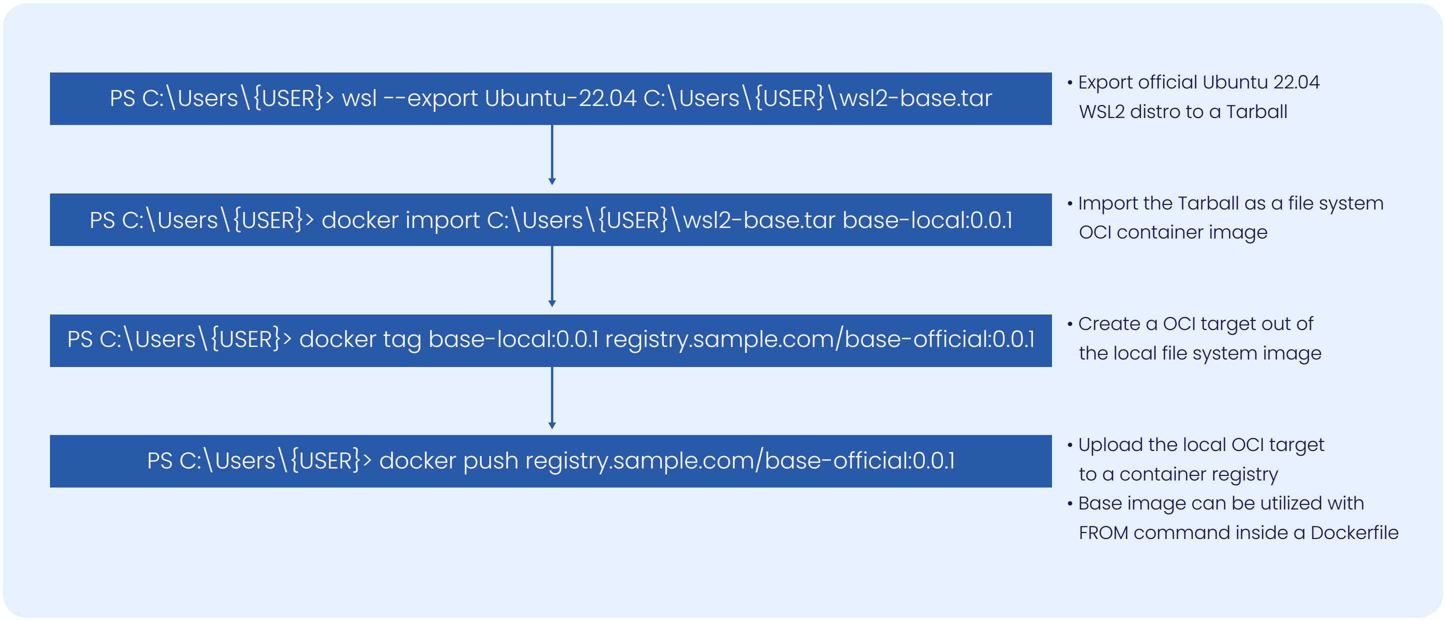

Malicious container
• User pulls malicious container since missing RAM and / or IAM
WSL2 guest
Windows userland
Windows system
Malicious container
• Exploits through bind-mount
• Malware stays undected
• Starts Windows Apps through interop feature
• Switch to Windows container and exploits to SYSTEM

Figure 16: Possible threat path through a **malicious** container download into a fully-featured WSL2 distribution and the Windows container engine, resulting in a Windows `SYSTEM` exploit.

As a result, companies may want to implement additional hardening measures for their WSL2 instances. Because WSL2 distributions adhere to OCI standards, widely-used container management tools like *Docker (Desktop)* or *Podman (Desktop)* can be employed to deploy customized WSL2 distributions across multiple systems. The `wsl.exe` command-line interface offers capabilities for exporting and importing file system containers, which can then serve as base images in any *Dockerfile* or *Containerfile*. **Figure 17** outlines a potential workflow for generating and uploading an OCI-compliant image using *Microsoft's* official Ubuntu 22.04 WSL2 distribution.



```
PS C:\Users\{USER}> wsl --export Ubuntu-22.04 C:\Users\{USER}\wsl2-base.tar
PS C:\Users\{USER}> docker import C:\Users\{USER}\wsl2-base.tar base-local:0.0.1
PS C:\Users\{USER}> docker tag base-local:0.0.1 registry.sample.com/base-official:0.0.1
PS C:\Users\{USER}> docker push registry.sample.com/base-official:0.0.1
```

• Export official Ubuntu 22.04 WSL2 distro to a Tarball
• Import the Tarball as a file system OCI container image
• Create a OCI target out of the local file system image
• Upload the local OCI target to a container registry
• Base image can be utilized with FROM command inside a Dockerfile

Figure 17: Sample workaround to export Microsoft's official Ubuntu 22.04 WSL2 distribution into a Tarball, importing this Tarball as a file system OCI image into Docker Desktop and finally uploading this OCI image to any corporate container registry.

From here, it's up to each company to consider and apply appropriate security measures for the respective WSL2 setup. It is important to note that, currently, root access to WSL cannot be prevented, allowing users to potentially relax local security settings on their own. This demands to additionally establish terms of use for a safe but productive usage of WSL. However, employing the following tools can enhance the overall enterprise-grade of WSL:

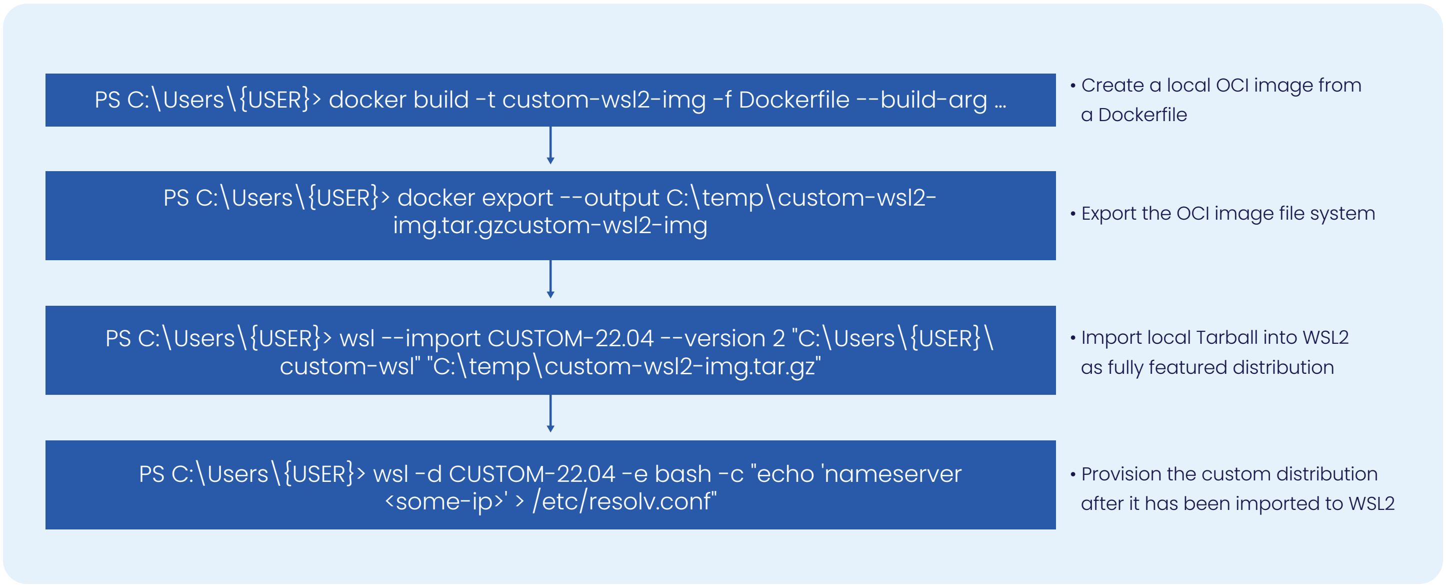
- `unattended-upgrades:`
  Automated installation of security updates and other selected packages without user intervention
- `ufw (uncomplicated firewall):`
  Easy to use interface for managing firewall rules on Linux systems. Alternatively, WSL administrators can utilize the Hyper-V firewall of the Windows host by enabling the `firewall=true` setting `inside .wslconfig` when operating on Windows 11.
- Apply private sources and use proprietary SSL certificates:
  Relying solely on private sources such as mirrored `apt` sources, `npm registries`, `PyPI` registries, etc.
- Disabling interoperability:
  To enhance the isolation of the WSL environment, disable the execution of any Windows process from within WSL. This can be achieved by setting `interop=false` inside `/etc/wsl.conf`. However, it is important to note that this will lead to significant drawbacks and will weaken the development experience.
- Add company DNS servers:
  To resolve private company domains, it's crucial to configure enterprise DNS servers in `/etc/resolv.conf`. However, it's important to note that this configuration cannot be applied during image build but must be done after deployment using the `wsl.exe --exec <COMMAND>` interface.
- Apply systemd as init process:
  Enhancing the usability of WSL, enabling systemd as the init daemon is essential as many tools rely on it, thereby improving the overall development experience. This can be achieved by activating the systemd=true setting in `/etc/wsl.conf`. It is important to note that a WSL distribution must initially be deployed without the systemd specification, requiring the use of the `wsl.exe --exec <COMMAND>` pattern during the post-deployment routine.
- Apply a custom user setup:
  Ensure that the local user needs to create an own Linux account when the first login to the WSL2 distribution.
- Apply a HTTP/HTTPS proxy:
  When using Windows 11, enabling `autoProxy=true` and `networkingMode=mirrored` settings will extend local proxy configurations from the Windows host to local WSL2 guests. While this setup allows tools like `curl` and `wget` to access proxy information from the environment, additional configuration is required for tools such as `apt` and `git`.

It is noteworthy that deploying an anti-malware solution that meets the demands of an enterprise environment poses significant challenges since internet access and user credentials are required for updating antivirus signatures, making centralized deployment a complex task. Moreover, it is essential to highlight that *Docker Desktop* version 4.27 or later is necessary when `interop` functionality is turned off as *Docker Desktop* persists the content of `%USERPROFILE%/config.json` to facilitate the use of the built-in credential helper. With `interop=false`, these helper processes become unattainable from WSL2. However, executing `docker login` or `docker pull` from any Windows shell interacting with those images from WSL2 works out of the box.

```
# See image above for how this base image was initially created
FROM registry.sample.com/base-official:0.0.1

# install tools such as ufw and unattended-upgrades
RUN apt update && apt upgrade -y && \
    DEBIAN_FRONTEND=noninteractive DEBCONF_NONINTERACTIVE_SEEN=true \
    apt install --no-install-recommends -f -y ufw unattended-upgrades … && \
    apt-get clean && rm -rf /var/lib/apt/lists/*

# apply APT mirrored sources; do not forget to sign them
RUN mv /local/path/to/apt-mirror.list /etc/apt/sources.list

# create an APT update timer for unattended-upgrades (do the same for apt upgrade)
RUN TIMER_PATH="/etc/systemd/system/%s.timer.d/override.conf" && \
    APT_UPDATE=$(printf $TIMER_PATH "apt-daily-update") && \
    mkdir -p $(dirname $APT_UPDATE) && \
    echo "[Unit]" > $TIMER_PATH && \
    echo "Description=Daily apt download activities" >> $TIMER_PATH && \
    echo "" >> $TIMER_PATH && \
    echo "[Timer]" >> $TIMER_PATH && \
    echo "OnCalendar=*-*-* 6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21:05" >> $TIMER_PATH && \
    echo "RandomizedDelaySec=23m" >> $TIMER_PATH && \
    echo "Persistent=true" >> $TIMER_PATH && …

# apply custom firewall rules
RUN ufw default deny incoming && \
    ufw default deny outgoing && \
    ufw allow out from any to {APT_MIRROR} comment Outbound to APT mirror server' && \    … \

# disable WSL2 to start host processes and prevent creation of resolv.conf to
# incorporate company’s DNS server(s)
RUN echo "[interop]\nenabled = false" > /etc/wsl.conf && \
    echo "[network]\ngenerateResolvConf = false" >> /etc/wsl.conf
```

Figure 18: Excerpt of a Dockerfile for applying hardening measures on a WSL2 guest. The base image can be retrieved from the standard Ubuntu WSL2 image, using the `wsl.exe --export` command for creating a TAR file and the docker import command to create a filesystem

Finally, a customized OCI image can easily be deployed as a fully-featured WSL2 distribution at scale, utilizing the same tools from above in reverse. **Figure 19** below shows a sample approach that converts an OCI image with *Docker Desktop* into a tarball and imports this tarball into the WSL2 environment. The distribution can, in turn, be further provisioned using the `wsl.exe --exec <COMMAND>` command-line interface.



```
PS C:\Users\{USER}> docker build -t custom-wsl2-img -f Dockerfile --build-arg ...
PS C:\Users\{USER}> docker export --output C:\temp\custom-wsl2-img.tar.gzcustom-wsl2-img
PS C:\Users\{USER}> wsl --import CUSTOM-22.04 --version 2 "C:\Users\{USER}\custom-wsl" "C:\temp\custom-wsl2-img.tar.gz"
PS C:\Users\{USER}> wsl -d CUSTOM-22.04 -e bash -c "echo 'nameserver <some-ip>' > /etc/resolv.conf"
```

• Create a local OCI image from a Dockerfile
• Export the OCI image file system
• Import local Tarball into WSL2 as fully featured distribution
• Provision the custom distribution after it has been imported to WSL2

Figure 19: Sample pattern to export an OCI image into a Tarball with the aid of Docker Desktop, which can be imported into WSL2 as a fully-featured distribution.

It is worth noting that companies should prioritize implementing the recommended administration and security features offered by *Microsoft Corp*, such as the new networking stack, *MDE* plugin support, and *Microsoft Intune* administration. However, if any of these tools are absent or if workstations encounter major issues like network connectivity problems, the introduced hardening measures serve as a viable fallback solution.

Unveiling Windows Containers best practices

Consider notable security aspects

As outlined in *Exploring complexities of Windows container engine architecture*, the Windows container engine `dockerd.exe` runs with elevated privileges on the host machine, which can be verified as shown in **Figure 20** below.

```
# List all services associated with Docker Desktop

PS C:\WINDOWS\system32>Get-Process-IncludeUserName | Where-Object { $_.ProcessName-
like'*docker*' }

Handles WS(K)    CPU(s) Id         UserName                 ProcessName

592              64136       796         NT AUTHORITY\SYSTEM      com.docker.service

...

116              26416       3182  <     DOMAIN>\<USER>           docker

204              49392       3868        <DOMAIN>\<USER>          Docker Desktop

351              45236       12144       NT AUTHORITY\SYSTEM      docker
```

Figure 20: Overview of Docker Desktop's processes that run with SYSTEM rights on the Windows host machine.

The `NT AUTHORITY\SYSTEM` represents a Windows security principle that grants its users the highest privileges within the Windows operating system. Processes initiated under this principle have almost unrestricted access to system resources. Thus, any Windows container launched with dockerd.exe inherits these privileges, enabling it to seamlessly interact with the underlying NT kernel, which, in turn, includes tasks such as altering system files or interfacing with system services.

Windows generally offers two container types with a distinct isolation level from the underlying host system. Firstly, there are so-called "process-isolated" containers, which are employed directly on the host machine and, thus, effectively operate as operating system features. If compromised by an attacker, these containers significantly increase the attack surface. Secondly, there are so-called "hyperv-isolated" Windows containers, which utilize a container within a stripped-down and isolated *Microsoft Hyper-V* VM named "DockerDesktopVM". Those containers run alongside the Windows NT kernel on the Hyper-V layer (see **Figure 3**), creating a robust security boundary since a hardware exploit is required to finally compromise the host. However, `dockerd.exe` grants `SYSTEM` permissions to any bind-mounted directory, potentially offering extensive system-level access. Consequently, this could present a significant vulnerability, particularly in production environments, in case sensitive directories such as `C:\Windows` are bind-mounted, as shown in **Figure 21**.

```
# Bind mount of Windows system directory into a Hyper-V container
PS C:\Users\{USER}> docker run -it --rm -v C:\Windows:C:\mnt windows:20H2 powershell
```

Figure 21 :Possible entry point for a Local Privilege Escalation (LPE) through a bind-mount on C:\Windows directory.

In addition, enterprises that deploy Docker Desktop with its Windows container engine at scale should consider that their developers could misuse Docker Desktop to gain unauthorized `SYSTEM` privileges of the local system. This could introduce the risks for *misconfiguration* and *local admin rights*. For instance, developers could expose the REST interface of the Windows container engine via TCP port 2375 without proper Transport Layer Security (TLS) protection. They could also relax the local Windows firewall to reach the engine outside the company network (see **Figure 22**). This could be a potential entry point for an attacker to compromise the local machine and the company's network. Therefore, it is highly advisable to implement a code of conduct to ensure responsible and compliant usage or to completely prevent the usage of Windows containers by using the `--no-windows-container` installer flag as outlined in *Crafting a smooth installation procedure*.

```
# Define content for exposing the REST interface
PS C:\Users\{USER}> $jsonContent = @{"hosts" =@("tcp://0.0.0.0:2375")

}

# Convert the PS object to JSON format
PS C:\Users\{USER}> $jsonString = $jsonContent | ConvertTo-Json

# Apply changes to Windows container engine
PS C:\Users\{USER}> $jsonString | Set-Content -Path
"C:\ProgramData\docker\config\daemom.json"

# Define request body for running a Windows container with bind-mount on "C: \Windows"

PS C:\Users\{USER}> $restBody = @{
        Image = "my-container-image"
HostConfig = @{
    Binds = @("C:\Windows:C:\mnt")
        }
        Privileged = $true
} | ConvertTo-Json

# Create the container via REST request
PS C:\Users\{USER}> Invoke-WebRequest -Uri "<YOUR-IP>/containers/
                                                    create?name=mycontainer " `
                                                  -Method Post `
                                                  -Body $restBody `
                                                  -ContentType 'application/json'

# Run the container via REST request
PS C:\Users\{USER}> Invoke-WebRequest -Uri "$apiUrl/containers/mycontainer /start"
                                                  -Method
```

Figure 22: Sample pattern to expose the REST interface of the Windows container engine, dockerd.exe, via TCP and running a Windows container with bind-mount on the sensitive `C:\Windows` drive via a REST request.

In the security context, working solely with "hyperv-isolated" containers is highly recommended due to their strong isolation from the host system. Nevertheless, the utilization of these container types includes several drawbacks, including the absence of GPU acceleration support, missing hardware device integration, and the potential for malware to exploit a container vulnerability and remain undetected within the container, as those containers are "black-boxed" for antivirus solutions safeguarding the host, too. Thus, enterprises are strongly recommended to implement global control mechanisms that ensure only Windows containers built with best practices can be deployed for both inner development tasks and used for productional targets. Best practices include allowing solely unprivileged users, like the built-in `ContainerUser`, to run a container or enabling natural Windows host protection mechanisms such as antivirus solutions within a Windows container. Additionally, it is noteworthy that the business customer features *Registry Access Management (RAM)* and *Image Access Management (IAM)*, outlined in *Choosing The Right Pricing Model*, shine since they provide global control over images' access.

Unveiling network connectivity issues

Windows containers operating within a complex enterprise environment with strict firewall rules, corporate Domain Name Servers (DNS), a Virtual Private Network (VPN), etc. often encounter notable networking issues. These issues prevent smooth development workflows in real-world scenarios, such as accessing license servers, retrieving artifacts from remote sources, or providing their services to remote agents. We first need to look at the standard network setup when launching Windows containers on a Windows 10 (22H2) host machine to unveil the main networking challenges.

When launching Docker Desktop's Windows container engine for the first time, Docker Desktop will create a default network with the name "nat". This network is configured as a NAT-mode network and integrates an internal *Hyper-V Virtual Switch*. Any freshly instantiated container will automatically connect to this network unless another network is designated using the `--network` flag, as shown in **Figure 25** below.

```
# Run a standard Windows 20H2 container

PS C:\Users\{MY_USER}> docker run --name foo mcr.microsoft.com/windows:20H2 powershell.exe

# Inspect the default "nat" network
PS C:\Users\FIXCYBJ> docker network inspect nat
[
     {
          "Name": "nat",
          ...
          "IPAM": {
    "Driver": "windows",
    "Options": null,
    "Config": [
           {
          "Subnet": "172.23.160.0/20",
          "Gateway": "172.23.160.1"
           }
    ]
         },
         ...
         "Containers": {
    "873c787f755e2fa40efde5b59ad68c8b8d16d50abf4118e1de1d1dff718be9e7": {
            "Name": "foo",
            ...
            "IPv4Address": "172.23.173.43/16",
            ...
    }
         },
          "Options": {
    "com.docker.network.windowsshim.hnsid": "78B73D13-9DEF-40F5-9F5C-5C45174C3FF7",
    "com.docker.network.windowsshim.networkname": "nat"
         },
          "Labels": {}
    }
]
```

Figure 23: The instantiated Windows container named "foo" automatically connects to the Docker's default "nat" network. The sample IP 172.23.173.43 of the container is assigned from the internal prefix IP range of 172.23.160.0/20

Upon establishing an interactive session with any instantiated container, as shown in **Figure 24**, we can verify the container's network interface by executing the command `ipconfig /all`. In addition, it can be revealed that the container, for currently unknown reasons, fails to ping the default gateway of its own network. Additionally, this gateway is configured as the default DNS server of a Windows Container, which leads to the issue that commands such as `Invoke-WebRequest` or `nslookup`, as well as tools such as `pip, npm, etc`., cannot resolve company-specific domain names and thus, are not working correctly, even in case the container can connect to the gateway.

```
# Launch standard Windows 10 (20H2) container
PS C:\Windows\Users\{MY_USER}>docker exec-it foo powershell.exe

# Inspect network settings and interface
PS C:\> ipconfig /all

Windows IP Configuration
     Host Name . . . . . . . . . . . . : 873c787f755e
     ...
     DNS Suffix Search List. . . . . . : {COMPANY_SUFFIX}

Ethernet adapter Ethernet:
     Connection-specific DNS Suffix . : {COMPANY_SUFFIX}
     Description . . . . . . . . . . . : Microsoft Hyper-V Network Adapter
     ...
     Link-local IPv6 Address . . . . . : fe80::9cc8:6bd8:d3d7:fbfae%4(Preferred)
     IPv4 Address. . . . . . . . . . . : 172.23.173.43(Preferred)
     Subnet Mask . . . . . . . . . . . : 255.255.240.0
     Default Gateway . . . . . . . . . : 172.23.160.1
     ...
     DNS Servers . . . . . . . . . . . : 172.23.160.1
                             10.111.11.35
     NetBIOS over Tcpip. . . . . . . . : Disabled

# Reach the gateway of the "nat" network
PS C:\> ping 172.23.160.1

Pinging 172.23.160.1 with 32 bytes of data:
Request timed out.
Request timed out.
```

Figure 24: Default network setup of a running Windows container where the default DNS server is set to the network gateway. Under certain but unknown conditions, the container is not able to reach the gateway and, thus, to connect to any internal orexternal target

To resolve these issues, the first step involves configuring the Windows container engine to prevent utilizing the default gateway as the default DNS server for NAT-mode networks. This configuration can be accomplished by deploying the admin-settings.json file, as shown in **Figure 14**, where the value for `com.docker.network.windowsshim.disable_gatewaydns` is set to false. Unfortunately, this global configuration is not automatically applied to the default "nat" network. Consequently, the second crucial step involves creating a custom NAT-mode network to which any instantiated container needs to be connected by providing the `--network` flag. It is essential to ensure that the first three octets of both the gateway and subnet align, as depicted in **Figure 25**. This ensures that the container can resolve its network-related problems.

```
# Creating a custom NAT-mode network named "nat2"
PS C:/Users/{USER}> docker network create-d nat--gateway172.9.128.1--subnet 172.9.128.0/24nat2

# Connecting a Windows container to the custom "nat2" network
PS C:/Users/{USER}> docker run--rm-it--network nat2 mcr.microsoft.com/windows:20H2powershell.exe

# Try to ping the default gateway
PS C:/> ping 172.9.128.1
Pinging 172.9.128.1 with 32 bytes of data:
Reply from 172.9.128.1: bytes=32 time=20ms TTL=248
CTRL + C

Ping statistics for 172.9.128.1:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 20ms, Maximum = 20ms, Average = 20ms
```

Figure 25: Creating a custom NAT-mode network with modified network settings to overcome networking issues for any local Windows container that connects to this network

Finally, persistent networking issues might arise from the Windows host's strict firewall rules, which actively deny specific network traffic from local Windows containers. In this context, it is essential to precisely review the Windows firewall log files under `C:\Windows\System32\LogFiles\Firewall` and define suitable ALLOW rules to regulate both incoming and outgoing network traffic fine-granularly. **Figure 26** below illustrates how the Windows Defender Firewall logs an actively dropped ping command.

```
2024-02-09 10:00:00 DROP ICMP 172.9.128.54 172.9.128.109 10:00:00 DROP ICMP 172.9.128.54
172.9.128.1--6060----8 08 0-RECEIVE
```

Figure 26: Sample log of a dropped ping initiated by a Windows container with IP 172.9.128.55 to the default gateway with destination IP 172.9.128.1. The DROP, in turn, demonstrates enabled network connetivity of a Windows containers

Further enterprise considerations

Docker Desktop strongly emphasizes human-machine interaction, making it an ideal container manager for the local developer workstation. However, many organizations find a need to supplement capabilities with other tools in CI/CD environments. Consequently, enterprises shall utilize both Windows and Linux Docker Community Edition (CE) or any other suitable container manager when integrating container workloads into their pipelines.

Furthermore, it is highly recommended that companies incorporate available container security tools into their development workflows. A highly advisable tool is *Docker Bench For Security*, which is a comprehensive and user-friendly self-assessment tool that validates Linux-based images against best practices. By utilizing this tool, container creators can enhance the quality of their images and ensure adherence to security standards. Unfortunately, *Docker Bench for Security* does not support Windows Containers. In addition, the usage of *Docker Scout*, which is available for business customers, significantly increases the image quality as it can be used for vulnerability scanning, creation of so-called *Software Bills of Materials* (SBOMs), and gives proactive advice on overcoming existing vulnerabilities.

Finally, weakly designed container images, such as those utilizing outdated base images, sharing sensitive information, granting unnecessary read or write permissions, or initiating as privileged users during startup, present significant security risks to the host system, particularly when deployed in production environments. For instance, running a Linux container with `root` privileges or a Windows container as the `containeradministrator` user allows attackers a straightforward pathway to compromise the container and the underlying host system by launching elevated processes. For example, attackers could exploit this vulnerability by leveraging package managers to install tools for SSH usage, inject private SSH keys, and execute privilege escalations. Hence, enterprises must establish a centralized approach to oversee and control the image build and distribution process.

References

- Cyberark Software Ltd. (2023, April 19). *Breaking Docker Named Pipes SYSTEMatically. Docker Desktop Privilege Escalation.* Retrieved from https://www.cyberark.com/resources/threat-research-blog/breaking-docker-named-pipes-systematically-docker-desktop-privilege-escalation-part-2
- Docker Inc. (2023). *Docker Bench Security Official Repository*. Retrieved from https://github.com/docker/docker-bench-security
- Docker Inc. (2023). *Hardened Desktop Whitepaper.* Retrieved from https://www.docker.com/wp-content/uploads/2023/01/hardened-desktop-whitepaper.pdf
- Docker Inc. (2024). *Docker Desktop Documentation*. Retrieved from https://docs.docker.com/desktop
- Loewen, C. (2023, 11 15). *Windows Command Line*. Retrieved from https://devblogs.microsoft.com/commandline/new-enterprise-grade-security-controls-for-the-windows-subsystem-for-linux/
- Microsoft Corp. (2024). *Container Security*. Retrieved from https://learn.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/container-security
- Microsoft Corp. (2024). *Windows Container Engines*. Retrieved from Get started: Prep Windows for containers: https://learn.microsoft.com/en-us/virtualization/windowscontainers/quick-start/set-up-environment
- Microsoft Corp. (2024). *WSL Repository*. Retrieved from WSL Repository: https://github.com/microsoft/WSL